Aprendizaje Reforzado y Aprendizaje Profundo en Aplicaciones de Robótica de Enjambre

Eduardo Andrés Santizo Olivet

UVG UNIVERSIDAD DEL VALLE DE GUATEMALA

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Aprendizaje Reforzado y Aprendizaje Profundo 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,

2021

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

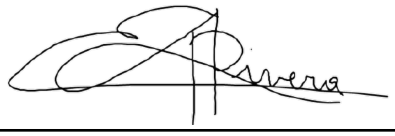
Aprendizaje Reforzado y Aprendizaje Profundo en Aplicaciones de Robótica de Enjambre

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

Guatemala,

2021

Vo.Bo.:



(f) _____

Dr. Luis Alberto Rivera Estrada

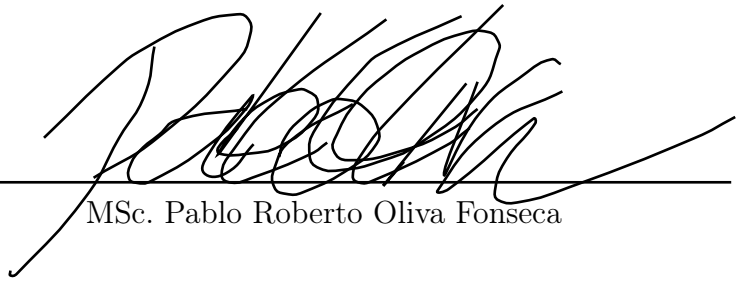
Tribunal Examinador:

(f) _____

Dr. Luis Alberto Rivera Estrada

(f) _____

MSc. Miguel Enrique Zea Arenales



(f) _____

MSc. Pablo Roberto Oliva Fonseca

Fecha de aprobación: Guatemala, 13 de Enero de 2021.

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 (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 such as 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 7,700 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 example *Gridworld* [3] was used as a basis. In this example, an agent moves through a state space represented as 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 increases 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 (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.

¿Existe 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 facilitar la realización de pruebas también se presenta un conjunto de herramientas auxiliares programadas en Matlab (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"[2], 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.

[2] 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 [1], 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 [9].

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 (ver Figura 1). 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 [1], 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.

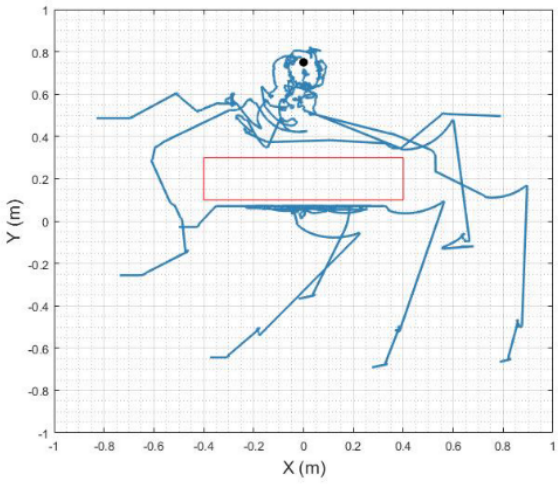


Figura 1: Trayectorias seguidas por E-Pucks en caso A alrededor del obstáculo colocado [2].

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[3]. 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 [10], 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

[3]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 [11], 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 *Markov Decision Process* (MDP) pueden ser construidas según las observaciones conjuntas de múltiples robots explorando de manera simultánea el ambiente [12]. 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 diseñó 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 servirá para mejorar al 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.

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[4], 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 [13] y experimentaron con el mismo. Luego de múltiples pruebas, se percataron que el algoritmo presentaba las cualidades de un método de optimización. Según 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* ($\vec{g}(t)$).

[4]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 [14], presentaba un comportamiento oscilatorio o divergente en ciertas situaciones, por lo que en 2002, [15] 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"":

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

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

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

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. Aprendizaje profundo

En la actualidad, los términos "aprendizaje automático" y "aprendizaje profundo" se han convertido en sinónimos de inteligencia artificial, pero en muchas ocasiones se desconoce la

diferencia entre ambos. De acuerdo con [16], 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 [16].

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.

$$\begin{aligned} \boldsymbol{Z} &= \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{B} \\ \boldsymbol{A} &= \sigma(\boldsymbol{Z}) \end{aligned} \tag{3}$$

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).

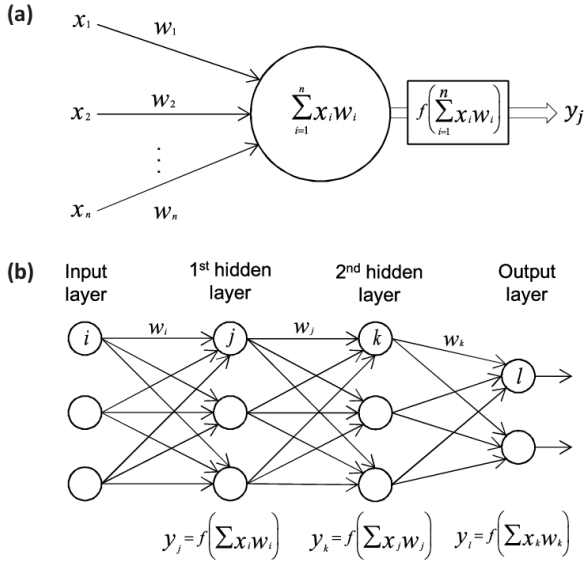


Figura 2: (a) Estructura de neurona. (b) Ejemplo de una red neuronal [17].

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 [16].

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* [18].

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 (como el observado en la Figura 3), esta única neurona tiende a "desplegarse" con fines demostrativos.

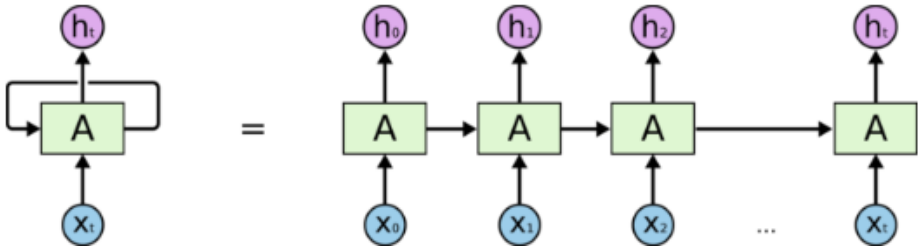


Figura 3: Representación desplegada de una red neuronal recurrente [19].

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.

$$\begin{aligned} a^{<t>} &= g\left(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a\right) \\ \hat{y}^{<t>} &= g\left(W_{ya}a^{<t>} + b_y\right) \end{aligned} \tag{4}$$

En este conjunto de ecuaciones, la función g representa una función de activación. Co-

múnmente para el cálculo de $a^{<t>}$ se emplea la función $ReLU$ o tangente hiperbólica, mientras 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 [18]. Esta notación es comúnmente simplificada para obviar el uso de múltiples constantes W en la ecuación para $a^{<t>}$

$$\begin{aligned} a^{<t>} &= g\left(W_a\left[a^{<t-1>}, x^{<t>}\right] + b_a\right) \\ \hat{y}^{<t>} &= g\left(W_{ya}a^{<t>} + b_y\right) \end{aligned} \tag{5}$$

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 5, 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 dos 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)

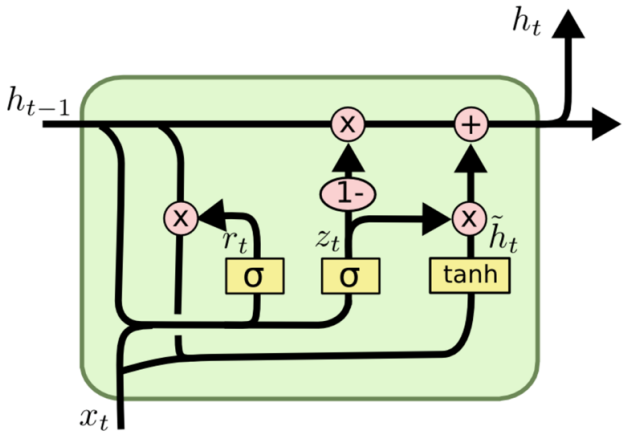


Figura 4: Estructura interna de una neurona GRU [20].

Simplificación de la neurona LSTM. Debido a su estructura y complejidad reducida (Figura 4), estas comúnmente tienden a utilizarse en mayores números para capturar dependencias entre muestras temporales (*time steps*) que se encuentran muy lejanas entre sí temporalmente. Las ecuaciones que rigen dicha neurona, son una versión modificada de las ecuaciones 4 y 5.

$$
\begin{aligned}
z_t &= \sigma\left(W_z \cdot [h_{t-1}, x_t] + b_z\right) \\
r_t &= \sigma\left(W_r \cdot [h_{t-1}, x_t] + b_r\right) \\
\tilde{h}_t &= \tanh\left(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h\right) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
\tag{6}
$$

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) [18].

Long Short Term Memory **(LSTM)**

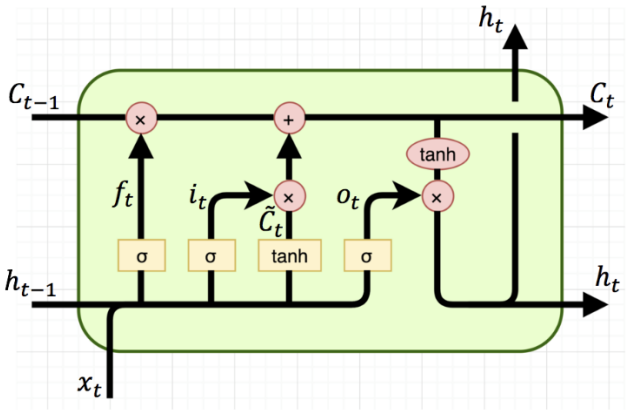


Figura 5: Estructura interna de una neurona LSTM [20].

Versión generalizada de la neurona GRU (Figura 5). 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 [18]. Las ecuaciones que rigen el comportamiento de esta neurona son las siguientes:

$$
\begin{aligned}
f_t &= \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \\
i_t &= \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \\
o_t &= \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) \\
\tilde{C}_t &= \tanh\left(W_c \cdot [h_{t-1}, x_t] + b_c\right) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
h_t &= o_t \odot \tanh\left(C_t\right)
\end{aligned}
\tag{7}
$$

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) [18].

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).

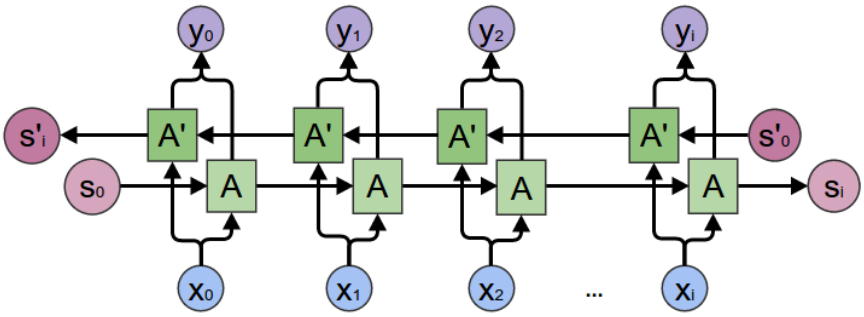


Figura 6: Representación desplegada de una red neuronal recurrente bidireccional [21].

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.

6.2.4. Conceptos útiles en aprendizaje profundo

En una red neuronal existen diferentes parámetros, medidas, métricas y conceptos que se utilizan muy comúnmente para referirse a diferentes elementos del proceso de entrenamiento y validación. Algunos de los más importantes se definen a continuación.

- *Batch size*: Número de muestras que la red neuronal es capaz de procesar en cada iteración del algoritmo de entrenamiento. Cuando se habla de un *Mini Batch Size*, se hace referencia al hecho que los *batches* alimentados a la red contienen menos muestras que el número total de muestras existente en la totalidad de los datos de entrenamiento [22].
- *Epoch*: Cantidad de veces que la red neuronal procesa los datos de entrenamiento en su totalidad [23].
- Iteraciones (aprendizaje profundo): Número de *batches* que se deben alimentar a la red neuronal para completar una *Epoch*.
- Datos de validación: Para determinar si una red está generando predicciones adecuadas sobre los datos de entrenamiento, se puede emplear un conjunto de datos ajeno a las muestras de entrenamiento, para "validar" si el modelo entrenado produce resultados aceptables incluso utilizando muestras nunca antes vistas por la red [18].
- Frecuencia de validación: Cada cuantas iteraciones se valida el modelo entrenado utilizando los datos de validación.
- *Learning rate*: Parámetro que controla "la cantidad de cambio" que experimentan las constantes de la red neuronal en función de su error. Valores muy bajos pueden causar que el proceso de optimización se atore en valles de la función de costo, mientras que valores muy altos pueden llevar a la convergencia temprana del modelo empleando constantes sub-óptimas. Cuando se establece un *Initial Learning Rate* se hace referencia a un valor inicial para el *learning rate* que posteriormente será alterado durante el entrenamiento para auxiliar en el proceso de minimización del error [24].
- *Learning Rate Drop Period:* Cada cuantas *epochs* el algoritmo disminuye el *learning rate* en cierto factor.
- *Learning Rate Drop Factor:* Factor en el que se reduce el *learning rate* luego del *learning rate drop period*.
- *Overfitting:* Fenómeno en el que la red neuronal aprende a imitar perfectamente los datos de entrada, perdiendo la capacidad de generalizar su comportamiento ante la presencia de nuevos datos [18].

6.2.5. *Dropout Layers*

De acuerdo con [25] en redes neuronales que cuentan con un gran número de parámetros (como es el caso en redes neuronales recurrentes), el fenómeno del *overfit* es muy común. Para mitigar su efecto, a las redes neuronales se les puede acoplar un tipo de capa conocida como *dropout layer*.

La idea de estas consiste en aleatoriamente "apagar" ciertas neuronas de la capa previa (siguiendo una cierta probabilidad) para así prevenir que las mismas se "co-adapten". Una "co-adaptación" ocurre cuando dos o más neuronas parecen optimizar sus parámetros conjuntamente para que al combinar sus salidas, generen exactamente los resultados presentes en los datos de entrenamiento. En el momento en que estas neuronas "co-adaptadas" se activan por si solas con datos nunca antes vistos, los resultados son sumamente pobres [25].

6.3. Aprendizaje reforzado

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* (Figura 7) 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

$$
\begin{aligned}
q_*(a) &\doteq \mathbb{E}\left[R_t \mid A_t = a\right] \quad \forall a \in \{1, \ldots, k\} \\
&= \sum_r p(r \mid a) r \\
\text{Donde:} \quad q_*(a) &= \text{Valor de la acción} \\
a &= \text{Acción arbitraria} \\
A_t &= \text{Acción en el tiempo } t \\
R_t &= \text{Recompensa en tiempo } t
\end{aligned} \tag{8}
$$

En otras palabras, el valor se puede re-expresar como la suma de todas las posibles recompensas según su probabilidad de ver dicha recompensa[5]. Si se supiera el valor de cada 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 "acciones avariciosas". 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 "acciones no avariciosa". 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* $t - 1$.

$$
Q_t(a) \doteq \frac{\text{suma de recompensas cuando } a \text{ es tomada previo a } t}{\text{veces que se ha tomado } a \text{ previo a } t} \tag{9}
$$

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, ya que toma el promedio de un conjunto de muestras de las recompensas disponibles.

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

Métodos para elegir acciones

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

$$A_t \doteq \arg\max_a Q_t(a) \tag{10}$$

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 $q_*(a)$

$$A_t \leftarrow \begin{cases} \text{argmax}_a\, Q_t(a) & \text{con probabilidad } 1-\epsilon \\ a \sim \text{ Uniforme } (\{a_1 \dots a_k\}) & \text{con probabilidad } \epsilon \end{cases} \tag{11}$$

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:

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1} \tag{12}$$

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[6]. 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.

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

$$
\begin{aligned}
Q_{n+1} =& \frac{1}{n}\sum_{i=1}^{n} R_i \\
=& \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\
=& \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\
=& \frac{1}{n}\left(R_n + (n-1)Q_n\right) \\
=& \frac{1}{n}\left(R_n + nQ_n - Q_n\right) \\
=& Q_n + \frac{1}{n}\left[R_n - Q_n\right]
\end{aligned} \tag{13}
$$

Donde: Q_n = Estado previo
n = Número de estimado
R_n = Recompensa recibida previamente

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

$$
\text{Nuevo estimado} \leftarrow \text{Viejo estimado} + \text{Step Size}(\underbrace{\text{Objetivo} - \text{Viejo estimado}}_{\text{Medida de Error}}) \tag{14}
$$

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

Algoritmo 1: *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[7].

[7]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*.

Seguimiento 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.

$$Q_{n+1} \doteq Q_n + \alpha \left[R_n - Q_n\right] \tag{15}$$

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.

$$\begin{aligned} Q_{n+1} =& Q_n + \alpha \left[R_n - Q_n\right] \\ =& \alpha R_n + (1-\alpha)\, Q_n \\ =& \alpha R_n + (1-\alpha) \left[\alpha R_{n-1} + (1-\alpha) Q_{n-1}\right] \\ =& \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} \\ =& \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 \alpha R_{n-2} + \\ & \cdots + (1-\alpha)^{n-1} \alpha R_1 + (1-\alpha)^n Q_1 \\ & = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha (1-\alpha)^{n-i} R_) \end{aligned} \tag{16}$$

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: 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 "promedio muestral" (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 según 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 "promedio muestral" 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

$$\beta_n \doteq \alpha / \bar{o}_n \tag{17}$$

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

$$\bar{o}_n \doteq \bar{o}_{n-1} + \alpha \left(1 - \bar{o}_{n-1}\right) \tag{18}$$

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 "promedio exponencial ponderado" como en el método *promedio muestral*.

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

$$A_t \doteq \underset{a}{\operatorname{argmax}} \left[\underbrace{Q_t(a)}_{\text{Explotación}} + c \underbrace{\sqrt{\frac{\ln t}{N_t(a)}}}_{\text{Exploración}} \right] \tag{19}$$

Donde: $t =$ *Time step* actual

$N_t(a) =$ Veces que una acción a ha sido seleccionada antes del *time step* t[8].

$c > 0 =$ 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).

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

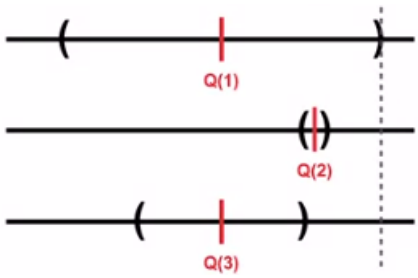


Figura 8: Tres estimados diferentes para el valor de una acción. La línea punteada representa el valor a tomar dado que se trata del límite superior de la incertidumbre [26].

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"[9]. Ambos elementos interactúan de forma continua (como se observa en la Figura 9):

- **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.

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

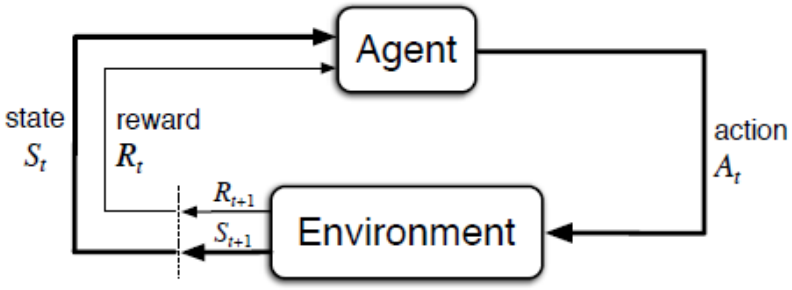


Figura 9: Interacción agente-ambiente en un proceso de decisión de Markov [3].

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 vectores que codifican estos tienen dimensiones pre-determinadas). 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"

$$p\left(s', r \mid s, a\right) \doteq \Pr\left\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\right\} \tag{20}$$

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

$$p\left(s' \mid s, a\right) \doteq \Pr\left\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\right\} = \sum_{r \in \mathcal{R}} p\left(s', r \mid s, a\right) \tag{21}$$

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

$$r(s, a) \doteq \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a\right] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p\left(s', r \mid s, a\right) \tag{22}$$

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

$$r\left(s, a, s'\right) \doteq \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s'\right] = \sum_{r \in \mathcal{R}} r \frac{p\left(s', r \mid s, a\right)}{p\left(s' \mid s, a\right)} \tag{23}$$

Comúnmente se utiliza la notación dependiente de cuatro 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 tres 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

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{24}$$

$$\text{Donde:} \quad T = \textit{Time step} \text{ final}$$

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 reinicio 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:

$$\begin{aligned} \mathrm{S} &= \text{Estados no terminales} \\ \mathrm{S}^+ &= \text{Estados futuros no terminales} \\ T &= \text{Tiempo de finalización} \end{aligned}$$

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"

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{25}$$

$$\text{Donde:} \quad 0 \leq \gamma \leq 1 = \text{Tasa de descuento}$$

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:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots \right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{26}$$

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:

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma} \tag{27}$$

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).

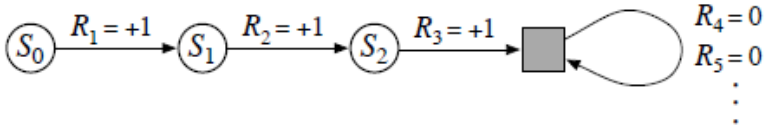


Figura 10: Sumar las recompensas de los primeros tres estados es lo mismo que sumar la serie infinita, ya que luego se pasa al estado de absorción (cuadro gris) [3].

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:

$$G_t \doteq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{28}$$

Donde: $t =$ *Time step* actual
$k =$ Siguiente *time step*

Políticas 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 política. En otras palabras, una política consiste de la forma en la que un agente selecciona una acción. Existen dos tipos:

- Determinista: 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 política solo depende del estado presente. No puede depender de factores como "tiempo" u otros estados. Políticas que dependen de otros factores se consideran "inválidas".

CAPÍTULO 7

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 [27] (e incluso en otros algoritmos PSO [28]).

En la propuesta presentada a continuación, se da vuelta a esta idea: "¿Por qué 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 [29], la regla de actualización de posición y velocidad del algoritmo de PSO original tenía la siguiente forma

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

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

$\odot$ = Multiplicación por elementos (*element-wise product*)[10]

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 en [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 en [30] se propuso una modificación a la regla de actualización de la velocidad de las partículas: La adición de un coeficiente ω multiplicando a la velocidad actual.

$$v(t+1) = \omega v(t) + \vec{U}\,(0,\phi_1) \odot (\overrightarrow{p_{\text{local}}} - \overrightarrow{x_i}) + \vec{U}\,(0,\phi_2) \odot (\overrightarrow{p_{\text{global}}} - \overrightarrow{x_i}) \tag{30}$$

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 [31].

Finalmente, el método más reciente para limitar la velocidad consiste de aquel propuesto en [15]. 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:

$$\begin{aligned} v(t+1) &= \chi\left(v(t) + \vec{U}\,(0,\phi_1) \odot (\overrightarrow{p_{\text{local}}} - \vec{x}_i) + \vec{U}\,(0,\phi_2) \odot (\overrightarrow{p_{\text{global}}} - \vec{x}_i)\right) \\ \chi &= \frac{2\kappa}{\phi - 2 + \sqrt{\phi^2 - 4\phi}} \\ \phi &= \phi_1 + \phi_2 > 4 \end{aligned} \tag{31}$$

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:

$$\begin{aligned} v(t+1) &= \chi v(t) + \chi\vec{U}\,(0,\phi_1) \odot (\overrightarrow{p_{\text{local}}} - \overrightarrow{x_i}) + \chi\vec{U}\,(0,\phi_2) \odot (\overrightarrow{p_{\text{global}}} - \overrightarrow{x_i}) \\ v(t+1) &= \chi v(t) + C_1 \odot (\overrightarrow{p_{\text{local}}} - \overrightarrow{x_i}) + C_2 \odot (\overrightarrow{p_{\text{global}}} - \overrightarrow{x_i})) \end{aligned} \tag{32}$$

[10] 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 de [1]. Esto implica que la ecuación para la actualización de la velocidad de las partículas PSO, tomará la siguiente forma:

$$\begin{aligned} v(t+1) &= \chi\left(\omega v(t) + \vec{U}\left(0,\phi_1\right) \odot (\overrightarrow{p_{\text{local}}} - \vec{x}_i) + \vec{U}\left(0,\phi_2\right) \odot (\overrightarrow{p_{\text{global}}} - \vec{x}_i)\right) \\ \chi &= \frac{2\kappa}{\phi - 2 + \sqrt{\phi^2 - 4\phi}} \\ \phi &= \phi_1 + \phi_2 \end{aligned} \tag{33}$$

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 [15] 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. Sin embargo, el llegar a estructurar ambos elementos de forma óptima trae consigo un largo proceso de prueba y error. Como consecuencia de esto, durante las primeras pruebas preliminares realizadas en la construcción del *PSO Tuner*, no se obtuvieron los resultados deseados, pero si se marcó un precedente para el proceso de diseño posterior.

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*" [32]. 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 tres ecuaciones diferenciales conocido como "Lorenz 63".

$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= x(\rho - z) - y \\ \dot{z} &= xy - \beta z \end{aligned} \tag{34}$$

Se definen las tres ecuaciones, se coloca el valor de las constantes β, ρ y σ, y luego se procede a utilizar un solucionador de ecuaciones diferenciales para simular el sistema. Se realiza un total de 100 ejecuciones del solucionador, 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 periodo de muestreo o *time step* t e intentará estimar su estado en $t+1$. De aquí el nombre *time stepper* empleado previamente.

La red neuronal resultante, es luego comparada contra una resolución analítica de estas ecuaciones por medio del solucionador. Generando la trayectoria del sistema dentro del espacio de estados (Figura 11), se hace evidente que la red es capaz de replicar la trayectoria perfectamente al inicio, pero comienza a perder precisión luego de cierto punto [32].

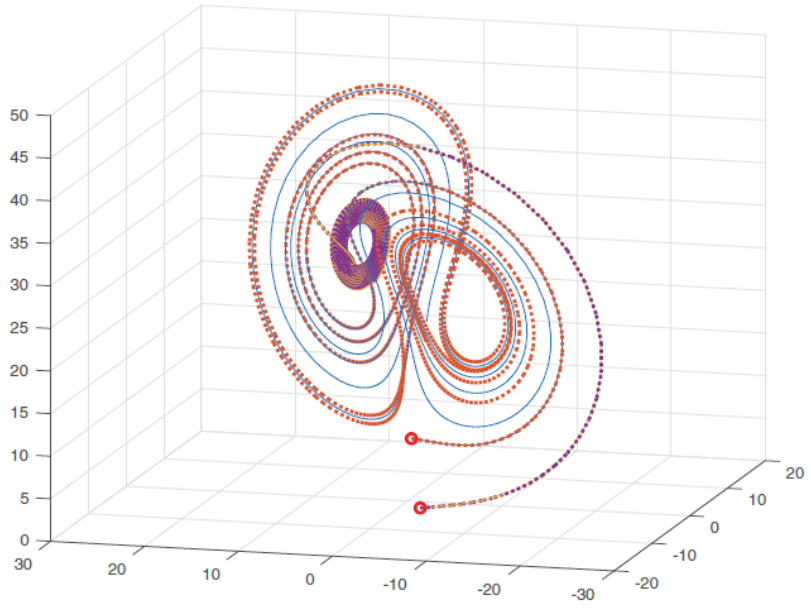


Figura 11: Comparación entre la evolución natural del sistema de Lorenz y la predicción de la red neuronal (línea punteada) para dos condiciones iniciales aleatorias [32].

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 ejecuciones. Por lo tanto, las matrices de input y output consistían de matrices de 100000×2000.

Al intentar entrenar la misma arquitectura de red propuesta por [32] (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 *gigabytes* de RAM para realizar el cálculo, una cantidad exorbitante para una red tan pequeña.

Código 7.1: Creación y entrenamiento de 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 [18].

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 100000×2000, 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 [11].

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 "característica". Por lo tanto, se decidió construir un "vector de características" 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 "características". 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".

Arquitectura	Opciones de entrenamiento					
	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

[11] 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" [33].

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.

Esto se puede evidenciar más claramente en la Figura 12, donde se presenta la posición de las partículas sobre la superficie de costo (superficie azul) durante la iteración final del algoritmo PSO. Como se mencionó previamente, si dicha posición se observa directamente desde arriba (a manera que el eje Z apunte en la dirección del observador) las partículas parecen ubicarse en las esquinas de la mesa de trabajo (representada por la "caja" amarilla).

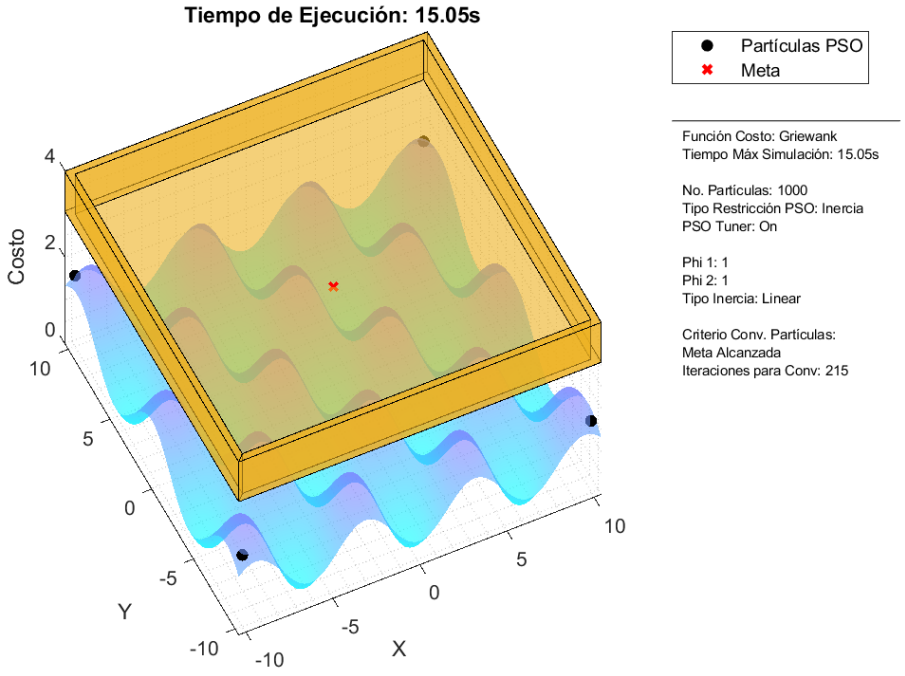


Figura 12: Partículas en las esquinas de la región de búsqueda luego de la ejecución de la primera prueba del PSO Tuner con neuronas LSTM.

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 [18]. 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. Entradas y salidas de red neuronal

Luego de las dos pruebas previamente descritas, era claro que se requería de una reestructuración del vector de entrada para la red neuronal. La estructura previamente propuesta (consistente de todas las coordenadas, medias y desviaciones estándar concatenadas verticalmente) 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 *vector de características* 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 es la utilización de un método capaz de "comprimir" la información contenida en las "características" hacia una representación de menor dimensionalidad. Para esto existen diferentes estrategias: *Principal Component Analysis* (PCA), *auto-encoders*, redes neuronales con una gradual reducción en el número de neuronas por capa, entre otros.

No obstante, una reducción de dimensionalidad previa a través de estos métodos puede traer consigo un incremento significativo en la carga computacional asociada a cada iteración del algoritmo PSO. Para evitar este problema, se puede optar por realizar este proceso de reducción de dimensionalidad de manera "manual", a través de la selección de un conjunto de métricas que describan el estado actual del enjambre. Algunas de las métricas consideradas se presentan a continuación.

7.3.1. Métricas de PSO

Diámetro y radio de enjambre

De acuerdo con [34], el diámetro del enjambre se define como la distancia máxima existente entre cualquier pareja de partículas en el enjambre. Matemáticamente esto se puede describir de la siguiente manera:

$$|D| = \max_{(i \neq j) \in [1,|S|]} \left(\sqrt{\sum_{k=1}^{|K|} (x_{ik} - x_{jk})^2} \right) \tag{35}$$

Donde $|S|$ consiste del número de partículas, $|K|$ representa la dimensionalidad de la superficie sobre la que se desplazan las partículas y x_{ik} / x_{jk} consisten de las posiciones de las partículas i y j en la k-ésima dimensión. Para el caso del radio, este se define como la distancia máxima existente entre el centro del enjambre ($\overline{\mathbf{x}}$: La media de todas las posiciones de las partículas en el enjambre) y la partícula más lejana a dicho centro. Esta se define de la siguiente manera:

$$|R| = \max_{i \in [1,|S|]} \left(\sqrt{\sum_{k=1}^{|K|} (x_{ik} - \overline{\mathbf{x}}_k)^2} \right) \tag{36}$$

Ambas métricas se pueden utilizar para cuantificar la dispersión del enjambre, con valores altos representando una mayor dispersión y valores bajos, un mayor nivel de convergencia. La desventaja de estas métricas es que son altamente susceptibles a valores atípicos, ya que si una única partícula del enjambre se encuentra en una posición extrema, la métrica retornará un valor alto, no importando si el resto del enjambre ha convergido.

Diversidad normalizada

Calculado de la misma manera que el radio de enjambre, con la excepción que en este caso, en lugar de obtener la distancia máxima, todas las distancias hasta el centro del enjambre son promediadas y luego normalizadas dividiendo el promedio dentro del diámetro de enjambre. Debido a su normalización, sus valores fluctúan en el rango (0,1) [35].

$$\mathcal{D}^N = \frac{1}{|S| \cdot |D|} \sum_{i=1}^{|S|} \sqrt{\sum_{k=1}^{|K|} (x_{ik} - \overline{\mathbf{x}}_k)^2} \tag{37}$$

Esta consiste de una medida de la dispersión de las partículas, con un valor bajo indicando convergencia y un valor alto un mayor grado de dispersión. Sin normalizar, esta métrica es ligeramente más robusta que el diámetro o radio de enjambre, ya que el uso del promedio mitiga en cierta medida el efecto de valores atípicos. A pesar de esto, la introducción del diámetro de enjambre como variable de normalización trae de regreso la sensibilidad a valores atípicos. Cabe mencionar que si se desea, también se puede utilizar al radio de enjambre ($|R|$) como variable de normalización [34].

Promedio de la Distancia Promedio entre Todas las Partículas del Enjambre

Se toma cada partícula en el enjambre y se calcula la distancia promedio hacia todas las demás partículas. Todos los promedios por partícula son luego promediados para obtener un único valor escalar por enjambre.

$$\mathcal{D}_{all} = \frac{1}{|S|}\sum_{i=1}^{|S|}\left(\frac{1}{|S|}\sum_{j=1}^{|S|}\sqrt{\sum_{k=1}^{|K|}(x_{ik}-x_{jk})^2}\right) \tag{38}$$

Esta métrica mide la dispersión relativa de todas las partículas en el enjambre con respecto a sus vecinas. Debido a la adición de un segundo promedio, el efecto de valores atípicos sobre esta métrica se reduce con respecto a la diversidad o el diámetro de enjambre. La desventaja de esta métrica es que es considerablemente más compleja de calcular, requiriendo de $|S|$ operaciones más que la diversidad [34].

Coherencia

Razón entre la velocidad del centro del enjambre y la velocidad promedio de las partículas. Debido a que consiste de un cociente, sus valores fluctúan en el rango de (0,1).

$$S_c = \frac{\mathbf{v}_s}{\overline{\mathbf{v}}} \tag{39}$$

Donde $\mathbf{v}_s$ consiste de la velocidad del centro del enjambre

$$\mathbf{v}_s = \frac{1}{|S|}\left\|\sum_{i=1}^{|S|}\tilde{\mathbf{v}}_i\right\|_2 \tag{40}$$

y $\overline{\mathbf{v}}$ consiste de la velocidad promedio de todas las partículas en el enjambre.

$$\overline{\mathbf{v}} = \frac{1}{|S|}\sum_{i=1}^{|S|}\|\tilde{\mathbf{v}}_i\|_2 \tag{41}$$

En ambas cantidades, la operación $\|v\|_2$ representa el cálculo de la velocidad resultante por medio de la combinación de sus componentes. En esta métrica, un valor alto se puede generar por dos factores: Una velocidad de centro de enjambre alta (numerador alto) o una velocidad promedio por partícula muy baja (denominador pequeño). Por lo tanto, una coherencia alta es señal de convergencia (velocidades promedio bajas) o de un movimiento coordinado de las partículas en una dirección específica (todas las velocidades individuales comparten signo, causando que el centro del enjambre se comience a desplazar).

Un valor bajo, por otro lado, se puede generar por: Una alta velocidad promedio por partícula (denominador alto) o una velocidad de centro muy baja (numerador bajo). Entonces una coherencia baja, es señal de dispersión (velocidades promedio muy altas por partícula) o de estancamiento del centro del enjambre (las direcciones de las partículas causan que sus velocidades se cancelen entre si, generando una velocidad de centro baja).

Por lo tanto, cuando la métrica tenga un valor alto, el enjambre estará "trabajando en equipo"; mientras que, cuando tenga un valor bajo, las partículas trabajarán más como unidades individuales. De aquí el nombre "coherencia". Debido a su uso de dos promedios, esta métrica es útil ante la aparición de valores atípicos, aunque siempre se presenta vulnerable ante valores extremos [34].

Desviación estándar promedio

Basado en la forma en la que [1] cuantificó la dispersión de las partículas en el algoritmo MPSO. Esta consiste del promedio de la desviación estándar sobre todas las dimensiones del problema a tratar.

$$\overline{\sigma} = \frac{1}{|K|} \sum_{k=1}^{|K|} \sqrt{\frac{\sum_{i=1}^{|S|} \left(x_{ik} - \overline{\mathbf{x}}_k\right)^2}{|S|}} \tag{42}$$

Donde el término dentro de la raíz consiste de la desviación estándar de las coordenadas sobre la dimensión k y la sumatoria exterior consiste del promedio de las desviaciones estándar sobre todas las $|K|$ dimensiones. Alternativa de mayor rigor estadístico para el resto de métricas de dispersión.

Distancia de meta a *Global Best* normalizada

Métrica para determinar la precisión del enjambre. En esta se calcula la distancia euclideana existente entre la posición del actual *global best* de las partículas (x_{gb}, y_{gb}) y la meta que deben alcanzar las mismas (x_m, y_m). Para su normalización, se toma la distancia entre meta y *global best* y se divide dentro de la distancia máxima que existe entre la meta actual y cada una de las esquinas que definen los límites del área de búsqueda para las partículas (Lims X y Lims Y). Para el caso bidimensional, esta se calcula de la siguiente manera

$$\begin{aligned}
&LimsX = [x_{min} \quad x_{max}] \\
&LimsY = [y_{min} \quad y_{max}] \\
&Meta = [x_m \quad y_m] \\
&\\
&d_x = max(|x_m - x_{min}|, |x_m - x_{max}|) \\
&d_y = max(|y_m - y_{min}|, |y_m - y_{max}|) \\
&\\
&d_{max} = \sqrt{d_x^2 + d_y^2} \\
&d_{gb} = \sqrt{(x_m - x_{gb})^2 + (y_m - y_{gb})^2} \\
&\\
&D_{gb}^N = d_{gb}/d_{max}
\end{aligned} \tag{43}$$

Para problemas de mayor dimensionalidad, el número de distancias (d_x, d_y, ...) incrementa por cada dimensión adicional agregada.

7.3.2. Selección de entradas

Durante las pruebas preliminares, el segundo problema que se tuvo fue el de mediciones ruidosas que impedían que la red dedujera patrones sobre los datos. Entonces, al momento de ensamblar el nuevo "vector de características" de la red neuronal, se intentó elegir métricas que no produjeran datos ruidosos. Debido a esto, se descartaron todas aquellas métricas que tuvieran relación con el diámetro y radio de enjambre, ya que como se describió en la sección 7.3.1, estas mediciones son altamente susceptibles a datos atípicos, lo que implica que pueden llegar a generar datos ruidosos.

Por lo tanto, retirando todas las métricas relacionadas a estos parámetros (diámetro, radio y diversidad normalizada), se obtuvo un total de cuatro entradas para la red neuronal: Promedio de la distancia promedio entre todas las partículas del enjambre, coherencia, desviación estándar promedio y distancia de meta a *global best* normalizada.

Cabe mencionar que, de acuerdo con [36], una de las mejores prácticas al momento de entrenar a una red neuronal, consiste en normalizar los datos de entrenamiento para que su media se encuentre cercana a 0. Esto permite que redes neuronales con funciones de activación "tanh", converjan más rápidamente hacia un conjunto de parámetros óptimos. Erróneamente, esta idea fue interpretada como la acotación de los datos alrededor del rango entre (0,1)[12], por lo que se tomaron las dos cantidades no acotadas del "vector de características" (D_{all} y $\overline{\sigma}$) y se normalizaron en función de las dimensiones del espacio sobre el que se desarrollaría el problema. Dado que se consideró exclusivamente el caso bidimensional para el movimiento de las partículas, el ajuste fue el siguiente:

$$D_{all}^{N} = \frac{D_{all}}{d_{max}} \qquad \sigma_x^N = \frac{\sigma_x}{|x_{max} - x_{min}|}$$

$$\sigma_y^N = \frac{\sigma_y}{|y_{max} - y_{min}|}$$

$$\overline{\sigma}^N = \frac{\sigma_y^N + \sigma_x^N}{2}$$

Tomando estos cambios en cuenta y concatenando todas las métricas verticalmente, el "vector de características" resultante tomó la forma de un vector de 4x1. En la sección 13.1 se presenta el código utilizado para calcular cada una de estas métricas dentro de Matlab, haciendo uso de diferentes funciones de la *SR Toolbox*.

$$\text{Vector de características} = \begin{bmatrix} \overline{\sigma}^N \\ S_c \\ D_{gb}^N \\ D_{all}^N \end{bmatrix}$$

[12] Acción que no necesariamente es incorrecta. Según [36], una normalización en el rango de (0,1) es útil para redes que hacen uso de la función de activación "sigmoide". No obstante, en el caso de una red neuronal recurrente, todas las funciones de activación consisten de "tanh" por defecto

7.3.3. Selección de salidas

Para las salidas de la red neuronal se decidió continuar con la misma estructura de salidas correspondiente a la segunda prueba preliminar (sección 7.2.2): ω, ϕ_1, ϕ_2 y el número de iteraciones para converger.

Siguiendo las ideas observadas en el caso del vector de entrada, se consideró también acotar las salidas en el intervalo (0,1). Realizar esto implicaba seleccionar un valor máximo arbitrario para ω, ϕ_1 y ϕ_2, a manera que, al dividir dichos valores dentro de su máximo, retornen una cantidad entre 0 y 1. No obstante, esto limitaba significativamente la libertad brindada al algoritmo y agregaba hiper parámetros adicionales a ajustar. Entonces, se decidió obviar este paso para los primeros tres parámetros.

La única cantidad que contaba con una "variable de normalización natural", era el número de iteraciones, ya que el algoritmo PSO posee un número de iteraciones máximo seleccionado por el usuario. Entonces, se tomó el número de iteraciones para converger (Iter_{conv}) y se dividió dentro del número de iteraciones máximo (Iter_{max}).

$$\text{Iter}^N = \frac{\text{Iter}_{conv}}{\text{Iter}_{max}} \tag{44}$$

La variable normalizada se concatenó verticalmente con los tres parámetros restantes para dar forma al "vector de respuestas" de la red neuronal.

$$\text{Vector de Respuestas} = \begin{bmatrix} \tilde{\omega} \\ \tilde{\phi_1} \\ \tilde{\phi_2} \\ \text{Iter}^N \end{bmatrix}$$

7.4. Datos de entrenamiento

7.4.1. Estructura de datos

En la sección 7.2.2, se hizo mención de la estructura empleada para los datos de entrenamiento de la primera red LSTM: Una matriz bidimensional con tantas filas como "características" y tantas columnas como *time steps* en la secuencia de entrenamiento. La estructura de las salidas era similar: tantas filas como "respuestas" y tantas columnas como *time steps*.

Esta estructura es útil cuando se desea entrenar a la red sobre una única secuencia de datos. No obstante, para el *PSO Tuner*, se desea generar una gran cantidad de simulaciones y alimentar a la red con la información de cada una de estas. Una opción posible es concatenar horizontalmente todas las secuencias generadas en una sola matriz de gran tamaño (colocando cada simulación "una al lado de la otra"). No obstante, si se le alimenta de esta

manera los datos a la red, esta considerará a todas las simulaciones como una única y muy larga secuencia de datos. Esto puede traer consigo resultados inesperados y adversos en las respuestas generadas, por lo que se debe utilizar otra estructura.

Para datos de carácter secuencial, Matlab permite la creación de un vector columna de celdas, con cada fila de este vector conteniendo la totalidad de una secuencia de datos. Esto implica que, en lugar de alimentar a la red con una sola matriz que contenga una secuencia, esta puede entrenarse con tantas secuencias o muestras como se desee. Cuántas secuencias o muestras (M) se alimenten de manera simultánea a la red se establece a través del parámetro `MiniBatchSize`.

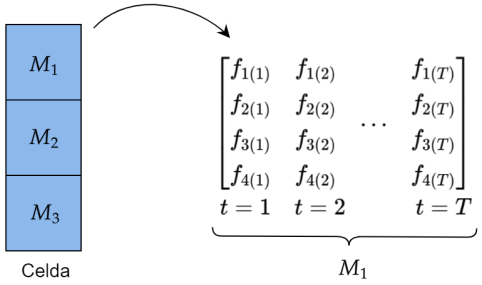


Figura 13: Estructura para los datos de entrenamiento de la red neuronal.

Por lo tanto, los datos de entrenamiento consistirán de dos vectores columna de celdas: Uno para entradas y otro para salidas. Dentro de cada fila del vector existirá una matriz con la forma previamente descrita: tantas filas como "características" o "respuestas", y tantas columnas como *time steps* existan en la simulación (Figura 13). El largo de cada secuencia puede variar entre muestras, pero no entre entrada y salida. Esto implica que para la secuencia contenida en la fila 1 de la celda de entradas, deben haber tantos *time steps*, como en la secuencia contenida en la fila 1 de la celda de salidas.

7.4.2. Generación de datos

Para la generación de los datos de entrenamiento, se tomó el script principal de la *SR Toolbox* (Ver capítulo 9) y se le removieron todas las secciones relacionadas a la visualización de gráficos propias del simulador. Una vez en su forma más básica, el simulador era únicamente capaz de realizar una prueba a la vez, por lo que todos sus componentes se ajustaron para que las condiciones iniciales y parámetros utilizados pudieran cambiarse automáticamente en cada ejecución.

Esta versión del simulador (nombrada `Generar_Dataset.m`) es capaz de tomar una lista de los parámetros que pueden ser alterados y genera una cierta cantidad de simulaciones del algoritmo PSO, utilizando todas las combinaciones posibles de dichos parámetros. Específicamente, el simulador tiene la capacidad de alterar la función de costo a optimizar, el tipo de restricción a emplear por el algoritmo PSO y, en caso la restricción sea igual a "inercia", el tipo de inercia a emplear[13]. A continuación se listan todas las opciones disponibles para

[13]El simulador también es capaz de alterar los obstáculos presentes en la mesa de trabajo en caso la función de costo a optimizar sea de tipo *APF*. A pesar de esto, durante la recolección de datos, el simulador eligió aleatoriamente el tipo de obstáculo a colocar.

estos parámetros.

- Función de costo (11): Banana, Dropwave, Levy, Himmelblau, Rastrigin, Schaffer F6, Sphere, Booth, Ackley, APF y Griewank[14].
- Tipo inercia (5): Constant, Linear, Chaotic, Random y Exponent1.
- Restricción (3): Inercia, Constricción y Mixto.

Para cada función de costo, el *script* realiza un conjunto de simulaciones por tipo de restricción. Si la restricción es igual a "inercia", también se ejecuta el mismo número de simulaciones por tipo de inercia. Tomando esto en cuenta, el número de simulaciones o muestras totales que se recolectarán ejecutando el *script* se puede calcular de la siguiente manera:

$$\begin{aligned} \text{No. Muestras} &= |FC| \times |S| \times (|R| - 1) + |FC| \times |S| \times |I| \\ \text{No. Muestras} &= |FC| \times |S| \times ((|R| - 1) + |I|) \end{aligned} \tag{45}$$

Donde $|FC|$ consiste del número de funciones de costo, $|S|$ del número de simulaciones del algoritmo PSO, $|R|$ del número de restricciones e $|I|$ del número de tipos de inercia. Considerando que se eligió un total de 100 simulaciones por cada permutación de los parámetros, el número total de muestras recolectadas es igual a 7700 secuencias de datos.

$$\text{No. Muestras} = 11 \times 100 \times ((3 - 1) + 5) = 7700$$

Durante cada simulación del algoritmo PSO, se simuló un total 1000 partículas bidimensionales (con posiciones y velocidades en 2 dimensiones) por un máximo de 500 iteraciones. La región de búsqueda para la función de costo se encontraba centrada en el origen y contaba con un ancho y alto de 20 unidades (los límites superiores e inferiores para cada dimensión eran iguales a -10 y 10 respectivamente).

En cada iteración de la simulación, se computaba un *time step* del algoritmo PSO, se calculaban las 4 métricas seleccionadas para el vector de entrada (sección 7.3.2) y se tomaba nota del valor de los parámetros actuales de restricción (ω, ϕ_1 y ϕ_2). En caso las partículas llegaran a converger en algún punto de la función de costo (no necesariamente la meta), se daba fin al algoritmo y se procedía a construir las matrices de entrada y salida.

Para la matriz de entrada, se concatenaba el historial de cada métrica de forma vertical (el historial de cada métrica es una fila en la matriz de entrada), asegurándose de "recortar" la matriz hasta la columna correspondiente a la última iteración del algoritmo. Este proceso se repitió para la matriz de salida, con la diferencia que las filas concatenadas verticalmente consistían de los historiales de los tres parámetros ω, ϕ_1 y ϕ_2, y de una última fila consistente de copias del número de iteraciones para converger normalizadas (se tomaba el número de iteraciones requeridas para converger, se normalizaba y luego se copiaba el resultado obtenido en cada columna de la última fila de la matriz de salida).

[14]Se provee una muestra de la forma general y ecuación correspondiente a cada tipo de función de costo en la sección 13.2

En total, debido a la gran cantidad de ciclos *for* anidados en el *script*, el proceso de generación datos de entrenamiento tomó un total de 1 hora y 45 minutos en una computadora con un CPU Intel i7-4790k de 4.4 GHz, 16 GB de RAM y una GPU NVidia GTX 780. El resultado fueron dos celdas de entrada y salida de 3.51 MB y 1.28 MB respectivamente.

Todo el proceso anterior se repitió una segunda vez para la generación de los datos de validación. El único parámetro alterado fue el número de simulaciones por permutación de parámetros, el cual se redujo a 20. Con esta modificación, el proceso tomó 32 minutos y produjo un total de 1540 muestras de validación.

7.5. Sistema de muestras

Para generar las predicciones de la red LSTM, el *script* de simulación hace uso de la función `predictAndUpdateState()`. Este permite que la red recurrente (que está entrenada para producir secuencias a partir de secuencias) pueda generar sus valores de salida un *time step* a la vez. A pesar de esto, en algunas ocasiones no se recomienda tomar esta vía de acción, ya que las predicciones producidas pueden ser ruidosas o inestables [18].

No obstante, el *PSO Tuner* consiste de una herramienta que genera mejoras en el algoritmo PSO "durante" la ejecución del mismo. Entonces, considerando que las redes recurrentes podrían llegar a requerir de una mayor cantidad de información simultánea para realizar sus estimados, se decidió implementar un intermedio entre la predicción de una secuencia completa y una predicción un *time step* a la vez: el sistema de "muestras". A continuación se describe su funcionamiento.

A medida que se generan métricas, el programa llena un *buffer* de "vectores de características" concatenados horizontalmente. Cuando el *buffer* alcanza el tamaño especificado por el usuario, este se introduce a la red neuronal, produciendo tantas estimaciones de salida como muestras se le alimentaron. El programa toma la última columna de la predicción (correspondiente al último *time step*) y la utiliza como el estimado de los parámetros del algoritmo PSO. En iteraciones posteriores, el *buffer* elimina su muestra más "antigua", desplaza todas sus muestras una columna hacia la izquierda y concatena el siguiente "vector de características" en su última columna. La Figura 14 ilustra este proceso.

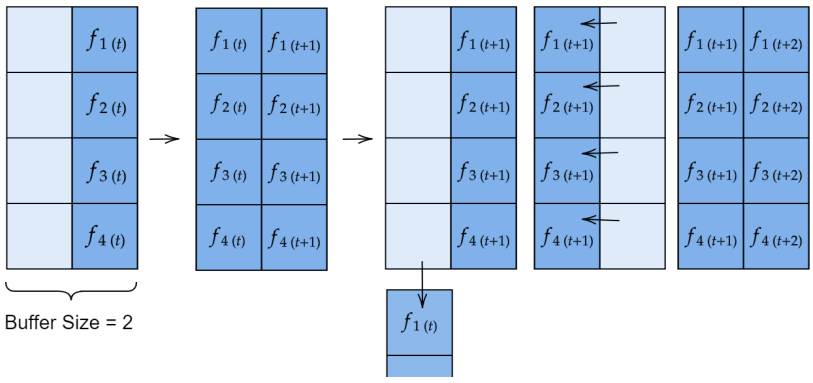


Figura 14: Representación gráfica del funcionamiento del sistema de muestras.

7.6. Tipos de red entrenadas

En total se decidió realizar pruebas con los tres tipos de redes neuronales recurrentes que ofrece la *deep learning toolbox* de Matlab: LSTM, GRU y BiLSTM.

En términos de su capacidad, las neuronas BiLSTM son las más poderosas, siendo capaces de aprender patrones dependientes de información pasada y futura. A estas le siguen las neuronas LSTM, únicamente capaces de emplear información pasada para generar sus estimaciones. Finalmente, se tiene a las neuronas GRU, las cuales fueron específicamente diseñadas para ser una simplificación de la neurona LSTM, por lo que cuentan con una menor capacidad (para más información sobre su estructura y características, consultar las secciones 6.2.1 y 6.2.2)

Se decidió experimentar con estos tres tipos de capa recurrente para observar si variaba la capacidad de aprendizaje de las mismas según las expectativas previamente descritas. Además, dado que la complejidad computacional de los cálculos realizados disminuye con cada descenso en la "jerarquía de capacidad" (mayor carga computacional para BiLSTM y menor para GRU), se deseaba observar cuál era la red más simple capaz de generar resultados satisfactorios.

7.7. Ajuste de hiper-parámetros de redes neuronales

Una red neuronal cuenta con una gran cantidad de variables con las que el investigador puede experimentar: Número de capas, número de neuronas, funciones de activación, *batch size*, *learning rate*, número de *epochs*, etc[15].

En el caso de las tres redes entrenadas, el proceso de ajuste de estos hiper parámetros conllevó a un extenso proceso de prueba y error en el que se alteraban ligeramente los parámetros, se entrenaba la red y luego se analizaban los efectos del cambio realizando una simulación del algoritmo PSO. Para la simulación, nuevamente se empleó una versión modificada del *SR Toolbox* (`Pruebas_DeepPSO.mlx`), a la cual se le retiró la parte asociada al movimiento de robots diferenciales y se le implementaron algunas modificaciones para acoplar la red neuronal al algoritmo PSO.

En primer lugar, se incluyó una sección que calcula en cada iteración del algoritmo las diferentes métricas empleadas para la construcción del "vector de características". Una vez construido, el vector columna de 4×1 es alimentado a la red, produciendo un estimado de ω, ϕ_1, ϕ_2 y el número de iteraciones para converger normalizadas.

El valor de ω, ϕ_1 y ϕ_2 se sustituye dentro del algoritmo PSO para su uso en la siguiente iteración del algoritmo. El estimado para el número de iteraciones, por otro lado, se "des-normaliza" multiplicándolo por el número de iteraciones máximas actuales. El usuario también puede elegir el método de restricción que desea utilizar para definir el valor inicial de los parámetros de restricción ω, ϕ_1 y ϕ_2 (inercia, restricción y mixto).

[15]Para una definición de estos parámetros ver sección 6.2.4

7.7.1. Ajuste de Red LSTM

Debido a que la red LSTM consistió del primer tipo de red neuronal recurrente implementada (sección 7.2.2), se decidió iniciar el proceso de ajuste con la misma. El proceso de diseño para esta red requirió de un mayor número de pruebas, pero los resultados obtenidos fueron sumamente útiles al momento de llevar a cabo el proceso de ajuste de los dos tipos de red restantes.

Prueba 1: *Dropout Layer*

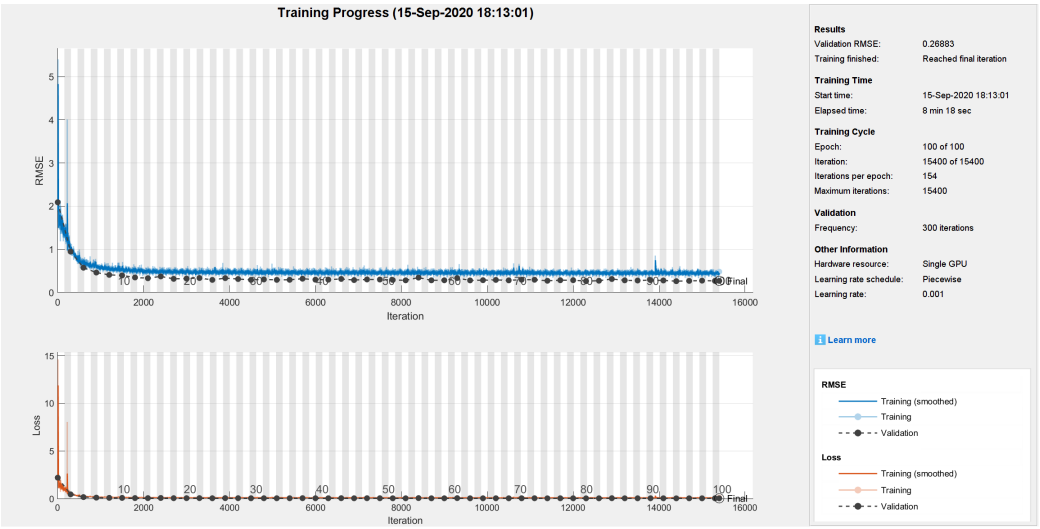


Figura 15: Gráfica del progreso de entrenamiento para la prueba 1 con red LSTM.

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Gradient Threshold	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor
Inputs (4) Neuronas LSTM (200) Fully Connected (50) Dropout (50 %) Ouputs (4) Capa de regresión	ADAM	100	50	1	0.001	1500	0.2

Cuadro 2: Arquitectura y opciones de entrenamiento para la prueba 1 con red LSTM

La primera arquitectura propuesta se basó en la estructura de red empleada en la segunda prueba preliminar (sección 7.2.2). El único elemento adicionado fue una *dropout layer* luego de la *fully connected layer*. Esta adición se realizó con la esperanza de eliminar al fenómeno de *overfit* de la lista de posibles causas para el mal desempeño conseguido durante las pruebas preliminares. A pesar de esto, las partículas continuaban disparándose hacia las esquinas de la región de búsqueda. Una observación importante es que, luego de la introducción de esta capa, el RMSE[16] del proceso de entrenamiento disminuyó significativamente (aproximadamente en dos órdenes de magnitud), indicando cierto grado de mejora.

[16]Error cuadrático medio. Dado que la red fue entrenada para generar una regresión no lineal entre valores, la precisión de sus estimaciones se mide en términos de esta medida.

Prueba 2: Duplicando neuronas en Fully Connected Layer

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas LSTM (200) Fully Connected (100) Dropout (20 %) Ouputs (4) Capa de regresión	ADAM	100	10	0.001	2000	0.2	18m 49s

Cuadro 3: Arquitectura y opciones de entrenamiento para la prueba 2 con red LSTM

Luego de un incremento del número de neuronas en la *fully connected layer* (100), una reducción en el *batch size* (10) y una disminución en el porcentaje de *dropout* (%), la red vio una mejora. Las partículas se disparaban agresivamente hacia las esquinas de la región de búsqueda, pero luego de una cierta cantidad de iteraciones, retornaban y se centraban en la meta. A pesar de esto, el enjambre no convergía. Las partículas permanecían aparentemente estáticas alrededor de la meta (-3,3) con una dispersión constante.

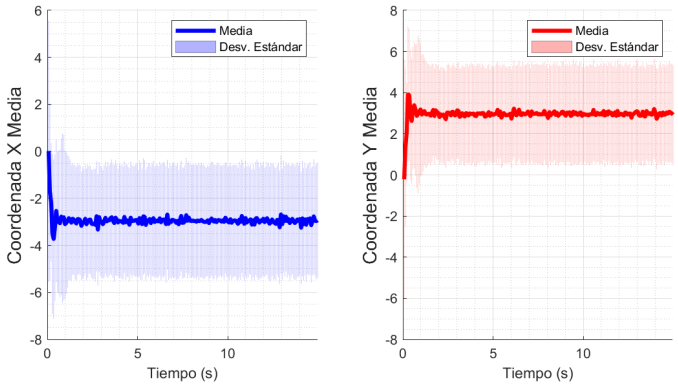


Figura 16: Posición media y dispersión de las partículas en la prueba 2 con red LSTM.

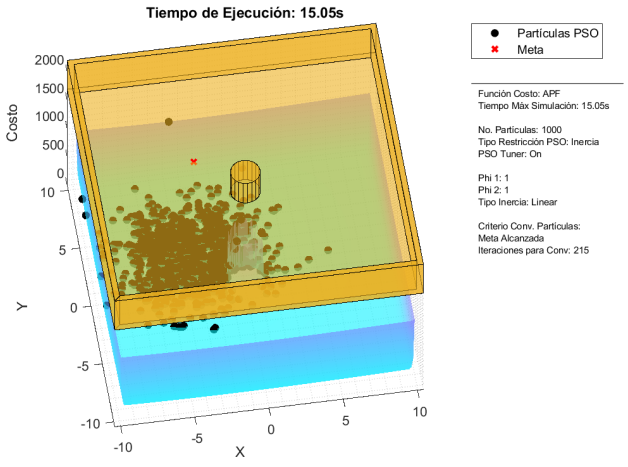


Figura 17: Partículas *estáticas* alrededor de la meta en la prueba 2 con red LSTM.

Prueba 3: 120 Neuronas en *Fully Connected Layer*

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas LSTM (200) Fully Connected (120) Dropout (20 %) Ouputs (4) Capa de regresión	ADAM	100	10	0.001	2000	0.2	18m 57s

Cuadro 4: Arquitectura y opciones de entrenamiento para la prueba 3 con red LSTM

En esta prueba se experimentó con el uso de múltiples muestras, pero no importando el número empleado, la red retornó al comportamiento errático observado en la prueba 1. También se alternó el método de restricción utilizado. De los tres disponibles, el método de restricción mixto propuesto por [1] produjo los mejores resultados. Estos resultados no se consideraron significativos ya que el método mixto tiene la cualidad de converger sumamente rápido, incluso antes de que el propio sistema de muestras finalice el llenado del *buffer*.

Prueba 4: 80 Neuronas en *Fully Connected Layer*

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas LSTM (200) Fully Connected (100) Dropout (20 %) Ouputs (4) Capa de regresión	ADAM	50	10	0.001	2000	0.2	8m 47s

Cuadro 5: Arquitectura y opciones de entrenamiento para la prueba 4 con red LSTM

Luego de reducir el número de neuronas en la *fully connected layer* (80) no se observó mejora alguna. Las partículas continuaban disparándose hacia las esquinas de la región de búsqueda no importando el tipo de restricción o función de costo utilizada. Se decidió entonces, retornar al número de neuronas en la *fully connected layer* que habían generado los mejores resultados hasta el momento (100).

Prueba 5: Intercambiando Dropout y Fully Connected Layers

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas LSTM (200) Dropout (20 %) Fully Connected (100) Ouputs (4) Capa de regresión	ADAM	50	10	0.001	2000	0.2	8m 18s

Cuadro 6: Arquitectura y opciones de entrenamiento para la prueba 5 con red LSTM

Al alterar el orden de la *fully connected* y *dropout layers* se produjo una ligera mejora en los resultados. Las partículas fueron capaces de converger en cualquiera de los tres métodos de restricción (mixto, constricción e inercia), aunque en algunas simulaciones las partículas retornaban a su comportamiento errático sin causa aparente. Esto se atribuyó al proceso de entrenamiento. Probablemente durante el mismo, la red encontró un conjunto de parámetros sub-óptimos para la red neuronal, los cuales causaban inestabilidad en las predicciones producidas. Debido a esta pequeña mejora, se decidió conservar el cambio.

Prueba 6: *Fully Connected Layer* Adicional

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas LSTM (200) Dropout (20 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	50	10	0.001	2000	0.2	12m 7s

Cuadro 7: Arquitectura y opciones de entrenamiento para la prueba 6 con red LSTM

Una de las pruebas más interesantes. El funcionamiento del *PSO Tuner* era dependiente de la función de costo a optimizar, y su grado de efectividad podía llegar a cambiar según el número de muestras elegidas. Para funciones con mínimos "profundos" ("Griewank", "Dropwave" y "Schaffer F6"[17]) se generaban buenos resultados en todos los métodos de restricción, utilizando una única muestra. Para funciones aparentemente "simples" de minimizar ("Esfera" o "Banana"), las partículas retornaban a su comportamiento errático. Para investigar la razón de esta explosión condicional se decidió graficar la evolución de las métricas y salidas producidas por la red neuronal (Figura 18).

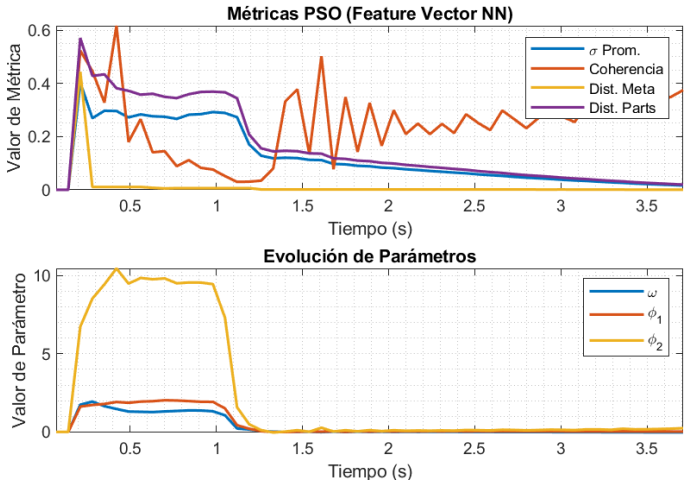


Figura 18: Gráfica de los parámetros de entrada y salida de la red LSTM. Prueba 6.

[17]Se provee una visualización de estas funciones de costo en la sección 13.2

Graficando la evolución temporal de las métricas del algoritmo PSO alimentadas a la red como entradas (gráfica superior de la Figura 18) y la evolución temporal de los parámetros ω, ϕ_1 y ϕ_2 generados como salidas (gráfica inferior de la Figura 18), se realizaron algunos descubrimientos importantes.

En primer lugar, se observó que la desviación estándar promedio (σ *Prom.*) y la distancia entre partículas (*Dist. Parts*) consisten de métricas "hermanas", ya que la distancia promedio parece consistir de una versión amplificada de la desviación estándar promedio. Debido a esta diferencia en la amplitud, la distancia promedio tiende a extenderse hacia valores por encima de 1 (Figura 19), indicando que su normalización no está correctamente aplicada. Por lo tanto, se podría llegar a argumentar que el uso simultáneo de ambas métricas es redundante e innecesario.

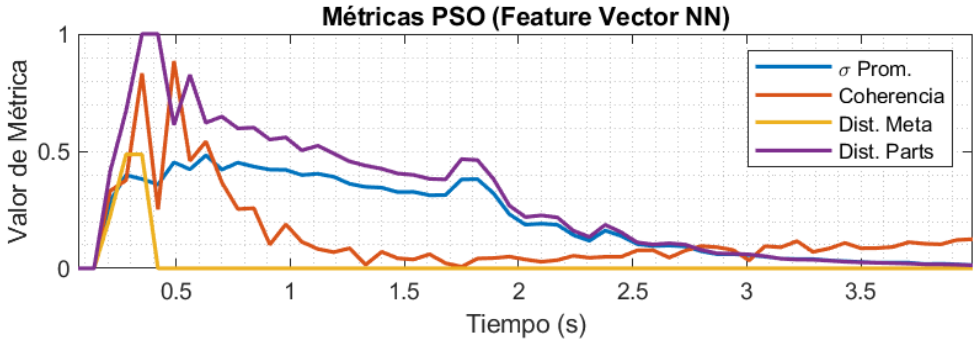


Figura 19: Parámetros de entrada de la red LSTM cuando la distancia entre partículas se disparaba por encima de 1. Prueba 6.

Además de esto, también se logró determinar que la razón por la que las partículas "explotaban" en las funciones de costo "simples" se debía a un incremento repentino en la inercia ω (Figura 20). Por lo tanto, para experimentar, se retiró el control de la red sobre este parámetro y al hacerlo, el *PSO Tuner* pasó a funcionar correctamente con todos los tipos de restricción y funciones de costo disponibles.

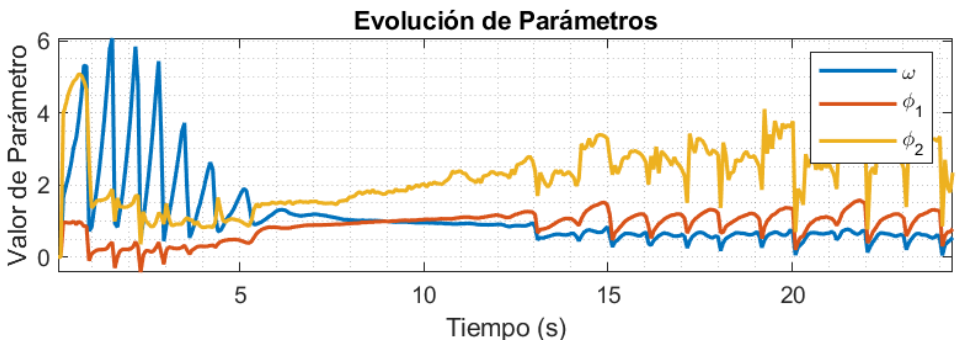


Figura 20: Parámetros de salida de la red LSTM cuando esta tendía a disparar el valor de la inercia por encima del valor de los otros dos parámetros (ϕ_1 y ϕ_2). Prueba 6.

Dado que los resultados parecían prometedores, se decidió mantener la *fully connected layer* adicional, además de implementar de manera permanente la habilidad de desactivar y re-activar el control de todas las redes sobre la inercia.

Prueba 7: Batch Size = 30

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas LSTM (200) Dropout (20 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	50	30	0.001	2000	0.2	5m 47s

Cuadro 8: Arquitectura y opciones de entrenamiento para la prueba 7 con red LSTM

De acuerdo con [37], el *batch size* es uno de los factores más importantes a elegir en una red neuronal recurrente, ya que se puede pensar de este valor como la cantidad de información simultánea que analiza la red para poder encontrar patrones. Realizando una analogía al mundo real: Si una persona desconoce el concepto de un pájaro, será mucho más fácil clasificar otras entidades como pájaros, si se brindan muchos ejemplos de diferentes tipos de pájaros. La situación es similar para una red neuronal, mientras más muestras se le alimenten de manera simultánea a la misma, esta conseguirá generar predicciones con mayor generalidad y precisión.

Según esta observación, se decidió incrementar el valor del *batch size* para observar si esto generaba cambios significativos en el desempeño de la red y en efecto, esta modificación trajo consigo una mejora sustancial. A pesar que la red perdió la capacidad de trabajar con una única muestra, empleando 2 muestras, esta fue capaz de generar un comportamiento óptimo (Figura 21). La red se probó utilizando todos los métodos de restricción y las 15 funciones *benchmark* disponibles y en ninguna combinación se produjo un comportamiento errático.

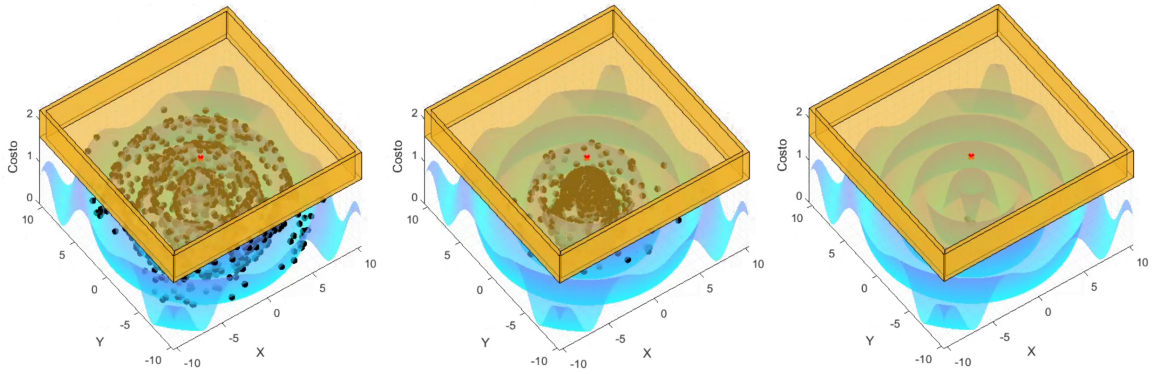


Figura 21: Comportamiento óptimo del algoritmo PSO auxiliado por el PSO Tuner en función de costo Schaffer F6.

También se investigó brevemente el efecto de alterar el número de muestras alimentadas al algoritmo y se pudo determinar que un incremento en el número de muestras producía una mayor "inseguridad" en las partículas al momento de converger (titubeaban antes de agruparse para evitar converger en mínimos locales). Otro comportamiento interesante fue aquel en el que el enjambre parecía copiar el comportamiento de las partículas en el método de restricción elegido. El método mixto producía una convergencia rápida, pero imprecisa,

el de inercia producía una convergencia extendida pero con una mayor exploración y el de constricción producía un intermedio entre ambos. La red parecía "inspirar" su selección de parámetros futuros en el valor inicial de las constantes de restricción.

Prueba 8: Batch Size = 50

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas LSTM (200) Dropout (20 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	50	50	0.001	2000	0.2	4m 39s

Cuadro 9: Arquitectura y opciones de entrenamiento para la prueba 8 con red LSTM

Luego de este segundo incremento al *batch size* la red adquirió la capacidad de trabajar con una sola muestra y comenzó a expresar en mayor medida el comportamiento de "mímica" de los diferentes tipos de restricción. Derivado de esto, la capacidad de minimización de la red se tornó significativamente más robusta. Para probar esto, se redujo el número de partículas simuladas a 10 y se observó el desarrollo de la simulación. Aunque la convergencia ocurrió luego de un periodo extendido de tiempo (858 iteraciones), esta logró alcanzar la posición del mínimo global eventualmente (Figura 22), una tarea compleja para el PSO estándar con pocas partículas. A pesar de esto, cabe mencionar que la capacidad de resolver este tipo de problemas dependía del tipo de restricción utilizada. El caso de 10 partículas únicamente se consiguió resolver con el método de inercia.

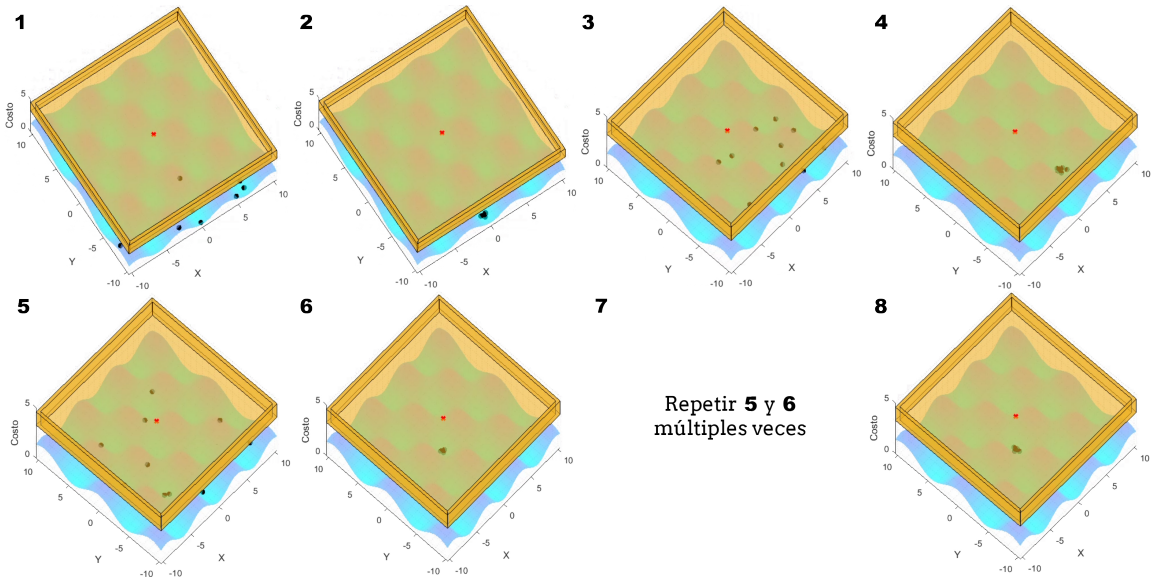


Figura 22: Proceso de minimización de la función de costo Griewank utilizando 10 partículas para la simulación del algoritmo PSO auxiliado por la red LSTM.

Prueba 9: Batch Size = 80

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas LSTM (200) Dropout (20 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	50	80	0.001	2000	0.2	6m 54s

Cuadro 10: Arquitectura y opciones de entrenamiento para la prueba 9 con red LSTM

Con un incremento aún mayor en el *batch size* (80), incluso el método de constricción fue capaz de resolver la minimización de algunas funciones de costo con 10 partículas. Los tiempos de convergencia en general se extendieron, pero los métodos restricción de inercia y constricción se tornaron casi equivalentes en términos de su tiempo de convergencia.

Prueba 10: Batch Size = 100 y Dropout = 40 %

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas LSTM (200) Dropout (20 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	50	100	0.001	2000	0.2	5m 38s

Cuadro 11: Arquitectura y opciones de entrenamiento para la prueba 10 con red LSTM

Esta consistió de la última prueba. Luego de incrementar el *batch size* una última vez (100), así como el porcentaje de *dropout* en la *dropout layer* (40 %), ya no se pudieron observar mejoras sustanciales en el desempeño de la red neuronal. Por lo tanto, este modelo se consideró la arquitectura final para la red LSTM.

Cabe mencionar que hacia el final de estas pruebas, Matlab comenzó a generar un error relacionado con las librerías CUDA que utiliza para poder procesar los datos de entrenamiento por medio de la GPU. El error causaba la finalización prematura del proceso de entrenamiento e impedía guardar el modelo generado por el *deep learning toolbox* de Matlab. El problema parecía desaparecer al reducir el tamaño de la red, no obstante, ya que se deseaba el desempeño obtenido por las redes de mayor complejidad, se optó por encontrar una solución a este problema. En la sección 13.3 de "Anexos" se detalla la solución que pareció generar los mejores resultados.

7.7.2. Ajuste de Red BiLSTM

El proceso de ajuste de la red BiLSTM fue significativamente más corto que el de la red LSTM. En total, se realizaron tres pruebas, todas siguiendo como guía la información descubierta al momento de entrenar a la red LSTM.

Prueba 1

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas BiLSTM (210) Fully Connected (50) Dropout (50 %) Ouputs (4) Capa de regresión	ADAM	100	50	0.001	1500	0.2	11m 27s

Cuadro 12: Arquitectura y opciones de entrenamiento para la prueba 1 con red BiLSTM

Para la primera prueba realizada, se decidió tomar la arquitectura de la primera prueba de la red LSTM (sección 7.7.1) y se modificó ligeramente. Específicamente, se incrementó el número de neuronas recurrentes (210) y el *batch size* (50). La red resultante generaba nuevamente un comportamiento errático, aunque en este caso, las partículas no se disparaban hacia las esquinas de la región de búsqueda. Estas se desplazaban en conjunto hacia una de las esquinas y seguido de cierto tiempo procedían a moverse conjuntamente hacia la esquina opuesta. En funciones de costo como "APF", que cuentan con valores de costo infinitos en los bordes, las partículas parecían desaparecer en una esquina, para luego re-aparecer dirigiéndose a la esquina opuesta.

Debido al comportamiento atípico obtenido, se decidió disminuir de regreso tanto el *batch size* como el número de neuronas BiLSTM y observar el efecto.

Prueba 2: 150 Neuronas BiLSTM y Batch Size de 10

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas BiLSTM (150) Fully Connected (50) Dropout (50 %) Ouputs (4) Capa de regresión	ADAM	100	10	0.001	2000	0.2	12m 44s

Cuadro 13: Arquitectura y opciones de entrenamiento para la prueba 2 con red BiLSTM

Disminuyendo el número de neuronas BiLSTM (150) y el *batch size* (10), la red presentó una mejora en el desempeño considerable. Al igual que en las últimas arquitecturas probadas para la red LSTM, la red BiLSTM comenzó a manifestar la capacidad de alterar su comportamiento base para intentar "imitar" los movimientos propios de un algoritmo PSO utilizando los diferentes tipos de restricción.

Dicho comportamiento se observó empleando una única muestra. Si se incrementaba el número de muestras, los resultados empeoraban y se generaban comportamientos erráticos. Para probar que tan robusto era el algoritmo, nuevamente se redujo el número de partículas y se observó si la red era capaz de hacerlas converger. Luego de un periodo extendido de tiempo utilizando los métodos de constricción e inercia, el algoritmo consiguió que un enjambre de 5 partículas convergiera correctamente en una función con múltiples mínimos locales (Griewank).

Debido al desempeño tan cercano a aquel observado en las últimas iteraciones de la selección de hiper parámetros con la red LSTM, se decidió tomar esta arquitectura como el modelo final a emplear para la red BiLSTM. A pesar de esto, se decidió realizar una prueba adicional agregando una segunda capa de neuronas BiLSTM.

Prueba 3: Segunda capa BiLSTM

Debido a la dimensión temporal que acompaña a las redes neuronales recurrentes, estas nunca tienden a utilizarse en altos números de capa. En el caso de aplicaciones altamente complejas, una red neuronal recurrente de entre 2 a 3 capas se considera suficiente e incluso excesivo [18]. Aún así, en esta prueba se decidió incrementar el número de capas (2). Luego de este cambio, se pudo llegar a observar que este incremento únicamente dañó el desempeño al hacer que la red retornara al comportamiento de explosión de las partículas hacia las esquinas de la región de búsqueda. Este comportamiento se presentaba no importando la función de costo, tipo de restricción o el número muestras alimentadas a la red.

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas BiLSTM (200) Fully Connected (50) Neuronas BiLSTM (110) Dropout (20 %) Ouputs (4) Capa de regresión	ADAM	100	50	0.001	1500	0.2	17m 14s

Cuadro 14: Arquitectura y opciones de entrenamiento para la prueba 3 con red BiLSTM

7.7.3. Ajuste de red GRU

A pesar de haber obtenido una arquitectura con desempeño aceptable en la primera iteración de diseño, el comportamiento de la red GRU nunca llegó a asemejarse a aquel presentado por las redes BiLSTM y LSTM. En el afán de obtener un comportamiento similar, se continuó experimentando con los hiper parámetros de la red, pero todas las pruebas retornaron un resultado similar: Un movimiento de alta dispersión con dificultades para converger.

Primer modelo

Dado que una neurona GRU consiste de una simplificación de una neurona LSTM, se decidió tomar la misma arquitectura final obtenida para la red LSTM, para el entrenamiento de la red GRU. La única alteración al modelo fue la remoción de la segunda *fully connected layer* de 50 neuronas.

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas GRU (200) Dropout (40 %) Fully Connected (100) Ouputs (4) Capa de regresión	ADAM	100	100	0.001	1500	0.2	6m 5s

Cuadro 15: Arquitectura y opciones de entrenamiento para el modelo inicial de red GRU

Inmediatamente los resultados se presentaron muy similares a los previamente observados en la red LSTM, con las partículas siendo capaces de escapar de mínimos locales y presentando patrones de movimiento similares a los métodos de restricción utilizados como base. La diferencia más importante sin embargo, era que el algoritmo era altamente "inseguro" y por lo tanto, en muy raras ocasiones consiguió converger.

En términos de su capacidad para encontrar el mínimo global de la función de costo, este era altamente robusto, siendo capaz de encontrarlo incluso con 100 y 10 partículas. Su desventaja era que al mismo le tomaba mucho tiempo escapar de los mínimos locales y por lo tanto, el tiempo que le tomaba llegar al mínimo global era significativamente más largo que en el caso de las redes LSTM y BiLSTM. Además, para su correcto funcionamiento era necesario el deshabilitar el control sobre la inercia.

Pruebas posteriores

Los resultados del primer modelo entrenado estaban muy cerca de conseguir el desempeño deseado, entonces se optó por continuar alterando diferentes hiper parámetros propios de la arquitectura y las opciones de entrenamiento para intentar alcanzar el comportamiento deseado. En total se realizaron 11 permutaciones a los parámetros (Cuadro 16), pero en casi ninguno de los casos se obtuvo un cambio sustancial.

	Arquitectura	Opciones de entrenamiento						
		Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Resultado
1	Inputs (4) Neuronas GRU (200) Dropout (50 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	100	100	0.001	2000	0.2	Movimiento más ruidoso
2	Inputs (4) Neuronas GRU (180) Dropout (50 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	100	100	0.001	2000	0.2	Movimiento agresivo
3	Inputs (4) Neuronas GRU (220) Dropout (50 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	100	100	0.001	2000	0.2	Sin cambios. Movimiento agresivo
4	Inputs (4) Neuronas GRU (250) Dropout (50 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	100	100	0.001	2000	0.2	Sin cambios. No utilizar más de 1 muestra o el movimiento empeora
5	Inputs (4) Neuronas GRU (200) Dropout (50 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	100	120	0.001	2000	0.2	Menos ruido
6	Inputs (4) Neuronas GRU (200) Dropout (50 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	100	150	0.001	2000	0.2	Aún menos ruido
7	Inputs (4) Neuronas GRU (200) Dropout (50 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	100	180	0.001	2000	0.2	Sin cambios
8	Inputs (4) Neuronas GRU (200) Dropout (50 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	100	50	0.001	2000	0.2	Sin cambios
9	Inputs (4) Neuronas GRU (200) Dropout (50 %) Fully Connected (50) Fully Connected (25) Ouputs (4) Capa de regresión	ADAM	100	100	0.001	2000	0.2	Ruido se potenció. Método de constricción funciona bien

	Arquitectura	Opciones de entrenamiento						
		Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Resultado
10	Inputs (4) Neuronas GRU (200) Dropout (50 %) Fully Connected (150) Fully Connected (100) Ouputs (4) Capa de regresión	ADAM	100	100	0.001	2000	0.2	Ruido impide la convergencia
11	Inputs (4) Neuronas GRU (200) Dropout (20 %) Fully Connected (100) Fully Connected (50) Ouputs (4) Capa de regresión	ADAM	100	100	0.001	2000	0.2	Movimiento ruidoso

Cuadro 16: Pruebas posteriores realizadas con red GRU para intentar mejorar su desempeño

Las pruebas con los mejores resultados fueron la prueba 9, 10 y 11. Estas fueron capaces de reducir el ruido en el movimiento lo suficiente como para permitir que el algoritmo convergiera. La velocidad con la que convergía el mismo variaba según el tipo de restricción elegida. Además, el correcto funcionamiento de la red dependía del uso de dos muestras y de la desactivación de la inercia.

Modelo final

Tomando en cuenta que hacia el final del proceso de diseño se observaron algunas mejoras en el desempeño de la red, se decidió combinar los diferentes parámetros obtenidos en estos modelos para generar un modelo que implementara todas las pequeñas mejoras conjuntas.

Arquitectura	Opciones de entrenamiento						
	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Tiempo de entrenamiento
Inputs (4) Neuronas GRU (200) Dropout (20 %) Fully Connected (150) Fully Connected (100) Ouputs (4) Capa de regresión	ADAM	100	100	0.001	2000	0.2	8m 5s

Cuadro 17: Arquitectura y parámetros de entrenamiento para el modelo final de red GRU

El modelo final no presenta un comportamiento muy alejado de su iteración inicial, con la única diferencia significativa siendo una pequeña reducción en el movimiento ruidoso o de dispersión alta. Requerimientos como el uso de dos muestras para el correcto funcionamiento y la desactivación de la inercia, se mantuvieron para la iteración final de este modelo.

7.8. Modelos finales de redes neuronales

A continuación se lista la arquitectura y opciones de entrenamiento utilizadas para cada uno de los tres tipos de redes neuronales entrenadas como parte del *PSO Tuner*. También se incluye información sobre el número de muestras que deben ser alimentadas a la red, así como si se debe deshabilitar o no la inercia para asegurar su correcto funcionamiento.

		Opciones de Entrenamiento							
Tipo Red	Arquitectura	Optimizador	Max Epochs	Mini Batch Size	Initial Learn Rate	Learn Rate Drop Period	Learn Rate Drop Factor	Número de Muestras	Inercia Habilitada
GRU	Inputs (4) Neuronas GRU (200) Dropout (20 %) Fully Connected (150) Fully Connected (100) Outputs (4) Capa de regresión	ADAM	100	100	0.001	2000	0.2	1	No
LSTM	Inputs (4) Neuronas LSTM (200) Dropout (40 %) Fully Connected (100) Fully Connected (50) Outputs (4) Capa de regresión	ADAM	100	100	0.001	2000	0.2	2	No
BiLSTM	Inputs (4) Neuronas BiLSTM (150) Fully Connected (50) Dropout (50 %) Outputs (4) Capa de regresión	ADAM	100	10	0.001	2000	0.2	1	Si

Cuadro 18: Modelos finales para las redes neuronales recurrentes GRU, LSTM y BiLSTM

7.9. Modo de operación de PSO Tuner

7.9.1. Descripción general

En su comportamiento base (o aquel que presenta no importando el tipo de restricción elegido) todas las redes causan que las partículas exploren la región de búsqueda moviéndose conjuntamente (alta coherencia) mientras mantienen un diámetro de enjambre casi constante. Cuando encuentran un mínimo potencial, las partículas comienzan a aglomerarse para intentar converger. No obstante, si el mínimo encontrado no consiste del mínimo global (detectado gracias a la métrica de la distancia entre el *global best* y la meta), las partículas incrementan repentinamente su dispersión para intentar "escapar" del mínimo local al que han convergido. En caso consigan salir del mínimo local previo, la dispersión vuelve a incrementarse y se comienza nuevamente el proceso de exploración.

Cuando este comportamiento base se expone a cada uno de los tipos de restricción, su comportamiento muta para acoplarse a las características del tipo de restricción. En el caso de la inercia, el método con mayor dispersión, el método adquiere una mayor capacidad para escapar de mínimos locales una vez ha convergido en los mismos. Esto se debe a que el mayor movimiento relativo de las partículas individuales (dispersión alta y cohesión baja) le permite "acelerar" para generar la velocidad suficiente para escapar de los mínimos encontrados.

En el caso del método mixto, el algoritmo adquiere un alta susceptibilidad a mínimos locales. Cuando las partículas encuentran un mínimo, todas incrementan su velocidad para

apresurarse a alcanzar dicho punto. Si el mínimo no es el correcto, las partículas intentan "escapar", pero debido a la influencia del método de restricción mixto, estas carecen de la velocidad suficiente para conseguir salir del mínimo encontrado.

Finalmente, para el caso de la constricción, se consigue un resultado intermedio: Las partículas convergen rápidamente hacia el primer mínimo que encuentren, pero si se percatan que este consiste de un mínimo local, estas alteran su dispersión para intentar escapar. Su escape no se presenta de forma tan fácil e inmediata como en el caso de la restricción por inercia, pero luego de cierta cantidad de intentos lo consiguen.

7.9.2. Descripción según los parámetros de entrada y salida

El comportamiento emergente previamente descrito puede no hacer sentido al inicio, pero este puede explicarse de forma adecuada al observar la evolución de los parámetros de entrada y salida de la red a lo largo del tiempo (Figura 23).

Al inicio de su funcionamiento la red dispara agresivamente el valor de ϕ_2 mientras incrementa en menor medida el valor de ω y ϕ_1. Una vez las partículas encuentran la meta, la red disminuye ligeramente el valor de ω para limitar la dispersión. Esto genera a su vez una gradual (aunque ruidosa) disminución en la coherencia de las partículas, ya que el centro del enjambre permanece estático. Cuando la coherencia alcanza un valor lo suficientemente bajo, la red disminuye agresivamente el valor de ϕ_1, ϕ_2 y ω, reduciendo consigo el movimiento ruidoso característico del algoritmo PSO.

Los nuevos valores de ϕ_1, ϕ_2 y ω, consisten de valores muy cercanos a 0, pero de los tres, ϕ_2 tiende a presentar el valor más alto. Debido a que ϕ_2 regula la influencia de la memoria global del enjambre, un valor más alto de esta constante, en conjunto con un movimiento menos ruidoso, se traduce en un suave movimiento hacia la meta encontrada previamente. Curiosamente esto también coincide con un incremento gradual en la coherencia del enjambre.

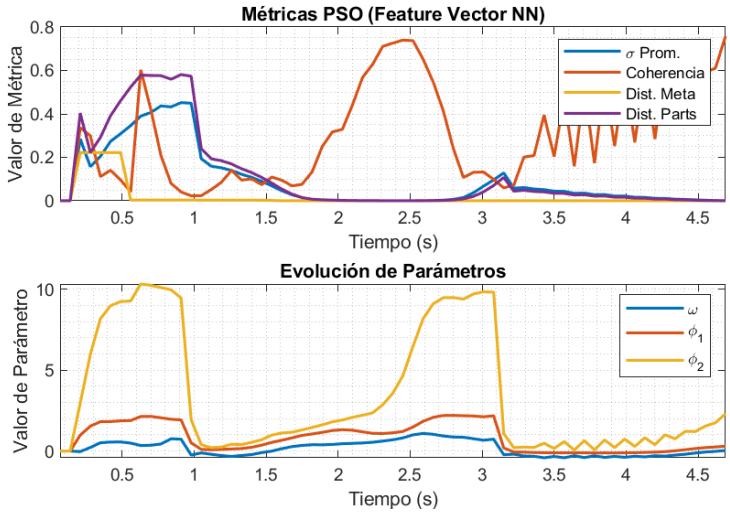


Figura 23: Evolución de los parámetros de entrada y salida de una red BiLSTM auxiliando la minimización de la función Schaffer F6.

7.10. Análisis de desempeño

7.10.1. Tiempo y precisión de convergencia

Para determinar si el *PSO Tuner* (PSOT) generaba una mejora significativa en el tiempo y precisión de la convergencia del algoritmo PSO, se realizaron un total de 200 simulaciones de 1000 partículas para el PSO estándar y para cada variación del PSO auxiliado por el *PSO Tuner* (GRU, LSTM y BiLSTM). Durante cada simulación, se tomaba nota del número de iteraciones para converger y del porcentaje de ocasiones en las que la posición de convergencia coincidía con la meta. Para dar fin a cada simulación se empleó un criterio de convergencia de "cercanía a la meta" y número de iteraciones máximas alcanzadas. Los resultados se generaron empleando diferentes combinaciones de tipos de restricción (constricción, inercia y mixto) y funciones de costo (Griewank, Schaffer F6 y APF).

En las figuras 24-32 se presentan los resultados de esta recolección de datos, con la gráfica a la izquierda presentando la media y dispersión del número de iteraciones para converger de cada método (dado un máximo de 438 iteraciones), y con la gráfica a la derecha presentando el porcentaje de ocasiones en las que cada variación del PSO consiguió alcanzar la meta o mínimo de la función de costo. El método PSO estándar se decidió abreviar como PSOT Off (ya que para este se desactivaba el *PSO Tuner*), mientras que los métodos auxiliados por el *PSO Tuner* se decidieron abreviar como "*PSOT + tipo red neuronal utilizada*" (BiLSTM, LSTM y GRU).

Función de costo: Griewank

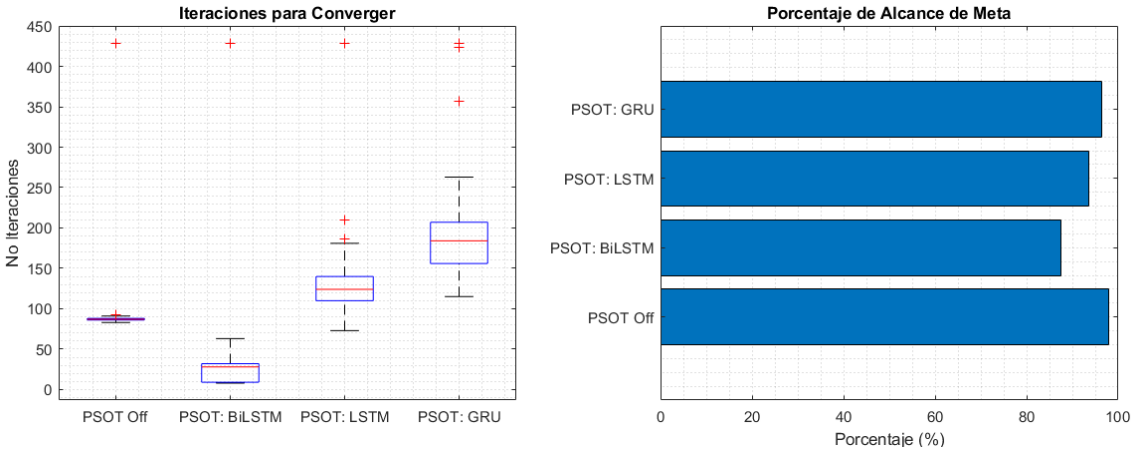


Figura 24: Comparación del tiempo y precisión de convergencia del algoritmo PSO con y sin el *PSO Tuner*. Función de costo: Griewank. Método de restricción empleado: Inercia.

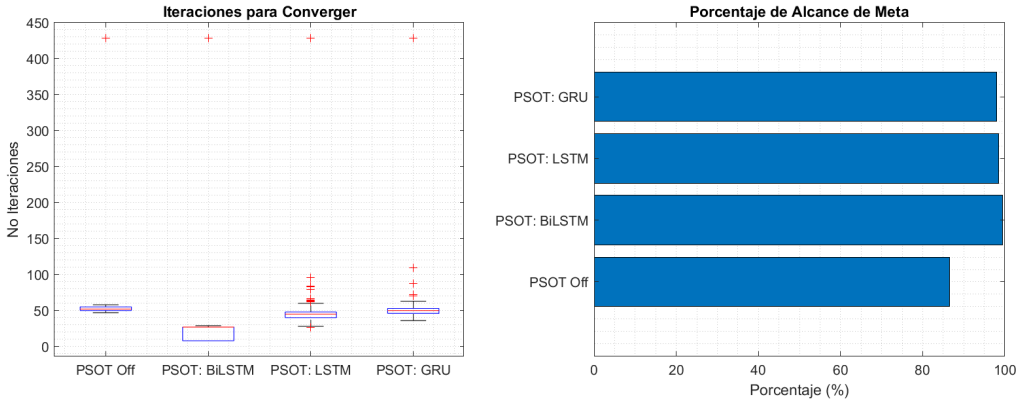


Figura 25: Comparación del tiempo y precisión de convergencia del algoritmo PSO con y sin el *PSO Tuner*. Función de costo: Griewank. Método de restricción empleado: Constricción.

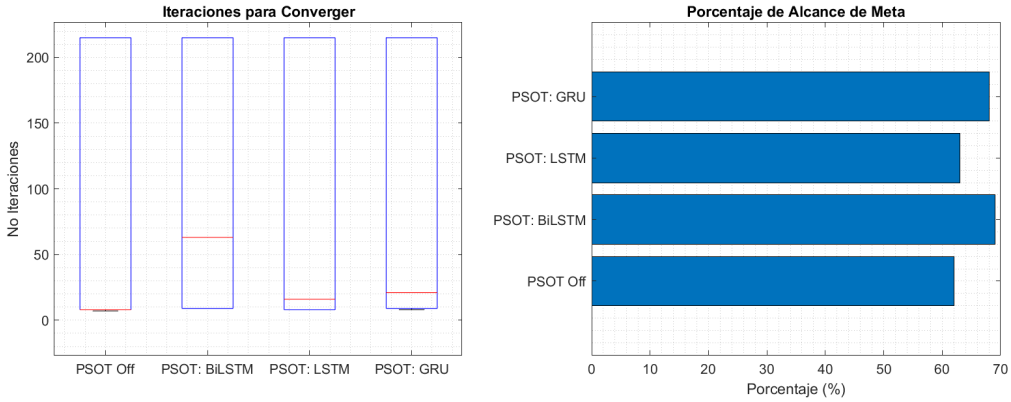


Figura 26: Comparación del tiempo y precisión de convergencia del algoritmo PSO con y sin el *PSO Tuner*. Función de costo: Griewank. Método de restricción empleado: Mixto.

Función de costo: Schaffer F6

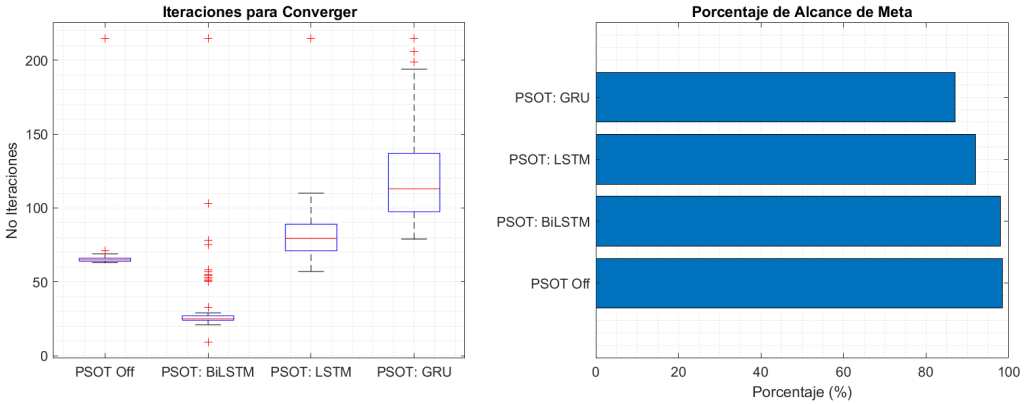


Figura 27: Comparación del tiempo y precisión de convergencia del algoritmo PSO con y sin el *PSO Tuner*. Función de costo: Schaffer F6. Método de restricción empleado: Inercia.

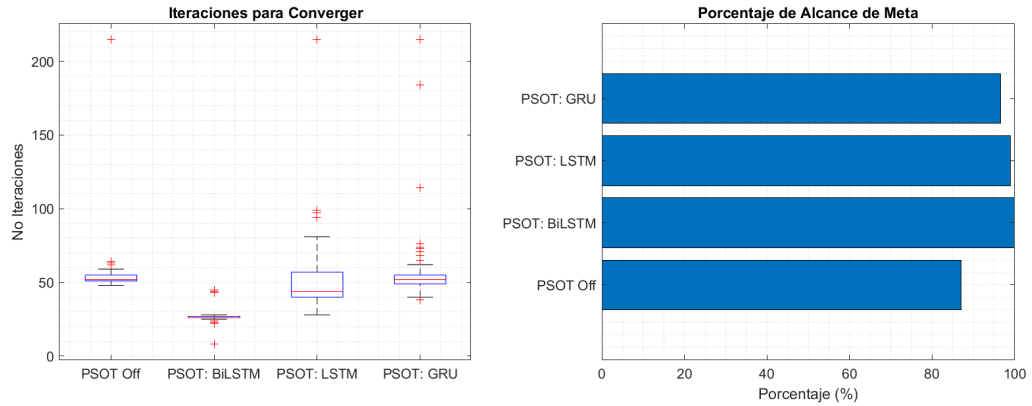


Figura 28: Comparación del tiempo y precisión de convergencia del algoritmo PSO con y sin el *PSO Tuner*. Función de costo: Schaffer F6. Método de restricción empleado: Constricción.

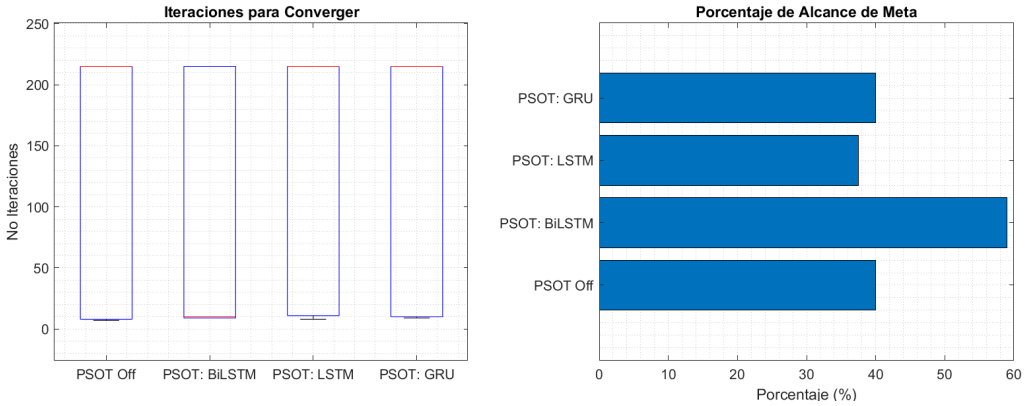


Figura 29: Comparación del tiempo y precisión de convergencia del algoritmo PSO con y sin el *PSO Tuner*. Función de costo: Schaffer F6. Método de restricción empleado: Mixto.

Función de costo: APF

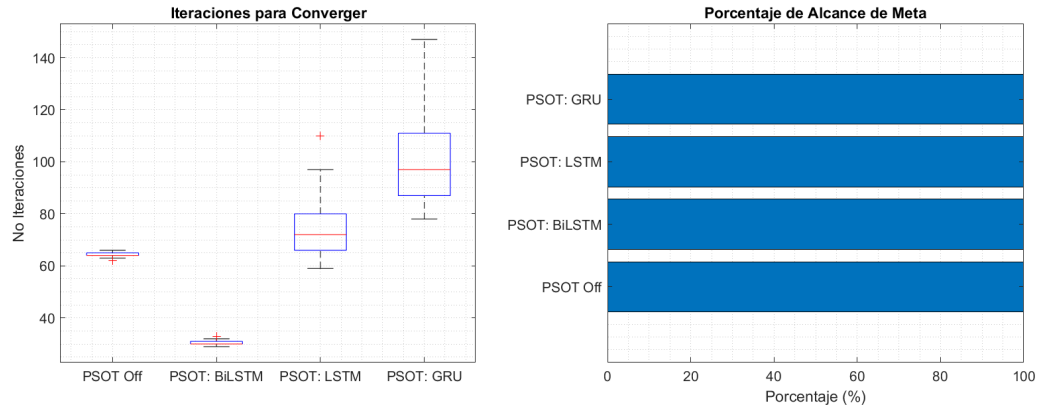


Figura 30: Comparación del tiempo y precisión de convergencia del algoritmo PSO con y sin el *PSO Tuner*. Función de costo: APF. Método de restricción empleado: Inercia.

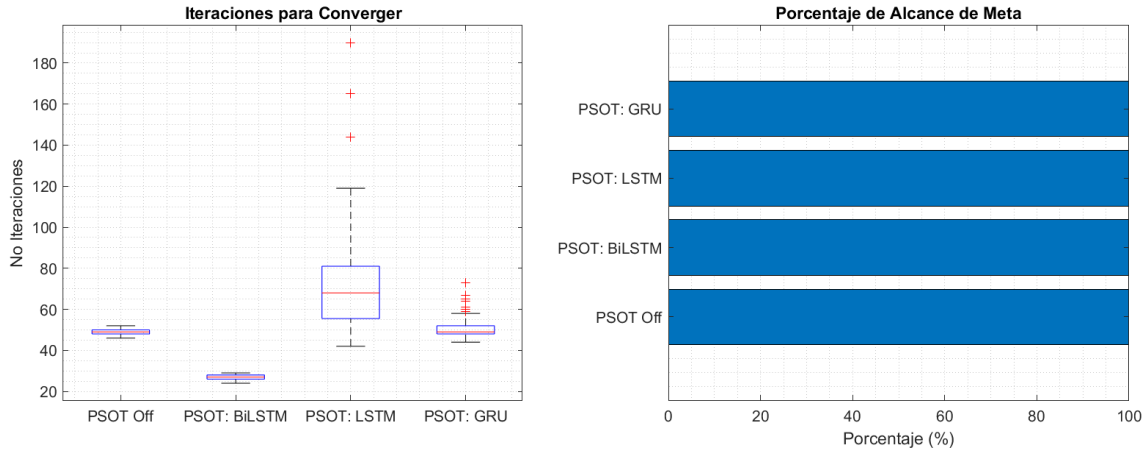


Figura 31: Comparación del tiempo y precisión de convergencia del algoritmo PSO con y sin el *PSO Tuner*. Función de costo: APF. Método de restricción empleado: Constricción.

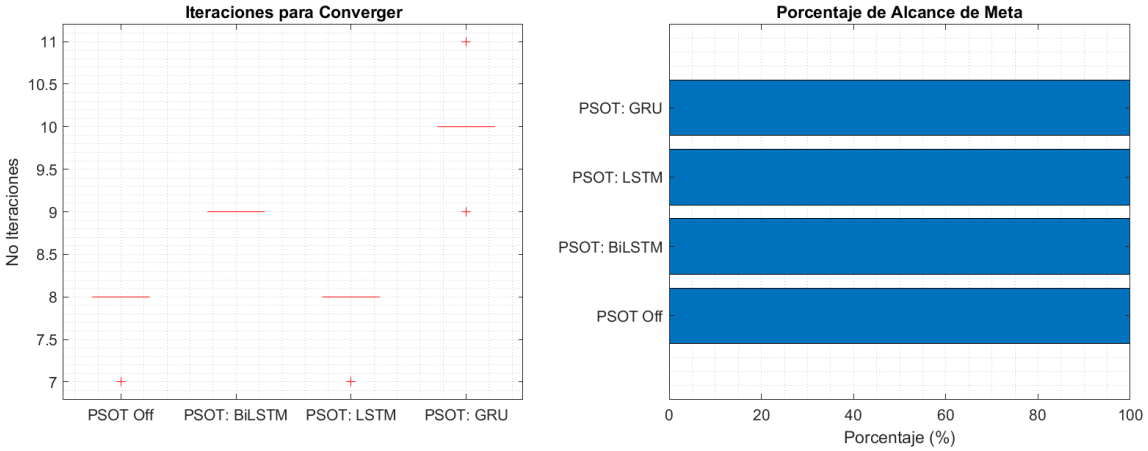


Figura 32: Comparación del tiempo y precisión de convergencia del algoritmo PSO con y sin el *PSO Tuner*. Función de costo: APF. Método de restricción empleado: Mixto.

Discusión de resultados

En términos de la velocidad de convergencia, la red BiLSTM se presenta como la alternativa de mayor rapidez. El único caso donde esta no se presenta como la opción superior, es en la función de costo "APF" en conjunto con el método de restricción mixto. En términos de la opción más lenta, la red GRU se presentó como la alternativa con mayor tiempo de convergencia en casi todos los casos. Las excepciones a esta regla consisten de los casos en los que se utiliza el método de constricción, el cual parece favorecer el correcto funcionamiento de la red GRU (incluso permitiendo que la misma supere por un pequeño margen al algoritmo PSO estándar en todas las funciones de costo). La red LSTM tiende a presentar un tiempo de convergencia igual o mayor al del PSO estándar. En los pocos casos en los que la media de las iteraciones son menores a las del PSO estándar, el algoritmo utiliza el método de constricción.

Para la precisión, no parece existir una tendencia clara. No obstante, uno de los casos más interesantes es el de la función "Schaffer F6" en conjunto con el criterio mixto. En este caso, la mayor parte de los métodos tendió a sufrir de una pérdida significativa en su precisión (menor o igual al 40 % para la mayoría). El único método que fue capaz de superar este valor pequeño fue la red BiLSTM, con un porcentaje de precisión de aproximadamente el 60 %. Aunque el valor aún continúa siendo bajo, esto prueba la capacidad de la red BiLSTM como un método robusto ante la presencia de mínimos locales.

En términos de la variabilidad en el número de iteraciones para converger, el método mixto parece generar los peores resultados. No obstante, estos resultados son engañosos. El amplio rango del cuartil superior e inferior del diagrama de caja y bigotes se debe a que el algoritmo empleado para generar estas estadísticas hace uso del criterio de convergencia "cercanía a la meta" para dar fin al algoritmo PSO.

Dado que el método mixto se caracteriza por una convergencia acelerada hacia el primer mínimo detectado, el número de iteraciones para converger en un ambiente con muchos mínimos locales puede consistir de valores o muy bajos (cuando converge en el mínimo global) o muy altos (cuando converge en un mínimo local). Al contar con únicamente 2 tipos de valores, el diagrama de caja y bigotes extiende los límites de la "caja" desde el valor inferior (iteraciones de convergencia a meta) hasta el superior (número de iteraciones máximas). La excepción a esto es la función de costo "APF".

Debido a que la misma consiste de una función "fácil" de minimizar (evidenciado en las figuras 30-32 por el hecho que todos los métodos fueron capaces de encontrar la meta un 100 % de las veces), los únicos valores que se obtuvieron fueron los valores de convergencia a la meta. Las pocas excepciones donde no se convergió se marcan como valores atípicos.

Dado que la forma de los diagramas de restricción mixta son todos iguales (exceptuando el caso con la función de costo APF), el mejor indicador de éxito en este caso consiste de la gráfica que indica la precisión de las convergencias. Tomando esto en cuenta, las redes LSTM y GRU parecen generar los resultados más precisos para el caso de una restricción mixta.

7.10.2. Dispersión y posición media

Para analizar la tendencia general del movimiento en cada tipo de PSO, se compara la media y desviación estándar de las coordenadas X y Y de las partículas del PSO estándar y del PSO auxiliado por el *PSO Tuner*. Al igual que en la sección 7.10.1, los resultados presentados consisten del promedio de un total de 200 simulaciones de 1000 partículas para cada tipo de PSO. Únicamente se incluyen los resultados más significativos[18]

En las figuras 33-36 se presenta la media y dispersión del movimiento del enjambre sobre el eje X (curvas azules) y Y (curvas rojas). Para las funciones Griewank y Schaffer, el enjambre converge hacia el mínimo ubicado en (0,0), mientras que para la función APF, el enjambre converge hacia el mínimo ubicado en (-3,3).

[18]En total se generaron nueve gráficas para esta sección, pero muchas de las gráficas que empleaban la misma función de costo permanecían inalteradas no importando el tipo de restricción. Debido a esto, se decidió obviar algunas de las gráficas para evitar redundancia.

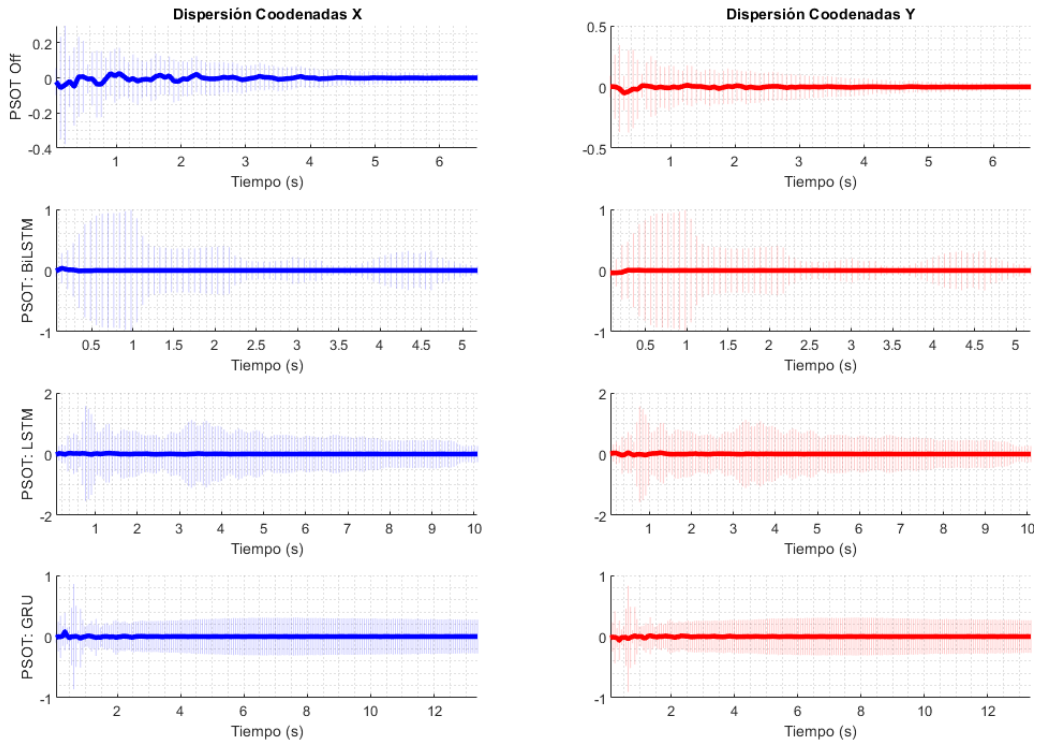


Figura 33: Comparación de la dispersión y movimiento de las partículas para el algoritmo PSO con y sin el *PSO Tuner*. Función de costo: Griewank. Método de restricción empleado: Inercia.

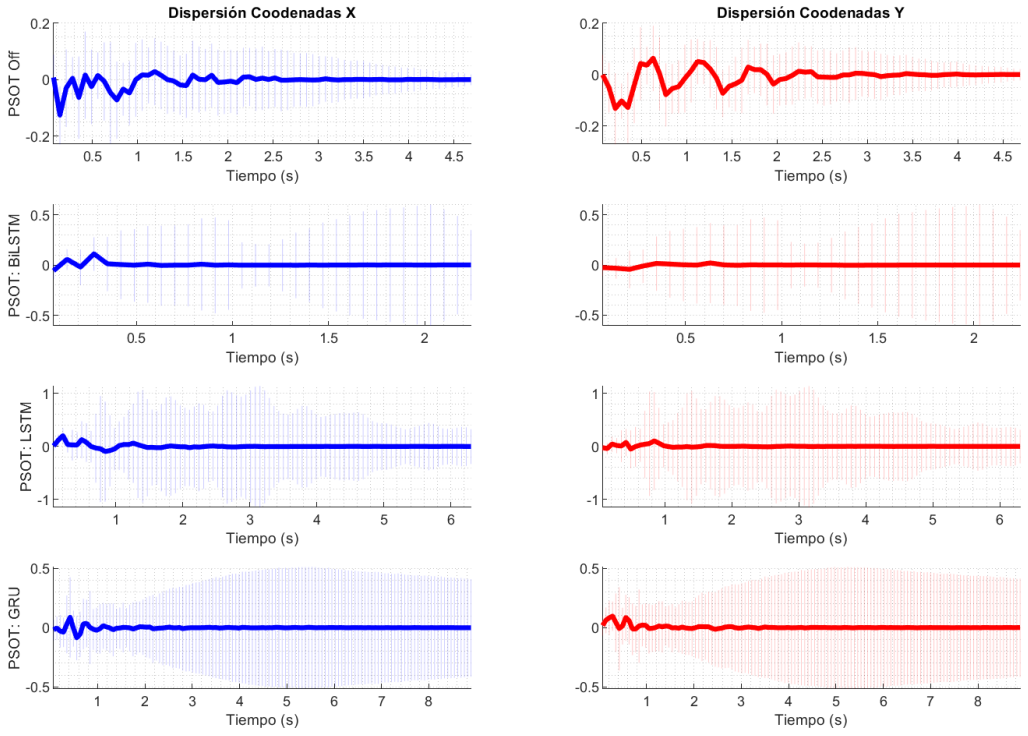


Figura 34: Comparación de la dispersión y movimiento de las partículas para el algoritmo PSO con y sin el *PSO Tuner*. Función de costo: Schaffer F6. Método de restricción empleado: Inercia.

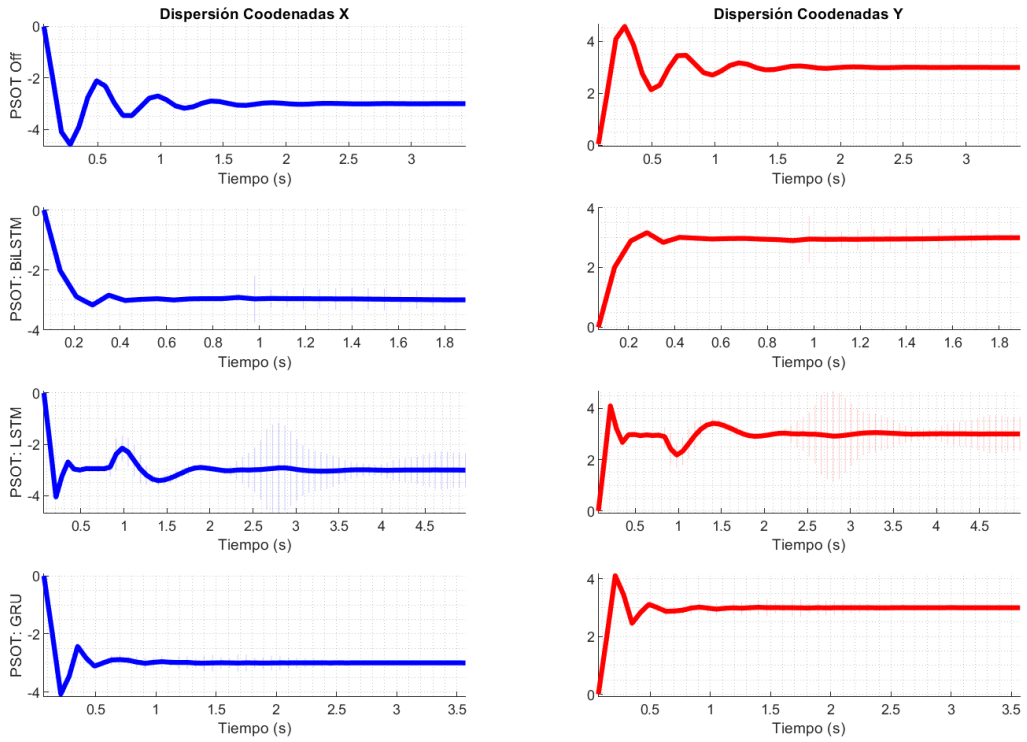


Figura 35: Comparación de la dispersión y movimiento de las partículas para el algoritmo PSO con y sin el *PSO Tuner*. Función de costo: APF. Método de restricción empleado: Constricción.

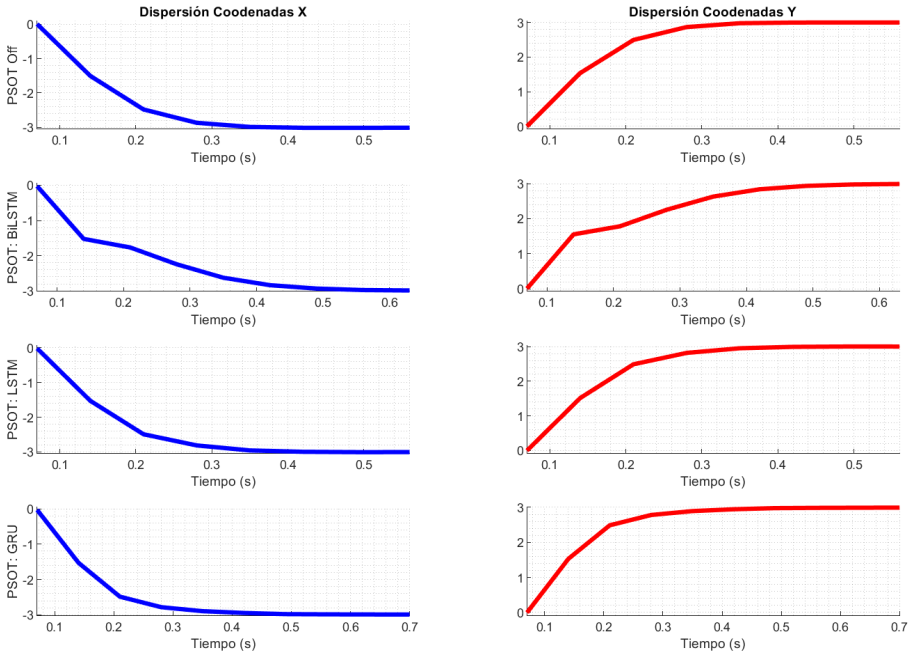


Figura 36: Comparación de la dispersión y movimiento de las partículas para el algoritmo PSO con y sin el *PSO Tuner*. Función de costo: APF. Método de restricción empleado: Mixto.

Discusión de resultados

Los movimientos del centro del enjambre para el PSO estándar tienden a presentar uno de tres elementos: patrones irregulares (Figura 34), oscilaciones (Figura 35) o una caída suave (Figura 36). En cualquiera de los casos, lo que hace la introducción de las redes neuronales es suavizar el movimiento del centro del enjambre. La calidad de la suavización es dependiente de la red aplicada.

Si se le analiza dentro del contexto de procesamiento de señales, la red BiLSTM parece reducir el "sobre-impulso" (*overshoot*) y el ruido de la posición, mientras incrementa su "tiempo de subida" (la escala de tiempo de la red BiLSTM tiende a dividirse en unidades más pequeñas que el resto de métodos, indicando que los cambios visualizados ocurren más rápido que en el resto de casos). Esto se traduce en un movimiento mucho más "intencionado" que se caracteriza por buscar la meta en lugar de orbitar alrededor de la misma. Esto coincide con el conocimiento que se posee sobre redes BiLSTM, el cual indica que las mismas utilizan información tanto pasada como futura para poder generar sus estimados. En palabras simples, se podría llegar a decir que la suavización ocurre porque la red "sabe hacia donde se dirige".

La suavización de las redes restantes se presenta ligeramente más pobre, conservando algunos fragmentos del ruido u oscilaciones originales. En el caso de la red GRU, las oscilaciones se traducen en movimientos en zig zag como consecuencia del movimiento errático que acompaña a este tipo de red. Para el caso de la red LSTM, esta no parece "filtrar" el movimiento completamente, sino que simplemente atenúa los cambios bruscos en la trayectoria seguida. En otras palabras, la red "sabe a donde debe ir, pero titubea en el camino".

La dispersión no parece exhibir un patrón definido, ya que su tendencia varía según la función de costo, tipo de restricción y tipo de *PSO Tuner*. Una constante en todos los casos es la falta de dispersión en el método de restricción mixto (Figura 36). Esto se debe a que, como se ha mencionado previamente, este método de restricción produce un movimiento conjunto de las partículas hacia el primer mínimo encontrado casi inmediatamente. Dado que todos los métodos de control por medio de redes neuronales parecen imitar este comportamiento con particular "intensidad", es de esperar que su movimiento presente una estructura casi idéntica.

7.10.3. Tiempo de predicción de red neuronal

A manera de determinar el tipo de red más eficiente en términos de su tiempo de computación de predicciones, a continuación se presentan los resultados de una comparación del tiempo promedio que le toma a cada red generar los parámetros de salida del *PSO Tuner* empleando diferentes métodos de restricción y funciones de costo. Las figuras 37-45 presentan la media y dispersión del tiempo de computación promedio (en milisegundos) observado a lo largo de 200 simulaciones. Nuevamente, los diferentes tipos de red comparados se decidieron abreviar como "*PSOT + tipo red neuronal*" (BiLSTM, LSTM y GRU).

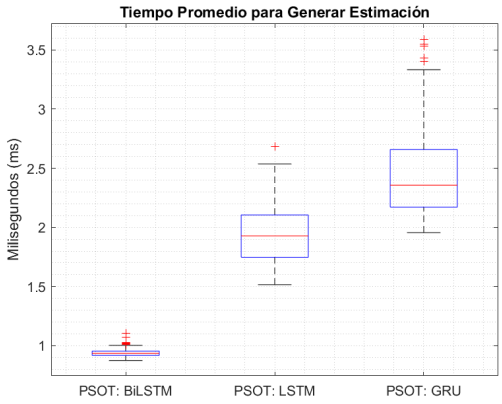


Figura 37: Comparación del tiempo de computación de predicciones entre redes. Función de costo: APF. Método de restricción: Inercia.

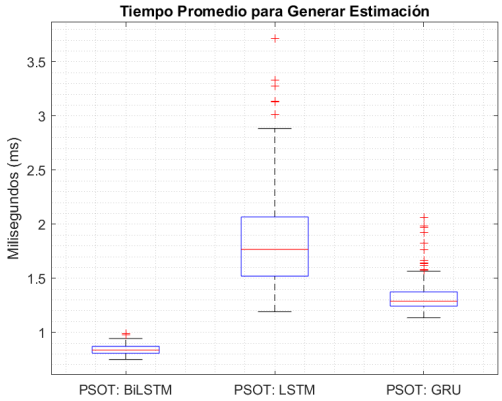


Figura 38: Comparación del tiempo de computación de predicciones entre redes. Función de costo: APF. Método de restricción: Constricción.

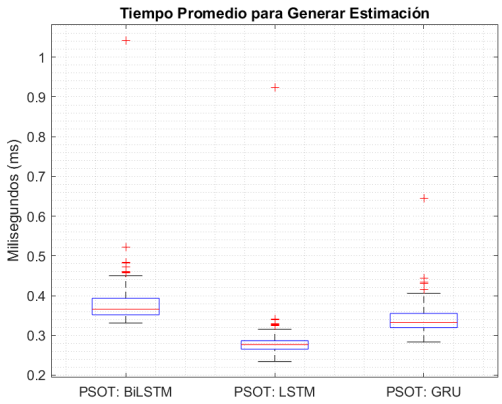


Figura 39: Comparación del tiempo de computación de predicciones entre redes. Función de costo: APF. Método de restricción: Mixto.

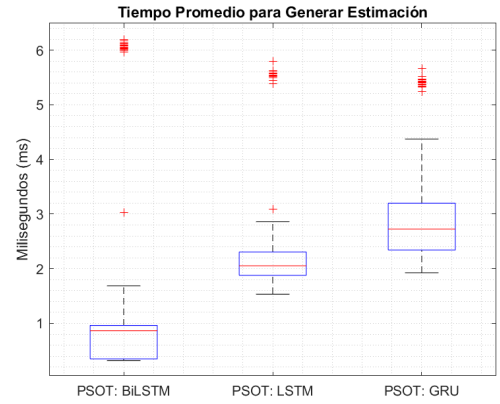


Figura 40: Comparación del tiempo de computación de predicciones entre redes. Función de costo: Griewank. Método de restricción: Inercia.

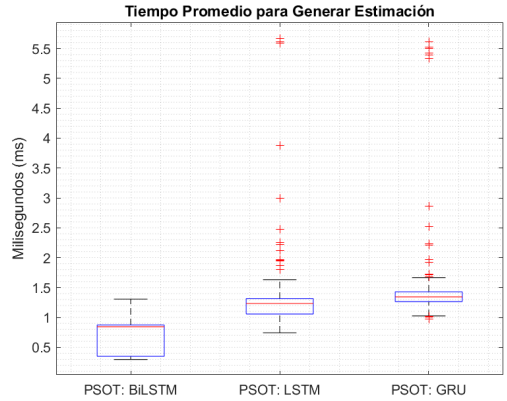


Figura 41: Comparación del tiempo de computación de predicciones entre redes. Función de costo: Griewank. Método de restricción: Constricción.

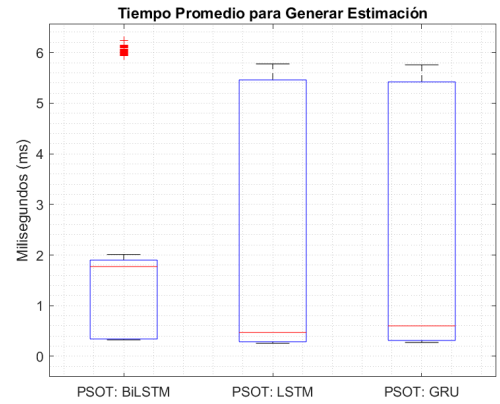


Figura 42: Comparación del tiempo de computación de predicciones entre redes. Función de costo: Griewank. Método de restricción: Mixto.

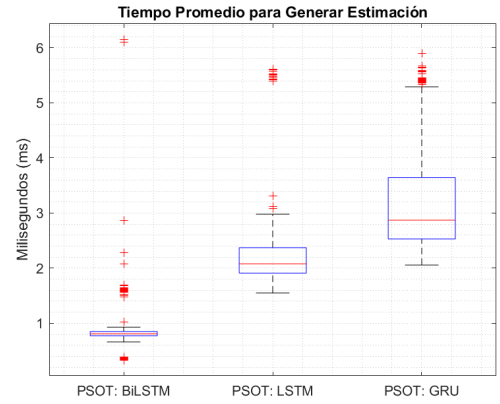


Figura 43: Comparación del tiempo de computación de predicciones entre redes. Función de costo: Schaffer F6. Método de restricción: Inercia.

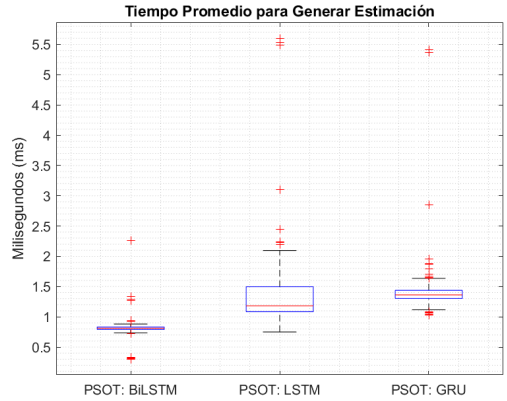


Figura 44: Comparación del tiempo de computación de predicciones entre redes. Función de costo: Schaffer F6. Método de restricción: Constricción.

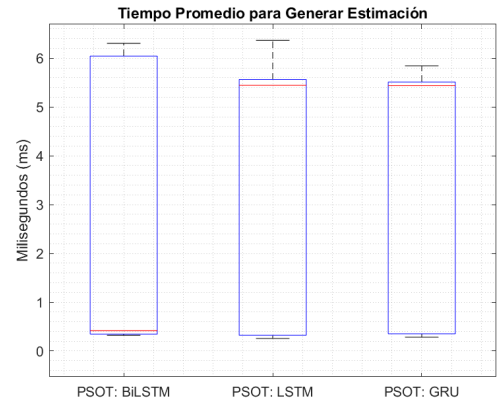


Figura 45: Comparación del tiempo de computación de predicciones entre redes. Función de costo: Schaffer F6. Método de restricción: Mixto.

Discusión de resultados

En promedio, la red BiLSTM parece consistir de la mejor opción en términos de su velocidad de predicción. Seguido de esta se encuentra la red LSTM y finalmente la GRU. La red BiLSTM presenta un tiempo medio de predicción inferior (entre 0.4 y 1 ms) en los casos que hacen uso de los métodos de restricción de inercia y constricción.

Inicialmente la ventaja de la red BiLSTM por sobre las redes restantes puede parecer contradictoria, después de todo, las neuronas de este tipo cuentan con un número mayor de parámetros internos y por lo tanto, requieren de un mayor número de operaciones para llevar a cabo el proceso de *feed forward*. No obstante, se debe considerar que la red BiLSTM final no solo presenta un menor número de capas (6 en total) sino que también un menor número de neuronas en cada una de ellas. Por lo tanto, en el caso de los resultados observados, el número reducido de neuronas resultó más influyente que la complejidad interna de cada neurona.

A su vez, también debe ser mencionado, que en el caso de las redes LSTM y GRU, se requiere del uso de dos muestras para obtener resultados óptimos. Esto conlleva a procesamiento adicional, ya que las matrices de entrada y salida de estas redes poseen una mayor dimensionalidad.

A pesar de todo esto, en los casos en los que se utiliza un tipo de restricción mixto, el tiempo de predicción parece no presentar una tendencia clara. En el caso de la Figura 39, la red LSTM parece contar con el tiempo más bajo de predicción, mientras que en los casos restantes, la dispersión de los datos es tan alta que resulta difícil deducir información al respecto.

7.10.4. Número reducido de partículas

Para probar la robustez del *PSO Tuner*, se decidió realizar un total de 200 simulaciones reduciendo el número de partículas a 10. Esto limita significativamente la información disponible para el algoritmo PSO, causando que este se torne altamente sensible ante los mínimos locales. Para que la prueba se realizara bajo el "peor caso posible", se decidió minimizar la función Griewank (función caracterizada por su alto número de mínimos locales) utilizando el método de restricción de inercia y constricción. Se obvió el método mixto, ya que este únicamente potenciaría el comportamiento de convergencia rápida a mínimos locales.

Restricción: Constricción

Para el caso de la restricción por constricción, el algoritmo PSO auxiliado por la red BiLSTM produjo los mejores resultados, alcanzando la meta casi el 80 % de las ocasiones (Figura 46). La red LSTM y el PSO estándar probablemente fueron capaces de converger un pequeño número de "ocasiones atípicas" gracias a un posicionamiento inicial favorable de las partículas sobre el espacio de búsqueda. La red GRU por otro lado, nunca fue capaz de alcanzar la meta.

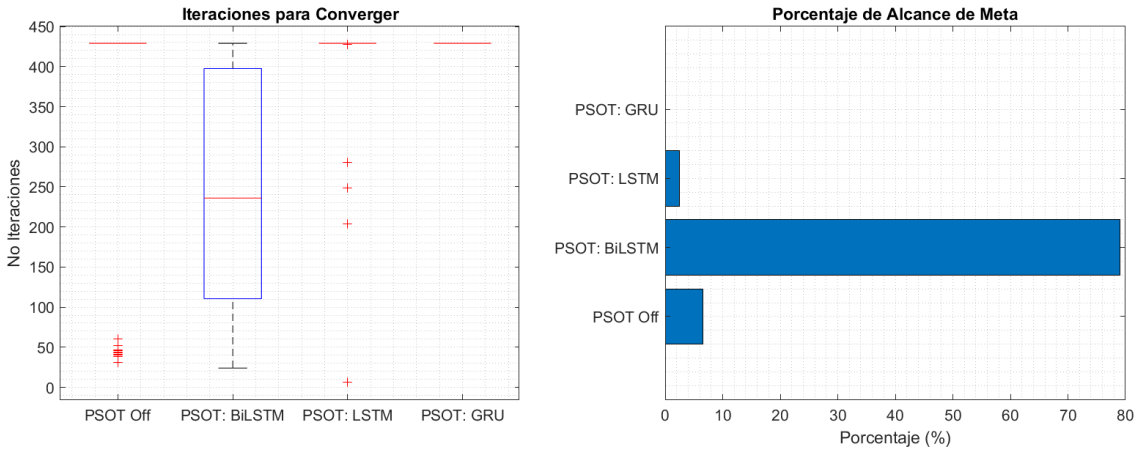


Figura 46: Número de iteraciones y precisión de convergencia para algoritmo PSO de 10 partículas. Función de costo: Griewank. Método de restricción: Constricción.

Cuando se analiza el movimiento que produjo estos resultados (Figura 47), se puede evidenciar que una disminución en el número de partículas trae consigo un incremento a la dispersión en los tres tipos de *PSO Tuner*. Esto es un claro indicador del comportamiento que utilizan las diferentes redes neuronales para alcanzar la meta: ante una falta de información (causada por el número reducido de partículas) estas presuntamente incrementan su dispersión para cubrir una mayor parte de la superficie de costo. El mantenimiento de dicha dispersión a lo largo de todo el movimiento es a su vez un indicador del comportamiento "inseguro" de las redes, las cuales evitan causar la convergencia de las partículas para evitar mínimos locales.

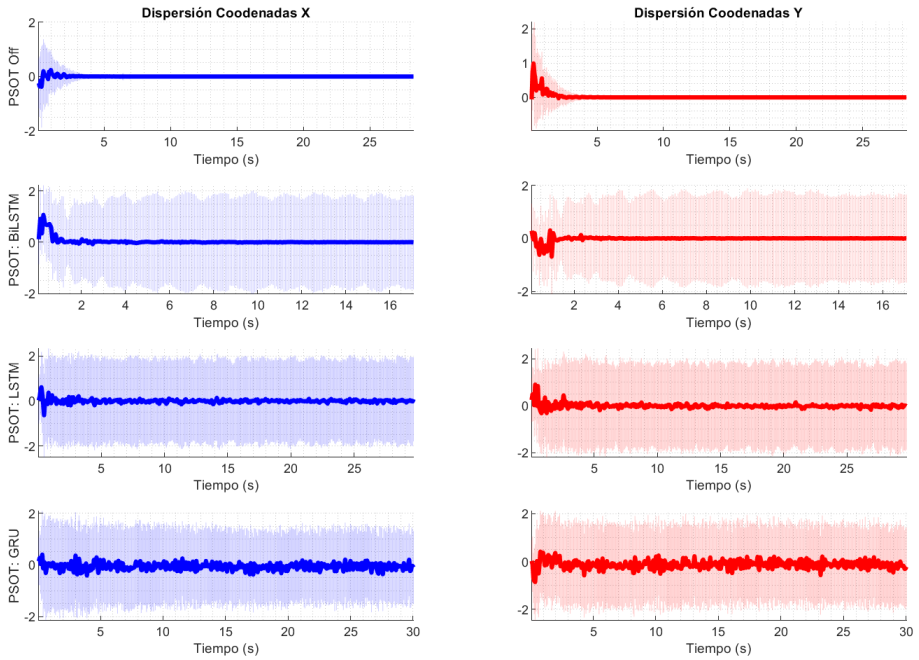


Figura 47: Posición media y dispersión de las partículas para algoritmo PSO de 10 partículas. Función de costo: Griewank. Método de restricción: Constricción.

Otra característica importante, es que el movimiento del centro de los enjambres controlados por las redes LSTM y GRU es mucho más ruidoso. Esto pudo haber causado que los enjambres controlados por las mismas orbitaran alrededor de la meta, pero nunca llegaran a converger, probando que estas probablemente son capaces de alcanzar la meta, pero requieren de un mayor número de iteraciones para conseguirlo.

Restricción: Inercia

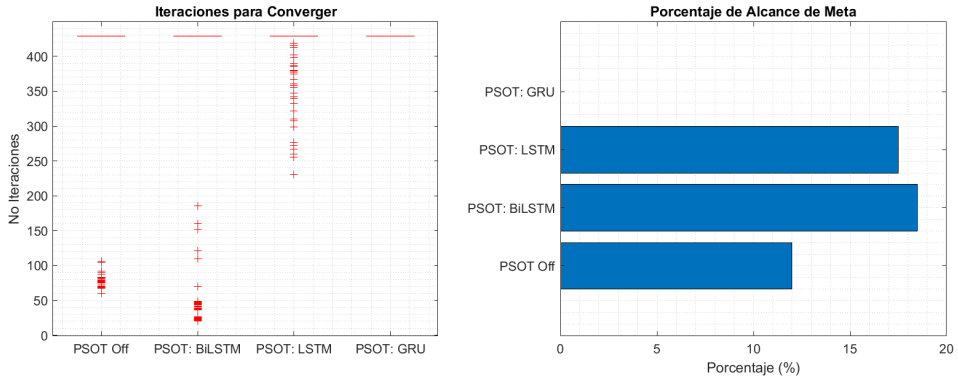


Figura 48: Tiempo y precisión de convergencia para algoritmo PSO de 10 partículas. Función de costo: Griewank. Método de restricción: Inercia.

Para el caso de la restricción por inercia, todos los métodos mantuvieron su desempeño previo, exceptuando por la red BiLSTM. Aunque esta continúa posicionándose como la opción más precisa, su precisión disminuyó del 80 % al 18 %. Por lo tanto, si se desea mejorar la convergencia en enjambres de pocas partículas, se debe hacer uso del PSO auxiliado por BiLSTM y empleando la restricción por constricción.

CAPÍTULO 8

Planificación de trayectorias con aprendizaje reforzado

8.1. Gridworld

Gridworld consiste de uno de los ejemplos más representativos del aprendizaje reforzado, ya que consiste de uno de los primeros y más simples ejemplos que se pueden emplear para introducirse en esta área de aprendizaje reforzado.

En este ejemplo se cuenta con un espacio cuadriculado a través del cual se desea navegar. Dentro de este espacio existen "celdas meta" y "celdas obstáculo". El agente que se desplaza dentro de dicha cuadrícula tiene la capacidad de moverse en cuatro direcciones: Arriba, abajo, izquierda o derecha. Si el agente alcanza la meta, se le brinda una recompensa positiva, de lo contrario se le castiga. La idea principal es generar un conjunto de acciones a tomar en cada celda, a manera que el agente sea capaz de esquivar las "celdas obstáculo" y llegar a las "celdas meta".

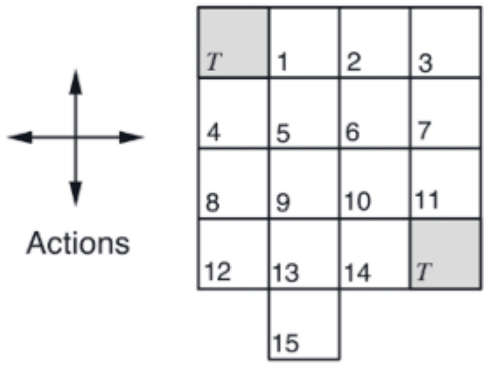


Figura 49: Representación gráfica de *Gridworld* [3]

Dentro del contexto de aprendizaje reforzado, este problema puede reformularse como un cadena de decisión de Markov [3], donde las diferentes celdas del entorno se pueden definir como los estados del sistema (un espacio de estados) y las direcciones de movimiento como las acciones.

8.2. Iteración de política

Para la implementación de este ejemplo dentro del contexto de navegación con robots diferenciales, se realizaron algunos cambios. En primer lugar, se buscó transferir el ambiente sobre el que se desplazarían los robots a una cuadrícula similar a aquella descrita en el ejemplo de *gridworld*.

Para conseguir esto, la mesa de trabajo se subdivide en celdas y luego se escanea celda a celda para determinar cuales son las celdas que contienen obstáculos o metas. El escaneo se realiza en el mismo orden en el que se numeran los estados: de arriba hacia abajo y de izquierda a derecha (Figura 50). Una vez finalizado el escaneo, se emplea la información adquirida para generar un espacio de estados. Este a su vez, se puede llegar a utilizar para generar una política de acción a través del algoritmo conocido como "iteración de política"

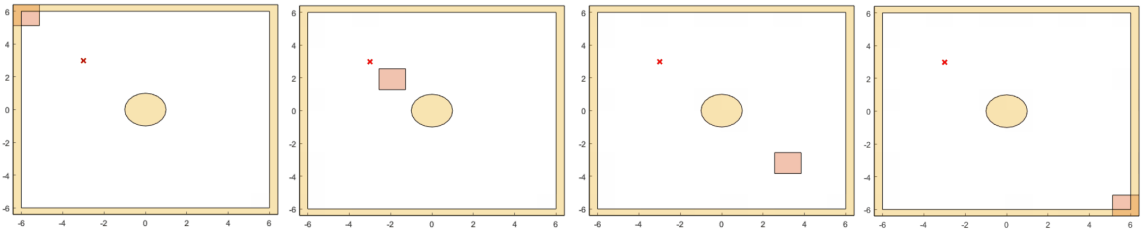


Figura 50: Proceso de escaneo de mesa de trabajo para el caso de un obstáculo cilíndrico de radio unitario ubicado en (0,0) y una meta ubicada en (-3,3).

En la primera etapa de este algoritmo, conocida como "evaluación de política", se toma cada uno de los estados del espacio de estados (cada celda en la cuadrícula) y se calcula el "valor de estado". Esta cantidad consiste de la suma de todos los "valores de acción" los cuales se calculan a su vez de la siguiente manera

$$q(s,a) = \pi(a|A)[r + \gamma V(s')] \tag{46}$$

Donde: r = Recompensa obtenida en caso se tome la acción a y se llegue al estado s'
γ = Factor de descuento[19]
$V(s')$ = Valor del estado al que se transicionaría si se tomara la acción elegida
$\pi(a|A)$ = Probabilidad de tomar la acción "a" dada la política actual "π"

Asumiendo una probabilidad inicial uniforme para todas las direcciones de movimiento, es posible calcular el valor para cada una de las diferentes acciones disponibles, y con su

[19]Constante entre 0 y 1 que establece que "tan a futuro" considera sus opciones el agente al momento de generar la política

suma, producir el valor para el estado específico analizado. Si la diferencia entre el "valor de estado" actual y el "valor de estado" previo es menor a cierto umbral (θ), se finaliza el ciclo de evaluación. Cabe mencionar que se le llama así a esta etapa porque se utiliza la política actual para "evaluar" el valor de los estados.

Luego de completar esta sección, se pasa a la etapa de "mejora de política". Para esta, se almacena el valor de la política actual en memoria (π_{old}) y se re-calculan los valores de acción correspondientes a cada estado en el espacio de estados. La diferencia con respecto a la etapa de "evaluación" es que en esta ocasión, en lugar de sumar los valores de acción, se extrae la acción que posee el valor máximo para cada estado ($arg\,\text{máx}_a$). Todas estas acciones "máximas" son agrupadas para forma una nueva política.

Se compara la nueva política (π) con aquella previamente almacenada (π_{old}) y si ambas son iguales, se considera que el algoritmo ha convergido en una "política óptima". Si este no es el caso, se repite el proceso hasta que no exista diferencia entre ambas políticas. Siguiendo este conjunto de acciones óptimas, no importando donde se posicione inicialmente el agente, este intentará desplazarse a manera de obtener la mayor cantidad de recompensa posible. Entonces, en este caso particular, es la tarea del usuario producir la "dinámica del sistema", o la función que establece las recompensas que se obtienen según el estado y acción actuales.

Algoritmo 2: Iteración de política

Inicialización:
$V(s) \leftarrow [0, 0, \dots]$
$\pi(s) \leftarrow$ Probabilidad uniforme
$\theta \leftarrow$ Número pequeño indicando precisión de estimación
politica-estable $\leftarrow 0$

while *politica-estable* $= 0$ **do**

Evaluación de política:
while $\Delta > \theta$ **do**
$\Delta \leftarrow 0$
for *cada estado s en espacio de estados* **do**
$v \leftarrow V(s)$
$V(s) \leftarrow \Sigma_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$
end
end

Mejora de política:
politica-estable $\leftarrow 1$
for *cada estado s en espacio de estados* **do**
$\pi_{old} \leftarrow \pi(s)$
$\pi(s) \leftarrow \text{argmax}_a \Sigma_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
if $\pi_{old} \neq \pi(s)$ **then**
politica-estable $\leftarrow 0$
end
end
end

8.3. Dinámica del generador de trayectorias

Dado que el generador de trayectorias se utilizará para planificar el movimiento de robots diferenciales capaces completar una rotación completa alrededor de su eje central, se decidió expandir el número de acciones disponibles para el agente, permitiendo que el mismo se desplace diagonalmente a 45 grados. Esto incrementa el número de acciones disponibles a ocho posibilidades.

Esto puede parecer incrementar significativamente la dimensión del problema a trabajar, no obstante, la escala del mismo permaneció virtualmente inalterada en comparación al ejemplo base. A pesar de esto si se requirió de la incorporación de algunos cambios. Por ejemplo, en *gridworld*, cuando el agente intenta tomar una acción que lo lleve a "entrar" a un obstáculo o salir de la cuadrícula, la dinámica detectará una colisión y retornará al agente a su estado original. Esto es fácilmente adaptable al caso con movimiento diagonal.

A pesar de esto, se debe de considerar una situación particular. Cuando el agente se ubique en un estado rodeado en todas sus direcciones cardinales por obstáculos, este debería de ser incapaz de transicionar hacia los estados ubicados en sus diagonales (estados 1, 3, 7 y 9 de la Figura 51). No obstante, dadas las reglas previas, el agente es capaz moverse a dichas celdas ya que las mismas consisten de estados sin obstáculos. Para tomar esto en cuenta, se incluyó una regla que impide el movimiento diagonal cuando existen dos obstáculos en la dirección general del movimiento diagonal (arriba y derecha, arriba e izquierda, abajo y derecha, abajo e izquierda). Al igual que en el caso de *gridworld*, se detecta una colisión y se retorna al agente a su estado original.

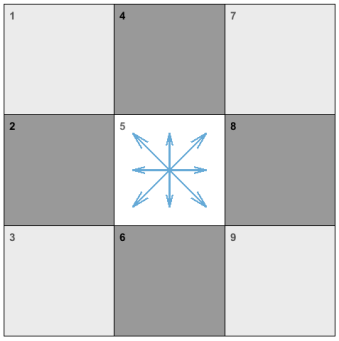


Figura 51: Situación en la que el agente (ubicado en celda blanca) cuenta con obstáculos (celda gris oscuro) en todas las direcciones cardinales.

Las recompensas entregadas al agente se otorgaban bajo 5 condiciones distintas: Al llegar a la meta (positiva), al dar un "paso" sin alcanzar la meta (negativa), al desplazarse en diagonal (negativa), al colisionar con un estado obstáculo (negativa) y al colisionar "parcialmente".

Se llamó "colisión parcial" al movimiento en el que el agente intenta desplazarse en diagonal con obstáculos a sus lados. Claramente, si este movimiento se acoplara a un robot real, el mismo se chocaría con la esquina del obstáculo. Entonces, para evitar esta situación, se incorporó una recompensa negativa para tomar en cuenta estas colisiones parciales.

8.4. Generación de puntos de trayectoria

Con la dinámica, es posible generar la política óptima, y con la política óptima es posible construir la trayectoria a seguir por el robot. Para esto, se toma la posición inicial del mismo y se establece como el primer punto en la trayectoria. A su vez, también se toma nota del estado en el que se encuentra el robot. Seguido de esto, se sigue la política generada hacia el siguiente estado y se coloca el centro del mismo como el siguiente punto en la trayectoria.

Se continúa este proceso de seguimiento de política hasta finalmente alcanzar el estado meta. Cuando se llega al mismo, se coloca el último punto de la trayectoria y se permite que los robots sigan la trayectoria generada (Figura 52). En caso la trayectoria generada no alcance el estado meta, se considera que la misma consiste de una "trayectoria inconclusa o cíclica" y se genera un error.

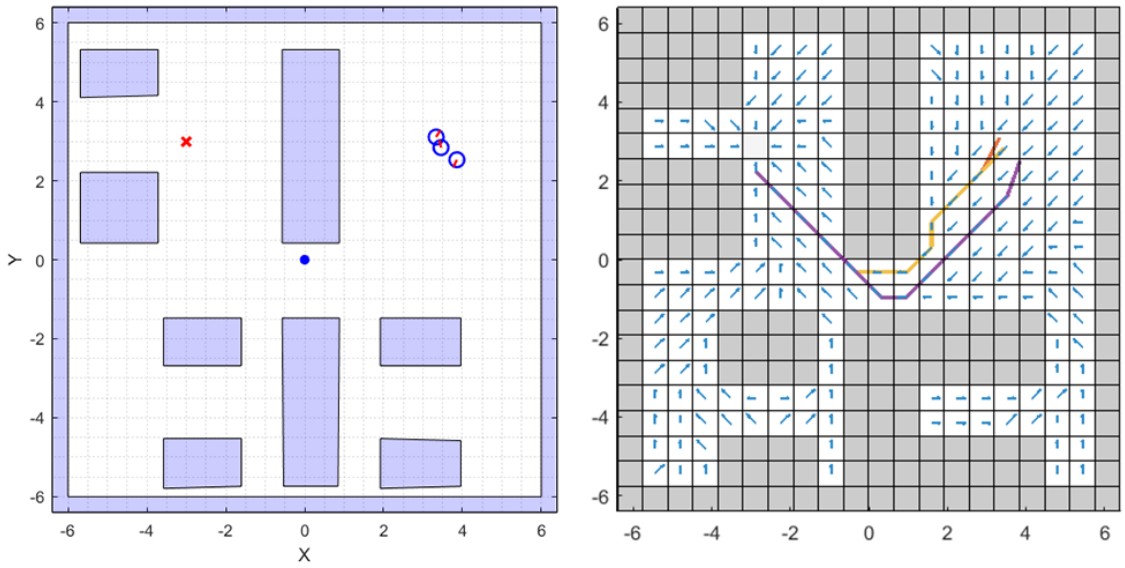


Figura 52: Conjunto de tres trayectorias generadas al seguir la política óptima (flechas azules) desde el punto de partida de cada robot hasta la meta (marcador rojo).

8.5. Parámetros de funcionamiento

A continuación se listan las diferentes recompensas y parámetros utilizados para poder generar un comportamiento aceptable por parte del generador de trayectorias por medio de aprendizaje reforzado

			Recompensas				
Threshold Eval. Política (θ)	Descuento (γ)	Dimensiones cuadrícula	Colisión borde	Colisión Parcial	Alcance meta	Paso sin llegar a meta	Movimiento diagonal
0.001	0.99	20×20	0	-1	+100000	-1	-1

Cuadro 19: Parámetros utilizados para el algoritmo de generación de trayectorias con aprendizaje reforzado.

8.6. Resultados

A continuación se presentan las trayectorias generadas en una variedad de configuraciones para los obstáculos presentes en la mesa de trabajo. Específicamente, las figuras 53-60 presentan las trayectorias generadas por medio del seguimiento de la política óptima obtenida (izquierda) y las trayectorias reales seguidas por tres robots diferenciales empleando un controlador de seguimiento punto a punto LQR (derecha). Cabe mencionar que, debido al carácter exploratorio y primordialmente experimental de este método de generación de trayectorias, los resultados carecen de rigor estadístico, por lo que a diferencia del capítulo previo, en este no se presentan análisis estadísticos o comparaciones entre otros métodos existentes.

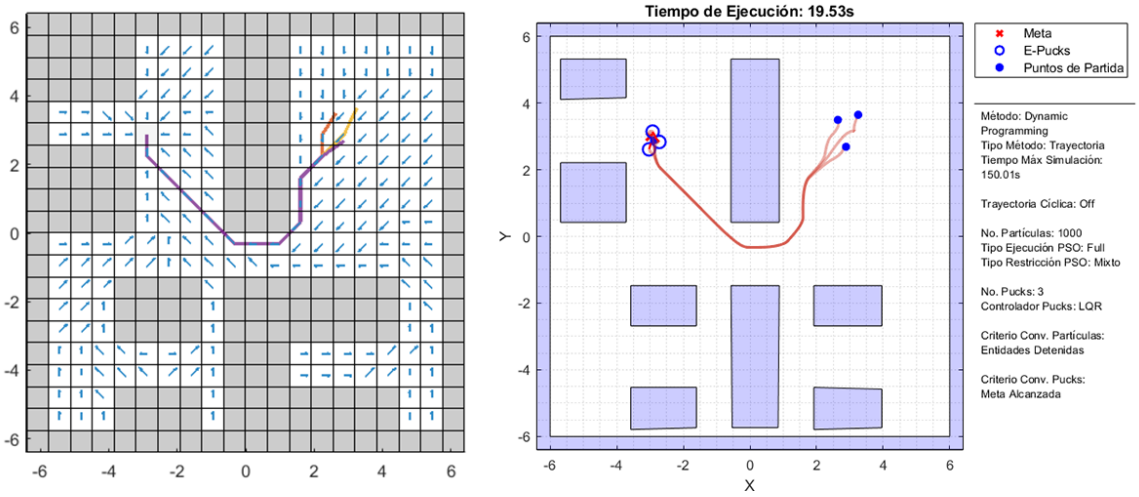


Figura 53: Trayectorias generadas (izquierda) y reales (derecha) para el mapa 1. Meta: (-3,3). Modificación del diseño propuesto por [38].

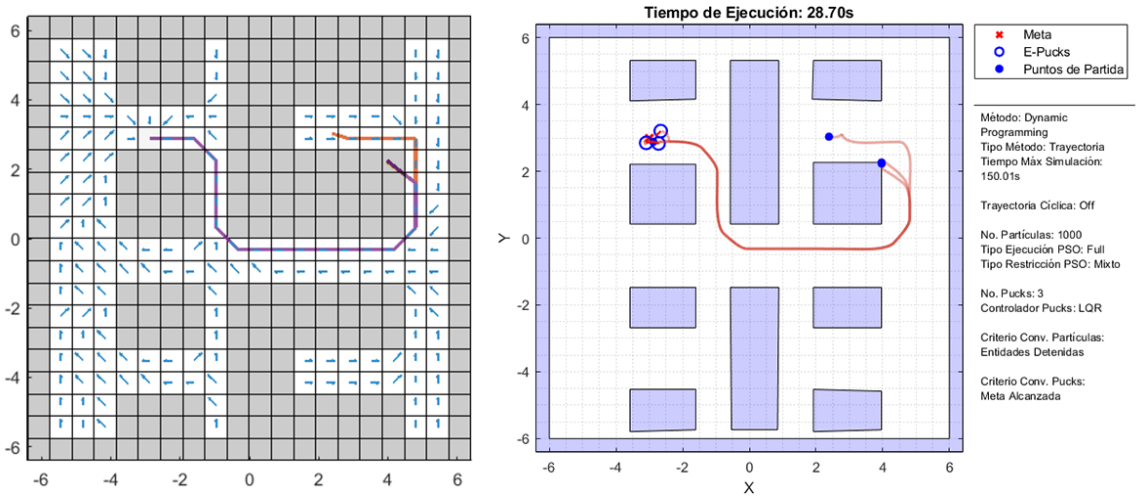


Figura 54: Trayectorias generadas (izquierda) y reales (derecha) para el mapa 2. Meta: (-3,3). Diseñado por [38].

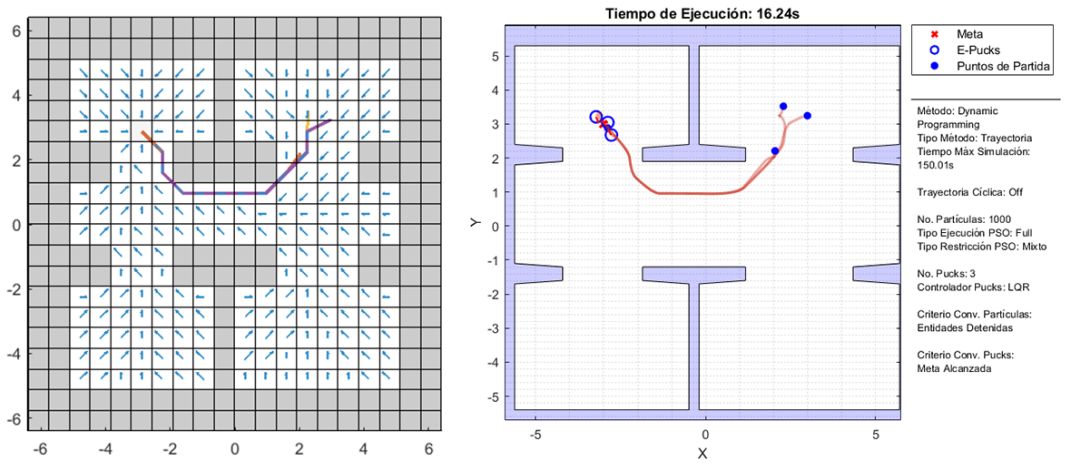


Figura 55: Trayectorias generadas (izquierda) y reales (derecha) para el mapa 3. Meta: (-3,3). Diseñado por [38].

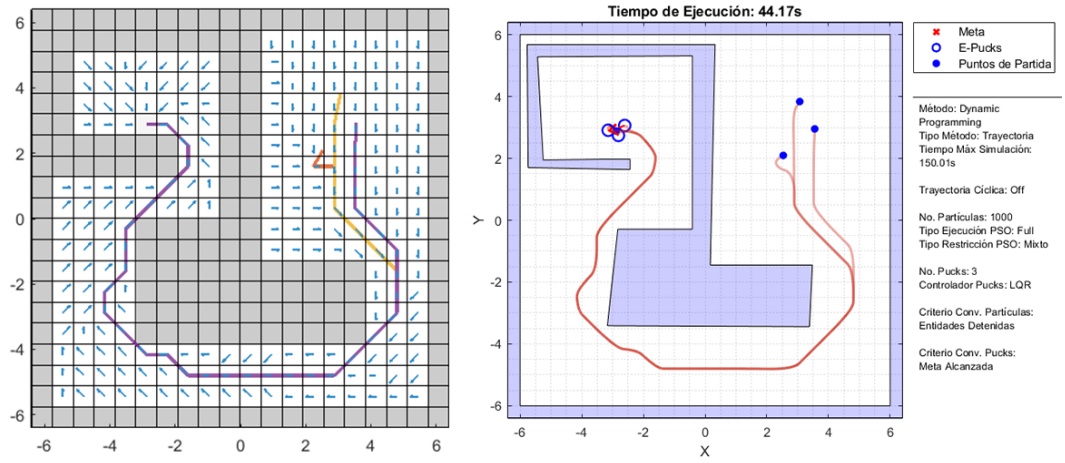


Figura 56: Trayectorias generadas (izquierda) y reales (derecha) para el mapa 4. Meta: (-3,3).

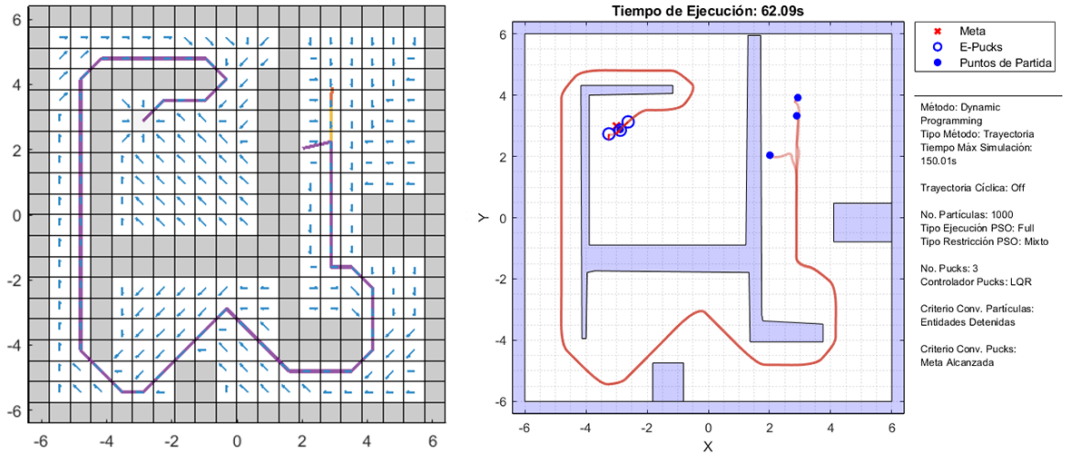


Figura 57: Trayectorias generadas (izquierda) y reales (derecha) para el mapa 5. Meta: (-3,3).

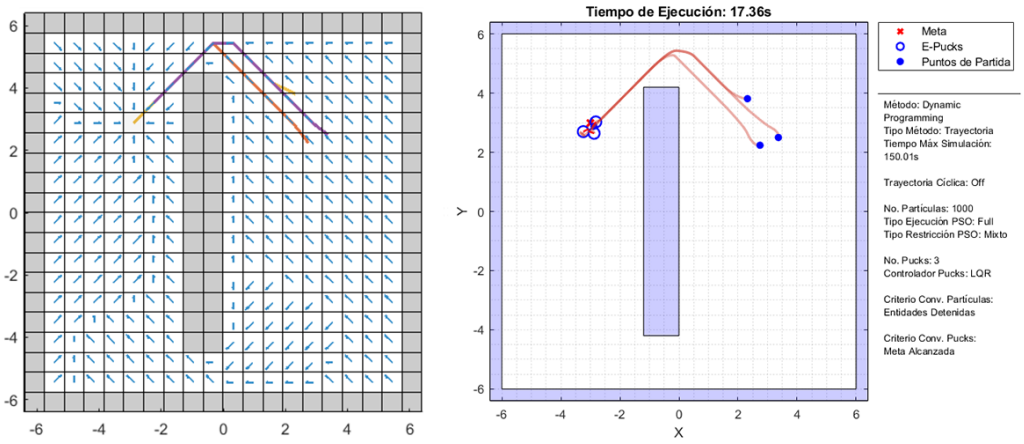


Figura 58: Trayectorias generadas (izquierda) y reales (derecha) para el mapa 6. Meta: (-3,3). Caso A de la tesis de [2].

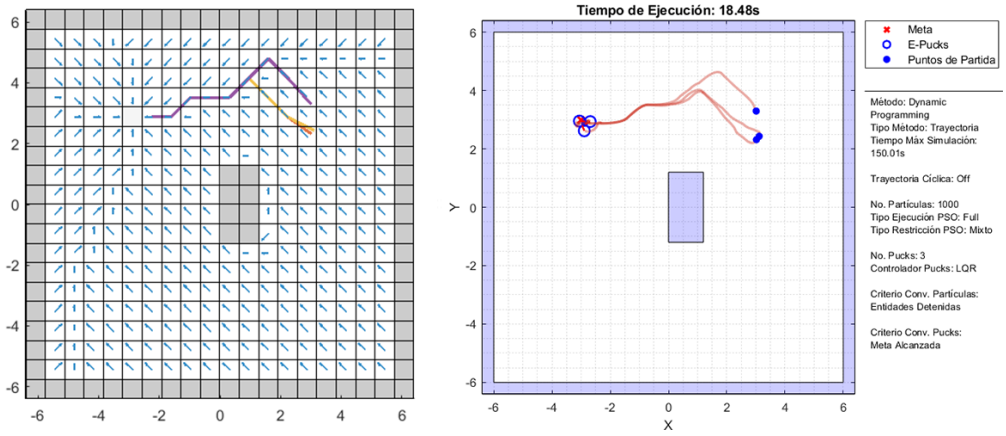


Figura 59: Trayectorias generadas (izquierda) y reales (derecha) para el mapa 7. Meta: (-3,3). Caso B de la tesis de [2].

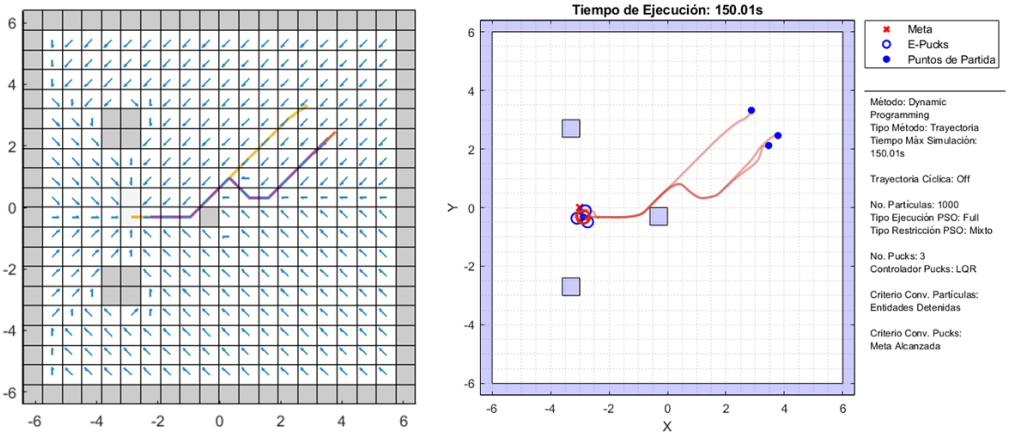


Figura 60: Trayectorias generadas (izquierda) y reales (derecha) para el mapa 8. Meta: (-3,3). Caso C de la tesis de [2].

8.7. Discusión de resultados

Inicialmente, considerando la complejidad de algunos de los escenarios presentados, los resultados pueden llegar a parecer altamente prometedores, no obstante, el método posee claras desventajas. En primer lugar, este método de planificación es de tipo *offline*, por lo que se requiere de conocimiento a *priori* del escenario que se navegará para poder generar las trayectorias. Esto retira la capacidad de los robots para poder reaccionar ante obstáculos dinámicos.

En segundo lugar, este algoritmo (sin intervención del usuario) es altamente propenso a generar trayectorias "inconclusas o cíclicas" en las que el agente nunca llega a alcanzar la meta. Esto se debe a que la trayectoria requerida para alcanzar la meta es sumamente larga, por lo que en estados muy alejados, los agentes carecerán de dirección o "motivación" para desplazarse en dirección a la meta. La solución que funcionó en la mayor parte de los casos consistió en incrementar en un orden de magnitud la recompensa obtenida al alcanzar la meta. Esto parece traer consigo una mejora a la capacidad del agente de "ver a futuro". De aquí que el valor de esta recompensa incrementara de 100 a 100000.

Finalmente, se tiene el problema de la resolución de la cuadrícula y el tiempo de procesamiento. Para mapas simples (como los presentados previamente), el planificador de trayectorias es capaz de capturar el detalle suficiente como para generar una trayectoria útil desde la posición inicial de los robots hasta la meta. No obstante, para mapas con diseños más intrincados, la captura de los detalles puede llegar consistir de un reto significativo. Para combatir esto se puede incrementar el número de celdas en las que se subdivide la mesa de trabajo. No obstante, esto trae dos problemas consigo: El incrementar el número de celdas también incrementa el "número de celdas de largo" de los caminos hacia la meta, así como el tiempo de procesamiento asociado.

Por lo tanto, para mapas de la misma dimensión que previamente se habían conseguido procesar correctamente, no solo se volverá a experimentar el problema donde el agente "pierde motivación" a medida que se aleja de la meta (por el mayor número de celdas en el camino), sino que también el tiempo para llegar a calcular la política asociada al agente tomará mucho más tiempo (debido al incremento en el número de estados). En otras palabras, este método de generación de trayectorias presenta capacidades de escalabilidad limitadas. Este puede ser aplicado a cuantos robots se deseen sin mayor problema, pero la precisión con la que se generan las trayectorias se ve limitada por la capacidad computacional disponible.

CAPÍTULO 9

Swarm Robotics Toolbox

Una de las primeras tareas que se realizó como parte de esta tesis, fue comprender y unir el contenido de las tesis de [1] y [2] 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*[20]

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 depuración (*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.

9.1. *Livescripts*

La extensión `.mlx` del script principal corresponde a un *livescript*. Un livescript permite el uso de código, imágenes, texto, ecuaciones, índices, secciones y otras características útiles dentro del mismo archivo.

[20]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.

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[21], 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.

9.2. 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.

9.3. Setup Path y limpieza de Workspace

Figura 61: Secciones iniciales del *Livescript* para el *SR Toolbox*.

Al apenas abrir el script, se encontrará con el índice y su primera sección: *Setup de Path* (Figura 61). 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

[21] 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.

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 características de *autocomplete*[22].

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 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.

9.4. 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:

9.4.1. 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 dinámicos (métodos que no requieren de planeación previa para realizar la exploración de la mesa de trabajo). En el caso de los métodos PSO, escribir en consola `help CostFunction`, para más información. En la sección 13.2, se provee la visualización y ecuaciones para algunas de las funciones de costo disponibles como parte de la *Toolbox*.

9.4.2. Dimensiones de mesa de trabajo

- `AnchoMesa`: Ancho de la mesa de trabajo. Unidades en metros (Figura 62).
- `AltoMesa`: Alto de la mesa de trabajo. Unidades en metros (Figura 62).

[22]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.

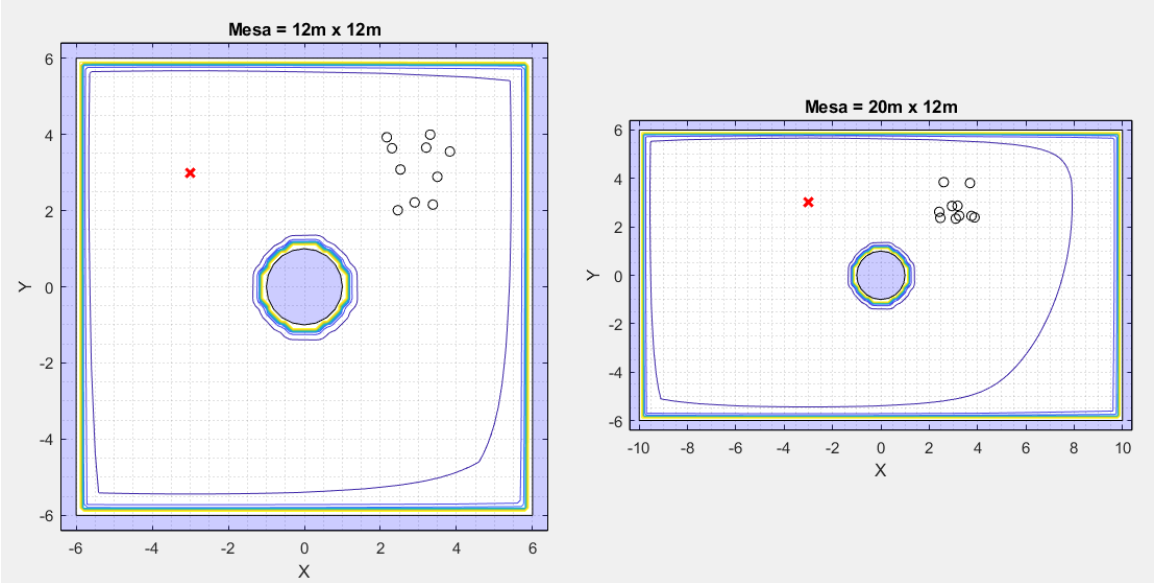


Figura 62: Efectos de alterar el ancho y alto de la mesa de trabajo.

- `Margen`: Ancho del margen uniforme que existirá alrededor de los bordes de la mesa de trabajo. Unidades en metros (Figura 63).

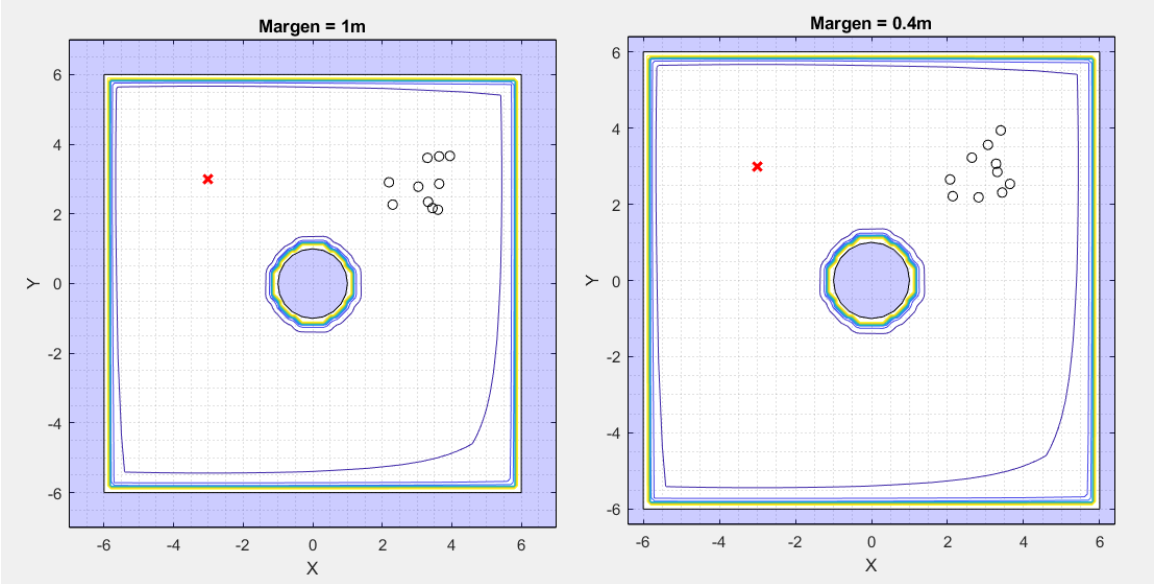


Figura 63: Efectos de alterar el tamaño del margen de la mesa de trabajo.

9.4.3. Ajustes 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.

9.4.4. Ajustes 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`.

9.4.5. Ajustes 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.

9.4.6. Ajustes 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 64).

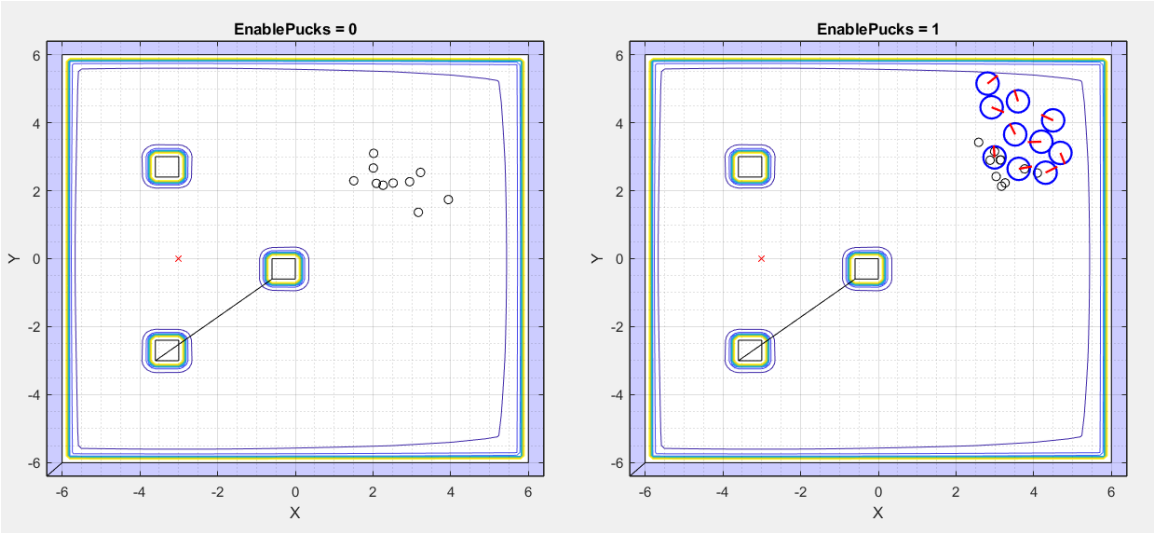


Figura 64: Efectos de alterar el parámetro `EnablePucks`.

- `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 cinco 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. Para más información escribir en consola `help getControllerOutput`.
- `CriterioPuck`: Similar al parámetro de `CriterioPart`. Determina el criterio de convergencia que utilizará el ciclo principal para determinar el momento en el que debe finalizar su ejecución.

9.4.7. 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 65).

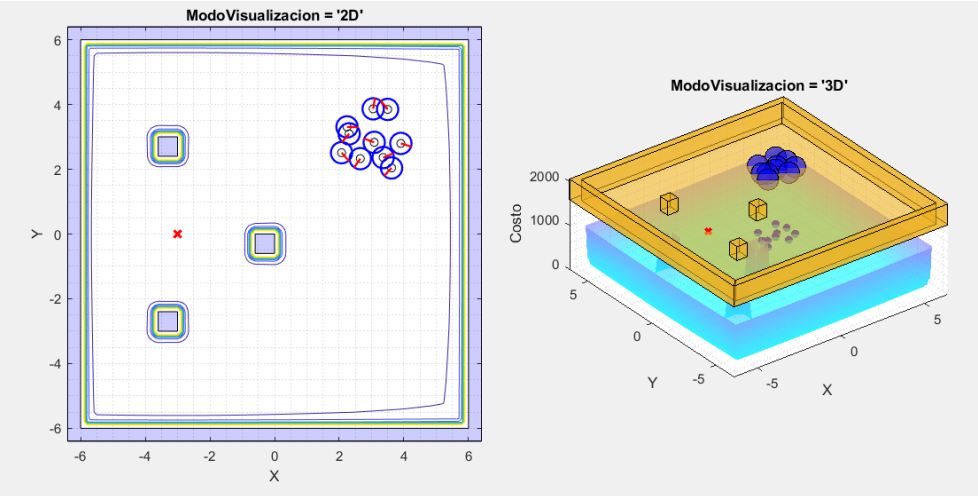


Figura 65: Efectos de alterar el parámetro `ModoVisualizacion`.

- `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 ciclo principal. 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.

9.4.8. Guardado de 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 *Media\Frames\Nombre de simulación actual*. Útil para crear GIFS dentro de Latex utilizando el paquete *animate*.

- `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*.

9.4.9. Ajustes del generador de números aleatorios

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.

9.5. Reglas de método a usar

Como se mencionó previamente, en el *Toolbox*, los robots pueden recorrer la mesa de trabajo utilizando tres 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[23], 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.

9.6. 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 66).

[23]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

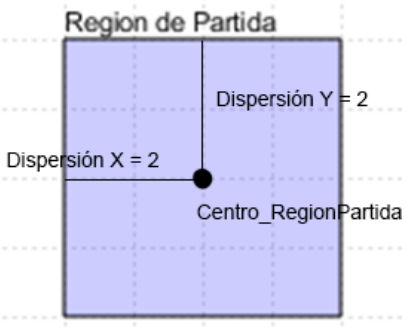


Figura 66: Explicación visual de cómo funciona la dispersión para la región de partida.

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 67).

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 vector que contiene las trayectorias consiste de un vector bidimensional, con cada fila representando una nueva meta en la secuencia.

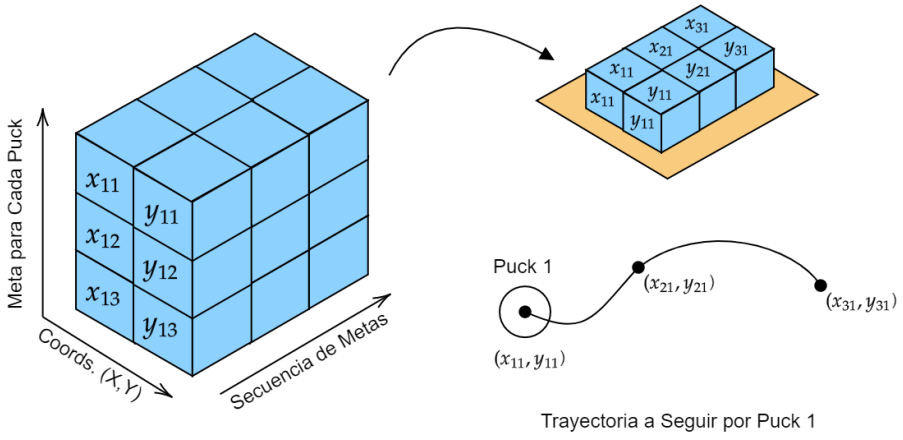


Figura 67: Estructura del vector de trayectorias para el caso *Multi-meta*.

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 emplea las coordenadas de la meta retornadas por la salida opcional (`varargout`) de la función `CostFunction`. En caso el usuario implemente una nueva función de costo, este debe asegurarse de colocar dicha meta de forma explícita o el algoritmo asumirá la meta dada por el usuario.

9.7. 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.

9.7.1. 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 (Figura 68).

Con esta herramienta únicamente se puede crear un único obstáculo para la mesa de trabajo[24] (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*.

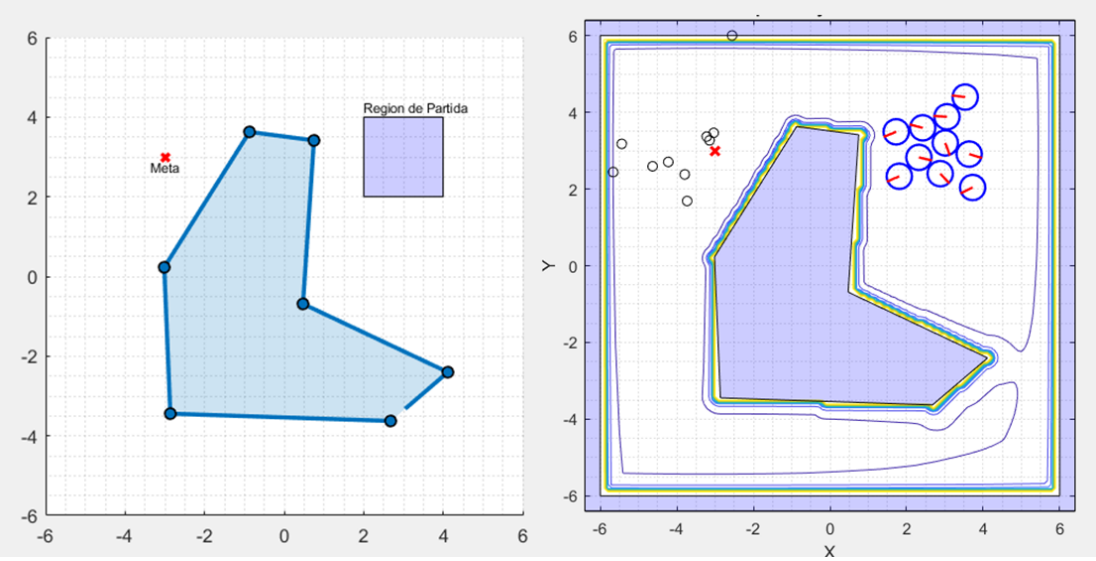


Figura 68: Creación de obstáculo poligonal.

9.7.2. 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`.

[24]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 complicado, por lo que se optó por abandonar la idea.

9.7.3. 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 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.

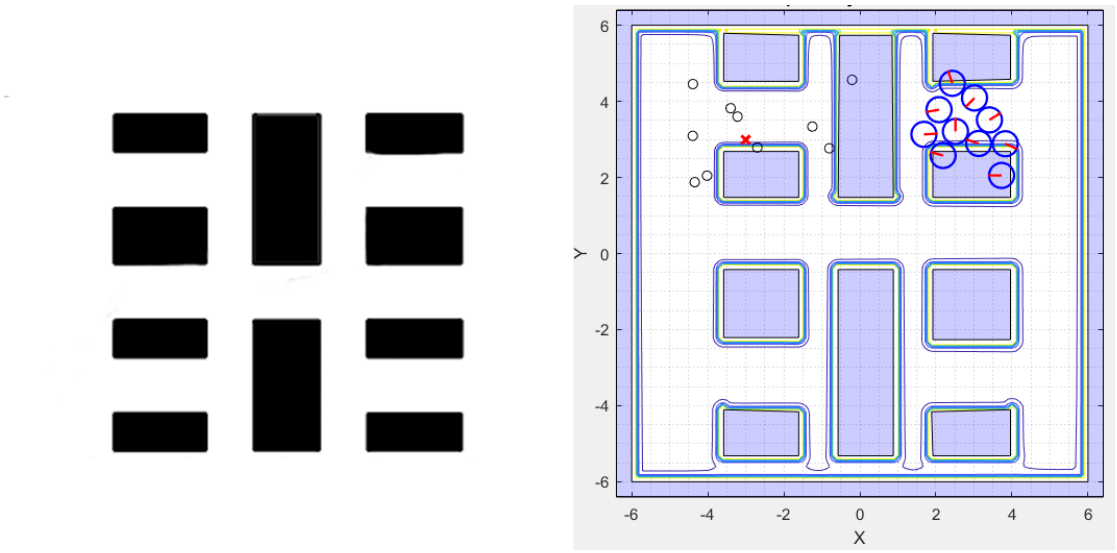


Figura 69: Creación de obstáculos basados en una imagen en blanco y negro.

9.7.4. Caso A

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

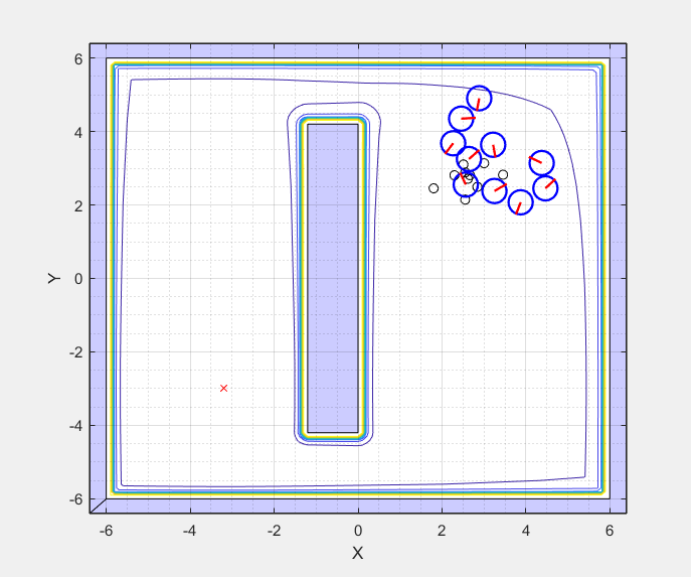


Figura 70: Caso A en tesis de Juan Pablo Cahueque

9.7.5. Caso B

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

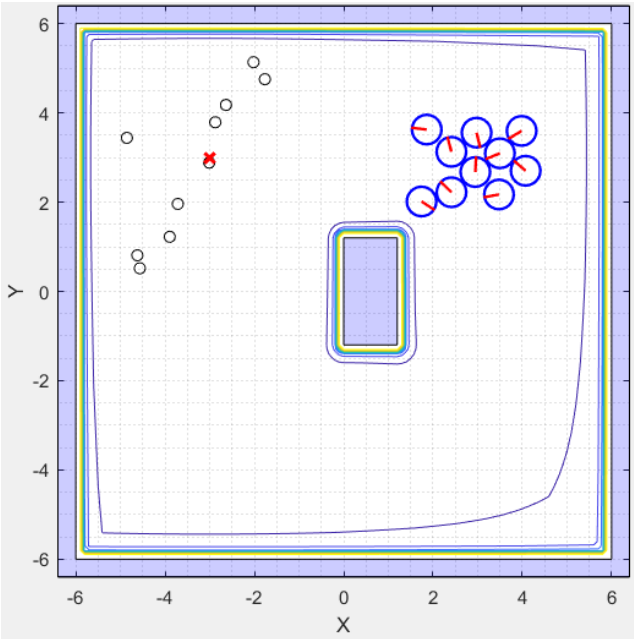


Figura 71: Caso B en tesis de Juan Pablo Cahueque

9.7.6. Caso C

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

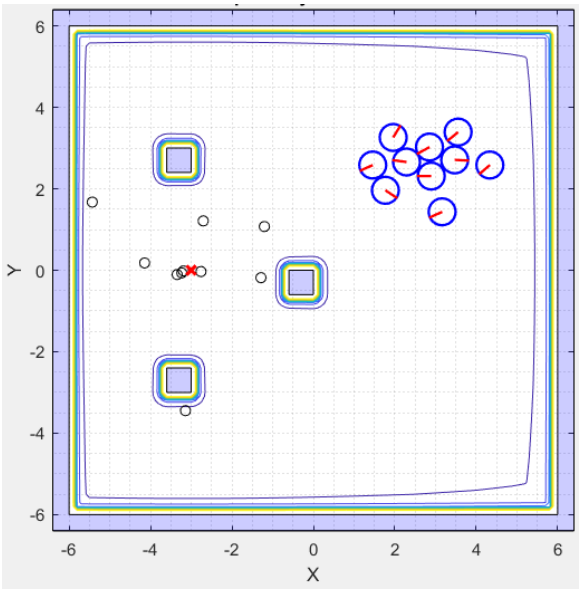


Figura 72: Caso C en tesis de Juan Pablo Cahueque

9.8. Ajustes 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.

9.8.1. 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).

9.8.2. 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`

9.8.3. 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.

9.8.4. 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.

9.8.5. 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 [14], el usuario puede obviar la ecuación 2 y utilizar el valor de inercia que desee. Se ofrecen 5 tipos diferentes de inercia. Para más información escribir en consola: `help ComputeInertia`
- Constricción: Criterio de convergencia propuesto por [14]. Este criterio asegura la convergencia del algoritmo siempre y cuando $\kappa = 1$ y $\phi_1 + \phi_2 > 4$.

- Mixto: Uso simultáneo de un tipo de inercia (por defecto exponencial natural), en conjunto con los parámetros de constricción propuestos por [14]. Utilizado por [1] en su tesis.

9.9. Ajustes de 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 ciclo principal, 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 tres partes: La leyenda, la descripción y la región de simulación (Figura 73).

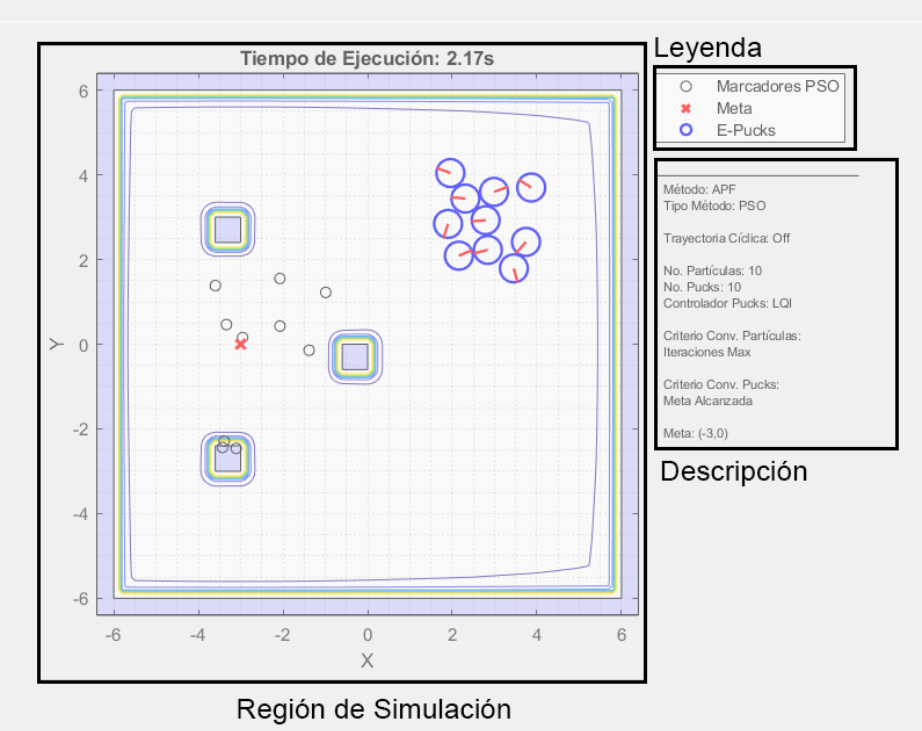


Figura 73: Partes de la figura de simulación.

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[25].

[25]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`.

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 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`.

Código 9.1: Ejemplo de inclusión del marcador de meta en la leyenda de la figura de simulación

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

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.

9.10. Ciclo principal

El ciclo principal o *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 ciclo principal 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.

9.11. Análisis de resultados

Al finalizar la simulación, se pueden generar 4 gráficas informativas sobre el movimiento de los robots sobre la mesa de trabajo.

9.11.1. Evolución del Global Best

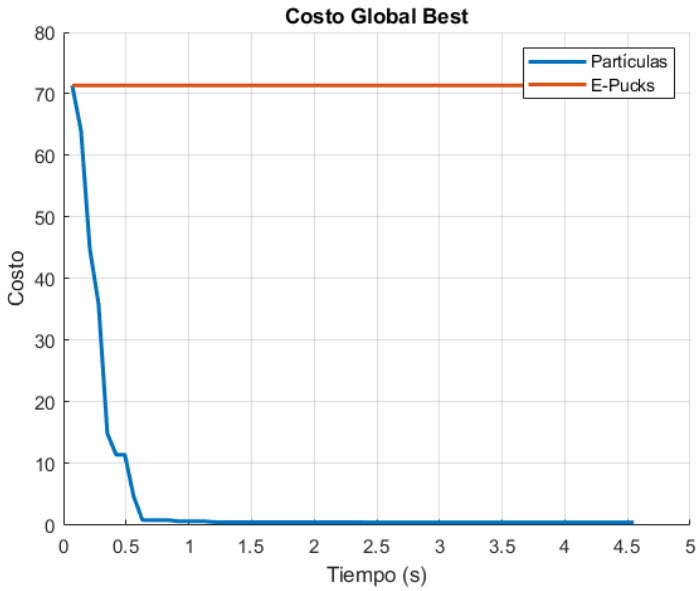


Figura 74: Evolución de la minimización hacia el *global best* de la función.

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 (Figura 74) puede utilizarse para determinar si las partículas alcanzaron el mínimo global o un mínimo local.

9.11.2. Análisis de dispersión de partículas

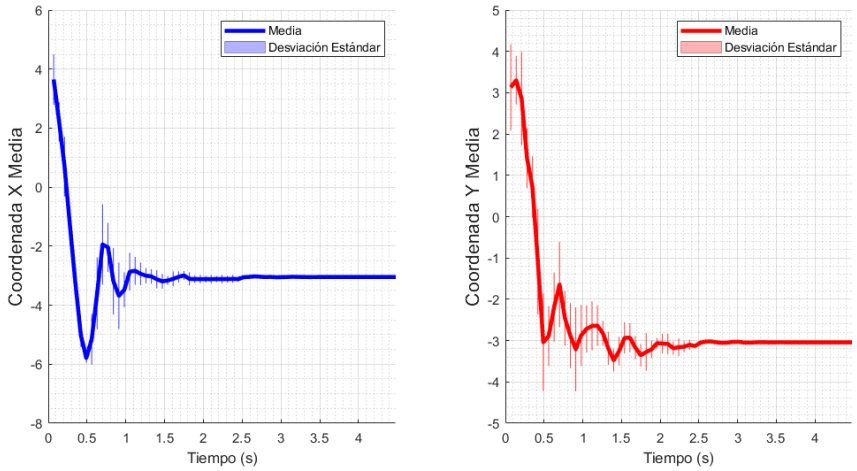


Figura 75: Dispersión de las partículas sobre el eje X y Y.

Dos cualidades importantes de las partículas del PSO es su capacidad de exploración y la precisión de su minimización. Con estas gráficas (Figura 75), 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.

9.11.3. Velocidad de motores

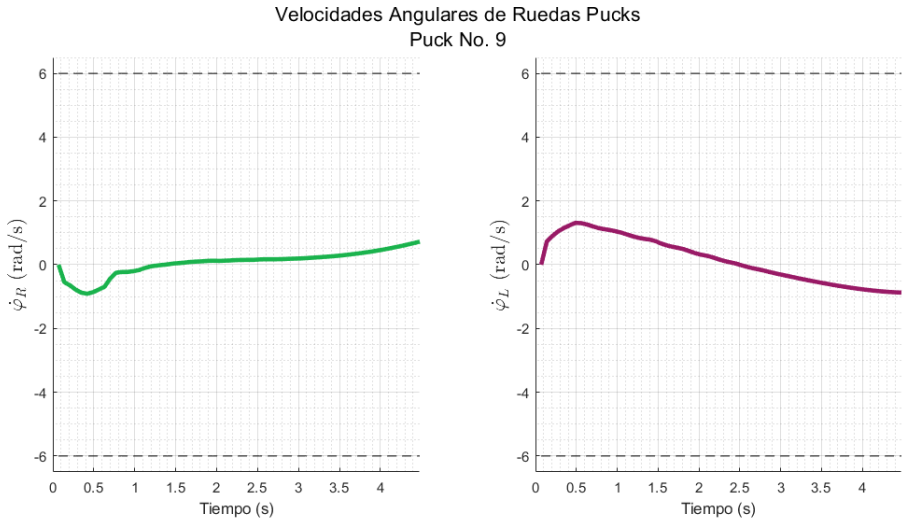


Figura 76: Velocidad angular observada en los motores del puck con los picos más altos de velocidad en dicha corrida.

Utilizando la cinemática inversa de un robot diferencial se calculan las velocidades angulares de las ruedas de todos los robots (Ecuación 47). 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 (Figura 76). 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`).

$$\dot{\varphi}_R = \frac{2v + 2\omega l}{2r}, \quad \dot{\varphi}_L = \frac{2v - 2\omega l}{2r} \tag{47}$$

9.11.4. Suavidad de velocidades

Basado en el criterio de evaluación empleado por [1] en su tesis. Se realiza una interpolación de los puntos que conforman la curva de velocidades angulares de las ruedas, y luego se calcula y grafica la energía de flexión de la curva (Figura 77). Si la energía de flexión es baja, la suavidad de operación es mucho mayor. Prueba ideal para diagnosticar cuantitativamente la suavidad de operación.

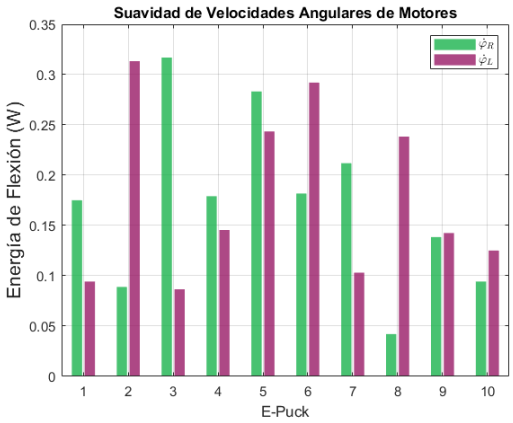


Figura 77: Energía de flexión observada en las velocidades angulares de las ruedas de cada puck.

9.12. Colisiones

Un entorno de simulación realista es necesario para obtener resultados útiles al momento de realizar pruebas. Debido a esto, se implementó detección de colisiones 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.

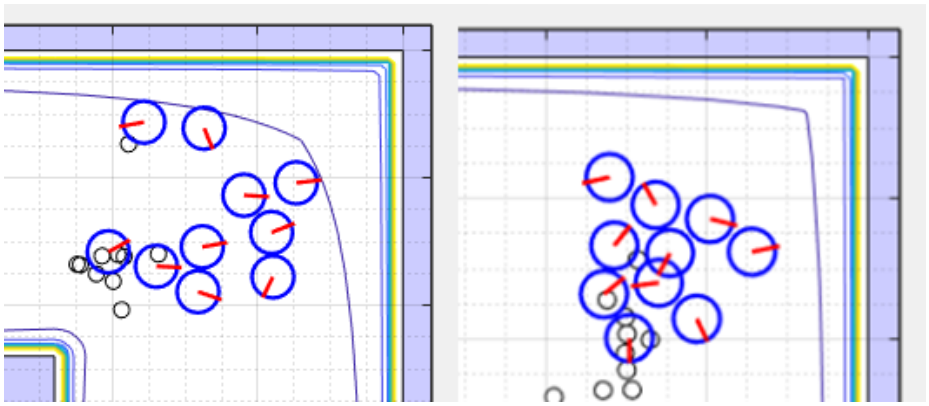


Figura 78: Con solución de colisiones (izquierda) y sin solución de colisiones (derecha).

No obstante, 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 causar que se bloquee 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 detección de colisiones mucho más robusto como "colisiones especulativas" que también incluya elementos como las paredes o los obstáculos como tal.

9.13. 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 el *Toolbox* se ofrecen cinco 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 dificultades de implementación) consistió del controlador de direccionamiento por lazo cerrado. En las figuras 79-83 se brinda una breve descripción del tipo de movimiento generado con cada tipo de controlador.

9.13.1. *Linear Quadratic Regulator* (LQR)

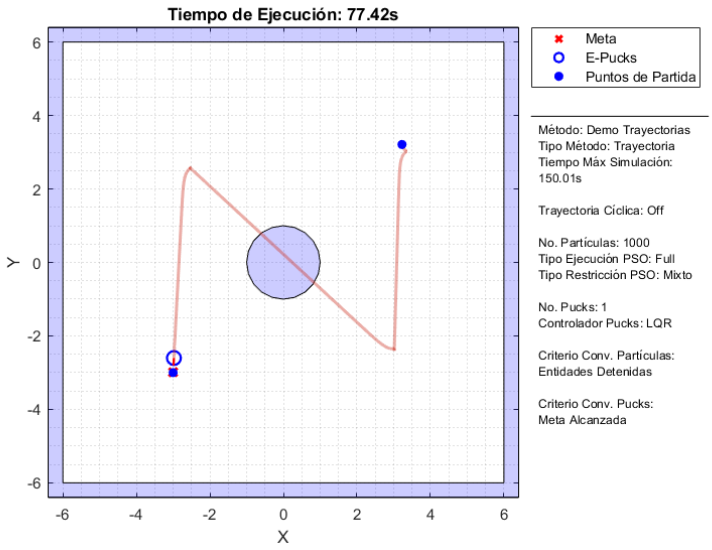


Figura 79: Seguimiento de la trayectoria (3,-3), (-3,3), (-3,-3) con un controlador LQR.

Movimiento rápido que desacelera conforme el robot se acerca a la meta. Los cambios de dirección involucran un detenimiento total del robot seguido de un giro.

9.13.2. *Linear Quadratic Integral Control* (LQI)

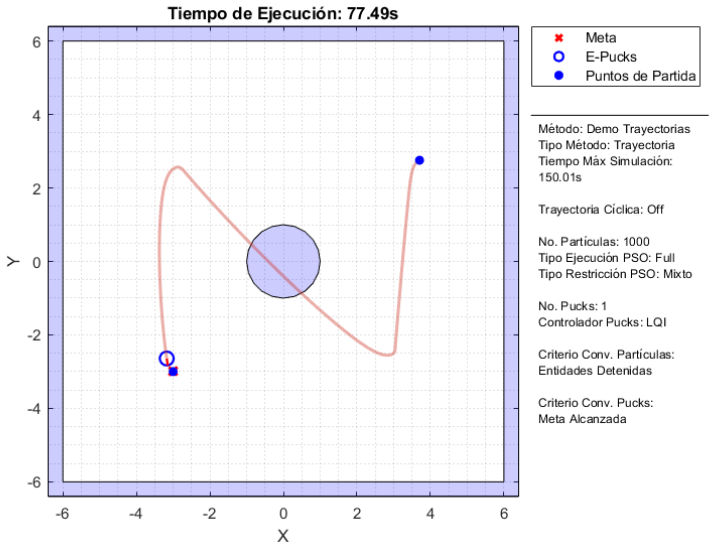


Figura 80: Seguimiento de la trayectoria (3,-3), (-3,3), (-3,-3) con un controlador LQI.

Movimiento comparable a aquel observado en el controlador LQR, pero con una desaceleración menos pronunciada y sin giros agudos en el cambio de meta a meta.

9.13.3. Controlador de pose simple

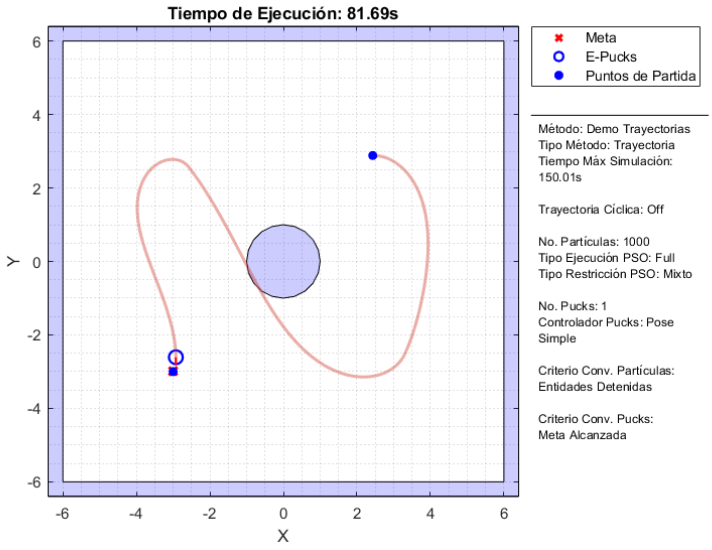


Figura 81: Seguimiento de la trayectoria (3,-3), (-3,3), (-3,-3) con un controlador de pose simple.

Movimiento con velocidad menor a aquella observada en los controladores LQR y LQI. Debido a su aceleración angular menor, las trayectorias generadas son más suaves y largas.

9.13.4. Controlador de pose con criterio de estabilidad de lyapunov

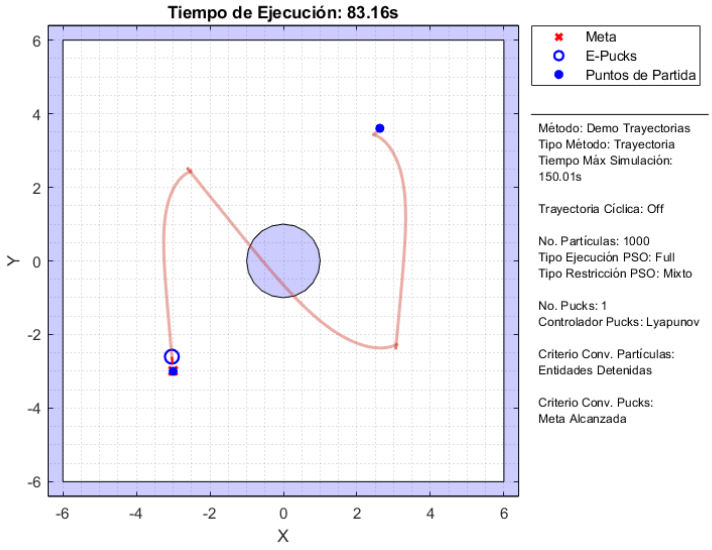


Figura 82: Seguimiento de la trayectoria (3,-3), (-3,3), (-3,-3) con un controlador de pose con criterio de estabilidad de Lyapunov.

Misma velocidad que en el controlador de pose simple. Giros agudos, pero aceleraciones angulares bajas al momento de iniciar el movimiento lineal.

9.13.5. Controlador de direccionamiento de lazo cerrado

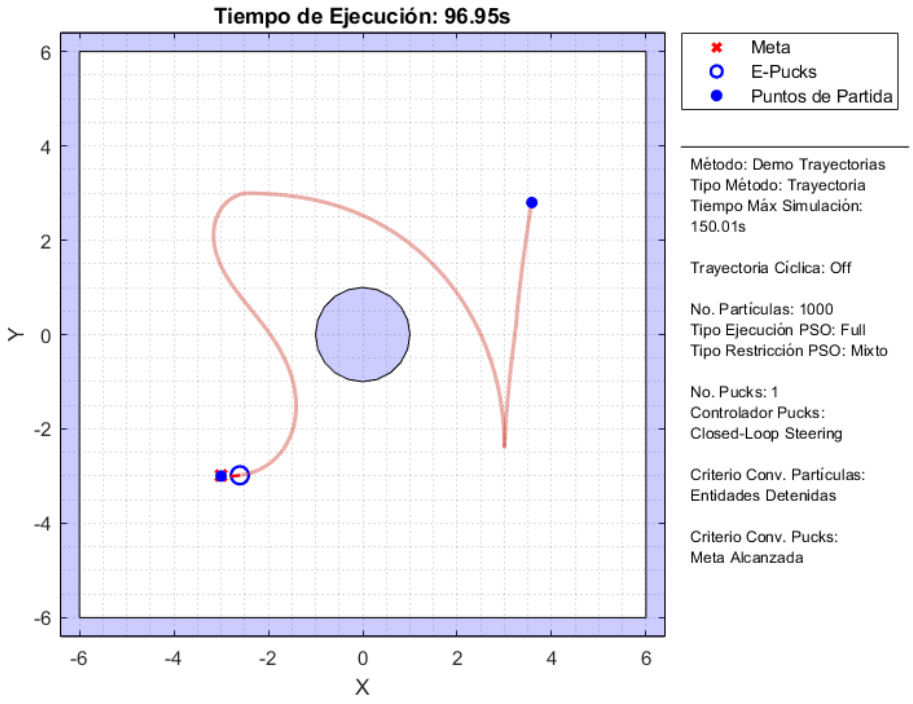


Figura 83: Seguimiento de la trayectoria (3,-3), (-3,3), (-3,-3) empleando un controlador de direccionamiento de lazo cerrado.

Controlador con la menor velocidad de entre los cinco presentados. El robot busca alinear su dirección con la meta, pero no su sentido. Por lo tanto, no importando si su eje +X (línea rojo vivo en Figura 83) o -X apunta en la dirección de la meta, este se moverá hacia la misma. Esto implica que según le sea conveniente, el robot se desplazará hacia adelante o en reversa hacia la meta. La aceleración angular es baja, produciendo giros sumamente suaves; no obstante, debido a la alta velocidad lineal asociada al movimiento, el robot tiende a desviarse ligeramente del punto hacia el que desea orientarse, causando que las trayectorias tengan una mayor longitud.

9.14. 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:

- `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. Como se puede observar en la Figura 84, el algoritmo se detiene a los 11.9s ya que las partículas han alcanzado la meta.

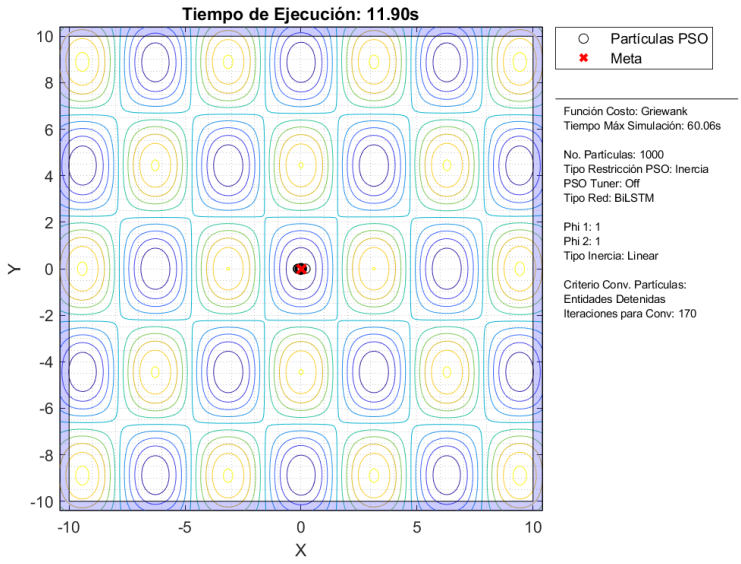


Figura 84: Utilización del criterio de convergencia de *meta alcanzada*.

- `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. En la Figura 85 se puede observar que el algoritmo se detuvo en el tiempo 0.91s (con el máximo siendo 60.6s) a pesar de no haber alcanzado la meta. Esto se debe a que las partículas convergieron y se detuvieron en un mínimo local.

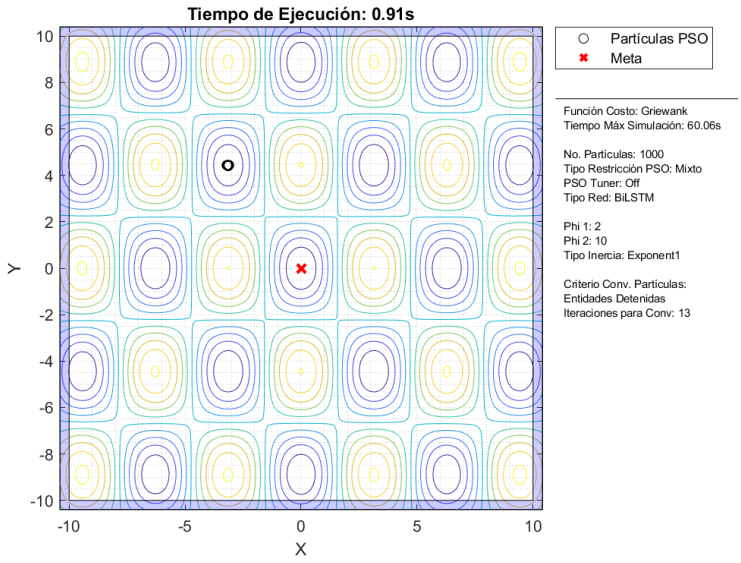


Figura 85: Utilización del criterio de convergencia de *entidades detenidas*.

- `Iter máx alcanzadas`: Se ha alcanzado el número máximo de iteraciones. En la Figura 86 se puede observar que el algoritmo se detuvo exactamente en el tiempo máximo de 60.6s a pesar de haber convergido previamente en la meta.

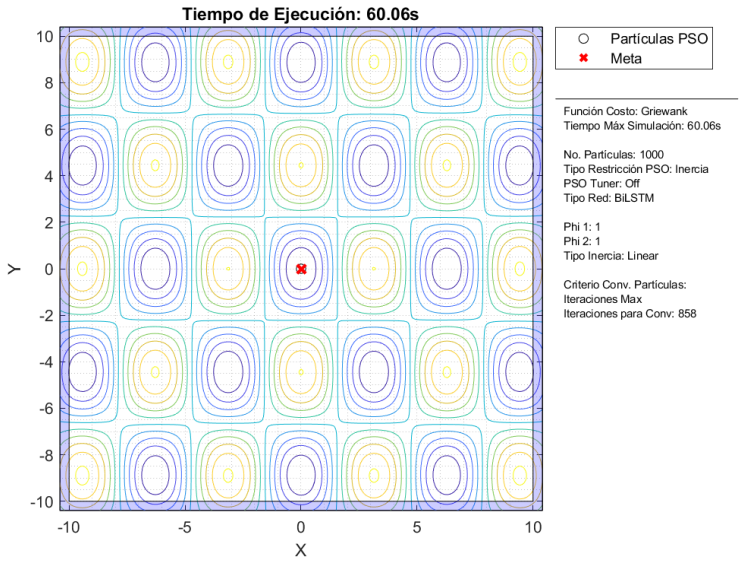


Figura 86: Utilización del criterio de convergencia de *iteraciones máximas alcanzadas*.

Cabe mencionar que 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 10

Conclusiones

- El *PSO Tuner* fue capaz de generar exitosamente predicciones dinámicas para los parámetros ϕ_1, ϕ_2 y ω, propios del algoritmo PSO estándar.
- El *PSO Tuner* es capaz de generar un comportamiento altamente resistente a los mínimos locales, gracias a que las redes entrenadas son capaces de generar parámetros que permiten que las partículas "escapen" de los mínimos locales.
- Las redes neuronales entrenadas produjeron un comportamiento emergente en el que intentaban "imitar" el movimiento característico de un algoritmo PSO según el método de restricción elegido. Dicho comportamiento no consiste de un caso de *overfit* ya que las constantes que producen dicho comportamiento fluctúan a lo largo del desarrollo del algoritmo.
- El *PSO Tuner* basado en una red BiLSTM, presentó el mejor rendimiento promedio en términos de su tiempo de predicción, tiempo de convergencia, precisión de convergencia y capacidad de "suavización" en la trayectoria seguida por el centro del enjambre.
- Para casos con un número bajo de partículas, el método más robusto para incrementar la precisión del algoritmo PSO estándar es la utilización de la red BiLSTM, en conjunto con el método de restricción por constricción. El peor método, consiste de la red GRU.
- Las redes LSTM y GRU, requieren de dos muestras simultáneas y de la desactivación sobre el control de la inercia para producir resultados adecuados.
- La red GRU consistió de la red con peor desempeño general en términos de su precisión y tiempo de convergencia. Esta no solo tendía a producir los resultados más pobres, sino aquellos con mayor variabilidad en los datos. La red era imprecisa e inconsistente.
- El método de generación de trayectorias basado en programación dinámica posee un alta efectividad, pero, a su vez, cuenta con múltiples limitantes: Carece de reactividad ante obstáculos dinámicos, requiere de constante intervención por parte del usuario y cuenta con capacidades de escalabilidad limitadas.

- El agente del algoritmo de generación de trayectorias tiende a perder "motivación" para llegar a la meta cuando este inicia en celdas muy alejadas de la meta. Para solucionar esto se puede incrementar en un orden de magnitud la recompensa obtenida al momento de alcanzar la meta.
- El método de generación de trayectorias es altamente escalable en términos del número de robots para los que se pueden planificar trayectorias de manera simultánea. Las limitantes en escalabilidad surgen al momento de incrementar la precisión de las trayectorias generadas.
- El *Swarm Robotics Toolbox* consiste de una herramienta altamente versátil, la cual puede llegar a ser fácilmente modificada para realizar diferentes tipos de pruebas: Desde pruebas con algoritmos de optimización como PSO, hasta pruebas de navegación de robots móviles.

CAPÍTULO 11

Recomendaciones

- Durante el ajuste de hiper parámetros de una red neuronal recurrente, se recomienda que el primer parámetro con el que se experimente sea el *batch size*.
- Al ensamblar una red neuronal recurrente asegurarse de no incluir más de una capa de neuronas recurrentes a menos que la aplicación sea particularmente exigente.
- En una red neuronal recurrente, nunca olvidar incluir una *dropout layer* directamente después de la capa de neuronas recurrentes para impedir *overfitting*.
- Un gran número de modificaciones al algoritmo PSO únicamente consisten de términos adicionados hacia el final de la ecuación de actualización de la velocidad (b).

$$v(t+1) = \chi\left(\omega v(t) + \vec{U}\,(0,\phi_1)\,(\overrightarrow{p_{\text{local}}} - \vec{x}_i) + \vec{U}\,(0,\phi_2)\,(\overrightarrow{p_{\text{global}}} - \vec{x}_i)\right) + b$$

  Se puede explorar la posibilidad de que el *PSO Tuner* genere como cuarta salida, el valor de esta constante adicional b. Esto permitiría que el algoritmo no solo tome información sobre el comportamiento del PSO tradicional, sino también de algunas modificaciones útiles al mismo.
- La métrica de "promedio de la distancia promedio entre partículas" consiste de una réplica incorrectamente normalizada y amplificada de la desviación estándar promedio normalizada. Para futuras iteraciones del *PSO Tuner* se recomienda eliminar esta métrica del vector de características de la red neuronal y agregar alguna métrica acotada entre (0,1) asociada al costo.
- El *PSO Tuner* emplea la distancia hacia el mínimo global de la función de costo, para poder corregir la posición en la que converge. En un problema tradicional de optimización, esta información no está disponible. Entonces, se sugiere entrenar a la red para producir una estimación de esta cantidad, la cual posteriormente es realimentada a la red para producir el siguiente estimado.

- Para la futura exploración del área de robótica con aprendizaje reforzado, se recomienda utilizar un entorno de simulación que incluya un "motor de física" ya incorporado, dado que se depende de la correcta interacción del agente con el ambiente para producir un proceso de aprendizaje adecuado.

- La forma tradicional de la programación dinámica empleada para la generación de trayectorias, produce una política completa y óptima al finalizar su ejecución. Existen versiones alternativas de este algoritmo basado en métodos *Monte Carlo* (métodos basado en la recolección de muestras) los cuales permiten la actualización dinámica de los valores de cada estado según estos se visitan. Por lo tanto, en lugar de generar una trayectoria de forma automática, se podría diseñar un método que utilice un enjambre "exploratorio" de robots para poder producir los valores de los estados y así generar la política que posteriormente podrán utilizar robots para llegar a la meta.

- Se puede llegar a modelar el espacio de estados de los robots como la pose de los mismos. Si se hace esto, entonces el espacio de estados se torna continuo y sus dimensiones crecen hacia valores virtualmente infinitos. Para solucionar esto, se puede hacer uso de aprendizaje reforzado profundo para estimar el estado actual del robot, y luego utilizando la generación dinámica de trayectorias por medio de un enjambre exploratorio, se pueden generar trayectorias con un alta precisión capaces de explorar incluso rincones reducidos del mapa (ya que se retira la necesidad de cuadricular la mesa de trabajo).

- Se puede asociar el proceso de generación de políticas de forma dinámica a agentes sujetos a un algoritmo genético que promueva la evolución de las mejores políticas. Se crea una población con diferentes variaciones aleatorias de una política y se disponen todas en un escenario de prueba. En función de su desempeño, pasan a los agentes progenitores los mejores aspectos de una política. Esto aceleraría la navegación dinámica a través de un escenario, ya que permite la realización de un gran número de pruebas de forma simultánea, manteniendo los mejores aspectos de cada prueba.

- Implementar un "modo manual" que permita el control de los robots diferenciales del *SR Toolbox* de forma similar a un carro a control remoto. Los datos recolectados pueden ser utilizados para realizar *aprendizaje reforzado inverso*, en el que el algoritmo intenta deducir las acciones requeridas para obtener el comportamiento óptimo dado como ejemplo por el usuario.

CAPÍTULO 12

Bibliografía

[1] A. Nadalini, *Algoritmo Modificado de Optimización de Enjambre de Partículas (MPSO)*, 2019.

[2] J. Cahueque, *Implementación de Enjambre de Robots en Operaciones de Búsqueda y Rescate*, 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*, 2019.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

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

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

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

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

[22] F. Gaillard, *Batch size (machine learning): Radiology Reference Article*. dirección: https://radiopaedia.org/articles/batch-size-machine-learning.

[23] S. Sharma, *Epoch vs Batch Size vs Iterations*, mar. de 2019. dirección: https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9.

[24] J. Brownlee, *Understand the Impact of Learning Rate on Neural Network Performance*, sep. de 2020. dirección: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/.

[25] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever y R. Salakhutdinov, «Dropout: A Simple Way to Prevent Neural Networks From Overfitting», *Journal of Machine Learning Research*, vol. 15, págs. 1929-1958, 2014, ISSN: 10636919. DOI: 10.1109/CVPR.2018.00797. arXiv: 1803.11095.

[26] M. White y A. White, *Fundamentals of Reinforcement Learning*.

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.

[32] 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.

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

[34] O. Olorunda y A. P. Engelbrecht, «Measuring exploration/exploitation in particle swarms using swarm diversity», *2008 IEEE Congress on Evolutionary Computation, CEC 2008*, págs. 1128-1134, 2008. DOI: 10.1109/CEC.2008.4630938.

[35] T. Krink, J. S. Vesterstrom y J. Riget, «Particle swarm optimisation with spatial particle extension», *Proceedings of the 2002 Congress on Evolutionary Computation, CEC 2002*, vol. 2, págs. 1474-1479, 2002. DOI: 10.1109/CEC.2002.1004460.

[36] T. Stottner, *Why Data should be Normalized before Training a Neural Network*, mayo de 2019. dirección: https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d.

[37] I. Rasin, *Batch Size*, 2017.

[38] I. Jabandzic y J. Velagic, «Particle swarm optimization-based method for navigation of mobile robot in unstructured static and time-varying environments», *Conference on Control and Fault-Tolerant Systems, SysTol*, vol. 2016-Novem, págs. 59-66, 2016, ISSN: 21621209. DOI: 10.1109/SYSTOL.2016.7739729.

[39] W. George, *Working around TDR in Windows for a better GPU computing experience*, 2016. dirección: https://www.pugetsystems.com/labs/hpc/Working-around-TDR-in-Windows-for-a-better-GPU-computing-experience-777/.

CAPÍTULO 13

Anexos

13.1. Matlab: Cálculo de métricas de PSO

13.1.1. Desviación estándar promedio normalizada

```
% Desviacion estandar para las coordenadas X
DesvEstPosX = std(Part.Posicion_Actual(:,1)) / (AnchoMesa/2);

% Desviacion estandar para las coordenadas Y
DesvEstPosY = std(Part.Posicion_Actual(:,2)) / (AltoMesa/2);

% Promedio de ambas dimensiones
DesvEstPosMedia = mean([DesvEstPosX DesvEstPosY]);
```

13.1.2. Coherencia

```
% Velocidad del centro del swarm (V_s)
VelCentroSwarm = norm(mean(Part.Velocidad));

% Velocidad promedio de las particulas (\overline{V})
VelPromedioParts = mean(vecnorm(Part.Velocidad,2,2));

% Coherencia (Ajustado para evitar division entre 0)
CoherenciaSwarm = (VelCentroSwarm + 0.01)/ (VelPromedioParts + 0.01);
```

13.1.3. Distancia de meta a *Global Best* normalizada

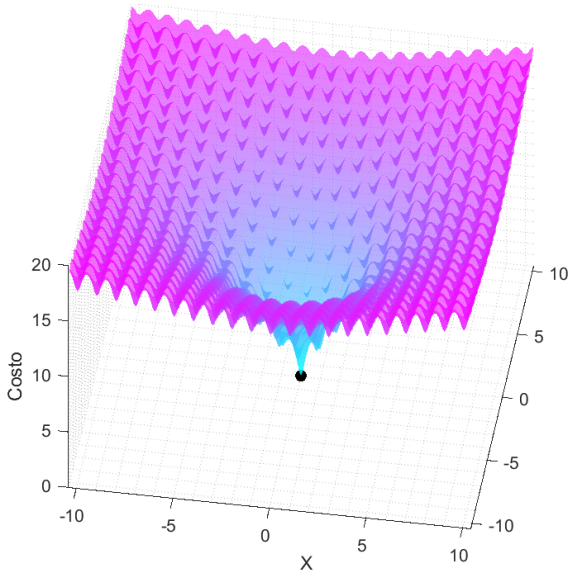
```
% Indice de la meta mas cercana al global best
IndDistMin = dsearchn(Meta, Part.Posicion_GlobalBest);

% Componente X de la distancia entre meta y global best
ComponenteXDist = Meta(IndDistMin,1) - Part.Posicion_GlobalBest(1);

% Componente Y de la distancia entre meta y global best
ComponenteYDist = Meta(IndDistMin,2) - Part.Posicion_GlobalBest(2);

% Distancia mas lejana desde la meta a una de las esquinas de la mesa
DistMasLejana = hypot(LimsX(2) - abs(Meta(IndDistMin,1)), ...
                      LimsY(2) - abs(Meta(IndDistMin,2)));

% Normalizacion de componentes y calculo de distancia
DistAMeta_Norm = hypot(ComponenteXDist, ComponenteYDist) / DistMasLejana;
```

13.1.4. Promedio de distancia promedio entre todas las partículas del enjambre

```
% Posicion del centro del swarm (Promedio de sus coordenadas)
MediaSwarm = mean(Part.Posicion_Actual);

% Distancias entre todas las particulas (Para mas informacion sobre
% getDistsBetweenParticles() escribir 'help getDistsBetweenParticles'
% en consola).
DistsEntrePartsSwarm = getDistsBetweenParticles(Part.Posicion_Actual,'Full');

% Distancia mas lejana entre la media de la swarm y los extremos
% de la region de busqueda (LimsX y LimsY)
DistMasLejana = hypot(LimsX(2) - abs(MediaSwarm(1)), ...
                      LimsY(2) - abs(MediaSwarm(2)));

% Promedio de todas las distancias promedio entre particulas.
% Division para la normalizacion de los datos
PromDistPromPartASwarm = mean(DistsEntrePartsSwarm,'all') / DistMasLejana;
```

13.2. Funciones de costo

13.2.1. Ackley

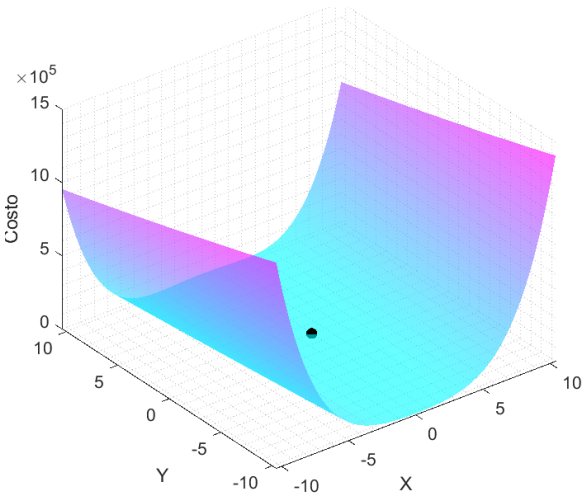


$$f(\mathbf{x}) = -a \exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1) \tag{48}$$

$a = 20$
$b = 0.2$
$c = 2\pi$
$d = 2$ (No Dims)

Figura 87: Visualización y ecuación de la función de costo Ackley. Mínimo: (0,0).

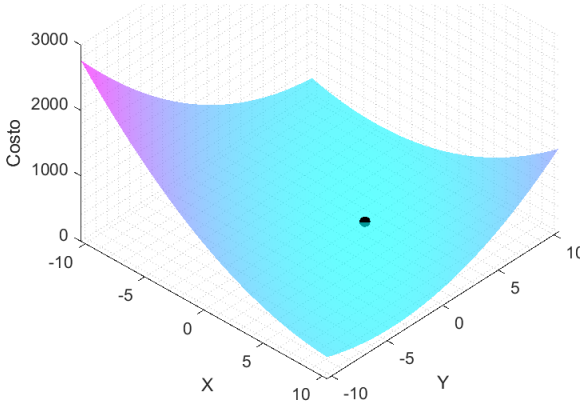
13.2.2. Banana / Rosenbrock



$$f(\mathbf{x}) = \sum_{i=1}^{d-1}\left[100\left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2\right] \tag{49}$$

Figura 88: Visualización y ecuación de la función de costo Banana / Rosenbrock. Mínimo: (1,1).

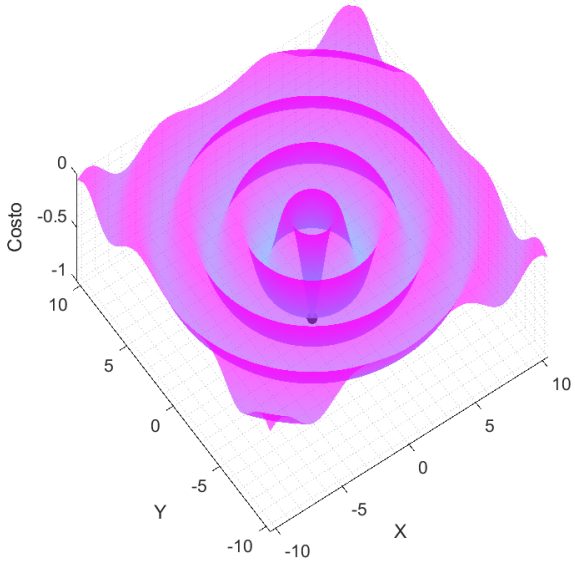
13.2.3. Booth



$$f(\mathbf{x}) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2 \tag{50}$$

Figura 89: Visualización y ecuación de la función de costo Booth. Mínimo: (0,0).

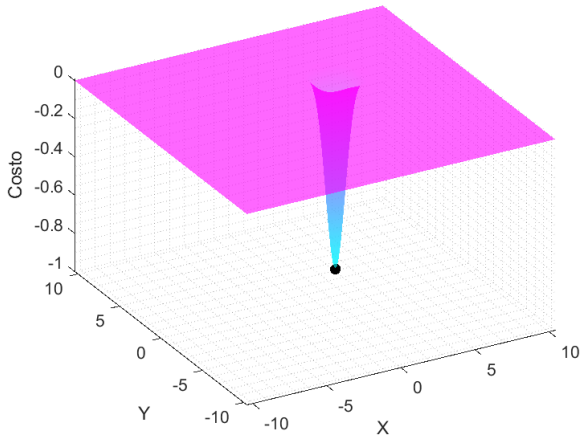
13.2.4. Dropwave



$$f(\mathbf{x}) = -\frac{1 + \cos\left(12\sqrt{x_1^2 + x_2^2}\right)}{0.5\left(x_1^2 + x_2^2\right) + 2} \tag{51}$$

Figura 90: Visualización y ecuación de la función de costo Dropwave. Mínimo: (0,0).

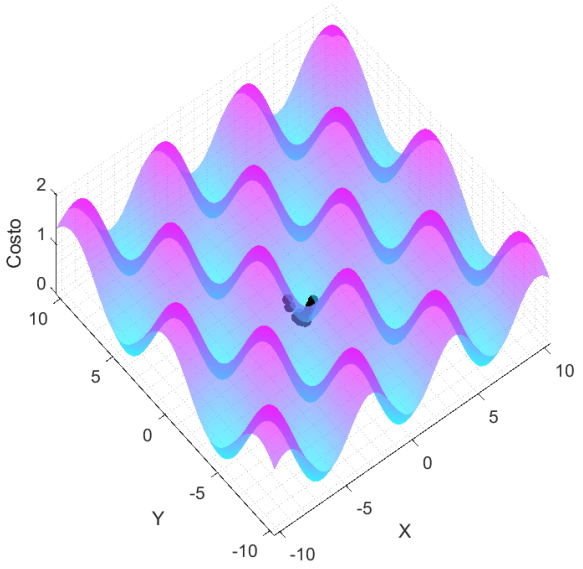
13.2.5. Easom



$$f(\mathbf{x}) = -\cos(x_1)\cos(x_2) \exp\left(-(x_1-\pi)^2 - (x_2-\pi)^2\right) \tag{52}$$

Figura 91: Visualización y ecuación de la función de costo Easom. Mínimo: (π, π).

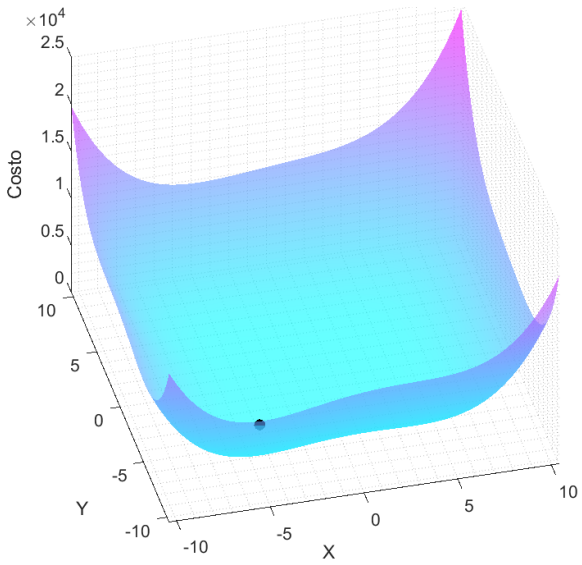
13.2.6. Griewank



$$f(\mathbf{x}) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \tag{53}$$

Figura 92: Visualización y ecuación de la función de costo Griewank. Mínimo: (0,0).

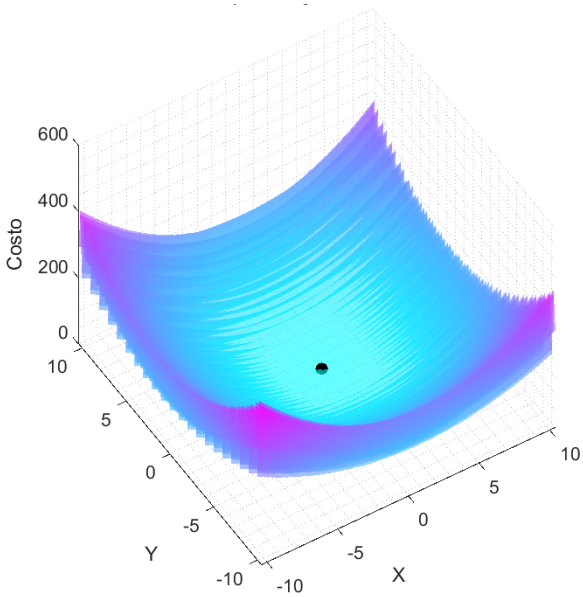
13.2.7. Himmelblau



$$f(x,y) = \left(x^2 + y - 11\right)^2 + \left(x + y^2 - 7\right)^2 \tag{54}$$

Figura 93: Visualización y ecuación de la función de costo Himmelblau. Múltiples mínimos: (3,2), (-2.8051 3.1313), (-3.7793 -3.2831) y (3.5844 -1.8481).

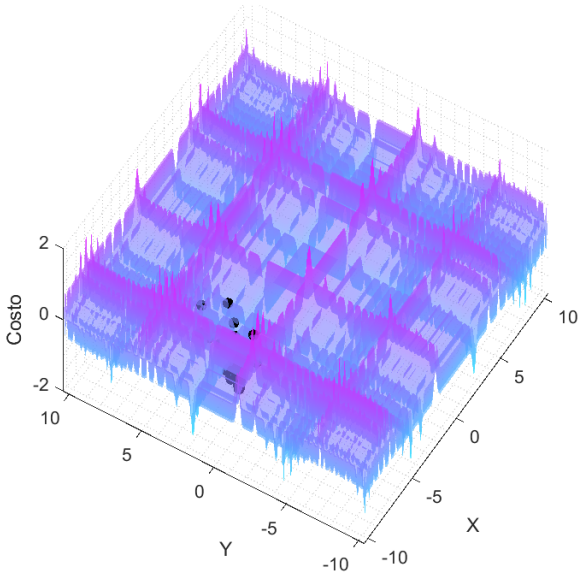
13.2.8. Levy No. 13



$$f(\mathbf{x}) = \sin^2(3\pi x_1) + (x_1 - 1)^2\left[1 + \sin^2(3\pi x_2)\right] + (x_2 - 1)^2\left[1 + \sin^2(2\pi x_2)\right] \tag{55}$$

Figura 94: Visualización y ecuación de la función de costo Levy No. 13. Mínimo: (1,1).

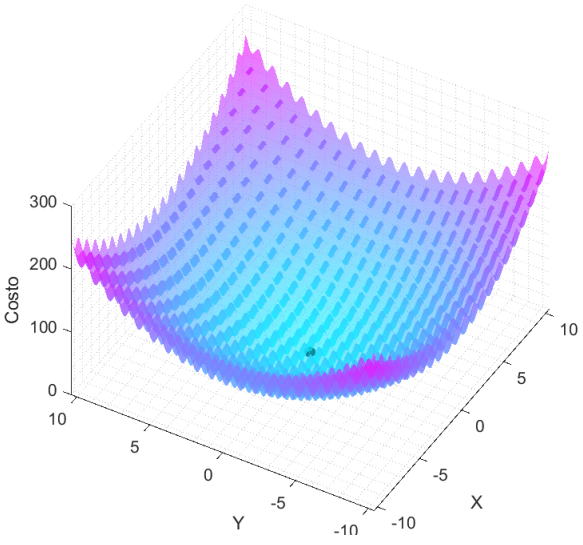
13.2.9. Michalewicz



$$f(\mathbf{x}) = -\sum_{i=1}^{d} \sin(x_i) \sin^{2m}\left(\frac{ix_i^2}{\pi}\right) \tag{56}$$

Figura 95: Visualización y ecuación de la función de costo Michalewicz. Mínimo: (2.2,1.57).

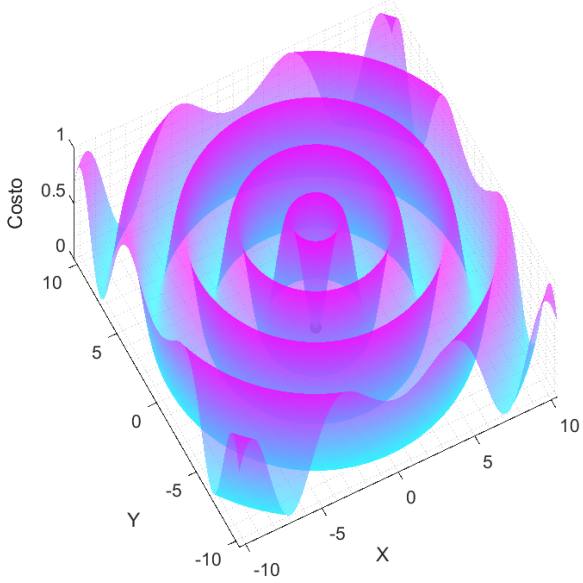
13.2.10. Rastrigin



$$f(\mathbf{x}) = 10d + \sum_{i=1}^{d} \left[x_i^2 - 10\cos(2\pi x_i)\right] \tag{57}$$

Figura 96: Visualización y ecuación de la función de costo Rastrigin. Mínimo: (0,0).

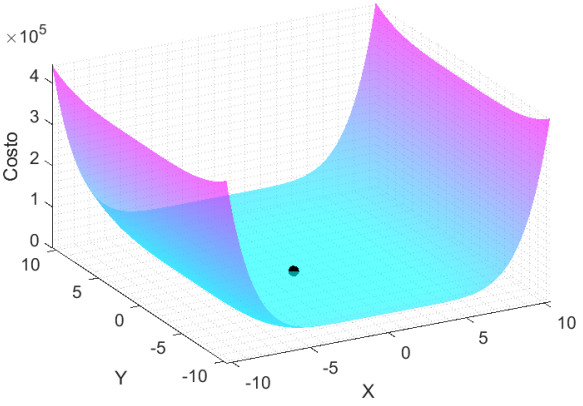
13.2.11. Schaffer F6 o Schaffer No. 2



$$f(\mathbf{x}) = 0.5 + \frac{\sin^2\left(x_1^2 - x_2^2\right) - 0.5}{\left[1 + 0.001\left(x_1^2 + x_2^2\right)\right]^2} \tag{58}$$

Figura 97: Visualización y ecuación de la función de costo Schaffer F6 o Schaffer No. 2. Mínimo: (0,0).

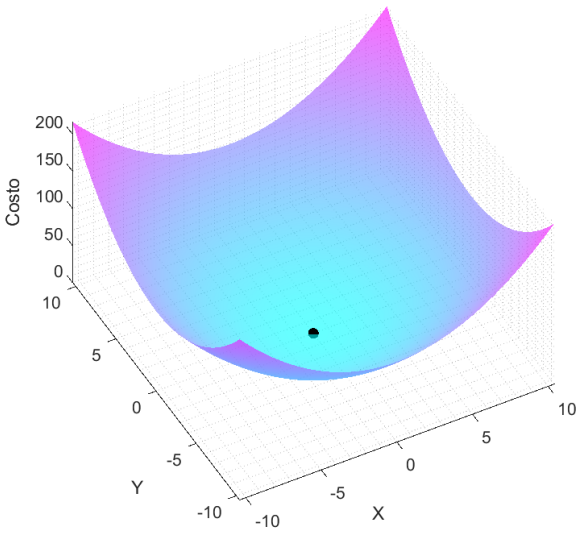
13.2.12. *Six-Hump Camel*



$$f(\mathbf{x}) = \left(4 - 2.1x_1^2 + \frac{x_1^4}{3}\right)x_1^2 + x_1x_2 + \left(-4 + 4x_2^2\right)x_2^2 \tag{59}$$

Figura 98: Visualización y ecuación de la función de costo *Six-Hump Camel*. Múltiples mínimos: (0.0898,-0.7126) y (-0.0898,0.7126).

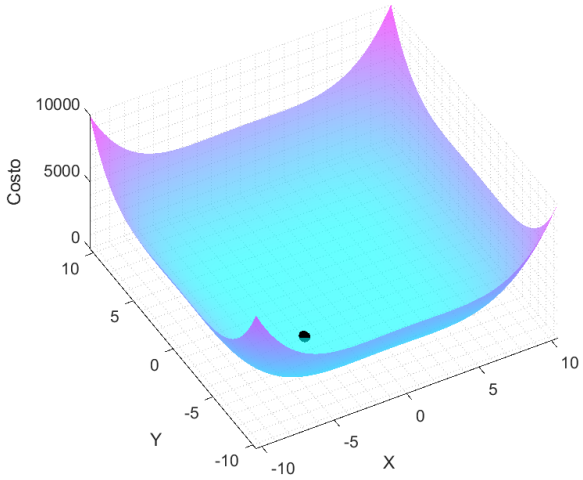
13.2.13. Esfera o Paraboloide



$$f(\mathbf{x}) = \sum_{i=1}^{d} x_i^2 \tag{60}$$

Figura 99: Visualización y ecuación de la función de costo Esfera o Paraboloide. Mínimo: (0,0).

13.2.14. Styblinski-Tang



$$f(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^{d} \left(x_i^4 - 16x_i^2 + 5x_i \right) \tag{61}$$

Figura 100: Visualización y ecuación de la función de costo Styblinski-Tang. Mínimo: (-2.903534,-2.903534).

13.2.15. Artificial Potential Fields (APF)

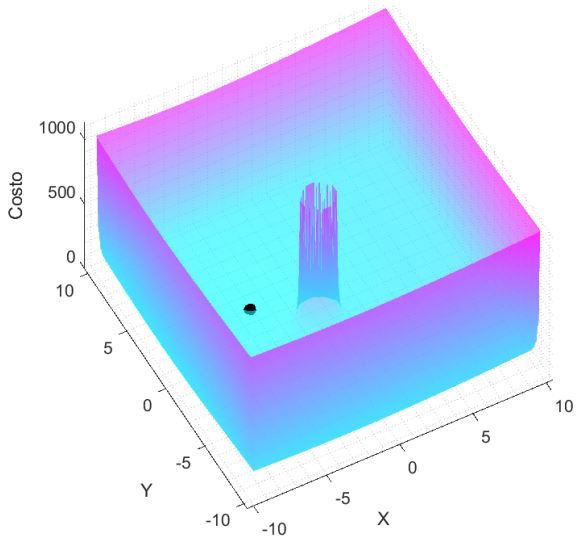


Figura 101: Visualización de la función de costo *Artificial Potential Fields* (APF). Mínimo: (-3,3).

13.3. Matlab: Error durante entrenamiento de redes neuronales

Durante el proceso de ajuste de hiper parámetros de las redes neuronales (ver sección 7.7), se alcanzó un punto en el que Matlab comenzó a detener consistentemente el proceso de entrenamiento y retornaba el siguiente error.

```
% Se entrena y guarda la red neuronal.
LSTM = trainNetwork(NetInput,NetOutput,layers,options);
```

```
Warning: An unexpected error occurred during CUDA execution. The CUDA error was:
CUDA_ERROR_LAUNCH_FAILED
Warning: An unexpected error occurred during CUDA execution. The CUDA error was:
CUDA_ERROR_LAUNCH_FAILED
Warning: An unexpected error occurred during CUDA execution. The CUDA error was:
CUDA_ERROR_LAUNCH_FAILED
```

Figura 102: Mensaje de error de Matlab durante el entrenamiento de las redes neuronales.

Luego de investigar [39], se descubrió que el error se debía a que la tarjeta gráfica utilizada para entrenar a la red era simultáneamente utilizada para producir la salida de video del sistema operativo. Cuando se intenta procesar los datos de entrenamiento, pero la tarjeta gráfica se encuentra "ocupada renderizando video", la librería CUDA (encargada de procesar los datos de entrenamiento) espera cierto tiempo a que esta se desocupe. Si luego de cierto tiempo, la tarjeta continúa ocupada, la librería simplemente retorna un error y finaliza el proceso de entrenamiento.

En Windows 10, este error puede solucionarse agregando uno de dos registros al sistema operativo:

- "TdrDelay": Tiempo que la librería CUDA espera antes de retornar el error. Su valor por defecto es 2s. Si este se incrementa a un número alto (30 por ejemplo), el tiempo que la tarjeta espera para que se desocupe la tarjeta gráfica se extiende. Esto retirará el error luego de un reinicio, pero puede causar que el proceso de entrenamiento se extienda.

- "TdrLevel": Variable de control para el sistema que detecta si la tarjeta gráfica está ocupada. Al igualar *TdrLevel* a 0, se desactiva el sistema encargado de esperar para que la tarjeta gráfica se desocupe, virtualmente dando prioridad al procesamiento de datos por sobre el procesamiento de video. Esto retirará las advertencias luego de un reinicio, pero a cambio, puede llegar a causar que la pantalla deje de actualizarse durante el proceso de entrenamiento, dando la impresión que el sistema se ha congelado.

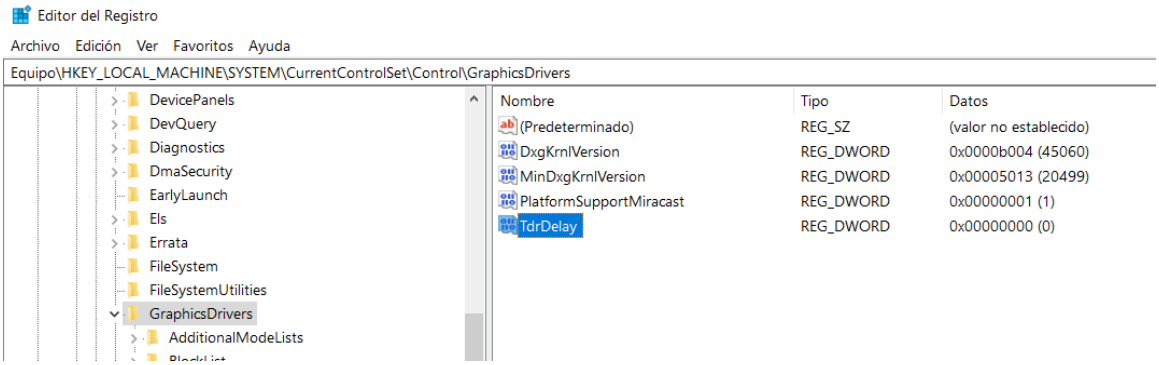


Figura 103: Directorio donde se deben crear los registros *TdrDelay* y *TdrLevel*

CAPÍTULO 14

Glosario

Batch Size: Número de muestras que la red neuronal es capaz de procesar en cada iteración del algoritmo de entrenamiento. Cuando se habla de un *Mini Batch Size*, se hace referencia al hecho que los *batches* alimentados a la red contienen menos muestras que el número total de muestras existente en la totalidad de los datos de entrenamiento [22]. 18, 49

Datos de validación: Para determinar si una red está generando predicciones adecuadas sobre los datos de entrenamiento, se puede emplear un conjunto de datos ajeno a las muestras de entrenamiento, para "validar" si el modelo entrenado produce resultados aceptables incluso utilizando muestras nunca antes vistas por la red [18]. 18, 48

Descuentos: 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". 30

Epoch: Cantidad de veces que la red neuronal procesa los datos de entrenamiento en su totalidad [23]. 18

Frecuencia de validación: Cada cuantas iteraciones se valida el modelo entrenado utilizando los datos de validación. 18

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". 29

Iteraciones (aprendizaje profundo): Número de *batches* que se deben alimentar a la red neuronal para completar una *Epoch*. 18

Learning Rate Drop Factor: Factor en el que se reduce el *learning rate* luego del *learning rate drop period*. 18

Learning Rate Drop Period: Cada cuantas épocas el algoritmo disminuye el *learning rate* en cierto factor. 18

Learning Rate: Parámetro que controla el cambio que experimentan las constantes de la red neuronal en función de su error. Valores muy bajos pueden causar que el proceso de optimización se atore en valles de la función de costo, mientras que valores muy altos pueden llevar a la convergencia temprana del modelo empleando constantes subóptimas. Cuando se establece un *Initial Learning Rate* se hace referencia a un valor inicial para el *learning rate* que posteriormente será alterado durante el entrenamiento para auxiliar en el proceso de minimización del error [24]. 18, 49

Overfitting: Fenómeno en el que la red neuronal aprende a imitar perfectamente los datos de entrada, perdiendo la capacidad de generalizar su comportamiento ante la presencia de nuevos datos [18]. 18