**Script Presentación**

Buenas noches. El día de hoy les presentaré la tesis “Reinforcement Learning y Deep Learning en aplicaciones de robótica de enjambre” desarrollada por su servidor, Eduardo Santizo.

Resumen (1 minuto. Real: 1 m 17 s)

Como ya conocen, en la Universidad del Valle contamos con el proyecto Robotat, una iniciativa similar al “Robotarium” de Georgia Tech. Este proyecto requiere de diferentes partes, desde el diseño físico de los robots que navegarán la mesa de trabajo, hasta el protocolo de comunicación utilizado por los mismos. Uno de los elementos más importantes, consiste del algoritmo que utilizarán los robots para moverse. En el caso de la universidad, se decidió utilizar el algoritmo de “Particle Swarm Optimization” o PSO como base para la movilidad de los robots, con los estudiantes Aldo Nadalini y Juan Pablo Cahueque adaptando el mismo para su utilización con robots reales. Aldo diseñó diferentes controladores para acoplar el movimiento de las partículas a los robots físicos y Juan Pablo modificó estos hallazgos para que los robots navegaran alrededor de obstáculos.

En el presente trabajo se emplearon diferentes técnicas de aprendizaje profundo y reforzado para mejorar los aportes propuestos por mis dos predecesores. Específicamente se proponen dos soluciones:

1. Un seleccionador automático de parámetros para el algoritmo PSO basado en redes neuronales recurrentes.
2. Y…. una alternativa al método de navegación de Juan Pablo basada en “Gridworld”, un ejemplo clásico en aprendizaje reforzado.

Objetivos (2 minutos. Real: 1 m 18 s)

La idea para ambas propuestas proviene del objetivo general del proyecto:

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

Así como de los objetivos específicos asociados:

* Combinar los trabajos sobre Artificial Potential Fields y PSO desarrollados en fases previas del proyecto Robotat.
* Construir una colección de datos de entrenamiento y validación para usar en métodos de aprendizaje reforzado y profundo, 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.

De todo esto, es importante resaltar dos términos recurrentes hasta el momento: “Aprendizaje profundo” y “aprendizaje reforzado”. Como se observó previamente, cada tipo de “machine learning” se empleó para una propuesta de solución distinta, por lo que a partir de este punto la presentación se dividirá en dos secciones: Una dedicada a la explicación del seleccionador automático de parámetros PSO y otra dedicada a la explicación del planificador de trayectorias.

*PSO Tuner*

Iniciamos con el seleccionador automático o PSO Tuner.

Introducción (5 minutos. Real: 3 m)

El algoritmo PSO consiste de un algoritmo de optimización estocástico basado en la exploración física de una superficie de costo por medio de “partículas”, elementos que tienen una posición y velocidad asociadas. Las partículas se colocan aleatoriamente dentro de una región de búsqueda y luego se deja que exploren la superficie de costo para encontrar el mínimo global. El “costo” de una partícula se obtiene al evaluar las coordenadas de su posición actual en la función de costo.

Durante su movimiento las partículas toman nota de dos cantidades: El “personal best” o la memoria personal de cada partícula de la posición con el costo más bajo hasta el momento; y el “global best”, la memoria de la posición con menor costo encontrada por todo el enjambre hasta el momento. En la forma base del algoritmo la posición, velocidad, “personal best” y “global best” se relacionan a través de estas dos ecuaciones.

El problema del PSO base es que la velocidad de las partículas no está restringida, por lo que existen situaciones en las que el algoritmo diverge. Para solucionar esto, se decidió agregar una constante adicional a la “ecuación de actualización de la velocidad” para evitar el crecimiento desmesurado de la misma. Dado que esta constante parecía modificar la “viscosidad” del medio en el que se movían las partículas, se le decidió llamar “inercia” a la constante “omega”. (EXPLICAR MÁS LA FIGURA)

El PSO con inercia presenta una gran mejora de rendimiento, pero siguen existiendo casos atípicos en los que aún diverge. Entonces, Shi y Eberhart decidieron analizar la ecuación de velocidad como un sistema dinámico y obtuvieron un conjunto de ecuaciones auxiliares que al acoplarlas al algoritmo PSO aseguraban la convergencia del mismo.

Comúnmente el “PSO con constricción” se le considera la versión “canónica” del algoritmo por lo que no se recomienda utilizarlo en conjunción con otras variaciones, no obstante, en 2019, Aldo Nadalini decidió combinar las constantes de constricción con el coeficiente de inercia y obtuvo resultados favorables. Sin embargo, uno de los aspectos que le tomó más tiempo, fue el encontrar el valor óptimo de las constantes Phi 1, Phi 2 y omega.

Para Phi 1 y Phi 2 podemos utilizar la restricción establecida por Clerc como una guía para asegurar la convergencia, mientras que para omega podemos basarnos en diferentes metodologías propuestas en papers para calcular la inercia. No obstante, como se mencionó antes, todas estas medidas solamente consisten de guías, no de una respuesta definitiva a los valores que deben tomar estos parámetros.

De aquí surge la idea del “PSO Tuner”, un seleccionador automático para el valor de estos parámetros.

Diseño Experimental (5 minutos. Real: 4 m 51 s)

El concepto del PSO Tuner es relativamente simple. A una red neuronal se le alimentan 4 métricas que cuantifican el desarrollo actual del enjambre y como resultado, esta retorna el valor para los parámetros “omega”, “phi 1” y “phi 2”, así como una predicción del número de iteraciones que le tomará al algoritmo converger.

Para permitir que esta metodología pueda ser utilizada en enjambres de todo tipo, las métricas elegidas deben provenir de una función que tome la información de todas las partículas (o sea su velocidad y/o posición) y la comprima en un único escalar representativo del enjambre, no importando el tamaño del enjambre o la dimensionalidad del problema.

Luego de diferentes pruebas, se eligieron 4 métricas, todas normalizadas para encontrarse en el rango entre 0 y 1 (mostrar la fórmula para cada una):

* Coherencia: Medida de que tan “en unísono” se mueven las partículas. Un valor alto se puede obtener tanto si las partículas convergen, como si todas se mueven con la misma velocidad y dirección.
* Promedio de la distancia promedio entre partículas: Esta métrica se obtiene al calcular la distancia euclideana promedio entre cada partícula y todas las demás. Las distancias promedio de todas las partículas luego se promedian para obtener un único valor.
* Desviación estándar promedio: Promedio de la desviación estándar de la posición de las partículas en cada dimensión.
* Distancia a meta: Distancia euclideana del “global best” al mínimo global de la función de costo.

Ahora bien, la red neuronal que recibe estas métricas se considera recurrente ya que emplea “neuronas recurrentes”. La principal característica de estas neuronas (y por consiguiente de estas redes), es que estas son capaces de “recordar” información del pasado, por lo que son muy útiles para procesar datos secuenciales.

Dentro del “Deep Learning Toolbox”, Matlab ofrece tres tipos de neurona recurrente: GRU, LSTM y BiLSTM. De estas tres, las neuronas GRU son las neuronas más simples, ya que están diseñadas para ser utilizadas en mayores números. Le sigue la neurona LSTM, la cual consiste de una versión generalizada y más poderosa de la neurona GRU. Finalmente, se encuentra la neurona BiLSTM, la cual consiste de dos neuronas LSTM acopladas, permitiendo que la neurona no solo tome en cuenta la información del pasado, sino también la información proveniente de una estimación del futuro cercano.

Se probaron tres arquitecturas diferentes para la red neuronal, todas variaciones de una arquitectura base donde se sustituía la capa de neuronas recurrentes por cada uno de los tipos de neurona previamente mencionados. La única excepción a esta regla fue la red BiLSTM, la cual tenía una arquitectura ligeramente distinta. La idea de mantener una estructura de red “similar”, mientras se variaba el tipo de neurona, era determinar cuál era el mejor tipo de neurona para la aplicación dada.

Para la generación de datos de entrenamiento se corrió una versión tradicional del algoritmo PSO y durante cada iteración del algoritmo, se calculaban las métricas que describen al enjambre y se tomaba nota del valor de Phi 1, Phi 2 y omega. Luego de converger, se contará con 2 matrices con tantas columnas como iteraciones: Una de 4 filas para las métricas o inputs y otra de 3 filas para los parámetros PSO u outputs de la red. En este punto también se agrega una fila adicional a la matriz de outputs, la cual consiste de copias del número de iteraciones requeridas por el algoritmo para converger.

Las 2 matrices obtenidas consisten de una única muestra. Para obtener más muestras, se continuaron realizando corridas del algoritmo variando factores como la restricción, tipo de inercia y función de costo. Dado que se realizaron 100 simulaciones por cada permutación de estos factores, en total se generaron 7700 secuencias o muestras de entrenamiento (Mostrar ecuación).

Luego de entrenar a las redes, se acopló el PSO Tuner a un algoritmo PSO tradicional de 1000 partículas, dándole control sobre el valor de “omega”, “Phi 1” y “Phi 2”. Específicamente se decidió realizar 200 corridas del algoritmo por cada una de las 3 funciones de costo minimizadas: Griewank, Schaffer F6 y APF.

Para medir la efectividad del PSO Tuner se decidió medir 4 factores:

* Velocidad de convergencia: Número de iteraciones que le tomó a cada tipo de PSO para converger. En este caso se consideró que las partículas convergían si las mismas se detenían en un punto o si alcanzaban el número máximo de iteraciones. En el eje X tenemos los tipos de PSO y en el Y el número de iteraciones.
* Precisión de convergencia: Porcentaje de veces que el algoritmo consiguió llegar al mínimo global de la función de costo. En este caso los ejes están invertidos: En el eje X tenemos el porcentaje de éxito y en Y cada tipo de PSO.
* Tiempo de computación: Tiempo que le tomaba a cada tipo de red producir sus predicciones para “Phi 1”, “Phi 2” y “Omega”. Sobre el eje Y tenemos el tiempo en milisegundos y sobre el X cada tipo de PSO.
* Dispersión de enjambre: Promedio de la posición de las partículas sobre cada eje. Cada columna representa un eje diferente, con la columna azul siendo el eje X y la roja el eje Y. Las líneas sólidas consisten del promedio de la posición y la “sombra” consiste de la dispersión o desviación estándar. En todas las gráficas el eje Y tiene los valores para la coordenada de la columna en la que se encuentra (X o Y) y el eje X consiste del tiempo.

Como se pudo observar, en todas las gráficas (exceptuando el tiempo de computación) se compararon los diferentes tipos de PSO Tuner con el algoritmo PSO tradicional (PSO Tuner Off).

Resultados (3.5 minutos)

Al analizar los resultados obtenidos, se pudo evidenciar que el PSO Tuner con neuronas BiLSTM consistía de la red con mejor rendimiento de entre las tres arquitecturas probadas, incluso llegando a superar al algoritmo PSO tradicional. Esta red permitió:

* Acelerar el tiempo de convergencia
* Incrementar la precisión del algoritmo
* Y suavizar el movimiento de las partículas

De hecho, esta red, a pesar de contener neuronas BiLSTM (que, recordando, consiste de la neurona más compleja por estar conformada por dos neuronas LSMT), llegó a producir sus estimaciones de salida con la mayor rapidez de entre las tres arquitecturas probadas. Esto puede parecer contradictorio, pero se puede atribuir al uso de una arquitectura con menos capas y neuronas recurrentes, y, por lo tanto, con menos parámetros internos.

A pesar de esto, curiosamente, la red se tornó altamente dependiente del valor inicial de los parámetros “Phi 1”, “Phi 2” y “Omega”, copiando el comportamiento general de enjambre que se hubiera esperado observar en un algoritmo PSO tradicional con esos valores iniciales. En otras palabras, si se iniciaba el algoritmo con la restricción de “Inercia”, el PSO Tuner buscaba imitar el movimiento observado en un algoritmo PSO tradicional con este tipo de restricción. Lo más interesante de todo esto, es que, aunque esto podría llegar a considerarse una instancia de “overfitting”, la red neuronal en realidad aprendió a producir estas tendencias de movimiento a partir de cambios dinámicos al valor de sus parámetros (como se puede observar en esta gráfica, donde las tendencias de cada parámetro son totalmente diferentes).

De entre todos los tipos de valor inicial para “Phi 1”, “Phi 2” y “Omega”, se observó que iniciar con los valores correspondientes a una restricción por inercia exponencial decreciente, retornaba los mejores resultados (son los resultados previamente observados), incluso adquiriendo la capacidad de “recuperarse” de convergencias tempranas en mínimos locales.

El único caso donde este no fue el caso, fue cuando el algoritmo contaba con un número reducido de partículas. Al realizar pruebas con apenas 10 partículas, el método de restricción por constricción retornó los mejores resultados, mejorando significativamente la precisión del algoritmo al incrementarla a casi el 80% a comparación de los demás métodos. Todo esto se realizó en la función “Griewank”, una difícil de optimizar por su gran cantidad de mínimos.

Recomendaciones

Como le salga joven.

*Planificador de Trayectorias*

Ahora pasaremos a la segunda parte, el planificador de trayectorias basado en aprendizaje reforzado.

Introducción (5 minutos)

En su tesis, Aldo Nadalini fue capaz de acoplar el movimiento de las partículas del algoritmo PSO al movimiento de robots reales. La desventaja de su solución, era que los robots se desplazaban en línea recta hacia la meta, ignorando si existían obstáculos en el camino. Para solucionar esto, Juan Pablo Cahueque propuso la utilización de “Artificial Potential Fields” o APFs como una sustitución para las funciones de costo que empleaba Aldo en su tesis.

Un APF consiste de una función de costo personalizada, la cual presenta su mínimo en el punto al que se desea llegar y “montañas” de alto infinito en las regiones que se desean esquivar (obstáculos). Al “liberar” las partículas del algoritmo PSO en este tipo de funciones, las partículas tenderán a evitar las montañas de costo alto, mientras se dirigen hacia el punto de costo mínimo o la meta. No obstante, cuando se introdujeron robots reales en la simulación, a los mismos les tomaba mucho tiempo llegar a la meta y en ocasiones se chocaban con los obstáculos presentes en el camino.

Debido a esto, se decidió implementar una alternativa a este método de esquivado de obstáculos utilizando Reinforcement Learning. Inicialmente esto se hubiera conseguido modificando los parámetros del algoritmo PSO, sin embargo, se consideró que esto únicamente sobre complicaría la implementación, ya que resultaría mucho más práctico controlar directamente a los robots.

Entonces, para que la solución propuesta consistiera de una alternativa a los aportes de Juan Pablo, se buscó cumplir con los mismos supuestos que él empleó:

* En primer lugar, se conoce “a priori” la forma del entorno
* La solución implementada es de carácter “offline” o se procesa previo a iniciar el movimiento de los robots.
* Y la solución puede utilizarse en números variables de robots sin mayor problema.

Para cumplir con todos estos requisitos, se decidió emplear una versión modificada del ejemplo de aprendizaje reforzado “gridworld”. En este problema, se cuenta con un espacio cuadriculado a través del cual se desea navegar. Dentro de este espacio existen “celdas meta” y “celdas obstáculo”. Un agente (o la entidad que aprende en el algoritmo de aprendizaje reforzado) debe desplazarse dentro de dicha cuadrícula únicamente moviéndose en cuatro direcciones: Arriba, abajo, izquierda o derecha. Si el agente alcanza la meta, se le recompensa, de lo contrario se le castiga. La idea es que el agente aprenda en base de estas recompensas cual es la dirección óptima de movimiento en cada celda.

Este problema puede fácilmente replantearse como una cadena de decisión de Markov, y solucionarse por medio del algoritmo de “iteración de política”. Dentro de este planteamiento, las diferentes celdas del entorno consisten del espacio de estados, mientras que las direcciones de movimiento consisten de las acciones. Para cada estado, todas las acciones tienen una probabilidad de llevarse a cabo. Al conjunto de todas estas probabilidades se le denomina “política”.

El algoritmo de “iteración de política” busca emplear un método eficiente para encontrar la política de acción óptima (o las mejores acciones a tomar en cada celda para maximizar la recompensa recibida). Este algoritmo se divide en dos etapas: “Evaluación de política” y “Mejora de política”.

En la primera etapa, conocida como “evaluación de política” se toma cada uno de los 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 a su vez se definen como “que tan bueno es ejecutar una acción estando en un estado”. Por ejemplo, en el estado 13, si el agente intenta subir este se chocará porque no hay celda a la cual moverse, entonces el valor de “subir” en el estado 13 es bajo. Por otro lado, si el agente baja, este se moverá sin problemas al estado 14, por lo que el valor de “bajar” es más alto.

La fórmula para obtener el “valor de acción” es la siguiente

Asumiendo una probabilidad inicial uniforme para todas las direcciones y valores de estado iniciales iguales a 0, es posible calcular el valor para todas las acciones y seguido de esto, sumar estos valores para obtener el valor del estado. Si la diferencia entre el valor de estado actual y previo es menor a cierto umbral “theta”, se finaliza la etapa de evaluación. Se le llama evaluación porque se utiliza la política “pi” para calcular o evaluar el valor de los estados.

Luego se pasa a la segunda etapa, la “mejora de política”. La diferencia con la etapa de “evaluación” es que, en esta etapa, en lugar de sumar los valores de acción, se extrae la acción con el valor más alto para cada estado. A todas las acciones “máximas” se les asigna la misma probabilidad de ocurrencia, para dar forma a una nueva política. Si la nueva política es igual a la política previa, se considera que el algoritmo ha convergido en una “política óptima”. De lo contrario, se repite el proceso de evaluación y mejora. Luego de converger, la política óptima contendrá las mejores acciones a tomar en cada celda para que el agente pueda navegar correctamente el espacio cuadriculado.

Diseño Experimental (5 minutos)

Para el planificador de trayectorias se decidió modificar algunos aspectos de “gridworld” para adaptar más fácilmente las “acciones” a las capacidades de movilidad de los robots. Específicamente, se decidió duplicar el número de acciones capaces de ser ejecutadas, al permitir que el agente se desplazara diagonalmente.

Con este cambio, se debió también alterar la dinámica del espacio de estados. La dinámica en este caso, consiste de una función que toma la acción y estado actual, y a partir de estos determina la recompensa a recibir y el estado futuro al que se transicionará. En “gridworld” la dinámica es sumamente sencilla: Si el agente se desplaza en alguna dirección, este se moverá a la celda adyacente que queda en esa dirección. Si la celda que se ocupará consiste de un obstáculo, se detecta una colisión, el agente regresa a la celda en la que estaba y recibe un castigo o “recompensa negativa”. Si en lugar de esto, la celda a ocupar consiste de una “celda meta”, el agente se mueve a dicha celda y se le brinda una alta recompensa positiva. Si el estado al que se transiciona no consiste de una meta u obstáculo, entonces también se le da un pequeño castigo negativo para que el agente se “apresure a llegar a la meta”.

Todos estos elementos se mantienen para el planificador de trayectorias. La única diferencia, es que debido a que el agente puede desplazarse diagonalmente, existen nuevas formas de colisionar que no estaban cubiertas por la dinámica previa. La primera de estas, se puede ejemplificar en el caso en el que el agente cuenta con una “celda obstáculo” arriba y a su derecha. Si el agente se desplaza diagonalmente hacia arriba (mostrar flechita), bajo la dinámica previa esto consistiría de una transición válida. No obstante, no es muy difícil observar que, si un robot real trata de hacer este movimiento, el mismo se va a quedar atorado. Debido a esto, cualquier movimiento que cumpla con un esquema similar de obstáculos, se pasó a considerar una colisión y será castigado.

Ahora consideremos el caso donde ya no existe un obstáculo en la parte de arriba, y la política óptima dicta que el agente se debe mover diagonalmente hacia arriba. En este caso, un robot real si será capaz de alcanzar el estado presente en la diagonal, pero lo conseguirá “topando” la esquina del obstáculo. A este caso se le pasó a denominar “colisión parcial” y se le asignó un pequeño castigo para prevenir este hecho.

A continuación, se presenta una tabla con los valores para las diferentes recompensas utilizadas dentro de la dinámica.

Con la dinámica modificada lista, se debía extraer una representación simplificada del mapa a navegar en la forma de celdas. Para esto se toma la geometría del mapa a navegar, se subdivide el mismo en una cuadrícula de 20x20 y se escanea cada celda para determinar si esta contiene una meta u obstáculo. Al finalizar se genera una representación simplificada del mapa en la forma de celdas, la cual puede ser optimizada por el algoritmo de “iteración de política”.

Para generar una trayectoria a partir de la política, se toma la posición inicial de los robots y se coloca como el primer punto en las trayectorias. Se toma nota de la celda en la que se encuentran los robots y se sigue la acción óptima que sugiere la política. El siguiente punto en la trayectoria consistirá del centro de la celda que se encuentra en la dirección de la acción óptima. Se continúa este proceso de seguimiento de política hasta finalmente alcanzar la “celda meta”. Cuando se llega a esta, se coloca el último punto de la trayectoria en el punto meta y se permite que los robots sigan la trayectoria generada.

Resultados (3.5 minutos)

Cabe mencionar que, debido al carácter principalmente exploratorio de este método, los resultados presentados a continuación no cuentan con un análisis estadístico riguroso que los respalde, pero si pueden ser analizados cualitativamente. Al observarlos desde esta perspectiva, los resultados obtenidos parecen muy prometedores. Las animaciones presentan la forma de las trayectorias “teóricas” generadas por la iteración de política, con un grupo de 3 robots (con controladores LQR) siguiendo dichas trayectorias. En todas estas animaciones, se puede observar como el planificador fue capaz de generar trayectorias exitosas y eficientes a lo largo de diferentes tipos de obstáculo, incluyendo:

* Los diferentes “casos” u obstáculos propuestos por Juan Pablo Cahueque (A, B y C)
* Variaciones de los mapas creados por Jabandzic en su literatura científica.
* Y algunos ejemplos más complejos creados por mí para mostrar las capacidades del planificador.

En todos estos casos se hizo evidente que, siempre y cuando la cuadrícula fuera capaz de capturar la geometría del mapa, el planificador sería capaz de crear una trayectoria exitosa. En los casos en los que el planificador no fue capaz de conseguir esto, este produjo trayectorias “inconclusas” o “cíclicas”, trayectorias donde la política guía a los robots a uno o más puntos distintos de la meta. Esto se pudo atribuir a dos factores: Desmotivación y baja resolución de cuadrícula.

* Desmotivación: Esto fue observado cuando existían regiones del mapa separadas por una gran distancia. En estos casos, el agente en las celdas lejanas parecía “perder las ganas de buscar la meta”, al generar una política que no llevaba a ningún lado. Para solucionar esto, se incrementó la recompensa por llegar a la meta, mejorando la capacidad del agente de “ver a futuro”.
* Baja resolución de cuadrícula: Cuando las celdas son demasiado grandes como para distinguir entre dos obstáculos muy cercanos, el escaneo de cuadrícula colocará una “pared” donde en realidad existe espacio libre, bloqueando el camino a la meta. Esto se puede solucionar disminuyendo el tamaño de celda pero trae consigo un incremento en el tiempo de computación.

En conclusión, se podría indicar que este método es muy efectivo, aunque presenta debilidades claras, en particular en términos de su carácter “automático”, ya que funciona muy bien para muchos casos, pero en algunos requiere de intervención por parte del usuario.

Recomendaciones

Como le salga joven.

Conclusiones (5 minutos)

Entonces, con todo lo anterior en mente, regresemos a los objetivos de la investigación para observar si estos se cumplieron, iniciando con el objetivo 1.

¿Se optimizó la selección de parámetros en algoritmos de inteligencia de enjambre mediante el uso de Reinforcement learning y Deep learning?

* Si. El PSO Tuner fue capaz de mejorar el rendimiento del algoritmo PSO al auxiliar en la selección de los parámetros “Phi 1”, “Phi 2” y “Omega”.

¿Se combinaron los trabajos sobre APF’s y PSO desarrollados en fases previas del proyecto Robotat?

* Esto ya había sido cumplido parcialmente por Juan Pablo Cahueque durante su tesis, no obstante, la unión de ambos elementos forma parte de la denominada “Swarm Robotics Toolbox”, el último elemento que se implementó como parte de esta tesis y que dio vida a todas las figuras, pruebas, animaciones y simulaciones que observaron previamente. Más de este “Toolbox” se presentará en la sección de “presentación de prototipos”.

¿Se construyó 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?

* Si. Se emplearon 7700 secuencias de entrenamiento y 2300 de validación para entrenar al PSO Tuner, todas generadas a partir de múltiples corridas del algoritmo PSO.

Y finalmente, ¿Se desarrolló e implementó un algoritmo que determine automáticamente los mejores parámetros para los algoritmos de robótica de enjambre?

* Si. Los excelentes resultados obtenidos por medio del PSO Tuner (basado en redes BiLSTM) demuestran que este consiste de un método eficaz para obtener los mejores parámetros para el algoritmo PSO.

Ahora bien, como se pudo observar en las conclusiones anteriores, el uso de Deep Learning fue suficiente para cumplir con todos los objetivos propuestos en la investigación. Dado que en todos estos también se hacía mención al Reinforcement Learning, se decidió emplear este tipo de “Machine Learning” en un nuevo aporte para el proyecto Robotat: El planificador de trayectorias. Dado que esta propuesta consiste de una alternativa viable para los aportes previos y consiste de un excelente fundamento para futuras investigaciones, considero que se podría establecer que se cumplió con un objetivo adicional: Explorar y emplear Reinforcement learning para aplicaciones de robótica de enjambre.

Muchas gracias.