UNIVERSIDADE DA CORUÑA

Escola Politécnica Superior

Trabajo Fin de Máster

CURSO 2018/19

ESTUDIO DEL APRENDIZAJE EN TIEMPO REAL DE MODELOS DE UTILIDAD EN ROBÓTICA COGNITIVA

Máster en Ingeniería Industrial

ALUMNO

Álvaro Fernández de la Torre

TUTORES

Francisco Javier Bellas Bouza

Alejandro Romero Montero

FECHA

SEPTIEMBRE 2019

TÍTULO Y RESUMEN

ESTUDIO DEL APRENDIZAJE EN TIEMPO REAL DE MODELOS DE UTILIDAD EN ROBÓTICA COGNITIVA

Este Trabajo Fin de Máster (TFM) se enmarca dentro del proyecto europeo de investigación DREAM que se lleva a cabo en el Grupo Integrado de Ingeniería (GII) de la UDC. En dicho proyecto, el objetivo es dotar a los robots autónomos de modelos cognitivos inspirados en el ser desarrollo intelectual de los humanos, de modo que puedan aprender por sí mismos en entornos reales a lo largo de grandes periodos de tiempo.

Uno de los componentes fundamentales de este proyecto es el sistema de motivación de los robots, cuya principal función es la obtención automática de modelos de utilidad que permitan establecer las recompensas esperadas en los diferentes estados sensoriales. Hasta el momento, en el marco del DREAM [1] se ha abordado el aprendizaje de estos modelos utilizando representaciones unidimensionales (denominadas SURs) y multidimensionales (denominadas Value Functions, VF) de manera aislada, mostrando que cada una de ellas es adecuada para cierto tipo de problemas. En este TFM se busca desarrollar un esquema operativo de aprendizaje on-line que gestione el uso de estos dos tipos de modelos. Para ello, se utilizará un experimento representativo del campo de la robótica cognitiva en un entorno simulado.

ESTUDO DO APRENDIZAXE EN TEMPO REAL DE MODELOS DE UTILIDADE EN ROBÓTICA COGNITIVA

Este Traballo Fin de Máster (TFM) enmarcase dentro do proxecto europeo de investigación DREAM que se leva a cabo no Grupo Integrado de Enxeñaría (GIE) de la UDC. En dito proxecto, obxectivo fundamental é dotar aos robots autónomos de modelos congnitivos inspirados no ser desenvolto intelectual dos humáns, de modo que poidan aprender por si mesmos en entornos reais ao longo de grandes periodos de tempo.

Un dos compoñentes fundamentais de este proxecto é o sistema de motivación dos robots, cuxa principal función é a obtención automática de modelos de utilidade que permitan establecer as recompensas esperadas nos diferentes estados sensoriais. Ata o momento, no marco do DREAM abordáronse o aprendizaxe destes modelos utilizando representacións unidimensionais (denominadas SURs) e multidimensionais (denominadas Value Funtions, VF) de maneira illada, mostrando que cada unha de elas é adecuada para certo tipo de problemas. Neste TFM buscase desenvolver un esquema operativo de aprendizaxe on-line que xestione o uso destes dous tipos de modelos. Para isto, utilizarase un experimento representativo no campo da robótica cognitiva nunha contorna simulada.

STUDY OF REAL-TIME LEARNING OF UTILITY MODELS IN COGNITIVE ROBOTICS

This Master's Degree Project is part of the European project DREAM that is carried out in the Integrated Engineering Group for Engineering research of the UDC. In this project, the objective is to provide autonomous robots with cognitive models inspired by the intellectual development of humans, so they can learn for themselves in real environments over long periods of time.

One of the fundamental components of this project is the robot motivation system, which main function is the automatic obtaining of utility models that allow to establish the expected rewards in the different sensory states. So far, within the framework of DREAM [1] the learning of these models has been approached using one-dimensional (called SURs) and multidimensional (called Value Functions, VF) representations in isolation, showing that each of them is suitable for a certain kind of problems This project aims to develop an operational

on-line learning scheme that manages the use of these two types of models. To do this, a representative cognitive robotics experiment will be implemented and studied using a simulated environment.

UNIVERSIDADE DA CORUÑA

Escola Politécnica Superior

TRABAJO FIN DE MÁSTER

CURSO 2018/19

ESTUDIO DEL APRENDIZAJE EN TIEMPO REAL DE MODELOS DE UTILIDAD EN ROBÓTICA COGNITIVA

Máster en Ingeniería Industrial

Documento

MEMORIA

ÍNDICE DE MEMORIA

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 INTRODUCCIÓN

La investigación en robótica, tradicionalmente se ha centrado en las tareas de control y el procesamiento de las lecturas de los sensores, la planificación de caminos y el diseño de manipuladores. Por el contrario, la robótica cognitiva, que está sufriendo un boom en los últimos años, se centra en desarrollar robots que puedan desenvolverse solos en el medio que los rodea sin necesidad de ayuda externa. Este tipo de robótica busca como finalidad obtener robots que sean capaces de aprender por sí mismos en un entorno real, descubriendo estados que les proporcionan recompensa o utilidad y asociando a estos estados modelos, lo que permitirá encontrar la recompensa de manera intencionada. En la robótica cognitiva, el aprendizaje es abierto (open-ended), es decir, no existe un instante estipulado en el que se deja de aprender, o una meta que alcanzar, sino que el robot se supone que continuará aprendiendo toda su vida útil. Para lograrlo, la robótica cognitiva se basa en los modelos cognitivos humanos, y los robots no están dotados de programa de control, sino de una arquitectura cognitiva que contiene modelos y memorias que permiten el aprendizaje y el desarrollo autónomo.

La línea de investigación de robótica cognitiva es muy importante para el grupo de investigación del Grupo Integrado de Ingeniería (GII) de la UDC. Disponen de numerosas plataformas robóticas para la realización de experimentos entre los que se encuentran el Baxter, NAO, Robobo, etc. Muestra de su importancia es la presencia en un proyecto de investigación llamado DREAM. En él participan instituciones europeas de diferentes países y está financiado por el programa de investigación Horizonte 2020 de la Unión Europea. Este proyecto pretende incorporar los procesos del sueño, en los que el ser humano asimila los conceptos aprendidos durante el día, en la arquitectura de un robot autónomo. Esto permitirá un aprendizaje más eficiente, dejando la creación de modelos complejos para etapas ajenas a la operación en tiempo real.

La arquitectura cognitiva que se ha desarrollado en el proyecto DREAM se denomina MDB (Multilevel Darwnist Brain) [2], e incluye tres componentes fundamentales: un sistema de motivaciones, un sistema de aprendizaje y un sistema de Memoria a Largo Plazo.

Este TFM se centra en el sistema motivacional desarrollado dentro del proyecto DREAM, y que se ha denominado MotivEn (Motivational Engine). Este es el encargado de gestionar qué tipo de motivación rige el comportamiento robot. Los dos tipos de motivaciones que se utilizan en el MotivEn son la motivación intrínseca y la motivación extrínseca. La motivación intrínseca se utiliza para explorar inicialmente el entorno que rodea al robot, mientras que le extrínseca está asociada a una recompensa exterior, impuesta por el entorno, o por un usuario humano. Una vez que tenemos un conocimiento básico del entorno, los datos generados con la motivación intrínseca servirán para guiar a la motivación extrínseca a controlar el comportamiento robot. Para lograr de manera repetida la recompensa extrínseca, MotivEn aprende modelos de utilidad, que le permiten asociar una utilidad esperada a cada estado sensorial.

Hasta el momento, en MotivEn se han tratado de aprender 2 tipos de modelos de utilidad de forma no óptima para el uso en un mecanismo de aprendizaje open-ended, como es el MDB. Los primeros son modelos unidimensionales simples denominados SURs, que se aprenden en las primeras iteraciones, y los segundos son modelos más completos que proporcionan la utilidad predicha para cualquier estado sensorial, denominados Value Functions (VF). Hasta el momento, dentro de MotivEn, las SURs se aprendían en tiempo de ejecución (on-line), pero no así las VF, que se aprendían a partir de las SURs en un modo de operación off-line no compatible con el MDB. En este TFM se plantea el desarrollo de un método para aprender las VFs a partir de SURs de forma automática y on-line.

2 OBJETIVOS

El objetivo principal de este proyecto es desarrollar un esquema operativo para el aprendizaje on-line de modelos de utilidad en entornos realistas de robótica autónoma, que esté integrado dentro del mecanismo motivacional MotivEn desarrollado por el Grupo Integrado de Ingenería de la UDC. Para comprobar el funcionamiento de este esquema se utilizará un experimento en simulación representativo del campo de la robótica cognitiva en el cual se enmarca este trabajo. Para lograr este objetivo principal, se proponen los siguientes subobjetivos parciales:

- Analizar los sistemas de aprendizaje on-line de redes neuronales más establecidos en la actualidad, para seleccionar el más adecuado al aprendizaje de modelos de utilidad
- Comprender el funcionamiento del entorno de programación para aprendizaje máquina Tensor Flow, en concreto la API Keras, para la ejecución del algoritmo y tipo de red seleccionadas de manera sencilla.
- Estudiar el funcionamiento del mecanismo motivacional MotivEn, de cara a integrar el API Keras en él para el aprendizaje de las Value Functions.
- Desarrollar un método para el aprendizaje on-line de las Value Functions a partir de las SURs de manera automática.
- Estudiar el funcionamiento del simulador 3D Gazebo, así como del entorno de programación ROS.
- Implementar en Gazebo un problema representativo de robótica cognitiva sobre el cual probar el esquema desarrollado.

3 ESTRUCTURA DE LA MEMORIA

La estructura de la memoria se ha dividido en los siguientes apartados:

1. ***Introducción:*** breve descripción del ámbito y alcance de este Trabajo Fin de Master
2. ***Objetivos:*** establecimiento del objetivo principal y los subobjetivos planteados en este trabajo.
3. ***Estructura de la memoria:*** descripción de los distintos apartados desarrollados en esta memoria y un breve resumen del contenido de cada punto.
4. ***Fundamentos teóricos y fundamentos tecnológicos:*** descripción de las distintas metodologías utilizadas, tanto teóricas como tecnológicas, en el desarrollo de este proyecto.
5. ***Antecedentes:*** descripción del mecanismo cognitivo MDB, del sistema motivacional MotivEn, y justificación de la necesidad de llevar a cabo las innovaciones introducidas con la nueva metodología expuesta en este trabajo.
6. ***Requisitos de diseño:*** especificaciones proporcionadas por los investigadores del GII para la realización del trabajo.
7. ***Desarrollo:*** en este apartado se va a explicar la nueva metodología desarrollada en este proyecto para llevar a cabo el entrenamiento de las VF de forma On-line.
8. ***Pruebas:*** apartado en el que se van a llevar a cabo dos tipos de pruebas. Primero se van a realizar pruebas Off-line para predimensionar la red neuronal a implementar posteriormente en el Simulador 3D. Se explica el programa desarrollado y los resultados a los cuales se han llegado. Después se van a llevar a cabo las pruebas On-line en el Simulador 3D y se va a explicar el experimento desarrollado y los resultados obtenidos de las distintas pruebas realizadas.
9. ***Análisis de resultados:*** análisis de los resultados obtenidos en las distintas pruebas llevadas a cabo.
10. ***Conclusiones y trabajo futuro:*** Análisis de las conclusiones a las que se lleva después de estudiar los resultados obtenidos en cada prueba y exposición de los posibles trabajos futuros que se podrían llevar a cabo en el marco de este proyecto.

4 FUNDAMENTOS TEÓRICOS Y TECNOLÓGICOS

4.1 MotivEn

4.1.1 Introducción

Para poder entender los principios del sistema motivacional MotivEn, se considera necesario definir previamente una serie de conceptos básicos [3]:

- **Estado perceptivo o sensorial (P):** array con los valores de los sensores del robot. Los valores son números reales ya que MotivEn opera en dominios continuos.
- **Acción (A):** ejecución de un actuador.
- **Impulso (drive):** medida que nos informa de lo cerca, o lejos, que está el sistema de alcanzar un objetivo. Su salida suele ser un número real que aumenta con la distancia entre el espacio de estados y el objetivo.
- **Utilidad (u):** medida que nos proporciona el valor de la satisfacción del impulso en un estado perceptivo P(t). La utilidad está determinada por la interacción del sistema con el entorno y es desconocida inicialmente.
- **Objetivo innato:** un punto o área en el espacio perceptivo que el diseñador establece en el momento del diseño como un estado deseable para el robot.
- **Necesidades:** Conjunto de criterios, incorporados externamente por el diseñador, que el robot debe cumplir para considerarse que funciona de manera óptima.
- **Objetivo:** un estado perceptivo P(t) en el que el robot obtiene la utilidad máxima. Es un concepto clave en la arquitectura cognitiva, ya que el funcionamiento del robot durante toda la vida se debe a los objetivos que debe alcanzar.
- **Utilidad esperada:** valor de utilidad que se espera obtener con el modelo de utilidad para un punto determinado del espacio de estados.
- **Modelo (o función) de utilidad (UM):** función que proporciona la utilidad esperada para cualquier punto en el espacio de estados.
- **Value Function (VF):** tipo particular de modelo de utilidad que representa la utilidad esperada absoluta precisa para cualquier estado de percepción:

$$Eu(t + 1) = V\ F(P(t + 1))$$

- **Trayectoria:** secuencia de puntos del espacio de estados por lo que ha pasado el robot en un intervalo de tiempo determinado.
- **Traza:** trayectoria de longitud predeterminada en la cual está incluida los valores de utilidad para cada punto de la trayectoria, que se asignan una vez se haya alcanzado el objetivo.
- **Buffer de trazas (TB):** memoria que almacena un número determinado de trazas.

MotivEn es el sistema motivacional desarrollado en el marco del proyecto Europeo DREAM [1], y será explicado con detalle en los siguientes apartados. Es importante comentar que el sistema motivacional de un robot cognitivo tiene como objetivo realizar cuatro procesos principales: descubrir nuevos objetivos de forma autónoma, aprender las funciones de utilidad,

seleccionar la activación de los objetivos en el estado perceptivo actual, y evaluar los distintos estados candidatos en función de los objetivos activados.

4.1.2 Arquitectura del MotivEn

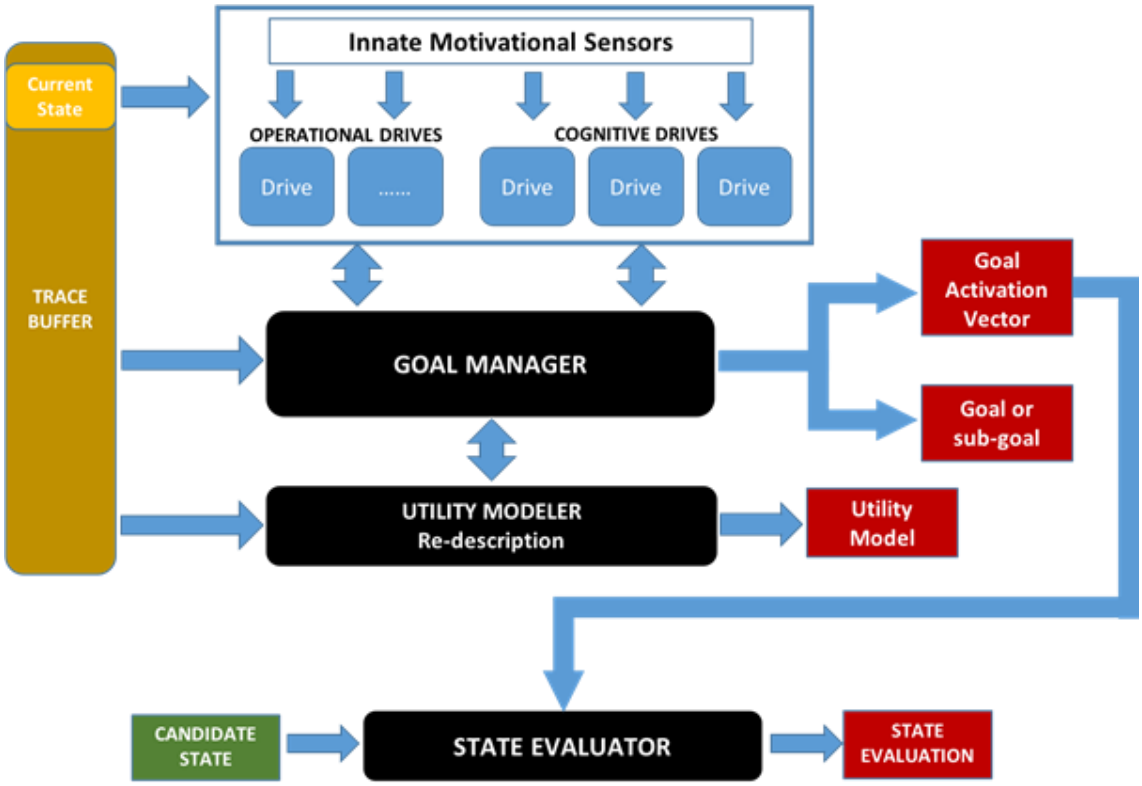


Figura 1: Diagrama motivacional DREAM.

En la **Figura [1]** se puede ver un esquema del motor motivacional en la arquitectura MDB, MotivEn [4]. Los bloques que realizan las principales operaciones del sistema son los de color negro, mientras que los bloques de color rojo son las salidas que se envían a la arquitectura cognitiva.

El bloque que encontramos de color amarillo a la izquierda del esquema es la Trace Buffer (Memoria de Trazas), en la que se encuentran las diversas trazas que se han generado durante la ejecución de los movimientos del robot en el intento de llegar a la meta (ya sea real o simulado). El primero de los dos bloques de color negro que se encuentran en el medio del esquema es el Goal Manager (Administrador de objetivos), que utiliza la Memoria de Trazas para definir nuevos objetivos, y el Estado Actual para establecer el vector de activación de cada objetivo. El segundo bloque es el Utility Modeler (Modelador de Utilidades), que utiliza las trazas para realizar el aprendizaje del modelo de utilidad. El State Evaluator (Evaluador del Estado) utiliza el vector de activación establecido por el Goal Manager para proporcionar la evaluación del Candidate State (Estado Candidato), denominado State Evaluation (Evaluación del Estado), que solicita el mecanismo cognitivo MDB.

4.1.3 Motivación intrínseca y extrínseca

El sistema motivacional, en el marco de robótica cognitiva, es una de las entidades que presentan mayor dificultad por la complejidad que conlleva establecer las estructuras de conocimiento que permiten relacionar las acciones que lleva a cabo el robot con las necesidades de obtener un objetivo. Este permite al robot explorar el espacio de estados con el propósito de establecer sus objetivos, los espacios sensoriales en el que robot quiere estar.

Se pueden diferenciar dos tipos de motivaciones para guiar las acciones que llevan a cabo el robot, motivación intrínseca y motivación extrínseca. Las primeras representan la exploración inicial que hace del robot del entorno que lo rodea en busca de estados sensoriales no visitados. Estas acciones permitirán al robot aprender como es el entorno que le rodea y como llegar hasta la meta, lo que a posteriori, permitirá guiar a la motivación extrínseca. Dentro de este tipo de motivación se encuentra la Novelty que ejecuta movimientos semi-aleatorios en busca de reducir la distancia de un objetivo. Las trazas obtenidas van a tener una gran ambigüedad, debido a la aleatoriedad en la ejecución de los movimientos a la hora de alcanzar el objetivo. Esto quiere decir que los puntos de la trayectoria que se incluyen en la traza no van a seguir un orden decreciente continuo del valor del sensor, si no que se van a alternar valores decrecientes y crecientes hasta llegar al objetivo. Un ejemplo simplificado de una traza obtenida mediante Novelty podría ser la mostrada en la **Figura [2]**, en la que se puede observar como el robot no llega de forma directa al objetivo. Con el paso del tiempo no se van a conseguir obtener ninguna mejora en la trayectoria que sigue el robot a la hora de llegar al objetivo porque con estos modelos no se realiza ningún aprendizaje.

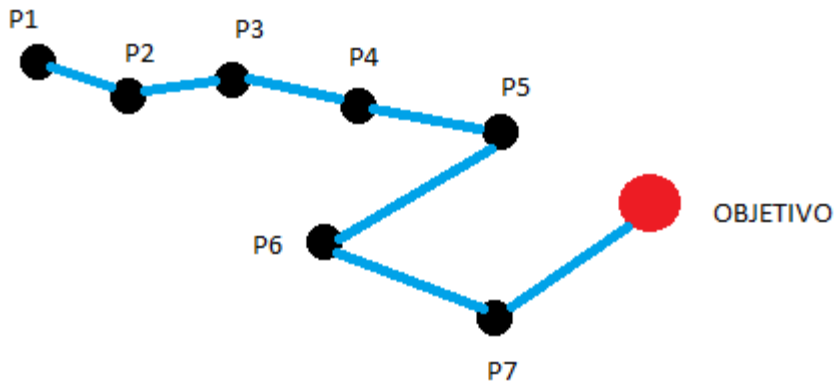


Figura 2: Traza Novelty.

Por otro lado, las motivaciones extrínsecas, al contrario que las motivaciones intrínsecas, están orientadas en la optimización de la forma de alcanzar un objetivo, los movimientos que ejecuta el robot son impulsados por el aprendizaje previo del entorno. Según vayan avanzando las simulaciones, con este tipo de motivaciones, se irán aprendiendo modelos mejores que permitan llegar cada vez de forma más rápida al objetivo. Por lo tanto, con este tipo de motivaciones se irán consiguiendo trazas cada vez con menores ambigüedades tras cada simulación, consiguiendo al final que todos los puntos de la trayectoria tengan un orden decreciente del sensor, lo que no se puede conseguir mediante motivación intrínseca. Esto es algo muy importante para realizar un buen entrenamiento de modelos de utilidad, sobre todo en el caso de modelos de utilidad muy complejos en los que es primordial entrenarlos con trazas con las menores ambigüedades posibles.

En concreto, el sistema motivacional MotivEn, se utiliza en la arquitectura cognitiva MDB para proporcionar la valoración de una serie de estados candidatos en un instante dado de tiempo. Es muy importante que se gestione de forma correcta la combinación de motivaciones intrínsecas y extrínsecas para elegir el estado final sensorial más adecuado, con el objetivo final de guiar al robot hacia la finalización de sus tareas. El Goal Activation Vector (Vector de Activación de Objetivo), que establece qué tipo de motivación se usa en cada instante de tiempo, es un elemento clave en MotivEn y, como resumen, funciona de la siguiente manera [10]:

1. En las etapas iniciales del aprendizaje, no se ha alcanzado ningún objetivo, y MotivEn utiliza principalmente una *Motivación Intrínseca Ciega* (Novelty), que guía

el comportamiento del robot hacia el descubrimiento de estados sensoriales no visitados que operan como un proceso intrínseco exploratorio.

2. Una vez que se alcanza un objetivo, se obtiene una traza, y ya se puede aprender un modelo de utilidad asociado, que se utilizará para evaluar los estados candidatos. Este modelo conduce a un objetivo y, por lo tanto, está guiado por una *motivación extrínseca*.
3. Dado que la fiabilidad del modelo de utilidad depende de cuánto se haya explorado su dominio para alcanzar el objetivo, se considera otro tipo de motivación intrínseca para fomentar la mejora de la certeza de las diferentes áreas del modelo de utilidad. Se llama *Motivación Intrínseca Basada en la Certeza*, y su objetivo es aumentar la confiabilidad del modelo de utilidad para todos los puntos dentro de su dominio.

Con todo esto la forma en la que se van a gestionar las trazas se podría definir de la siguiente manera. El conjunto de datos obtenido de la trayectoria del robot en su camino a alcanzar el objetivo se guarda, junto al valor de utilidad que se le otorga a cada punto de la trayectoria, a lo que se le llama traza. Siempre se van a empezar a guardar los últimos valores de la trayectoria, ya que van a ser los que menos ambigüedad van a tener, hasta llenar el tamaño de la traza predefinido. La mayor utilidad, de valor 1, se le otorgará al último punto de la trayectoria, en el que se alcanza el objetivo, y se irá disminuyendo el valor de utilidad a los siguientes puntos de la trayectoria de forma ordenada. Dependiendo del tamaño de traza otorgado por el diseñador entrarán mayor o menor número de puntos de la trayectoria en la traza. Una vez obtenida la traza esta se guardará dentro de la memoria de trazas de forma ordenada, sin mezclar los valores de las distintas trazas obtenidas. De este conjunto de trazas almacenadas, posteriormente, se seleccionará un grupo deseado para llevar a cabo el entrenamiento de la VF.

Un ejemplo podría ser el mostrado en la **Figura [3]**. Un elemento de desplaza por el espacio de estados teniendo una trayectoria formada por 10 puntos hasta llegar al objetivo. El diseñador ha considerado correcto almacenar un tamaño de traza de 5 valores por lo que esta quedaría de la siguiente forma observada en la **Tabla [1]**, pudiéndose ver en ella como se organiza la adjudicación de la utilidad correspondiente a cada punto de la trayectoria.

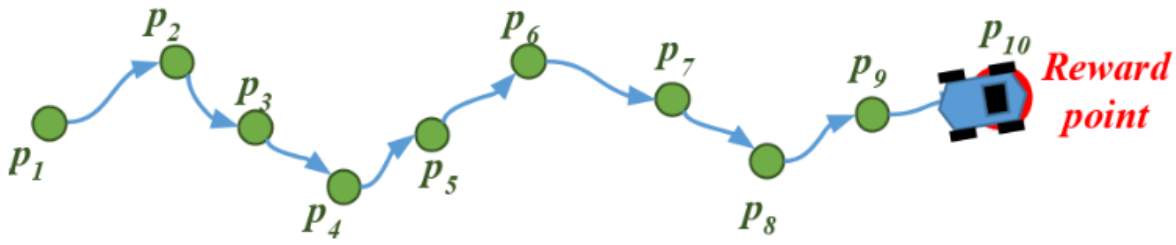


Figura 3: Puntos de la trayectoria del robot.

TRAYECTORIA	UTILIDAD
P10	1
P09	0.8
P08	0.6
P07	0.4
P06	0.2

Tabla 1: Valores de la traza

4.1.4 Modelos de utilidad

Una de las razones por las que resulta altamente complejo definir modelos de utilidad precisos se debe a que los dominios de actuación son continuos y de varias dimensiones. Por este motivo, los modelos implementados en diversas investigaciones se han basado en la consecución de objetivos locales centrados en áreas reducidas del espacio de estados. Por ejemplo, obteniendo una serie de subobjetivos que el robot pueda utilizar para desplazarse entre ellos hasta llegar hasta su objetivo final.

Un factor importante que va a influir a posteriori en la precisión de los modelos de utilidad es la cantidad de estados sensoriales visitados inicialmente. Si solo se ha explorado una pequeña parte del espacio de estados se tendrá un número muy pequeño de trazas, por lo que el modelo de utilidad tendrá una baja fiabilidad.

Dentro de los modelos de utilidad de pueden diferenciar dos tipos. El primer tipo son conocidos como Value Functions (VF), modelan todo el espacio perceptual de manera continua. El segundo tipo son modelos de utilidad más simples llamados SURs. Con este tipo de modelos de utilidad no es necesario conocer toda la sensorización del espacio de estados, sino que solo es necesario conocer un sensor, que es el que se va a tener en cuenta para configurar el mapa de estados de esa SUR. En los siguientes puntos se hablará más en profundidad de ambos tipos de modelos de utilidad.

4.1.5 Separable Utility Regions (SUR)

Separable Utility Regions (SUR) son una representación muy aproximada de las variaciones de utilidad en el espacio de estados.

Las SURs relacionan la dirección apropiada del movimiento del sistema en el espacio de estados con la variación de los valores de distancia dados por los sensores, para usarse esta relación como guía. Por lo tanto, se centra en utilizar la disminución de los valores de distancia como guía de la dirección apropiada de los movimientos a llevar a cabo por el robot. Además, es importante que la región del dominio donde se aplica la correlación deba definirse de modo que se pueda suponer que dentro de esa región la utilidad aumenta en la dirección de la correlación con un sensor en particular. Esta región de dominio se denomina área o mapa de certeza [3].

Una función de certeza es una función que asocia un valor de certeza con respecto a una función de utilidad dada a puntos en el espacio de estado. Esta función opera basándose en los estados visitados por el robot que se almacenan como tranzas en la memoria de trazas. La certeza de un punto se calcula en función de su distancia de los puntos que ya se han explorado.

Para crear los mapas de certeza son necesarias las trazas de los movimientos del robot. Se diferencian tres posibles tipos de trazas:

- Trazas exitosas (trazas-s): son las trazas creadas por el robot cuando alcanza el objetivo dentro de un área de certeza, estando bajo a la influencia de la motivación extrínseca.
- Trazas fallidas (trazas-f): son las trazas creadas por el robot cuando no alcanza el objetivo y se encuentra en un área de certeza, estando bajo la influencia de la motivación extrínseca.
- Trazas débiles (trazas-w): son las trazas creadas por el robot cuando alcanza el objetivo, pero no estaba bajo la influencia de la motivación extrínseca. Tienen menor influencia positiva que las trazas exitosas.

Cada traza obtenida por el robot se incorpora al mapa de certeza consiguiendo que el conjunto de regiones se actualice con el avance de las simulaciones. La adición de trazas

exitosas amplía el área de valores de alta certeza, mientras que la adición de trazas fallidas lo reduce.

Inicialmente el área abarca la mayor superficie posible porque aún no se conocen las trayectorias correctas para alcanzar el objetivo. Con el paso del tiempo este mapa se va reajustando en regiones que se correlacionan con los estados sensoriales explorados. Con esto se va a ir consiguiendo que cada vez se alcancen los objetivos de cada SUR de forma más óptima obteniendo trazas con menores ambigüedades.

Durante el proceso de aprendizaje, las áreas de certeza correspondientes a diferentes tendencias del sensor se codifican utilizando las trazas que se generan para que, después de algunas pruebas, solo el sensor correcto muestre una certeza positiva en el área correspondiente a su correlación, mientras que las otras devuelven un valor de cero debido a la presencia de trazas fallidas. De esta manera, el sistema convergerá a la activación de la SUR apropiada en aquellas regiones donde sea útil.

4.1.6 Value Functions (VF)

Las VF son un conjunto de funciones que permiten obtener un sistema que resuelva los problemas de forma óptima, en ellas el modelador de utilidad debe ocuparse de todo el espacio perceptual, al ser modelos multidimensionales. Este tipo de modelos de utilidad se modelan como una función que otorga la utilidad esperada para cada uno de los puntos del espacio de estados. Este tipo de funciones aprenden que utilidad asignar a las trazas generadas por el simulador.

Es posible emplear directamente las trazas obtenidas de la VF para entrenar inicialmente la misma, pero este tipo de metodología no es la más óptima. Lo que se realiza para el entrenamiento de estos modelos de utilidad es utilizar las trazas obtenidas mediante modelos de utilidad más robustos aprendidos previamente. Con estos datos la VF puede actuar proporcionando valores de utilidad precisos, seleccionando la acción que proporcione al sistema la mayor utilidad.

4.2 ROS

ROS (Robot Operating System) es un framework de código abierto para para el desarrollo de software para robots [5]. Proporciona los servicios que esperaría de un sistema operativo, incluida la abstracción de hardware, el control de dispositivos de bajo nivel, la implementación de la funcionalidad de uso común, el paso de mensajes entre procesos y la administración de paquetes. También proporciona herramientas y bibliotecas para obtener, crear, escribir y ejecutar código en varios equipos.

Figura 4: ROS.

El objetivo principal de ROS es respaldar la reutilización del código en la investigación y el desarrollo de la robótica. Los Nodes existentes en ROS permiten que puedas diseñar individualmente los archivos que se van a ejecutar y después poder unirlos cuando se están ejecutando. Estos procesos se pueden agrupar en Packages y Stacks, los cuales se pueden compartir y distribuir fácilmente. Este diseño permite tomar decisiones independientes sobre el desarrollo y la implementación. Todo esto muestra que los principales objetivos de ROS es tener un sistema que facilite compartir información y la colaboración entre usuarios.

A parte de estos objetivos principales que tiene ROS, compartir y colaborar, existen otros:

- Simplicidad en el código para que se pueda usar fácilmente con otros marcos de software de robots. Un ejemplo es lo fácil de integrar con otros software de robots como OpenRAVE, Orocos y Player.
- Bibliotecas ROS-agnósticas con interfaces funcionales limpias.
- La independencia del lenguaje permite que el marco de ROS sea fácil de implementar en cualquier lenguaje de programación moderno. Ya se ha implementado en Python, C ++ y Lisp, y tienen bibliotecas experimentales en Java y Lua.
- ROS tiene un marco de prueba integrado de unidad / integración llamado rostest que facilita la activación y desactivación de dispositivos de prueba.
- ROS es apropiado para grandes sistemas de tiempo de ejecución y para grandes procesos de desarrollo.

En ROS es muy importante la colaboración entre los usuarios, esto se puede observar en que por ejemplo se cuenta con una red federada de repositorios de código, donde diferentes instituciones pueden desarrollar y lanzar sus propios componentes de software de robot. También existe la Wiki de la comunidad de ROS, que es un foro en el que se encuentra información sobre ROS. Cualquiera puede registrarse para obtener una cuenta y contribuir con su propia documentación, proporcionar correcciones o actualizaciones, escribir tutoriales y más.

La única pega que se le podría encontrar a ROS actualmente es que solo se ejecuta en plataformas basadas en Unix. El software para ROS se prueba principalmente en los sistemas Linux y Mac OS X. De hecho en este proyecto ha sido necesario la utilización de Ubuntu 14.04 LTS para su realización, contando con la versión de ROS Indigo. También es conveniente decir que se está estudiando un puerto a Microsoft Windows para ROS.

ROS se divide en dos niveles principalmente: el nivel de gestión de los archivos y el nivel de comunicación.

4.2.1 Gestión de los archivos

En este nivel se concentran los recursos de ROS que se encuentran en el disco, como:

- Packages (Paquetes): Los paquetes son la unidad principal en el que se encuentran los archivos generados en ROS. Son los elementos más pequeños que se pueden encontrar en ROS
- Manifiestos de paquetes: Manifiestos (package.xml) proporcionan información sobre un paquete, incluido su nombre, versión, descripción, información de licencia, dependencias, etc
- Tipos de mensajes (msg): las descripciones de los mensajes, definen las estructuras de datos para los mensajes enviados en ROS.
- Tipos de servicio (srv): las descripciones del servicio, definen las estructuras de datos de solicitud y respuesta para los servicios en ROS.

4.2.2 Comunicación

La comunicación se fundamenta en la red de igual a igual de los procesos de ROS que procesan datos en forma conjunta. Los conceptos básicos de gráficos de computación de ROS son:

- Nodes (Nodos): Los nodos son procesos que realizan computación. Estos se comunican entre ellos usando topics y services. Al final lo que encontramos son un conjunto de nodos, donde cada uno se encarga de controlar un elemento del sistema como puede ser un sensor, un motor de una rueda, etc. Después se puede acceder a cada nodo para leer información necesaria sobre ellos o para enviar información de un nodo a otro.
- Maestro: El ROS maestro es donde se guardan los datos de los nodos. Sin el Maestro, no podrías acceder a los nodos, intercambiar mensajes o invocar servicios.
- Servidor de parámetros: El servidor de parámetros permite almacenar los datos. Actualmente forma parte del Máster.
- Mensajes: Los nodos se comunican entre sí pasando mensajes. Un mensaje es simplemente una estructura de datos.
- Topics: Se utilizan para transmitir los mensajes entre distintos nodos. Un nodo envía un mensaje publicándolo en un topic determinado. El topic es un nombre que se utiliza para identificar el contenido del mensaje. Un nodo que esté interesado en cierto tipo de datos se suscribirá al topic apropiado, como si fuera un bus de datos. Cada topic tiene un nombre, y cualquiera puede conectarse al él para enviar o recibir mensaje.
- Servicios: permite la comunicación solicitud-respuesta, que se definen mediante un par de estructuras de mensajes: una para la solicitud y otra para la respuesta. Un nodo proveedor ofrece un servicio con un nombre y un cliente lo utiliza enviando el mensaje de solicitud y esperando la respuesta. Las bibliotecas cliente de ROS generalmente presentan esta interacción al programador como si fuera una llamada a un procedimiento remoto.
- Bolsas: Las bolsas son un formato para guardar y reproducir datos de mensajes ROS. Las bolsas son un mecanismo importante para almacenar datos.

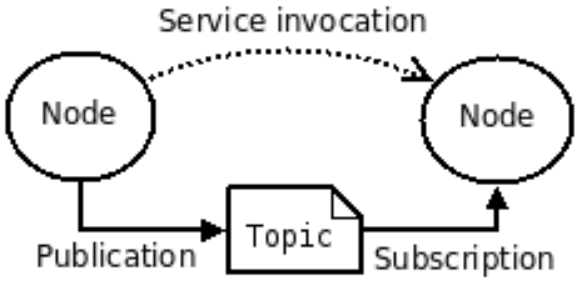


Figura 5: Comunicación entre nodos de ROS.

El ROS Master almacena topics y servicios de información de registro para nodos de ROS. Los nodos se comunican con el Maestro para reportar su información de registro. A medida que estos nodos se comunican con el Maestro, pueden recibir información sobre otros nodos registrados y realizar las conexiones según corresponda. El Maestro también realizará devoluciones de llamada a estos nodos cuando cambie esta información de registro, lo que permite a los nodos crear conexiones dinámicamente a medida que se ejecutan nuevos nodos.

Los nodos se conectan a otros nodos directamente y el Maestro solo proporciona información de búsqueda. Los nodos que se suscriben a un tema solicitarán conexiones de los nodos que publican ese tema y establecerán esa conexión a través de un protocolo de conexión acordado.

Esta arquitectura permite la operación desacoplada, donde los nombres son los medios principales por los cuales se pueden construir sistemas más grandes y complejos. Los nombres tienen un papel muy importante en ROS: los nodos, los topics, los servicios y los parámetros tienen nombres. Cada biblioteca de cliente de ROS admite la reasignación de nombres en la línea de comandos.

4.3 Gazebo

A la hora de simular el comportamiento de un robot es esencial elegir bien el software a utilizar. Un buen simulador permite probar algoritmos rápidamente, diseñar robots, realizar pruebas de regresión y entrenar el sistema de inteligencia artificial utilizando escenarios realistas. Dentro de este ámbito se encuentra Gazebo que es un simulador de robots open source, distribuido bajo la licencia Apache 2.0, que lleva largo tiempo utilizándose en ámbitos de investigación en robótica e Inteligencia Artificial. Tiene una interfaz simple muy fácil de aprender y utilizar [6].

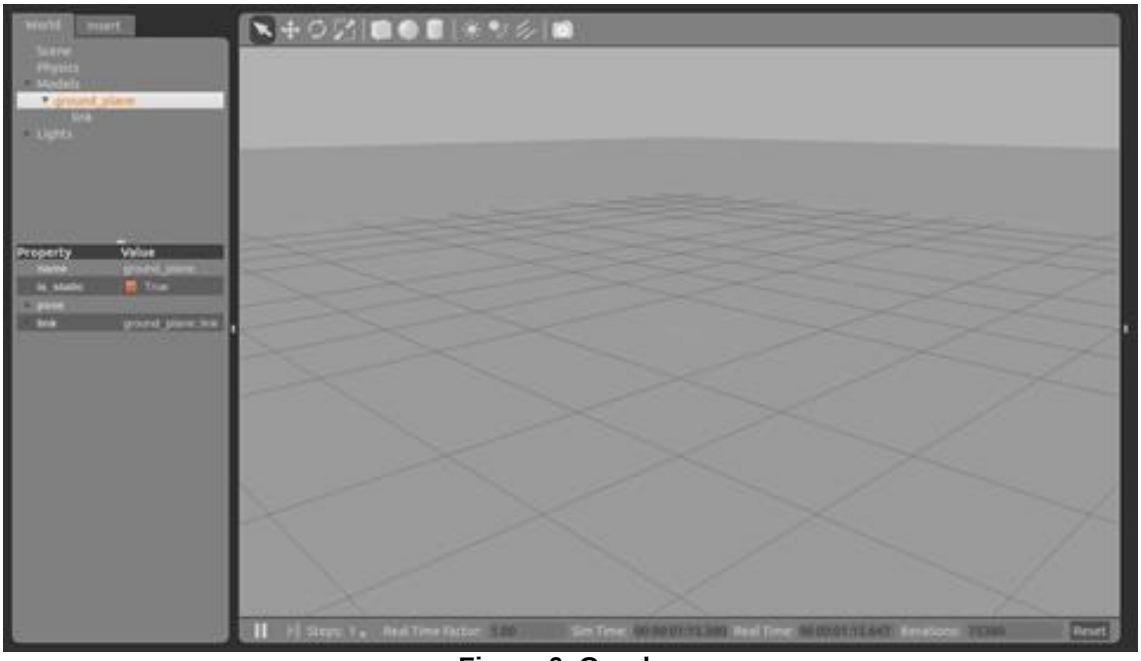


Figura 6: Gazebo.

Gazebo ofrece la capacidad de simular de manera precisa y eficiente las poblaciones de robots en entornos complejos interiores y exteriores. Cuenta con un motor de física robusto, gráficos de alta calidad y un conjunto de sensores e interfaces para usuarios y programas. Lo mejor de todo es que Gazebo es un simulador gratuito con una comunidad muy activa en foros, lo que permite poder solucionar posibles problemas durante el proceso de modelado.

Los usos típicos de Gazebo incluyen:

- Uso de algoritmos para robots.
- Diseño de robots.
- Realizar pruebas de regresión con escenarios realistas.

Algunas características clave de Gazebo incluyen:

- Motores de física múltiple.
- Una rica biblioteca de modelos de robots y entornos de simulación.
- Una amplia variedad de sensores para utilizar en el modelado de robots.
- Convenientes interfaces programáticas y gráficas.

La única pega que se le podría encontrar a Gazebo actualmente es que es necesario es sistema operativo Ubuntu para su utilización. De echo, para llevar a cabo este proyecto ha sido necesario la utilización del sistema operativo Ubuntu 14.04 LTS.

Para lograr la integración de ROS con Gazebo es necesario la utilización de un conjunto de paquetes ROS llamado gazebo_ros_pkgs. Proporcionan las interfaces necesarias para simular un robot en Gazebo mediante mensajes ROS, servicios y reconfiguración dinámica.

También es conveniente comentar que ROS posee comandos propios que posibilitan el lanzamiento de Gazebo. Un ejemplo es el comando roslaunch que se ha utilizado en este proyecto para el lanzamiento del mundo del simulador desarrollado.

4.3.1 Baxter

En este proyecto vamos a utilizar Gazebo como simulador para utilizar el robot Baxter en los experimentos.

Baxter es un robot industrial diseñado por Rethink Robotics en el 2011 [7]. Cuenta con dos brazos y una pantalla en la que se puede proyectar diversas caras, lo que le da una imagen más humana. Caba brazo cuenta con un gripper como mano lo que le permite interactuar con objetos cogiéndolos y soltándolos. La altura del Baxter se puede variar entre 1,75m y 1,90m y su peso ronda alrededor de los 140 kg. Sus brazos están diseñados para permitir el manejo de objetos, puede coger un objeto en un punto y desplazarlo y soltarlo en un segundo punto, pudiendo realizar trabajos simples y monótonos, de manejo de material, de forma sencilla y rápida.

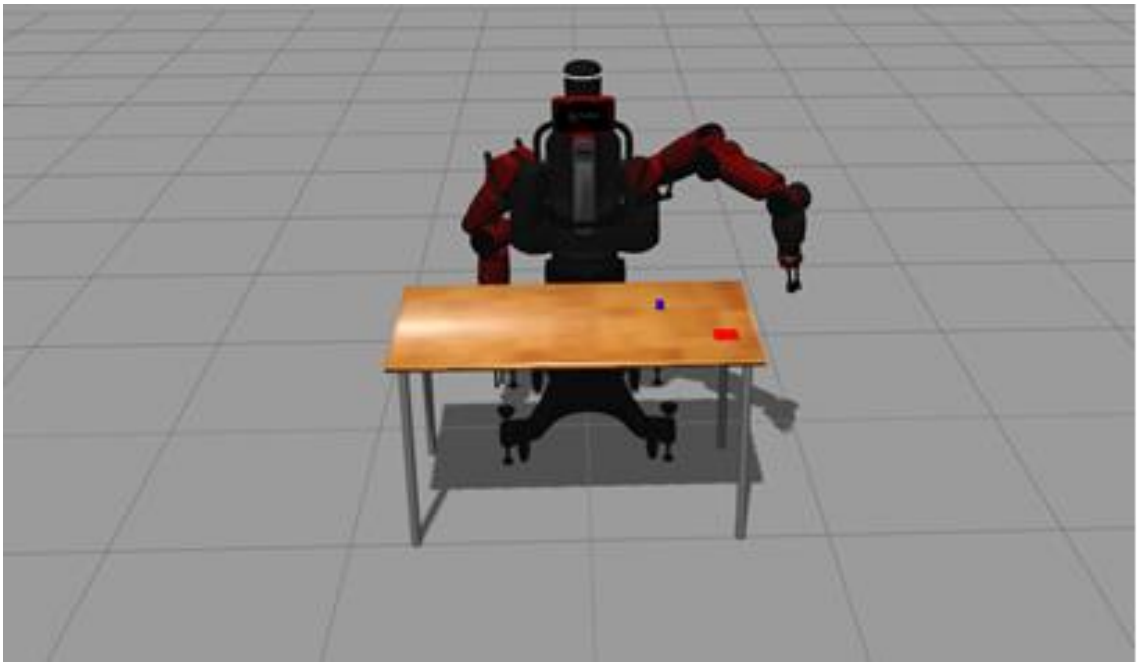


Figura 7: Baxter.

4.4 TensorFlow

TensorFlow es una librería de código abierto ampliamente utilizada para aprendizaje automático desarrollado por investigadores e ingenieros que trabajan en el equipo Google

Brain Team, dentro del departamento de investigación de Google de Machine Intelligence. Esta librería permite a principiantes y expertos en Machine Learning generar modelos para diseñar y entrenar redes neuronales. Estos modelos permitirán detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos [8][9].

Figura 8: Logo de la libraría de código abierto TensorFlow.

La arquitectura flexible de TensorFlow le permite implementar el cálculo a una o más CPU o GPU en equipos de escritorio, servidores o dispositivos móviles. Algunos de los ejemplos de aplicaciones actuales de TensorFlow son:

- Para mejorar la fotografía en los smarthphones.
- Para ayudar al diagnóstico médico, como por ejemplo para analizar radiografías.
- Procesamiento de imágenes.

En Tensorflow nos encontramos con distintas APIs de alto o bajo nivel. Dentro de las APIs de alto nivel se encuentra la API de Keras. Esta se utiliza para diseñar y entrenar modelos de Deep Learning de forma simple y rápida. Tiene dos ventajas fundamentales:

- Tiene una interfaz simple y consistente lo que permite diseñar redes neuronales de forma sencilla y rápida.
- Los modelos se realizan conectando bloques de construcción configurables entre sí, con pocas restricciones.

Debido a estas ventajas se ha decidido de la API de TensorFlow, Keras, será la utilizada para llevar a cabo el desarrollo de este proyecto.

5 ANTECEDENTES

La línea de investigación de robótica cognitiva es muy importante para el grupo de investigación del Grupo Integrado de Ingeniería (GII) de la UDC. Disponen de numerosas plataformas robóticas para la realización de experimentos entre los que se encuentran el Baxter, NAO, Robobo, etc.

En la actualidad, el GII se centra en utilizar modelos de utilidad más simples para guiar inicialmente el comportamiento del robot, como puede ser el caso de las SURs. El aprendizaje de estos modelos unidimensionales se realizará a partir de la motivación intrínseca. Con estos distintos tipos de modelos de utilidad se pretende conocer e interpretar el espacio de estados.

La SUR no es la manera más eficiente para moverse a través del espacio de estados hacia una meta, pero son bastante fáciles de obtener y son indicaciones muy sólidas de la correcta dirección a seguir para alcanzar el objetivo. No obstante, siempre que sea posible, sería interesante contar con representaciones que nos aporten información más exacta de los MU que permitan mejores procesos de decisión, como representaciones precisas de la función de valor, como las VF.

Este tipo de MU, las VF, a menudo son bastante difíciles de entrenar directamente a partir de las trazas obtenidas por motivaciones exploratorias intrínsecas. Sin embargo, la hipótesis que se sigue hoy en día en el GII es primero generar las áreas de certeza de las SURs mediante motivación intrínseca. Después, una vez que se obtiene una representación correcta basada en una jerarquía SUR de un MU, las trazas que produce el sistema cuando se alcanzan los objetivos que siguen a esta jerarquía SUR, se comportarán mucho mejor y presentarán un nivel de ambigüedad mucho menor que las trazas exploratorias mediante motivación intrínseca. Estas trazas obtenidas mediante SURs se utilizarán para entrenar posteriormente a la VF. En consecuencia, en muchos casos será posible producir MU basadas en funciones de valor precisas a partir de estas trazas [3][11].

Para lograr esto, han establecido un procedimiento de aprendizaje de VF. Primero MotivEn crea el número de SURs consideradas como necesarias para alcanzar la meta. A través de la concatenación de las distintas SURs se conduce al robot hasta el objetivo. Una vez conseguido esto el proceso de aprendizaje de la VF puede comenzar tratando de generalizar la respuesta de estas SURs. Por lo tanto, la idea de este procedimiento es, intentar generalizar la respuesta de estas SURs en una representación combinada más precisa mediante la VF. Esto es lo que han llamado un proceso de redescripción representativa del Modelo de Utilidad.

Más específicamente, en el GII, se ha diseñado y probado un procedimiento, inspirado en el utilizado en la arquitectura cognitiva de MDB, en que las VF se representarán a través de una Feed Forward ANN estándar (aunque se podría usar cualquier otro tipo de estructura), que tiene tantas entradas como el número de SUR seleccionados para generalizar, y una salida: el valor de utilidad. Las entradas corresponden a los valores de los sensores utilizados por los SURs considerados. La ANN se entrena utilizando un algoritmo clásico de descenso de gradiente sobre los buffers de rastreo de los SURs a generalizar, que se combinan en un solo buffer donde se asigna una utilidad decreciente siguiendo el esquema clásico de rastreos de elegibilidad utilizados en el aprendizaje de refuerzo.

Una vez que una ANN se entrena con éxito en estas trazas iniciales relacionadas con las SURs, sustituye a las SURs y se utiliza como el modelo de utilidad para guiar al robot hacia la meta. Inicialmente, esta ANN modela directamente el comportamiento de la jerarquía SUR sobre la que fue entrenado, así como su área de certeza. Sin embargo, a medida que se utiliza esta ANN, cada vez que el robot alcanza el objetivo se obtiene una nueva traza y se almacena

en el buffer de traza de VF, lo que implica la eliminación de la más antigua de este buffer y la actualización del área de certeza de VF. Estos buffers de rastreo se utilizan periódicamente en procesos de reentrenamiento que mejoran la representación de VF, incluida la de su área de certeza. En otras palabras, progresivamente, el VF podrá generalizar las correlaciones de sensores unidimensionales de las SURs a correlaciones de sensores multidimensionales que proporcionan un camino hacia el objetivo más directo.

Como podemos ver, el aprendizaje de VF es un proceso dinámico que se realiza de manera continua a medida que se obtienen nuevas trazas. El punto relevante aquí es que el aprendizaje de la VF comienza a partir de las trazas obtenidas con la metodología SUR, es decir, todos los puntos de percepción siguen una correlación hacia la meta (utilidad), lo que simplifica el espacio de búsqueda de la VF. Una vez aprendido, la VF es una redescripción de los modelos de utilidad esperados unidimensionales creados con las SURs, que generalizan su respuesta. La **Figura [9]** contiene una representación simplificada de este proceso, donde podemos ver dos regiones SUR 1 y 2, que llevan al objetivo después de seguir caminos unidimensionales en el espacio de estado. Con la generalización de la VF, su región de certeza es más amplia y generaliza las otras dos áreas de certeza, obteniendo rutas directas al objetivo en más de una dimensión (representadas mediante líneas verdes), [2][11].

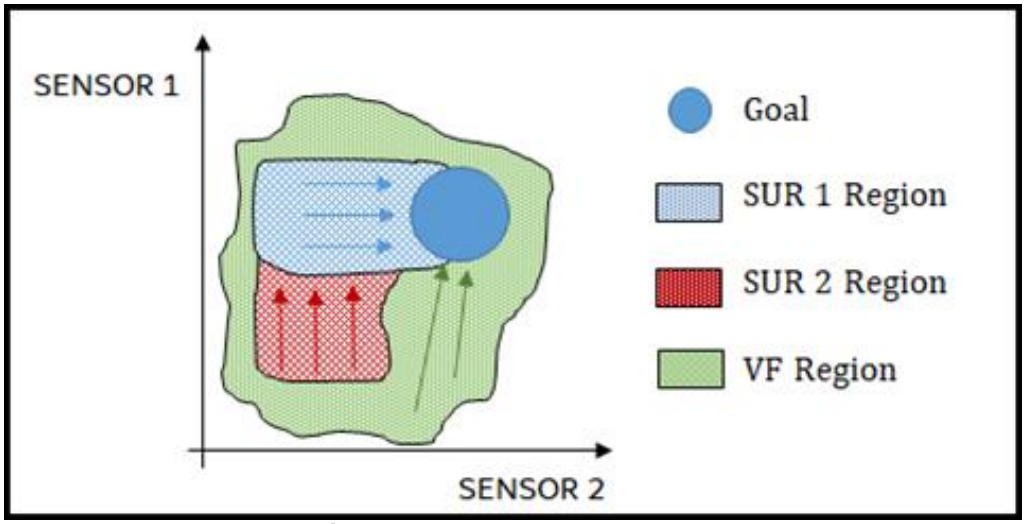


Figura 9: Identificación de subobjetivos y proceso de encadenamiento.

Es importante tener en cuenta que este proceso de redescripción de SUR-VF no siempre es posible o eficiente. Puede haber casos o áreas del espacio del estado donde no es factible obtener una MU basada en VF directa. En esos casos, el sistema utilizará la representación basada en SUR, que, aunque es menos eficiente, consigue realizar el trabajo de forma correcta.

Hasta el momento, este tipo de entrenamiento de VF se lleva haciendo de forma off-line, es decir, una vez que el aprendizaje on-line con las SURs se daba por terminado, se iniciaba un entrenamiento off-line de la VF hasta que este convergía. Evidentemente, esta metodología no es compatible con el funcionamiento open-ended y on-line del MDB. En consecuencia, en este proyecto se pretende desarrollar un método que permita el aprendizaje on-line y autónomo de las VF a partir de las SURs. Para poder realizar el aprendizaje on-line de las VF, se deberá integrar en MotivEn un algoritmo de optimización on-line basado en Keras.

El método propuesto se va a probar en un problema representativo de robótica cognitiva, utilizando para ello el simulador 3D Gazebo. Con él, a medida que el robot se desplaza por el conjunto de estados sensoriales, se va a poder obtener los distintos datos necesarios para llevar a cabo el entrenamiento de las funciones de utilidad.

A continuación, en los siguientes puntos de la memoria, se va a explicar con más detalle esta nueva metodología introducida con anterioridad en este apartado. En ellos se hablará del

desarrollo llevado a cabo para realizar este proyecto y los distintos resultados obtenidos en las diferentes simulaciones realizadas para comprobar el correcto funcionamiento de esta nueva metodología de entrenamiento de VF desarrollada.

6 REQUISITOS DE DISEÑO

A continuación, se van a describir los requisitos que deben cumplir todos los elementos que forman parte de este proyecto:

- En primer lugar, tal y como está establecido dentro del proyecto DREAM, todos los desarrollos que se realicen deben encontrarse bajo los estándares de ROS (Robot Operating System), por lo que, debe ser necesario implementar los comportamientos básicos en este nuevo sistema.
- El lenguaje de programación empleado debe ser Python 2.7.x para mantener la compatibilidad con el código de MotivEn y porque es el que se usa en el código del simulador en 3D utilizado como base para realizar este proyecto. Además, al ser Python el lenguaje más utilizado por otros diseñadores en el mismo robot, facilitará la incorporación de nuevas funcionalidades futuras desarrolladas por otros diseñadores si lo considerasen necesario.
- La implementación de la Value Function se debe realizar empleando la biblioteca TensorFlow, en concreto la API de Keras, y empleando el algoritmo de optimización Adam como ejemplo de su correcto funcionamiento en las pruebas.
- El escenario de simulación 3D para las pruebas debe realizarse empleando Gazebo. En consecuencia, los modelos robóticos empleados deben ser compatibles con este último.
- El sistema motivacional resultante debe mantener su correcto funcionamiento independientemente del entorno de simulación empleado.
- Los comportamientos físicos del modelo de simulación deberán ser lo más files posibles a los movimientos realizados por el robot real.

7 DESARROLLO

Como se ha mencionado anteriormente en el apartado *6. Antecedentes*, en el GII las pruebas que se llevan a cabo para el estudio del aprendizaje de VF se realizan de forma Off-line. Con la realización de este proyecto se pretende incorporar el entrenamiento de VF On-line, junto con el simulador 3D Gazebo.

El diseño del nuevo método de obtención de forma autónoma de VF a partir de SURs que se va a llevar a cabo en este proyecto, mediante entrenamiento On-line, se puede resumir en los siguientes puntos:

1. Inicialmente, el comportamiento del robot se va a guiar mediante la motivación intrínseca ciega, concretamente mediante Novelty. Con esta motivación se va a realizar un proceso exploratorio inicial que hace el robot del entorno que lo rodea en busca de estados sensoriales no visitados.
2. Una vez que se alcanza un objetivo, se obtiene una traza y se puede aprender un modelo de utilidad asociado a dicho objetivo, inicialmente, basado en SURs.
3. Con el paso de las iteraciones, gracias a las trazas obtenidas, se van a ir actualizando los mapas de certeza de cada SUR. Cuando el robot se encuentre dentro de un área de certeza, la SUR a la que le pertenece dicha área, gobernará el movimiento del robot, y cuando no se encuentre en ningún área de certeza, seguirá guiándose el comportamiento robot mediante motivación intrínseca. Cada vez que se llegue al objetivo, los mapas de certezas se van a ir actualizando y mejorando sus predicciones.
4. Una vez superado un umbral mínimo de veces que se llega al objetivo utilizando las SURs, comienza el aprendizaje on-line de la VF en paralelo con el de las SURs. En la primera iteración de entrenamiento de la VF, se utilizarán todas las trazas obtenidas hasta el momento. A partir de ahí, se entrena la VF con cada nueva traza. En el momento en que el error de predicción de la VF sea inferior a un umbral, esta sustituirá a las SURs como modelo de utilidad asociado a la motivación extrínseca.
5. A partir de este momento se van a utilizar como datos de entrenamiento las trazas obtenidas mediante la VF.

En la **Figura [10]** se puede observar un esquema en el cual se representa la metodología para obtener VF a partir de SURs de forma autónoma implementada en este proyecto.

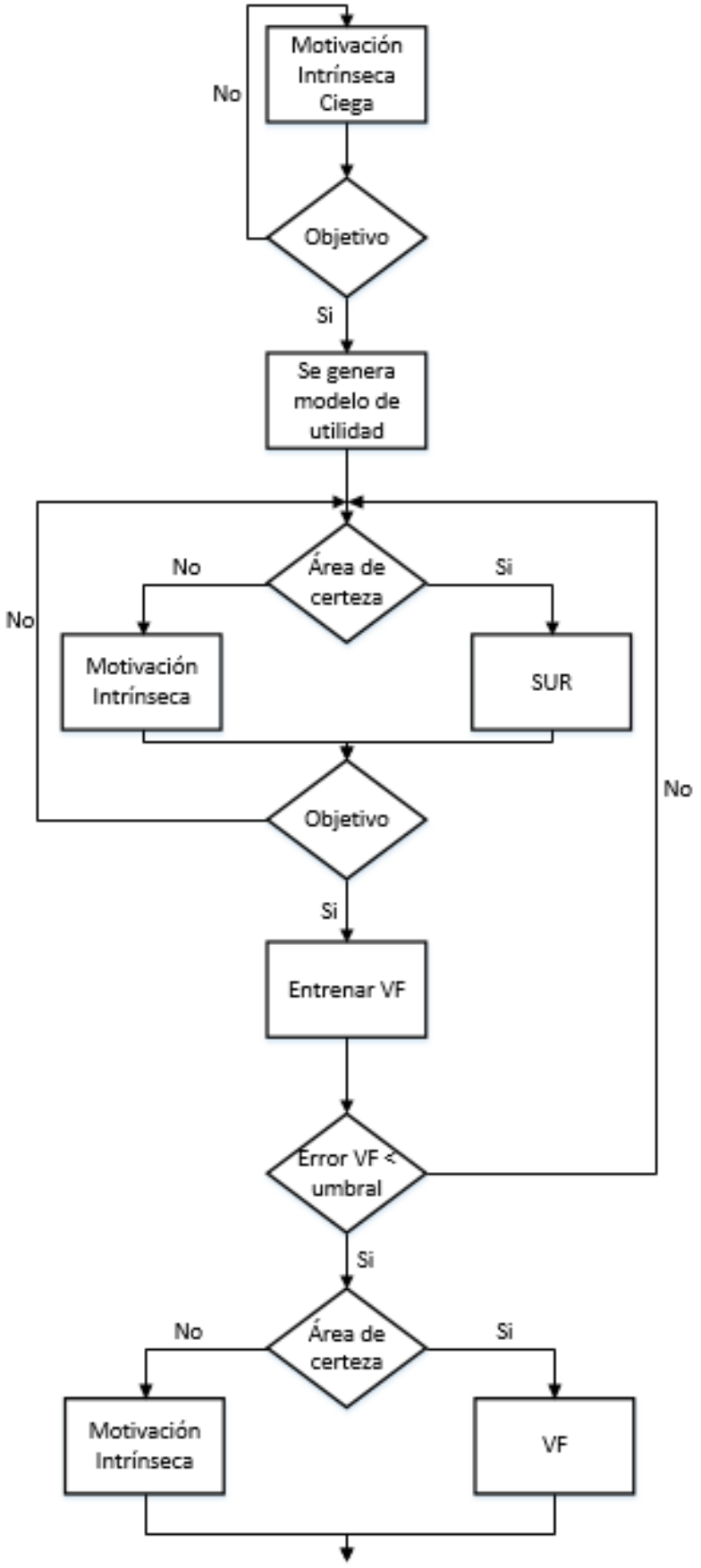


Figura 10: Diagrama de bloques metodología entrenamiento VF.

8 PRUEBAS

En este apartado se van a explicar todas las pruebas llevadas a cabo en este proyecto. Primeramente, se han realizado unas pruebas con trazas Off-line mediante un programa diseñado en Python 2.7 para elaborar un predimensionamiento de la red neuronal a utilizar para las pruebas On-line. En los siguientes puntos se explicará en profundidad el por qué de realizar esta prueba previa, como se ha realizado y los distintos resultados obtenidos.

Después se realizarán las pruebas On-line en el simulador 3D con la nueva metodología de entrenamiento de VF implementada en este proyecto.

8.1 Diseño red neuronal

Lo primero que se ha decidido realizar en este proyecto ha sido el diseño previo de la arquitectura de la red neuronal. Esto no va a ser posible llevarlo a cabo en la propia aplicación del simulador en 3D debido al alto tiempo de cómputo que sería necesario. Por esta razón se ha decidido desarrollar un programa implementado con Python 2.7 en el que se van a probar distintas tipologías de red para encontrar la que mejor se comporte para el experimento diseñado en el simulador en 3D. Una vez haya sido seleccionada una tipología de red, se podrán acometer las pruebas requeridas en el simulador en 3D con la arquitectura de red con la que mejores resultados se hayan obtenido.

Este archivo va a estar programado en Python 2.7 mediante Tensor Flow. En concreto se ha utilizado la API de Keras para el diseño de la arquitectura de la red. Esta API es muy utilizada para diseñar y entrenar modelos de Deep Learning debido a que se puede hacer de forma simple y rápida mediante bloques implementados en Keras.

Los datos que se van a utilizar para entrenar la red neuronal en este programa van a ser trazas Off-line obtenidas mediante un simulador en 2D, realizado por trabajadores del GII de la UDC, en el que se ha implementado el experimento desarrollado en el simulador en 3D (este experimento se puede observar en el apartado *8.2.1 Diseño del experimento en Gazebo*). Hay dos tipos de trazas que se van a utilizar para el entrenamiento, las primeras son obtenidas mediante motivación intrínseca, Novelty, y las segundas obtenidas mediante SURs. En ambos casos las trazas utilizadas para entrenamiento y test fueron sacadas de un conjunto original de datos separando las 82 primeras trazas para entrenamiento y las 7 siguientes para test.

8.1.1 Programa desarrollado

En este apartado se va a explicar cómo se ha realizado el programa para la realización del entrenamiento y diseño de la red neuronal. Este se puede dividir en 4 partes principalmente:

1. La primera parte va a ser la encargada de leer los datos obtenidos con el simulador 2D guardados en un archivo .csv. Este conjunto de trazas se va a guardar en vectores de valores tipo float para permitir su posterior manejo a la hora de realizar el entrenamiento de la red de una forma similar a la que se realizaría de forma On-line. Por lo tanto, se van a tener dos archivos .csv, uno de nombre TRAZAS ENTRENAMIENTO en el que se va a guardar las 82 trazas que se van a utilizar para entrenar el modelo, y otro llamado TRAZAS TEST en el que se va a almacenar las 7 trazas que se van a utilizar para el testeo del modelo. El tamaño máximo de cada traza se ha prediseñado anteriormente con un valor de 50 datos de máximo, por lo que solo van a contar, como máximo, con los últimos 50 puntos de la trayectoria del robot junto a sus respectivos valores de utilidad.

En la **Figura [11]** se puede observar el código implementado para la realización de lo explicado anteriormente.

```
def lectura_train():
    #Lectura de datos de entrenamiento
    lectura = pd.read_csv("/home/alvaro/Escritorio/PROGRAMA/DATOS/TRAZAS_ENTRENAMIENTO.csv", sep=";")
    X=lectura.iloc[0:,1:3].values.astype(float)
    Y=lectura.iloc[0:,3:4].values.astype(float)
    return X,Y

def lectura_test():
    #Lectura de datos de test
    lectura = pd.read_csv("/home/alvaro/Escritorio/PROGRAMA/DATOS/TRAZAS_TEST.csv", sep=";")
    X=lectura.iloc[0:,1:3].values.astype(float)
    Y=lectura.iloc[0:,3:4].values.astype(float)
    return X,Y
```

Figura 11: Código para lectura de datos.

2. Una vez que tenemos todos los datos guardados en vectores para su posterior uso ya podemos empezar a diseñar la red neuronal de tipo Feed Forward ANN. Para ello, como se ha explicado con anterioridad, se ha utilizado la API de Tensor Flow llamada Keras. La forma más simple para diseñar redes neuronales en Keras es utilizando el modelo Sequential(), que crea series de capas de neuronas de forma secuencial (una tras otra), y el tipo de capa Dense(), que es un tipo de capa "normal". Este tipo de modelado permite añadir la capa de entrada, las capas ocultas y la capa de salida, con la función. add() de forma muy sencilla y rápida como se puede observar en la **Figura [12]**.

```
def baseline_model():
    #Se configura la red neuronal
    model = Sequential()

    #Se añaden las distintas capas que forman la arquitectura de la red
    model.add(Dense(6, input_dim=2, kernel_initializer=normal, bias_initializer='ones', activation='relu'))
    model.add(Dense(6, kernel_initializer=normal, bias_initializer='ones', activation='relu'))
    model.add(Dense(1, kernel_initializer=normal,bias_initializer='ones', activation='relu'))

    # Se compila el modelo
    model.compile(loss='mse', optimizer='adam' , metrics=['mse'])

    return model
```

Figura 12: Diseño de la red neuronal.

En estas funciones es necesario introducir el número de neuronas en cada capa y la función de activación, que en este caso se ha decidido utilizar la función de activación relu. Como se puede ver en la **Figura [12]**, se ha diseñado un ejemplo de una red neuronal para comprender como se realizaría la configuración de ella. Este ejemplo cuenta con dos entradas, dos capas ocultas de seis neuronas cada una y una salida.

Una vez implementada todas las capas de la red neuronal, mediante la función. compile() se realiza la compilación del modelo definiendo en esta función el cálculo del error y el tipo de optimizador a utilizar, en este caso se ha decidido usar el optimizador tipo Adam, como se puede ver en la **Figura [12]**.

3. El siguiente paso a realizar va a ser el entrenamiento de la red neuronal. Para ello es importante comentar, que para simular el entrenamiento On-line que se va a realizar posteriormente con el simulador 3D, se ha tenido que implementar una forma determinada de selección de las trazas de entrenamiento, ya que, en este caso, contamos inicialmente con todo el conjunto de trazas obtenidas en el simulador 2D de forma Off-line. Esto se va a llevar a cabo mediante una serie de bucles que permiten entrenar solo con el conjunto de trazas deseadas por el diseñador. Esto se ha realizado siguiendo la condición de que el último valor de utilidad de cada traza siempre va a ser uno, con lo que podemos separar el vector de conjuntos de datos en las distintas trazas que lo forman. Aunque selecciones

un grupo de trazas de valor distinto a uno, este entrenamiento se realizará traza a traza de dicho conjunto.

Otro mecanismo de selección de datos implementado es la variación del tamaño de conjunto de trazas a entrenar. Se puede entrenar trazas a traza o con un conjunto de trazas para aumentar el conjunto de datos de entrenamiento. Por ejemplo, si se desea se podría entrenar con un conjunto de dos trazas, introduciendo número de trazas = 2. Este entrenamiento se realizaría de la siguiente forma. Primero se seleccionarían las dos primeras trazas y se entrenaría primero con la primera traza y posteriormente con la segunda. Acto seguido se cambiaría el conjunto de trazas incorporando a ella la segunda y la tercera traza, eliminando la primera traza para mantener un conjunto de dos trazas, y se volvería a entrenar el modelo primero con la segunda trazas y después con la tercera. Se seguiría con este mecanismo de entrenamiento de forma secuencial hasta que lleguemos hasta el conjunto formado por la última y penúltima trazas, con los que terminaría de entrenar el modelo y se saldría del bucle gracias a un break.

El entrenamiento se va a llevar a cabo mediante la función. fit() de Keras con la que realiza el entrenamiento de la red neuronal traza a traza con el tamaño de lote deseado. Se le introducen como entradas el vector formado por los puntos de la trayectoria, valores de los dos sensores de distancia entre la pelota y el gripper y la pelota y el objetivo, y como salida el vector con los valores de utilidad asignados a cada punto de la trayectoria. Por otro lado, también es necesario definir el número de epochs, el número de veces que se va a llevar a cabo el entrenamiento con este conjunto de datos. Esto se va a realizar introduciendo un máximo de 3000 epoch (epoch=3000) y una condición de parada (callback), que consiste en que cuando se estabilice el error de entrenamiento este se corta (esta condición de parada va a ser la predominante). Se elige un tamaño de batch (número de datos que forman el conjunto de entrenamiento) de valor la longitud de la traza (longitud del vector con valores de utilidad) y no se va a utilizar un conjunto de datos de validación, ya que, simplemente se hará una predicción del modelo más adelante. Se van a guardar los datos de error de entrenamiento y de tiempo de entrenamiento en archivos Excel para su posterior análisis. Todo el código explicado anteriormente se puede observar en la **Figura [13]**.

```
for i in ytrain:
    if i != 1.0:
        cont+=1
    if i == 1.0:
        cont+=1
        datos.append(cont)
        cont=0

for i in datos:
    suma= suma + i

n=numero_trazas
inicio=0
valor=0
for i in range(len(datos)):
    numero_datos = 0

    for j in datos[i:n]:
        numero_datos=numero_datos + j

    if inicio == 0:
        inicio = 1
        punto_inicio = 0
    else:
        punto_inicio = punto_inicio + datos[(i-1)]

    cont=0

    for k in xtrain[punto_inicio:(punto_inicio+numero_datos)]:
        xtrain2.append(k)

    for k in ytrain[punto_inicio:(punto_inicio+numero_datos)]:
        ytrain2.append(k)

    n += 1
```

```
    for f in range(len(xtrain2)):
        for l in xtrain2[f]:
            xtrain3.append(l)
        ytrain3.append(ytrain2[f])
        cont += 1
        if ytrain2[f]==1:
            xtrain3 = np.reshape(xtrain3, (cont, 2))
            ytrain3 = np.reshape(ytrain3,(cont,1))
            t0 = time.time()
            history = model.fit(xtrain3, ytrain3, epochs=3000, callbacks=[callback], batch_size=len(ytrain3),
                                verbose=0,validation_split = 0.0)
            t1 = time.time()
            t=t1-t0
            vector_tiempo.append(t)
            valor += 1
            for i in history.history['loss']:
                vector_error.append(i)
            store_info_3(xtrain3,ytrain3, valor)
            store_info_2(vector_tiempo,iteracion)
            store_info_1(vector_error, iteracion)
            xtrain3 = []
            ytrain3 = []
            cont=0

    xtrain2=[]
    ytrain2=[]

    if i == (len(datos)-numero_trazas):
        break
```

Figura 13: Metodología de realización del entrenamiento.

4. Ahora mediante la función. evaluate() de Keras se obtiene como salida, introduciéndole como argumento la trayectoria y la utilidad del conjunto de datos de test, el valor del error cuadrático medio del modelo para un determinado conjunto de datos y un diseño de arquitectura de red neuronal. Este valor se va a guardar en un archivo Excel para su posterior análisis.

   Con la función. predict() de Keras se realiza la evaluación del modelo, introduciendo como entrada la trayectoria del conjunto de datos de test. Con esta función se va a obtener como salida las predicciones de los valores de utilidad esperados por el modelo entrenado. Estos valores también se van a guardar en un archivo Excel junto con el error de posición, valor absoluto de la resta entre los

valores de utilidad del conjunto de test y los valores de salida de las predicciones realizadas mediante la función. predict() de Keras.

Todo lo mencionado en este punto se puede observar en la **Figura [14]**.

```
evaluacion = model.evaluate(xtest, ytest)
predictions = model.predict(xtest, verbose=0)
```

Figura 14: Función evaluate y predict.

Finalmente, en la función main se agrupan todas las funciones mencionadas anteriormente. Además, como con una simulación no es suficiente para corroborar la validez de los resultados obtenidos, se ha implementado un bucle con el que se permitirá realizar 10 veces estos cálculos descritos anteriormente, como se puede ver en la **Figura [15]**.

```
for i in range(10):
    xtrain,ytrain=lectura_train()
    xtest,ytest=lectura_test()
    pred=train(xtrain,xtest,ytrain,ytest,numero_trazas,i)
```

Figura 15: Función main con bucle de 10 iteraciones.

El conjunto completo del código descrito en este apartado se puede observar en el **Anexo 2**.

8.1.2 Resultados

En este apartado se van a exponer los resultados obtenidos con distintas arquitecturas de red y seleccionar la que mejor se comporta para este experimento en concreto. Se van a tratar resultados conseguidos mediante dos tipos de datos, como se ha comentado con anterioridad, primero unas trazas obtenidas mediante motivación intrínseca, concretamente mediante Novelty, y después otras trazas obtenidas mediante SURs.

Para el entrenamiento de la red neuronal se han probado distintas tipologías de red, pero siendo las mismas para ambos conjuntos de datos. Los datos están formados por dos entradas, los valores de los sensores de distancia que están situados en las dos primeras columnas, y una salida, el valor de utilidad que está situado en la tercera y última columna. Las redes neuronales han sido diseñadas con una capa oculta (2-6-1 / 2-8-1 /2-10-1), dos capas ocultas (2-6-6-1 / 2-8-8-1 / 2-10-10-1) y tres capas ocultas (2-6-8-6-1 / 2-5-10-5-1 / 2-8-10-8-1). También se han realiza pruebas con un mayor número de capas ocultas, pero al no obtener ninguna mejora en los resultados, se ha optado por no profundizar en ellas.

Aparte de los distintos ensayos llevados a cabo con diferentes arquitecturas de redes, también se van a hacer pruebas con tres tipos de tamaño de traza, en concreto se han probado con tamaño de traza de uno, dos y cinco, para observar con que diseño se obtienen mejores resultados.

A la hora de elegir la mejor tipología de red se va a comparar la media de la diferencia entre los valores de utilidad de los datos de test y los valores de utilidad estimados por la red neuronal una vez se haya entrenado. Todo el conjunto de los resultados obtenidos en estas pruebas se puede observar en el **Anexo 3**.

Ahora se van a exponer los resultados obtenidos con ambos tipos de datos:

8.1.2.1 Datos obtenidos mediante motivación intrínseca

En la **Tabla [2]** se puede observar una selección de algunos de los resultados obtenidos para este conjunto de datos. Se ha decidido mostrar en este apartado el mejor resultado con una capa oculta, con dos capas ocultas y con tres capas ocultas.

Nº SIM	2-10-1	2-6-6-1	2-8-10-8-1
1	0,139843	0,140014	0,206944
2	0,142902	0,156966	0,245665
3	0,128669	0,147692	0,123864
4	0,139842	0,120588	0,245212
5	0,139842	0,140428	0,147207
6	0,153534	0,127126	0,12432
7	0,144634	0,13312	0,127766
8	0,116642	0,139111	0,182451
9	0,139843	0,123302	0,120523
10	0,139842	0,132658	0,049928
Media	**0,138559**	**0,1361**	**0,157388**
Desviación	**0,000866**	**0,001119**	**0,034695**

Tabla 2: Resultados obtenidos con datos Novelty.

Como se puede observar, se obtienen mejores resultados con tamaños de red neuronal pequeñas, pocas capas ocultas y de pequeño número de neuronas. Cuando mayor es la red neuronal mayor valor de error y desviación obtenemos. Además, también se puede ver en el **Anexo 3**, donde se encuentran todos los resultados obtenidos, que los mejores resultados se obtienen para tamaños de conjunto de trazas pequeño, cuanto mayor este tamaño de traza la variabilidad de la media del error aumenta. En concreto la arquitectura de red con la que mejores resultados se ha obtenido, con este conjunto de datos, es la de dos capas ocultas de seis neuronas cada una (2-6-6-1), obteniendo una media de error de 0,1361. En la simulación número 4 se ha obtenido el mejor resultado que es de un error de valor 0,120588.

En la siguiente figura, para los resultados obtenidos con la arquitectura de red de 2-6-6-1 (dos entradas, dos capas ocultas de seis neuronas cada una y una salida), se ha comparado los valores de utilidad de los datos de test y los valores de utilidad predichos por la red neuronal. Se puede observar claramente la ambigüedad de los datos, hay varios puntos, de distintos valores de distancias del sensor, que tienen el mismo valor de utilidad (**Figura [16]).** Esto puede ser debido, a que como los movimientos son gobernados mediante Novelty no alcanzan el objetivo directamente, sino que, al ser el movimiento semi-aleatorio, el robot da muchas vueltas hasta que llega al objetivo. También se puede ver que para valores de utilidad altos la predicción se ajusta mejor a los datos reales porque, cuando más cerca del objetivo nos encontremos, menor ambigüedad en los datos habrá.

En la **Figura [17]** se muestra el error de entrenamiento en la misma simulación que en el caso de la **Figura [16]**. La media del error de entrenamiento es de 0,01044. Los puntos con valores de error de entrenamiento más altos en cada entrenamiento corresponden a las primeras epochs del entrenamiento, y este error va disminuyendo continuamente mientras se realiza el entrenamiento. Como se puede observar, debido a la gran ambigüedad en los datos obtenidos con esta motivación intrínseca, no se consigue ir disminuyendo el valor del error de entrenamiento inicial en cada simulación de entrenamiento respecto al entrenamiento anterior.

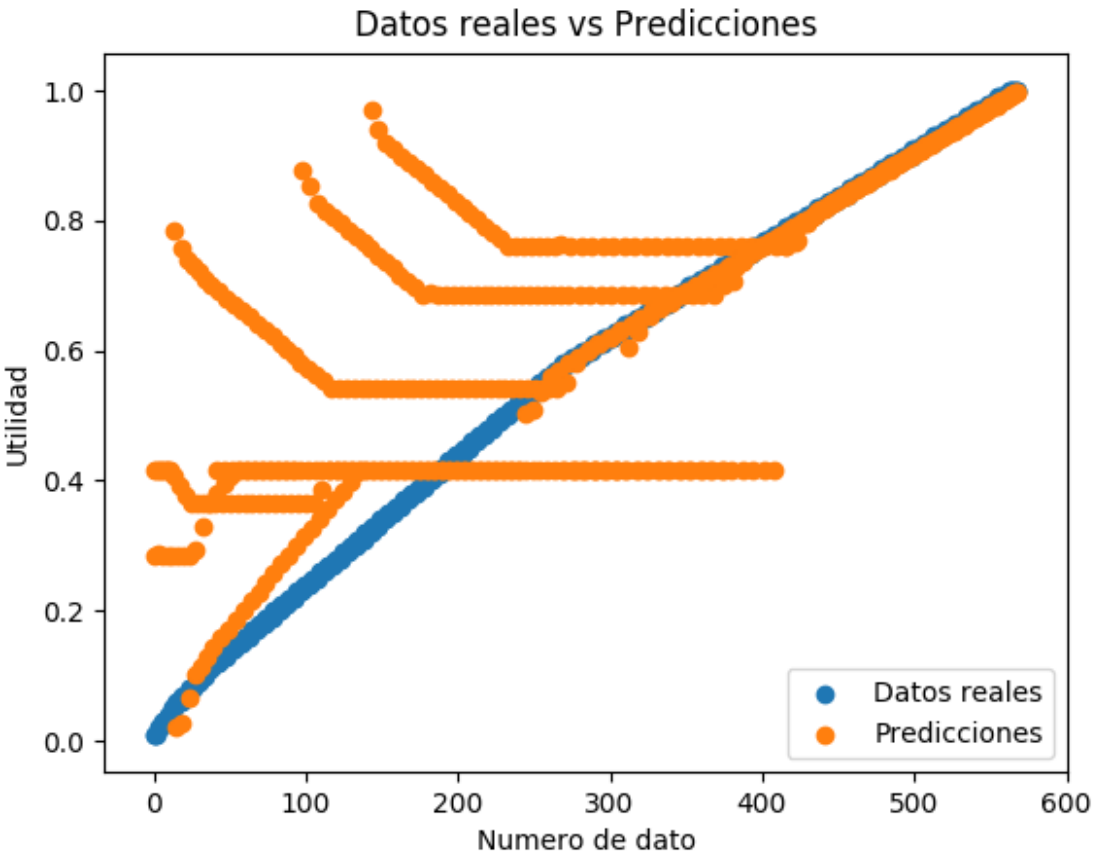


Figura 16: Datos reales vs Predicciones con datos Novelty.

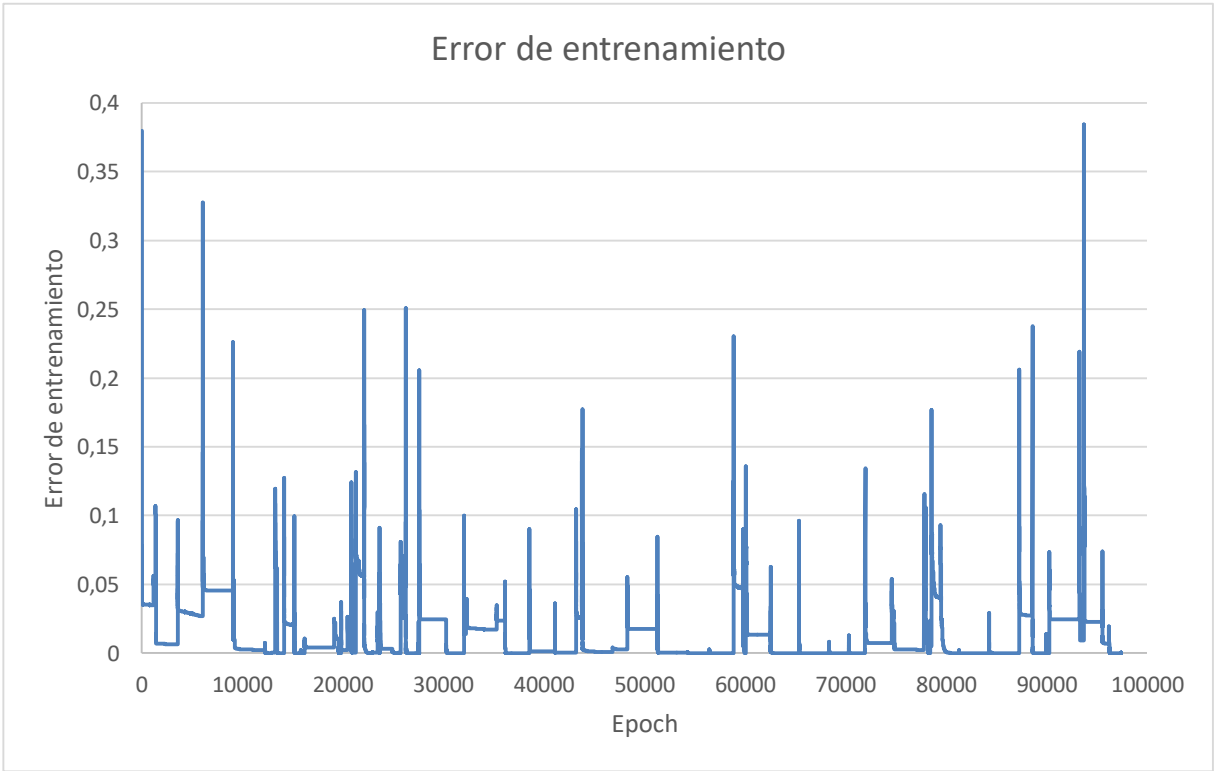


Figura 17 : Error de entrenamiento con datos Novelty.

8.1.2.2 Datos obtenidos mediante SURs

En la **Tabla [3]** se puede observar una selección de los mejores resultados obtenidos para este conjunto de datos. Se ha decidido visualizar en este apartado los mismos ejemplos que en el punto de datos obtenidos mediante Novelty para facilitar la comparación de los distintos resultados obtenidos.

Nº SIM	2-10-1	2-6-6-1	2-8-10-8-1
1	0,054709	0,053455	0,036113
2	0,060418	0,057024	0,036541
3	0,026609	0,022041	0,038516
4	0,026607	0,053567	0,040444
5	0,027667	0,042140	0,042159
6	0,026612	0,026329	0,045404
7	0,027542	0,041974	0,040601
8	0,056220	0,028692	0,039181
9	0,053462	0,022544	0,032899
10	0,026722	0,041230	0,041754
Media	**0,038657**	**0,038899**	**0,039361**
Desviación	**0,002081**	**0,001595**	**0,000114**

Tabla 3: Resultados obtenidos con datos SURs.

Como en el caso anterior, los mejores resultados se obtienen con redes neuronales de pocas capas ocultas y pequeño número de neuronas. También ocurre lo mismo con los tamaños de traza, los mejores resultados se obtienen con tamaño de conjunto de traza pequeño (se puede ver en el la tabla de resultados completa en el **Anexo 3**). En este caso con arquitecturas de red de menor cantidad de capas ocultas se obtienen resultados igualmente buenos, pero se ha decidido trabajar con el diseño de red que mejor funcione con los datos obtenidos mediante Novelty.

En la **Figura [18]** se ha comparado los valores de utilidad de los datos de test y los valores de utilidad predichos por la red neuronal en el caso de la simulación número tres, que ha sido con la que mejores resultados se han obtenido. Se puede observar que las predicciones de la red neurona (2-6-6-1) son buenas, ajustando la curva de dichas predicciones bastante bien a la curva de los valores reales. En este caso los movimientos siguen el descenso del valor de un sensor de distancia, por lo que, en este caso, al no ser movimientos semi-aleatorios, no se ven claras ambigüedades en los resultados. Principalmente, por la razón de la poca ambigüedad en los datos de entrenamiento, se consigue obtener un mejor aprendizaje de la red neuronal.

En la **Figura [19]** se muestra el error de entrenamiento en la misma simulación que en el caso de la **Figura [18]**. La media del error de entrenamiento es de 0,02448. Los puntos con valores de error de entrenamiento más altos en cada entrenamiento corresponden a las primeras epochs del entrenamiento, y este error va disminuyendo continuamente mientras se realiza el entrenamiento. Como se puede observar, a medida que van avanzando las simulaciones el valor del error de entrenamiento al comienzo de cada entrenamiento va disminuyendo respecto al entrenamiento anterior. A partir de la epoch 10.000 este valor de error de entrenamiento se vuelvo más o menos constante para todos los entrenamientos, sufriendo algunos pequeños picos puntuales.

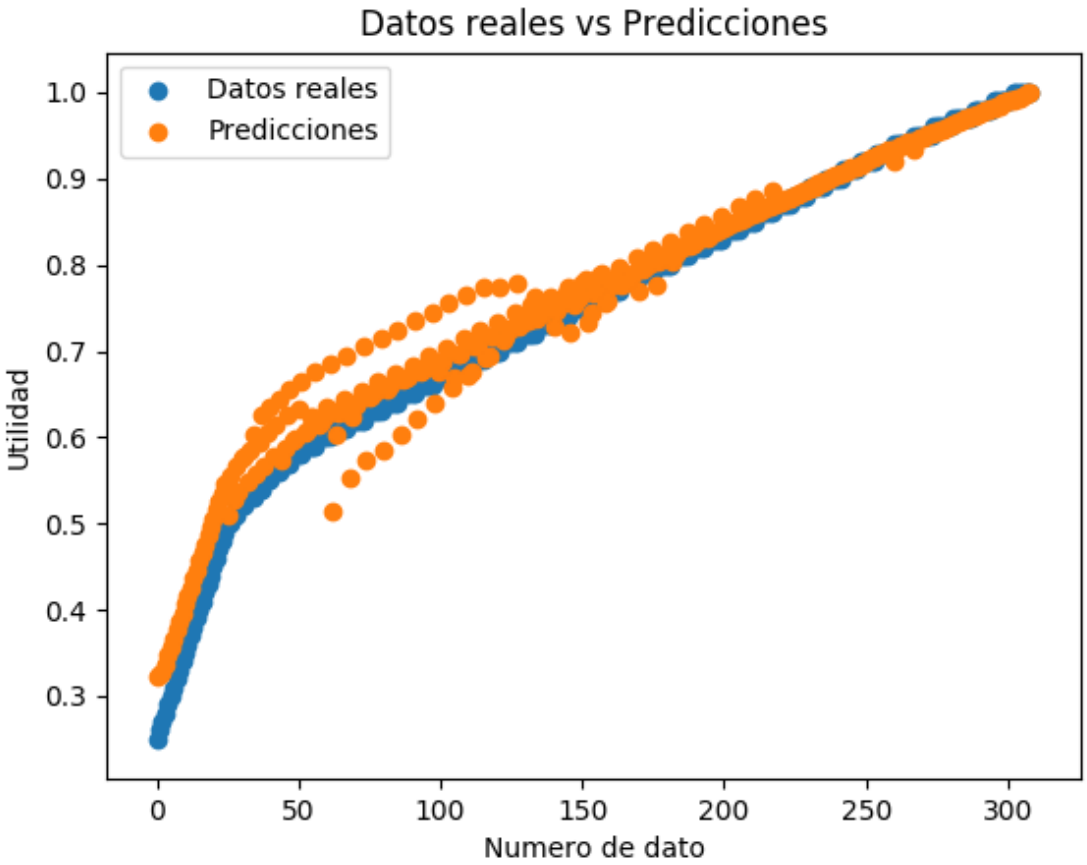


Figura 18: Datos reales vs Predicciones con datos SURs.

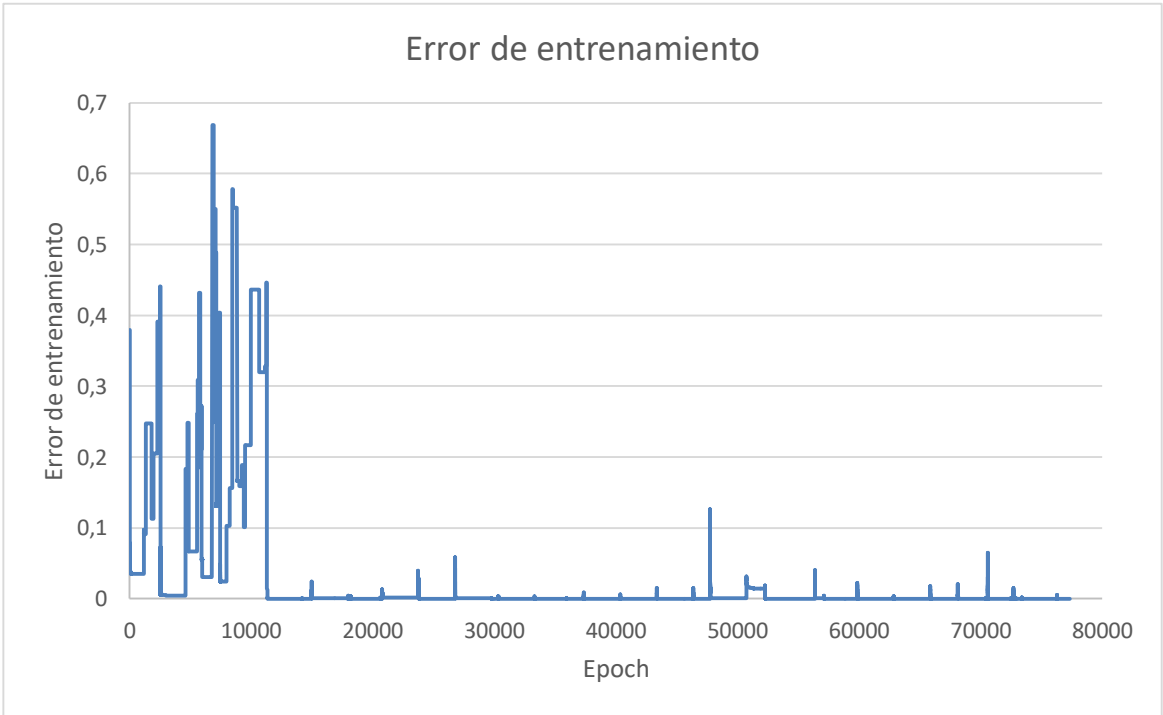


Figura 19: Error de entrenamiento con datos SURs.

8.1.3 Conclusiones

Como se puede observar en las dos figuras (**Figura [16] y Figura [18]**), para realizar un entrenamiento más óptimo de la red neuronal, en este experimento en concreto, es necesario partir de unos datos obtenidos mediante SURs. En el caso de los datos obtenidos mediante Novelty se ve una gran ambigüedad en los resultados, debido a la ambigüedad de los datos, por lo que no se obtiene un resultado correcto de aprendizaje de la red neuronal, ni con una arquitectura de red grande ni con una pequeña, ni tampoco con mayores tamaños de conjunto de traza. Lo que sí que es posible realizar, y es lo que se realiza en la metodología de entrenamiento de VF que se utiliza actualmente en el GII, es utilizar la motivación intrínseca antes de las SURs para guiar el comportamiento robot en las zonas en las que aún no estén definidas las áreas de certeza, y servirse de las trazas obtenidas con esta motivación para actualizar estas áreas de certeza.

También, en la **Figura [17] y Figura [19]** se puede observar las diferencias que hay en los errores de entrenamiento de cada caso. En la primera figura se ve que el error de entrenamiento inicial no va disminuyendo respecto al valor del entrenamiento predecesor debido a la gran ambigüedad que hay en los valores de las trazas de entrenamiento. Esto no es lo que ocurre en la segunda gráfica, en ella se puede observar como a medida que avanzan los entrenamientos, el error de entrenamiento inicial de cada entrenamiento va disminuyendo respecto al valor predecesor hasta llegar a la epoch 10.000 que es cuando empieza a reducir la diferencia entre ellos.

Por otro lado, igual con otra metodología de aprendizaje podría llegarse a obtener mejores resultados utilizando solo Novelty, pero este tema se encuentra fuera del alcance de este proyecto de Trabajo Fin de Máster, por lo que se ha decidido no profundizar en ello.

Con todo esto se puede concluir que la mejor arquitectura de red obtenida, con dos entradas y una salida, es la de dos capas ocultas con seis neuronas cada una (2-6-6-1), y la mejor forma de guiar los movimientos del robot para obtener los datos de entrenamiento de la VF es mediante las SURs.

Una vez obtenidos los resultados de este programa realizado en Python, se pretende implementar el diseño de la arquitectura de la red neuronal, con la que se han obtenido mejores resultados, en el modelo del simulador en 3D, comprobando que los resultados son los mismos o muy similares a los obtenidos de esta forma. En los siguientes apartados se pretende profundizar y explicar más en detalle la parte del proyecto.

8.2 Pruebas On-line

En este apartado se van a explicar el desarrollo del experimento implementado en el Gazebo para el simulador en 3D y las diversas modificaciones en la aplicación de MotivEn para hacer posible aplicar la nueva metodología mencionada en el punto *7. Desarrollo*.

8.2.1 Diseño del experimento en Gazebo

El experimento con el que se va a realizar la implementación de este nuevo método de análisis va a consistir principalmente en un robot Baxter. Se va a utilizar su brazo izquierdo para poder desplazar un objeto desde un punto de la mesa, donde se encentra una pelota, hasta otro punto, donde se va a encontrar una caja (objetivo). Más concretamente el Baxter va a mover su brazo, posicionar el gripper encima de la pelota y acto seguido descender el brazo y coger la pelota. Una vez realizado este movimiento llevará la pelota hasta la caja donde la depositará. En la siguiente figura se puede ver la disposición de los distintos elementos en el espacio del experimento diseñado en Gazebo. Los elementos, el bloque y la caja, van a ir modificando su posición aleatoriamente en la mesa cada vez que se reinicialice el experimento, pero siempre manteniendo una mínima distancia entre ellos de 0,20 m. A parte, las posiciones de estos elementos van a estar acotadas en las direcciones x e y,

teniendo que situarse en la dirección x entre los valores de 0,40 m y 0,75 m, y en dirección y entre 0,05 m y 0,50 m.

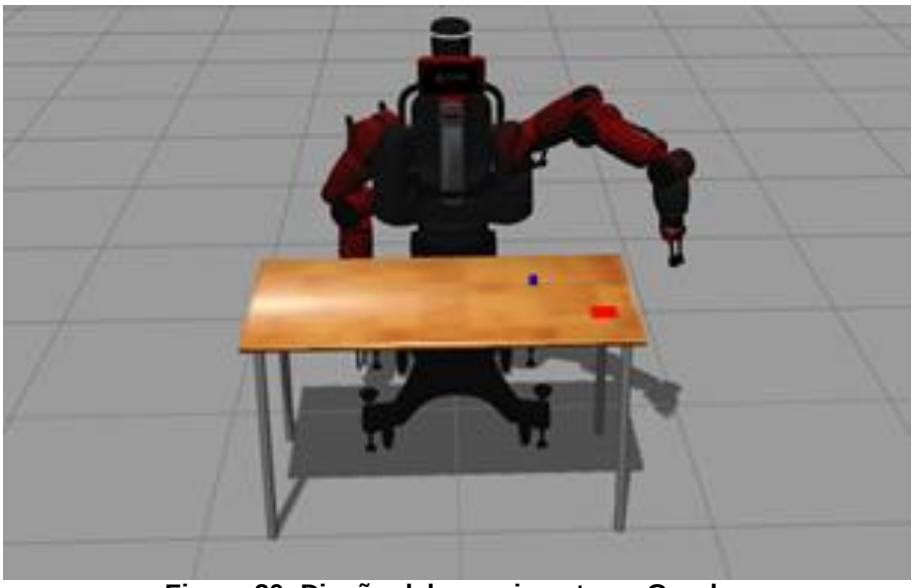


Figura 20: Diseño del experimento en Gazebo.

Para realizar el diseño del experimento en Gazebo mostrado en la **Figura [20]** van a ser necesarios los siguientes elementos:

- Robot Baxter: este robot cuenta con dos brazos móviles diseñados para permitir el manejo de objetos con ellos. Cada brazo cuenta con un gripper como mano lo que le permite interactuar con objetos cogiéndolos y soltándolos. La altura del Baxter se puede variar entre 1,75m y 1,90m y su peso ronda alrededor de los 140 kg. El diseño en Gazebo de este robot no se ha realizado en este proyecto, si no que se puede descargar desde su página oficial ("*http://sdk.rethinkrobotics.com/wiki/SimulatorInstallation"*).

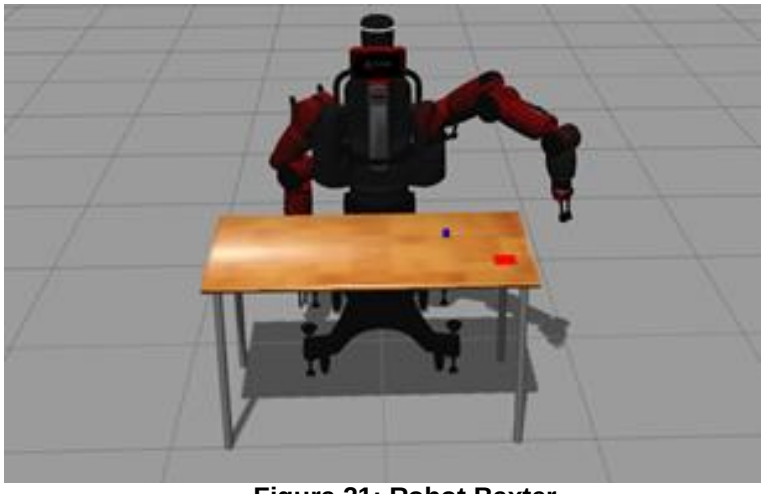


Figura 21: Robot Baxter.

- Mesa: este elemento está formado por cuatro patas de aspecto metálico en las que se apoya una tabla de aspecto de madera de dimensiones 1,5 m x 0,6 m y un espesor de 3 mm. La altura es de un metro y se va a utilizar para apoyar los distintos bloques con los que va a interactuar el robot Baxter. Está diseñada en Gazebo como archivo .sdf.

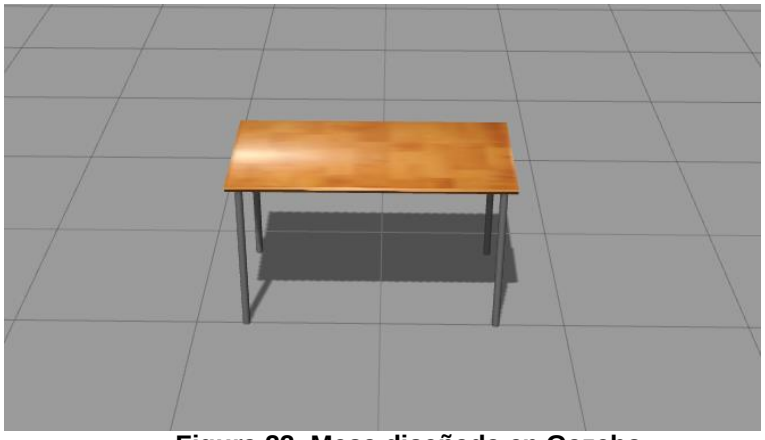


Figura 22: Mesa diseñada en Gazebo

- Pelota: este elemento está diseñada como un cubo de dimensiones 3 mm x 3 mm x 3 mm de color azul. Está diseñada en Gazebo como un archivo. urdf.

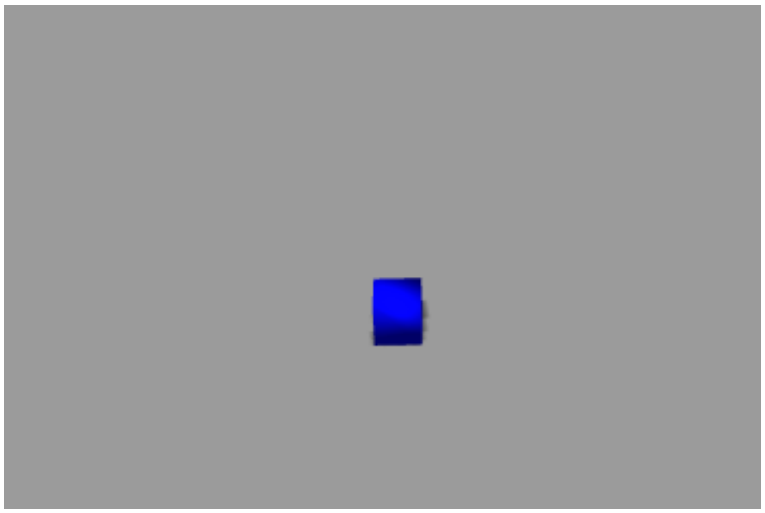


Figura 23: Pelota diseñada en Gazebo.

- Caja: este elemento está diseñado como una lámina de color rojo de dimensiones 0,1 m x 0,1 m y está considerado como objetivo, a donde tiene que llevar el Baxter la pelota para alcanzar el objetivo. También se programa en Gazebo como un archivo. urdf.

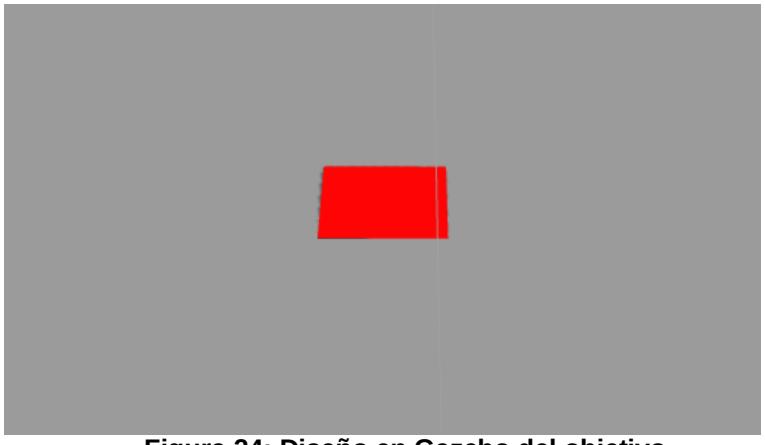


Figura 24: Diseño en Gazebo del objetivo.

Para incorporar, al mundo de Gazebo, todos los elementos que formar parte del experimento, excepto el robot Baxter, se ha implementado un archivo llamado simulator.py,

programado en Python, donde se realiza el Spawn de todos estos bloques. En el **Anexo 1** se puede observar el código mencionado anteriormente.

Posteriormente, para cargar el mundo en Gazebo solo es necesario lanzar el archivo simulator.launch, con el que se genera el mundo del robot Baxter y se cargan los distintos elementos con los que interactúa el robot. El código de este archivo se puede ver en la **Figura [25]**.

```
<?xml version="1.0" encoding="utf-8"?>
<launch>
  <!-- Cargar escenario del robot-->
   <include file="$(find baxter_gazebo)/launch/baxter_world.launch"/>
  <!-- Carga los objetos necesarios para el simulador -->
  <node pkg="simulator" type="simulator.py" name="simulator" />
</launch>
```

Figura 25: Código del archivo simulator.launch

8.2.2 Simulador en 3D

Una vez realizado el diseño del experimento en Gazebo el siguiente paso va a ser la configuración del simulador en 3D. Para esto, lo primero sería llevar a cabo la sensorización de cada elemento del experimento para calcular las futuras posiciones del brazo del Baxter y poder recoger los datos del simulador que, posteriormente, se utilizarán para la aplicación de MotivEn. Para ello se toma como partida los archivos implementados en el TFG de Yeray Méndez Montero con título *"Desarrollo de un módulo de redescripción de modelos de utilidad mediante Deep Learning para robótica cognitiva"*. En este TFG se realizó una aplicación que juntaba la metodología de MotivEn con un simulador en 3D implementado en Gazebo.

Este simulador en 3D implementado en Gazebo forma parte de una aplicación en la que también se incorporan archivos que permiten el cálculo de las futuras posiciones del brazo del robot Baxter, archivos en los que se definen los movimientos a realizar por el robot Baxter, archivos en los que se configuran las distintas motivaciones que pueden guiar el comportamiento robot y en qué orden lo van a hacer, archivos en los que se programa la VF y su entrenamiento a medida que se obtienen las trazas, lo que quiere decir, que esta implementado en la aplicación el sistema MotivEn desarrollado por el GII.

Se van a modificar los archivos de este TFG teniendo en cuenta que ahora solo son necesarios dos sensores, distancia de la pelota al gripper del baxter y distancia de la pelota al objetivo. Con esto va a ser necesario modificar todos los archivos para que se realicen los cálculos, necesarios en la aplicación, a partir de estos dos sensores. Al ser un experimento distinto, la forma de calcular las futuras posiciones del Baxter se va a modificar para que se ajusten a la forma de actuar el robot en este nuevo experimento.

Es necesario comentar que aparte de lo mencionado anteriormente, ha sido necesario corregir algunos errores que daba el código, sobre todo en el relacionado con el Baxter, producto de estar trabajando con una versión distinta del sistema operativo. Por otro lado, también se ha modificado el cálculo de las posiciones candidatas mediante Novelty, porque en este nuevo experimento se obtenían valores incorrectos.

Un punto importante para la realización del experimento es conocer el radio de actuación del brazo del Baxter para solo permitirle el movimiento en esa zona, impidiendo que calcule puntos fuera de ella que nos deriven en errores debido a que el robot no puede llegar a ellos. Esta zona va a estar limitada en la dirección x entre los valores de 0,35 y 0,80 y el en la dirección y entre los valores de 0,00 y 0,55.

Otro tema importante ha sido incorporar dos medidas de seguridad. La primera tiene que ver a la hora de calcular las futuras posiciones del Baxter. Si durante la realización de estos movimientos obtiene una posición que sale de los límites preestablecidos, se recalculará esta futura posición obteniendo una nueva de valor el límite de movimiento del brazo en esa dirección menos la suma de lo que se ha pasado del límite en el anterior cálculo del límite y

una contante de valor 0,05. La segunda tiene que ver con un error que puede darse una vez que tiene cogida la pelota el Baxter. Para corregir la posible caída de la pelota durante los movimientos que realiza el brazo del Baxter se ha forzado a que, una vez que se haya cogido la pelota, la distancia entre el gripper del brazo izquierdo del Baxter y la pelota sea siempre constante y de valor igual a 0. De esta forma, si durante la realización de los movimientos se cae la pelota, el modelo no tendría ningún problema en seguir la simulación como si no pasara nada, lo que evita tener que repetir las simulaciones y poder condicionar en el tiempo necesario de simulación.

Es importante recordar que al principio de cada simulación los elementos del modelo, excepto el baxter y la mesa, van a modificar su posición inicial de forma aleatoria respecto a las simulaciones anteriores. Estas posiciones van a estar dentro de la zona de actuación del brazo de Baxter y la distancia entre ambos nunca podrá ser menor de 0,20 m.

Se ha implementado también un método de obtención de VF a partir de SURs de forma autónoma. Esto se realizará mediante una condición, diseñada por el diseñador, de cambio del tipo de motivación extrínseca.

También se ha implementado en el código la introducción del algoritmo estocástico On-line de optimización Adam, en vez del algoritmo de Descenso de Gradiente, que se utiliza para actualizar los pesos de la red de forma iterativa en función de los datos de entrenamiento. Con este tipo de optimizador, con el que se está extendiendo su uso en Deep Learning debido a su gran potencia de cálculo, se pretende obtener mejores resultados de forma más rápida.

8.2.3 Análisis de resultados

En este apartado se van a exponer los distintos resultados obtenidos del entrenamiento On-line de la VF con el Simulador en 3D implementado en Gazebo. Se van a tratar tres tipos de metodologías para obtener VF a partir de SURs de forma autónoma:

- La primera va a estar diseñada con una condición que va a tener en cuenta el número de iteraciones llevadas a cabo en la simulación. Este primer diseño va a ser bastante burdo pero va a servir para tener una primera impresión rápida del comportamiento del sistema. El diseñador va a poder elegir en qué valor de iteración de la simulación va a pasar en la motivación extrínseca de SURs a VF.
- La segunda va a estar diseñada con una condición que va a tener en cuenta el número de veces que se alcanza el objetivo mediante motivación extrínseca, SURs. Este segundo diseño, aunque sigue siendo bastante simple, permite obtener un método basado en un valor que puede indicar una idea más correcta del buen funcionamiento de la simulación. El diseñador va a poder elegir qué número de veces se alcance el objetivo con las SURs antes de cambiar el tipo de motivación extrínseca, de SURs a VF.
- La tercera va a estar diseñada con una condición que va a tener en cuenta el error medio de entrenamiento de la VF. Este tercer diseño es el más complejo y el que permite tomar una decisión de cambio de motivación extrínseca más óptima y correcta. Esta se centra en que un vez que se alcance un número de veces el objetivo mediante motivación extrínseca, determinada por el diseñador, se va a empezar a realizar el entrenamiento de la VF. La primera vez se realizará con todas las trazas generadas hasta el momento. El valor del error medio de entrenamiento obtenido se va a comparar con un valor umbral, decido por el diseñador. Si este valor de error es mayor que el umbral se seguirá con las SURs como motivación extrínseca. La siguiente vez que se alcance el objetivo se va a volver a entrenar la VF, en este caso solo con la última traza obtenida, y se comparará el error medio de entrenamiento con el valor umbral. Si este error sigue siendo mayor que el umbral se seguirá con las SURs como motivación extrínseca y se volverá a repetir este último proceso hasta conseguir que la media del error de entrenamiento sea menor al umbral. Una vez que se alcanza la condición que

la media del error de entrenamiento sea menor al umbral se utilizará como motivación extrínseca la VF en vez de la SURs.

A la hora de comparar los resultados con las distintas metodologías se va a tener en cuenta el número de iteraciones que tarda en alcanzar el objetivo en cada ejecución del experimento con las distintas pruebas realizadas y las curvas de error de entrenamiento de cada prueba.

Ahora se van a exponer los resultados obtenidos con los distintos métodos para obtener VF a partir de SURs de forma automática:

8.2.3.1 En función del número de iteraciones

Con esta condición se van a realizar cuatro pruebas con distinto valor del número de iteraciones en el cual la motivación extrínseca va a pasar de SURs a VF, estas son 1000 iteraciones, 2000 iteraciones, 3000 iteraciones y 4000 iteraciones. Todos los resultados obtenidos en estas cuatro pruebas se pueden observar en el **Anexo 4**.

En la **Tabla [4]** se pueden observar los distintos valores de entrenamiento medio para las distintas condiciones de valor de iteraciones.

Iteraciones	Error de entrenamiento medio
1000	0,023
2000	0,016
3000	0,002
4000	0,003

Tabla 4: Valores de error de entrenamiento medio.

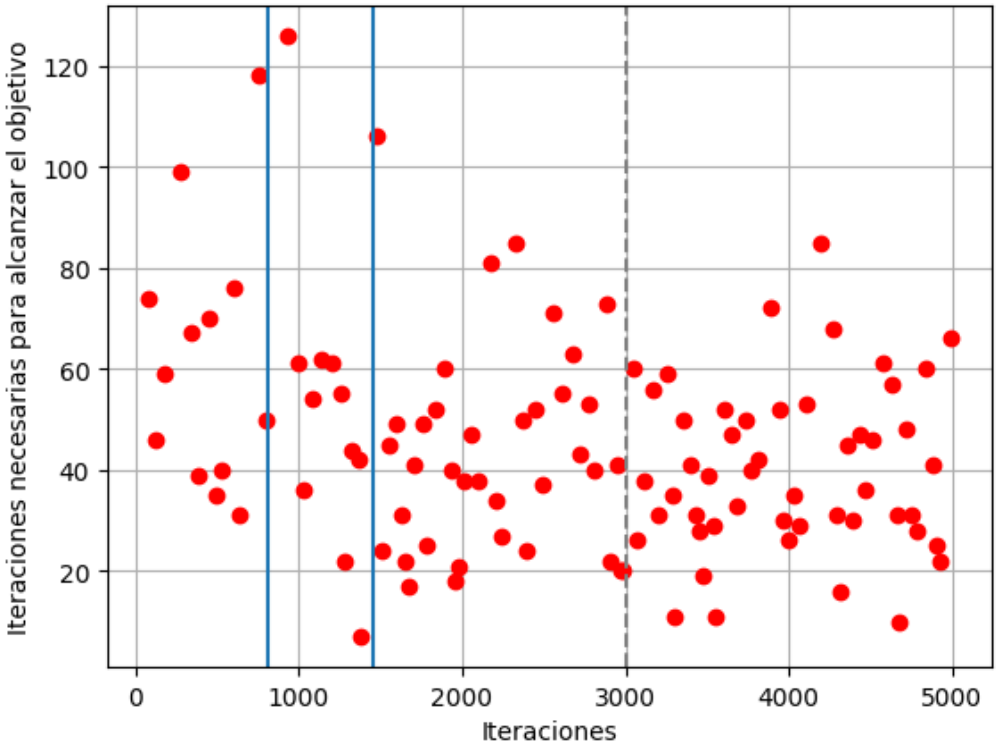


Figura 26: Valores de iteraciones necesarias para alcanzar el objetivo.

Después de analizar los distintos resultados se han obtenido como mejor condición la de 3000 iteraciones con valor medio de error de entrenamiento de 0,002. Centrándonos en la mejor solución, en la **Figura [26]** se puede observar los valores de las iteraciones necesarias para alcanzar los objetivos durante la ejecución de la simulación. Como se puede observar, al principio son necesarias un número mayor de iteraciones para alcanzar el objetivo, porque como se ve en la **Figura [27],** la motivación intrínseca está bastante activa. Con el paso del tiempo se van acotando el número de iteraciones necesarias para llegar al objetivo notando una ligera mejora una vez que se activa la VF como motivación extrínseca.

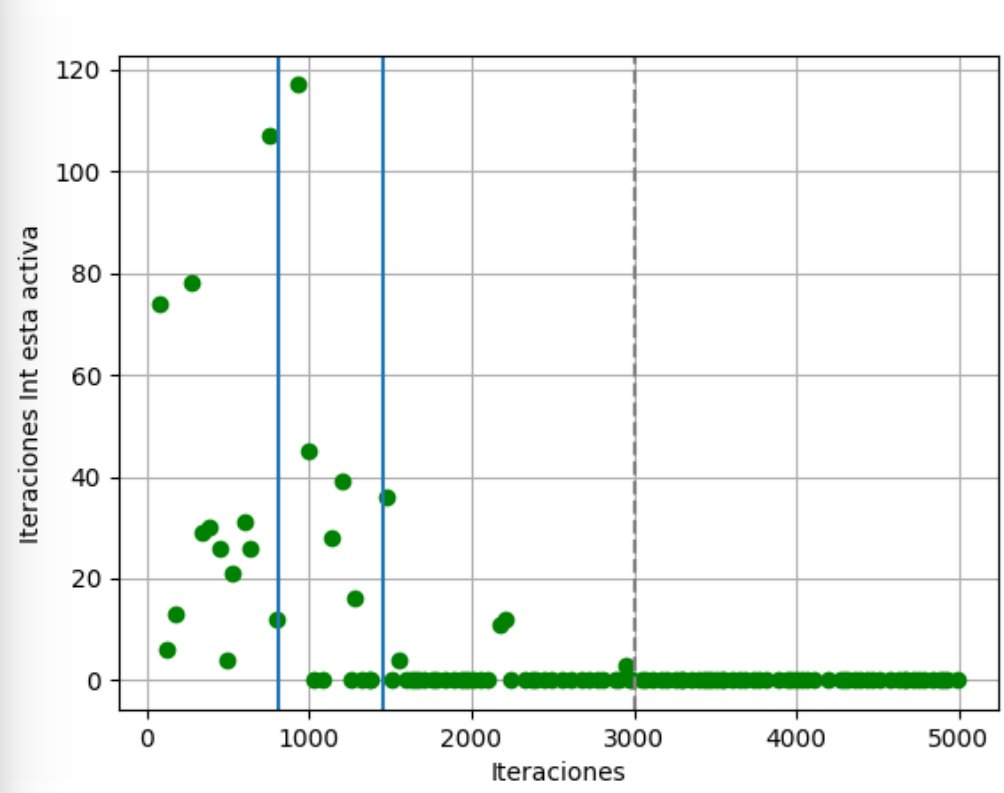


Figura 27: Número de iteraciones en las que la motivación intrínseca esta activa.

8.2.3.2 En función del número de veces que se ha llegado al objetivo con SURs

Con esta condición se van a realizar cuatro pruebas con distinto valor del número de veces que se ha llegado al objetivo con SURs necesarias para pasar de SURs a VF, estas son 20 veces, 30 veces, 50 veces y 70 veces. Todos los resultados obtenidos en estas cuatro pruebas se pueden observar en el **Anexo 4**.

En la **Tabla [5]** se pueden observar los distintos valores de entrenamiento medio para las distintas condiciones de valor de veces que se alcanza el objetivo con SURs.

Veces que se alcanza el objetivo	Error de entrenamiento medio
20	0,024
30	0,012
50	0,005
70	0,003

Tabla 5: Valores de error de entrenamiento medio.

Después de analizar los distintos resultados se han obtenido como mejor condición la de 70 veces alcanzar el objetivo mediante SURs con valor medio de error de entrenamiento de 0,003. Centrándonos en la mejor solución, en la **Figura [28]** se puede observar los valores de

las iteraciones necesarias para alcanzar los objetivos durante la ejecución de la simulación. Como se puede observar al principio son necesarias un número mayor de iteraciones para alcanzar el objetivo, porque como se ve en la **Figura [29],** la motivación intrínseca está bastante activa. Con el paso del tiempo se van acotando el número de iteraciones necesarias para llegar al objetivo notando una ligera mejora una vez que se activa la VF como motivación extrínseca.

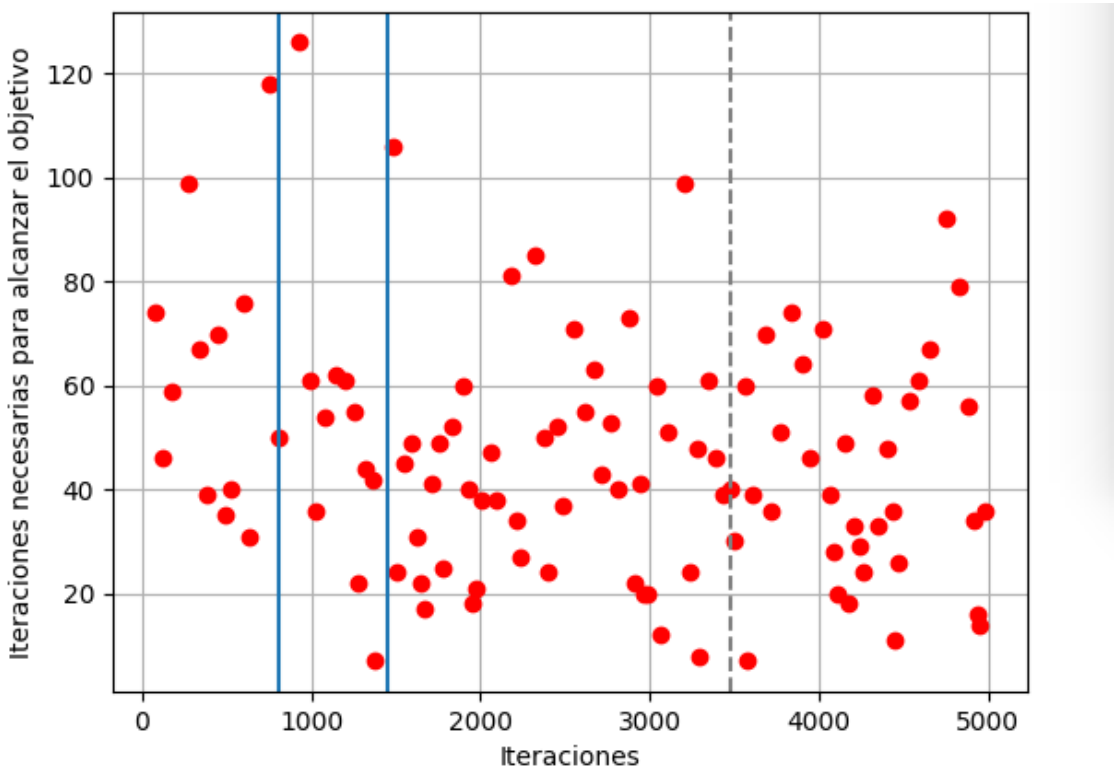


Figura 28: Valores de iteraciones necesarias para alcanzar el objetivo.

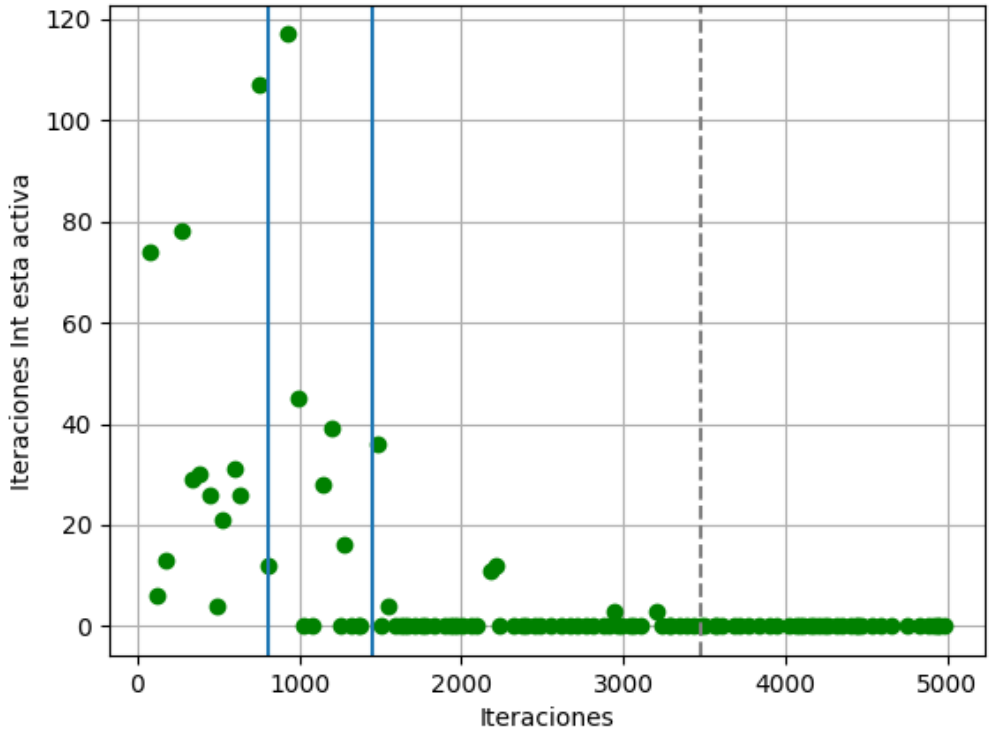


Figura 29: Número de iteraciones en las que la motivación intrínseca esta activa.

8.2.3.3 En función del error de entrenamiento de la VF

Esta condición es la más óptima de todas las estudiadas. Con ella se puede obtener un método de obtener VF a partir de SURs de forma automática. Una vez se alcance un número de veces el objetivo, mediante SURs, se empezara a entrenar la VF. Este valor se va a dimensionar en función de las pruebas realizadas anteriormente. Como el mejor valor de error de entrenamiento medio se dio con 70 veces alcanzar el objetivo se utilizará este valor como condición para empezar a entrenar la VF. Después cambio de SURs a VF se producirá en el momento en el que el la media del error de entrenamiento sea menor a un umbral que considere el diseñador.

Con esta metodología se va a realizar una serie de pruebas con distinto umbral de error de entrenamiento medio, estos son 0.01, 0.008, 0.005 y 0.003. Todos los resultados obtenidos se pueden observar en el **Anexo 4**.

En la **Tabla [6]** se pueden observar los distintos valores de entrenamiento medio para las distintas condiciones de valor de umbral de error medio de entrenamiento.

Umbral error medio entrenamiento	Error de entrenamiento medio
0,01	0,0029
0,008	0,0028
0,005	0,0023
0,003	0,0024

Tabla 6: Valores de error de entrenamiento medio.

Después de analizar los distintos resultados se han obtenido como mejor condición la de un umbral de 0,005 con valor medio de error de entrenamiento de 0,0023. Centrándonos en la mejor solución, en la **Figura [30]** se puede observar los valores de las iteraciones necesarias para alcanzar los objetivos durante la ejecución de la simulación. Como se puede observar al principio son necesarias un número mayor de iteraciones para alcanzar el objetivo, porque como se ve en la **Figura [31],** la motivación intrínseca está bastante activa. Con el paso del tiempo se van acotando el número de iteraciones necesarias para llegar al objetivo notando una ligera mejora una vez que se activa la VF como motivación extrínseca.

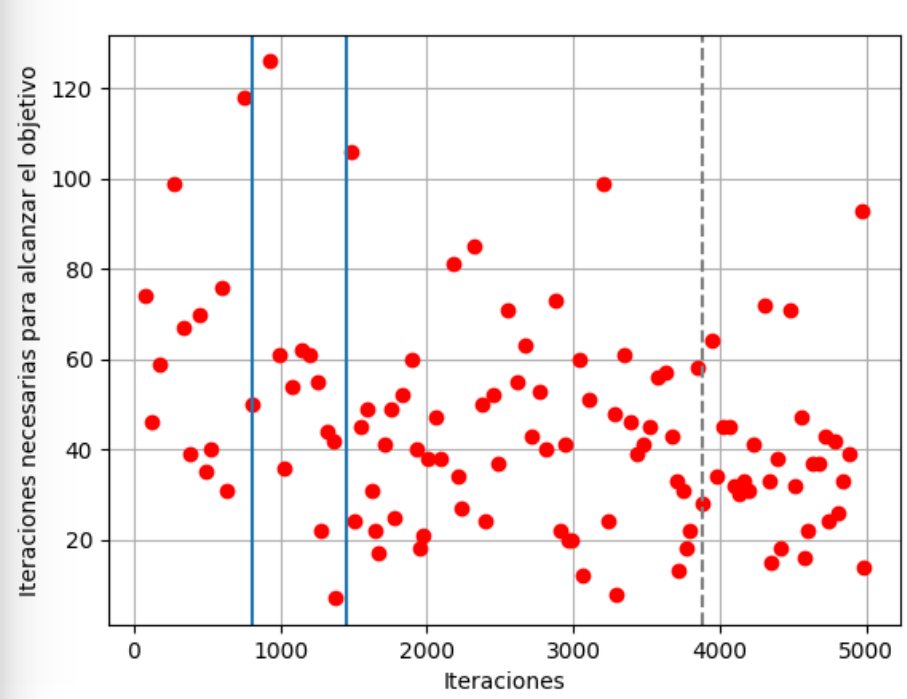


Figura 30: Valores de iteraciones necesarias para alcanzar el objetivo.

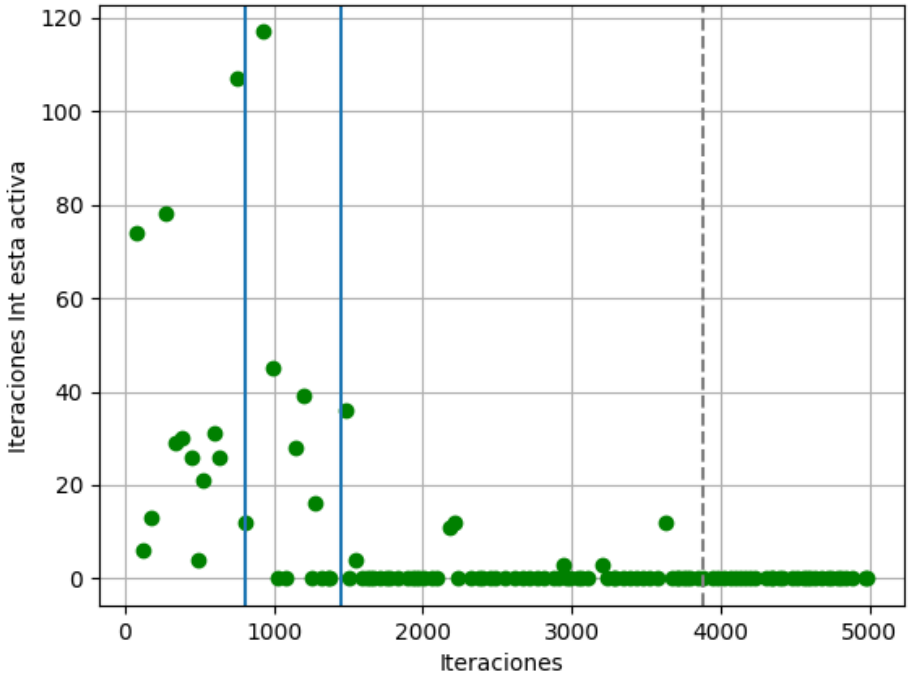


Figura 31: Número de iteraciones en las que la motivación intrínseca esta activa.

9 CONCLUSIONES Y TRABAJO FUTURO

9.1 Conclusiones

En el presente proyecto se ha conseguido llevar a cabo el principal objetivo planteado originalmente: diseñar y estudiar la viabilidad de un método de aprendizaje on-line de modelos de utilidad para MotivEn. Con este método se logra un avance notable en el MDB, ya que ahora el aprendizaje de los modelos de utilidad se puede llevar a cabo de forma autónoma y en tiempo real.

Para lograr este objetivo se ha implementado de forma satisfactoria el algoritmo estocástico On-line de optimización Adam en vez del algoritmo de Descenso de Gradiente que se utilizaba originalmente. Con este tipo de optimizador, con el que se está extendiendo su uso en Deep Learning debido a su gran potencia de cálculo, se han obtenido mejores resultados de forma más rápida.

Para implementar el algoritmo Adam se ha estudiado el campo del aprendizaje en tiempo real de redes neuronales mediante Tensor Flow. En concreto, se ha integrado la API Keras para el dimensionamiento y entrenamiento de las redes neuronales, en MotivEn. Esto permite tener el código del simulador en un lenguaje universal como Python, lo que permite realizar modificaciones a otros usuarios que deseen utilizarlo de forma sencilla.

Se ha implementado un experimento representativo del campo de la robótica cognitiva en el simulador 3D Gazebo mediante ROS, y sobre este se ha estudiado el método de aprendizaje propuesto. Como se puede observar en la sección de resultados, la gestión del uso de SURs y VF en base al error de entrenamiento es la más óptima, y con la que se ha obtenido mejores resultados. Con esta condición se ha diseñado un método con el cual se empieza a guiar el comportamiento robot mediante VF cuando el error medio de entrenamiento descienda de un valor que considere el diseñador.

Por lo tanto, se puede decir que se ha cumplido el objetivo principal de este proyecto, se ha desarrollado un esquema operativo para el aprendizaje on-line de modelos de utilidad en entornos realistas de robótica autónoma, que esté integrado dentro del mecanismo motivacional MotivEn desarrollado por el Grupo Integrado de Ingenería de la UDC.

9.2 Trabajo futuro

Los procedimientos realizados han permitido la consecución de los objetivos planteados inicialmente. Sin embargo, al tratarse de un proyecto de robótica cognitiva centrado en el campo del aprendizaje abierto, todavía se podrían realizar diferentes mejoras y experimentos que podrían contribuir a mejorar el sistema motivacional:

- Comportamiento del robot. Mejorar el desenvolvimiento de determinadas acciones realizadas por el robot Baxter en Gazebo, ya que a veces el agarre de objetos o su posicionamiento en su sitio no se hacían de la forma más correcta.
- Comprobar el experimento desarrollado en un entorno real. Comprobar el correcto comportamiento del simulador haciendo las mismas pruebas con el robot real.
- Aprendizaje en paralelo de los modelos de utilidad desde el inicio. Sería muy interesante realizar pruebas sobre el entrenamiento de la SURs y de la VF en paralelo desde el inicio, para poder seleccionar el momento más óptimo en el cual pasar de utilizar unas a utilizar otras. Esto implica el desarrollo de un sistema zonas de certeza para la VF.
- Probar la aplicación desarrollada en otros experimentos para comprobar su correcto funcionamiento.

10 BIBLIOGRAFÍA

[1] G. I. de Ingeniería, “Grupo integrado de ingeniería,” http://www.gii.udc.es/, 2014.

[2] DREAM, “Deferred restructuring of experience in autonomous machines,” http://www.robotsthatdream.eu/, 2015.

[3] Alejandro Romero, Abraham Prieto, Francisco Bellas and Richard J. Duro, “Simplifying the Creation and Management of Utility Models in Continuous Domains for Cognitive Robotics”, 2018.

[4] José Antonio Becerra, Alejandro Romero, Francisco Bellas and Richard J. D, “Motivational Engine and Long Term Memory Coupling within a Cognitive Architecture for Life-long Open-ended Learning”, 2019.

[5] http://wiki.ros.org/ROS/Introduction, 2018.

[6] Gazebo, “Gazebo,” http://gazebosim.org/, 2014.

[7] R. Robotics, “Baxter research robot,” http://sdk.rethinkrobotics.com/wiki/Home, 2015.

[8] ROBOLOGS, “Gazebo simulator: simular un robot nunca fue tan fácil,” https://robologs.net/2016/06/25/gazebo-simulator-simular-un-robot-nunca-fue-tan-facil/, 2016.

[9] https://www.tensorflow.org/

[10] A. Prieto, A. Romero, F. Bellas, R. Salgado and R.J. Duro, “Introducing Separable Utility Regions in a Motivational Engine for Cognitive Developmental Robotics”, 2018.

[11] A. Romero, F. Bellas, A. Prieto, R. J. Duro, A Re-description Based Developmental Approach to the Generation of Value Functions for Cognitive Robots.

UNIVERSIDADE DA CORUÑA

Escola Politécnica Superior

TRABAJO FIN DE MÁSTER

CURSO 2018/19

ESTUDIO DEL APRENDIZAJE EN TIEMPO REAL DE MODELOS DE UTILIDAD EN ROBÓTICA COGNITIVA

Máster en Ingeniería Industrial

Documento

ANEXOS

ÍNDICE

1 ARCHIVO SIMULATOR.PY

```
import rospy
import rospkg
import baxter_interface
from gazebo_msgs.srv import SpawnModel
from gazebo_msgs.srv import DeleteModel
from gazebo_msgs.msg import ModelStates
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import SetModelState
from geometry_msgs.msg import Twist
from geometry_msgs.msg import Pose
from geometry_msgs.msg import PoseStamped
from geometry_msgs.msg import Quaternion
from geometry_msgs.msg import Point
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
from tf.transformations import quaternion_from_euler
from std_msgs.msg import Header
from std_msgs.msg import Empty
from baxter_core_msgs.srv import SolvePositionIK
from baxter_core_msgs.srv import SolvePositionIKRequest
from baxter_interface import CHECK_VERSION
from random import uniform

MODEL_PATH = "/home/alvaro/catkin_ws/src/simulator/models/"
BLOCK = "block"
GOAL = "goal"
TABLE = "table"

BAXTER_POSITION_X_MIN = 0.3
BAXTER_POSITION_X_MAX = 0.9
BAXTER_POSITION_Y_MIN = -0.2
BAXTER_POSITION_Y_MAX = 0.5

def diference(x, y):
    dif = 0
    if x > 0 and y < 0:
        dif = x + y
    elif x < 0 and y > 0:
        dif = y + x
    else:
        dif = abs(x - y)

    return dif

def generate_random_pose_for_objects():

    x = round(uniform(0.4,0.74),4)
    y = round(uniform(-0.1,0.5),4)
    x1 = round(uniform(0.4,0.75),4)
    y1 = round(uniform(-0.1,0.5),4)

    while (diference(x, x1) < 0.2) and (diference(y, y1) < 0.2):

            x = round(uniform(0.4,0.74),4)
            y = round(uniform(-0.1,0.5),4)
            x1 = round(uniform(0.4,0.75),4)
            y1 = round(uniform(-0.1,0.5),4)

    return (x, y), (x1, y1)

(a_x, a_y), (b_x, b_y) = generate_random_pose_for_objects()

def delete_gazebo_models():
    try:
        delete_model = rospy.ServiceProxy('/gazebo/delete_model', DeleteModel)
        delete_model("table")
        delete_model("block")
        delete_model("goal")
        rospy.sleep(1.0)
```

```
    except rospy.ServiceException, e:
        rospy.loginfo("Delete Model service call failed: {0}".format(e))


def load_gazebo_models():
    world_frame = "world"
    block_pose = Pose(position=Point(x=a_x,y=a_y, z=1.2))
    table_pose = Pose(position=Point(x=0.6, y=-0.05, z=-0.17),
                      orientation=Quaternion(x=0,y=0,z=-1,w=1))
    goal_pose = Pose(position=Point(x=b_x, y=b_y, z=1.2))
    rospy.loginfo('***Cargando modelos.....')

    # Load Mesa SDF
    table_xml = ''
    with open (MODEL_PATH + "table/model.sdf", "r") as table_file:
        table_xml=table_file.read().replace('\n', '')

    # Load Bloque URDF
    block_xml = ''
    with open (MODEL_PATH + "block/model.urdf", "r") as block_file:
        block_xml=block_file.read().replace('\n', '')

    # Load Goal URDF
    goal_xml = ''
    with open (MODEL_PATH + "goal/model.urdf", "r") as goal_file:
        goal_xml=goal_file.read().replace('\n', '')


    #Spawn Mesa SDF
    rospy.wait_for_service('/gazebo/spawn_sdf_model')

    try:
        spawn_sdf = rospy.ServiceProxy('/gazebo/spawn_sdf_model', SpawnModel)
        spawn_sdf("table", table_xml, "/",table_pose, world_frame)
    except rospy.ServiceException, e:
        rospy.logerr("Spawn SDF service call failed: {0}".format(e))

    #Spawn Bloque y Goal RTDF
    rospy.wait_for_service('/gazebo/spawn_urdf_model')

    try:
        spawn_urdf = rospy.ServiceProxy('/gazebo/spawn_urdf_model', SpawnModel)
        spawn_urdf("block", block_xml, "/", block_pose, world_frame)
        spawn_urdf("goal", goal_xml, "/", goal_pose, world_frame)
    except rospy.ServiceException, e:
        rospy.logerr("Spawn URDF service call failed: {0}".format(e))

    rospy.loginfo('***Entorno cargado.')

def main():
    rospy.loginfo('***Inicializando nodo...')
    rospy.init_node("simulator")
    rospy.Rate(100)
    load_gazebo_models()
    rospy.on_shutdown(delete_gazebo_models)
    rospy.wait_for_message("/robot/sim/started", Empty)
    limb = 'left'
    left_arm = baxter_interface.Limb(limb)
    left_gripper = baxter_interface.Gripper(limb)
    rospy.loginfo("***Colocando baxter en posicion de inicio...")
    rospy.loginfo("***Obteniendo estado del baxter... ")
    robot_state = baxter_interface.RobotEnable(baxter_interface.CHECK_VERSION)
    robot_state.state().enabled
    rospy.loginfo("***Activando robot... ")
    robot_state.enable()
    left_arm.move_to_neutral()
    left_gripper.close()
    rospy.spin()

if __name__ == '__main__':
    main()
```

2 PROGRAMA PRUEBAS OFF-LINE

```
from tensorflow import keras
import pandas as pd
import numpy as np
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, InputLayer
from tensorflow.keras.wrappers.scikit_learn import KerasRegressor
import matplotlib.pyplot as plt
import tensorflow.keras.initializers as initializer
from tensorflow.keras.callbacks import TensorBoard, EarlyStopping
from tensorflow.keras.models import model_from_json
from sklearn.model_selection import train_test_split, KFold, cross_val_score, StratifiedKFold
import time
import h5py
import xlsxwriter


normal = initializer.random_normal(stddev=0.1, seed=101)
callback=keras.callbacks.EarlyStopping(monitor='loss', min_delta=0, patience=25, verbose=0,
mode='min')

def lectura_train():
    #Lectura de datos de entrenamiento
    lectura = pd.read_csv("/home/alvaro/Escritorio/PROGRAMA/DATOS/TRAZAS_ENTRENAMIENTO.csv",
sep=";")
    X = lectura.iloc[0:,1:3].values.astype(float)
    Y = lectura.iloc[0:,3:4].values.astype(float)
    return X,Y

def lectura_test():
    #Lectura de datos de test
    lectura = pd.read_csv("/home/alvaro/Escritorio/PROGRAMA/DATOS/TRAZAS_TEST.csv", sep=";")
    X = lectura.iloc[0:,1:3].values.astype(float)
    Y = lectura.iloc[0:,3:4].values.astype(float)
    return X,Y

def ordenados(p,yt,iteracion):
    ordenado = []

    for l in range(len(p)):
        vx.append((yt[l],p[l]))

    ordenado = sorted(vx)

    store_info_6(ordenado,iteracion)

def baseline_model():
    #Se configura la red neuronal
    model = Sequential()

    #Se añaden las distintas capas que forman la arquitectura de la red
    model.add(Dense(6, input_dim=2, kernel_initializer=normal, bias_initializer='ones',
activation='relu'))
    model.add(Dense(6, kernel_initializer=normal, bias_initializer='ones', activation='relu'))
    model.add(Dense(1, kernel_initializer=normal,bias_initializer='ones', activation='relu'))

    # Se compila el modelo
    model.compile(loss='mse', optimizer='adam' , metrics=['mse'])

    return model


def train (xtrain,xtest,ytrain,ytest,numero_trazas,iteracion):
    #Se realiza el entrenamiento.
    predictions = []
    evaluacion = []
    datos = []
    model = baseline_model()
    cont = 0
    suma = 0
    punto_inicio = 0
    xtrain2 = []
    ytrain2 = []
    xtrain3 = []
    ytrain3 = []
    vector_error = []
```

```
    vector_tiempo = []
    vector_diferencia = []

    for i in ytrain:
        if i != 1.0:
            cont += 1
        if i == 1.0:
            cont += 1
            datos.append(cont)
            cont = 0

    for i in datos:
        suma = suma + i

    n = numero_trazas
    inicio = 0
    valor = 0

    for i in range(len(datos)):
        numero_datos = 0

        for j in datos[i:n]:
            numero_datos=numero_datos + j

        if inicio == 0:
            inicio = 1
            punto_inicio = 0
        else:
            punto_inicio = punto_inicio + datos[(i-1)]

        cont = 0

        for k in xtrain[punto_inicio:(punto_inicio+numero_datos)]:
            xtrain2.append(k)

        for k in ytrain[punto_inicio:(punto_inicio+numero_datos)]:
            ytrain2.append(k)

        n += 1

        for f in range(len(xtrain2)):
            for l in xtrain2[f]:
                xtrain3.append(l)
            ytrain3.append(ytrain2[f])
            cont += 1
            if ytrain2[f] == 1:
                xtrain3 = np.reshape(xtrain3, (cont, 2))
                ytrain3 = np.reshape(ytrain3,(cont,1))
                t0 = time.time()
                history = model.fit(xtrain3, ytrain3, epochs=3000, callbacks=[callback],
batch_size=len(ytrain3),
                                    verbose=0,validation_split = 0.0)
                t1 = time.time()
                t = t1-t0
                vector_tiempo.append(t)
                valor += 1
                for i in history.history['loss']:
                    vector_error.append(i)
                store_info_3(xtrain3,ytrain3, valor)
                store_info_2(vector_tiempo,iteracion)
                store_info_1(vector_error, iteracion)
                xtrain3 = []
                ytrain3 = []
                cont = 0

        xtrain2 = []
        ytrain2 = []

        if i == (len(datos)-numero_trazas):
            break

    evaluacion = model.evaluate(xtest, ytest)
    predictions = model.predict(xtest,verbose=0)

    for i in range(len(ytest)):
        diferencia = abs(ytest[i]-predictions[i])
        vector_diferencia.append(diferencia)
```

```
    store_info_4(evaluacion, iteracion)
    store_info_5(predictions, ytest, vector_diferencia, iteracion)

    return predictions

def store_info_1(error,iteracion):

    file_name = 'ERROR_' + '{0}'.format(iteracion) + ".xlsx"
    sheet_name = 'Datos modelo'

    df = pd.DataFrame()
    df['Error'] = error

    writer = pd.ExcelWriter(file_name,engine='xlsxwriter')
    df.to_excel(writer,sheet_name=sheet_name)
    writer.save()

def store_info_2(tiempo, iteracion):

    file_name = 'TIEMPO_' + '{0}'.format(iteracion) + ".xlsx"
    sheet_name = 'Datos modelo'
    vector=[]

    for i in range(len(tiempo)):
        vector.append(tiempo[i])

    df = pd.DataFrame()
    df['Tiempo'] = vector

    writer = pd.ExcelWriter(file_name, engine='xlsxwriter')
    df.to_excel(writer, sheet_name=sheet_name)
    writer.save()

def store_info_3(xtrain3,ytrain3, cont):

    file_name = 'VALORES' + "_" + 'numero_{0}'.format(cont) + ".xlsx"
    sheet_name = 'Datos modelo'
    sensor_data_1 = []
    sensor_data_2 = []
    utility = []
    for i in range(len(ytrain3)):
        sensor_data_1.append(xtrain3[i][0])
        sensor_data_2.append(xtrain3[i][1])
        utility.append(ytrain3[i][0])

    df = pd.DataFrame()
    df['Sensor Data 1'] = sensor_data_1
    df['Sensor Data 2'] = sensor_data_2
    df['Utility'] = utility

    writer = pd.ExcelWriter(file_name,engine='xlsxwriter')
    df.to_excel(writer,sheet_name=sheet_name)
    writer.save()

def store_info_4(error, iteracion):

    file_name = 'VALIDACION_' + '{0}'.format(iteracion) + ".xlsx"
    sheet_name = 'Datos modelo'
    vector = []

    for i in range(len(error)):
        vector.append(error[i])

    df = pd.DataFrame()
    df['Error'] = vector

    writer = pd.ExcelWriter(file_name, engine='xlsxwriter')
    df.to_excel(writer, sheet_name=sheet_name)
    writer.save()
def store_info_5(predictions, ytest, diferencia, iteracion):

    file_name = 'PREDICCIONES_' + '{0}'.format(iteracion) + ".xlsx"
    sheet_name = 'Datos modelo'
    vector = []
    y=[]
    dif=[]
```

```
    for i in range(len(ytest)):
        vector.append(predictions[i][0])
        y.append(ytest[i][0])
        dif.append(diferencia[i][0])

    df = pd.DataFrame()
    df['Prediccion'] = vector
    df['utilidad'] = y
    df['diferencia'] = dif

    writer = pd.ExcelWriter(file_name, engine='xlsxwriter')
    df.to_excel(writer, sheet_name=sheet_name)
    writer.save()

def store_info_6(ordenado, iteracion):

    file_name = 'PREDICCIONES ORDENADAS' + "_" + 'numero_{0}'.format(iteracion) + ".xlsx"
    sheet_name = 'Datos modelo'

    datos_1=[]
    datos_2=[]

    for i in range(len(ordenado)):
        datos_1.append(ordenado[i][0][0])
        datos_2.append(ordenado[i][1][0])

    df = pd.DataFrame()
    df['Salida'] = datos_1
    df['Prediccion'] = datos_2

    writer = pd.ExcelWriter(file_name,engine='xlsxwriter')
    df.to_excel(writer,sheet_name=sheet_name)
    writer.save()

print("Empieza el entrenamiento.")

numero_trazas=1

for i in range(10):
    xtrain,ytrain=lectura_train()
    xtest,ytest=lectura_test()
    pred=train(xtrain,xtest,ytrain,ytest,numero_trazas,i)

    ordenados(pred,ytest,i)
```

3 RESULTADOS PRUEBAS OFF-LINE

3.1 Resultados con datos Novelty

3.1.1 Tamaño de traza de uno

En estas tablas se muestra el error de test promedio de cada simulación para los distintos diseños de arquitectura de red.

Nº SIM	2-6-1	2-8-1	2-10-1	2-6-6-1	2-8-8-1
1	0,148504	0,148501	0,139843	0,140014	0,148504
2	0,148503	0,152175	0,142902	0,156966	0,121364
3	0,148510	0,148503	0,128669	0,147692	0,187506
4	0,145690	0,152026	0,139842	0,120588	0,131289
5	0,148503	0,148500	0,139842	0,140428	0,187829
6	0,148503	0,152443	0,153534	0,127126	0,245740
7	0,148504	0,152161	0,144634	0,133120	0,181049
8	0,148277	0,151707	0,116642	0,139111	0,190992
9	0,148503	0,148502	0,139843	0,123302	0,126531
10	0,146016	0,152857	0,139842	0,132658	0,139511
Media	**0,147951**	**0,150738**	**0,138559**	**0,1361**	**0,166031**
Desviación	**1,11E-05**	**3,41E-05**	**0,000866**	**0,001119**	**0,013911**

Nº SIM	2-10-10-1	2-6-8-6-1	2-5-10-5-1	2-8-10-8-1
1	0,183400	0,132698	0,245741	0,206944
2	0,245653	0,189767	0,131137	0,245665
3	0,139627	0,184293	0,184298	0,123864
4	0,246435	0,148026	0,184295	0,245212
5	0,245669	0,121439	0,139870	0,147207
6	0,140388	0,245464	0,139841	0,124320
7	0,125906	0,140983	0,182093	0,127766
8	0,139597	0,130268	0,120890	0,182451
9	0,246342	0,245674	0,125461	0,120523
10	0,560721	0,120874	0,126825	0,049928
Media	**0,227374**	**0,165949**	**0,158045**	**0,157388**
Desviación	**0,147712**	**0,020919**	**0,01445**	**0,034695**

3.1.1 Tamaño de traza de dos y de cinco respectivamente

Nº SIM	2-6-6-1 (tamaño traza 2)	2-6-6-1 (tamaño de traza 5)
1	0,13984	0,13347
2	0,14850	0,13200
3	0,12918	0,24576
4	0,13705	0,24565
5	0,13984	0,13987
6	0,13984	0,13984
7	0,13695	0,16216
8	0,18416	0,13997
9	0,13216	0,24567
10	0,13265	0,13895
Media	**0,14202**	**0,17233**
Desviación	**0,00223**	**0,02366**

3.2 Resultados con datos SURs

3.2.1 Tamaño de traza de uno

Nº SIM	2-6-1	2-8-1	2-10-1	2-6-6-1	2-8-8-1
1	0,026605	0,026608	0,054709	0,053455	0,025370
2	0,026605	0,026606	0,060418	0,057024	0,031114
3	0,026603	0,026606	0,026609	0,022041	0,028300
4	0,026605	0,026553	0,026607	0,053567	0,034053
5	0,026601	0,026608	0,027667	0,042140	0,026607
6	0,026603	0,026607	0,026612	0,026329	0,026174
7	0,026605	0,026606	0,027542	0,041974	0,029004
8	0,026606	0,026576	0,056220	0,028692	0,028844
9	0,026605	0,026607	0,053462	0,022544	0,026606
10	0,026596	0,026608	0,026722	0,041230	0,041824
Media	**0,026604**	**0,026599**	**0,038657**	**0,038899**	**0,02979**
Desviación	**7,54E-11**	**3,18E-09**	**0,002081**	**0,001595**	**0,000221**

Nº SIM	2-10-10-1	2-6-8-6-1	2-5-10-5-1	2-8-10-8-1
1	0,032575	0,030391	0,053462	0,036113
2	0,026758	0,027809	0,042586	0,036541
3	0,021529	0,040627	0,053462	0,038516

Nº SIM	2-10-10-1	2-6-8-6-1	2-5-10-5-1	2-8-10-8-1
4	0,037748	0,038519	0,029426	0,040444
5	0,030588	0,036239	0,039506	0,042159
6	0,026609	0,030071	0,038334	0,045404
7	0,028497	0,041204	0,042490	0,040601
8	0,023842	0,043215	0,031677	0,039181
9	0,041858	0,026000	0,026624	0,032899
10	0,029011	0,042039	0,050401	0,041754
Media	**0,029901**	**0,035611**	**0,040797**	**0,039361**
Desviación	**0,000342**	**0,000376**	**0,00084**	**0,000114**

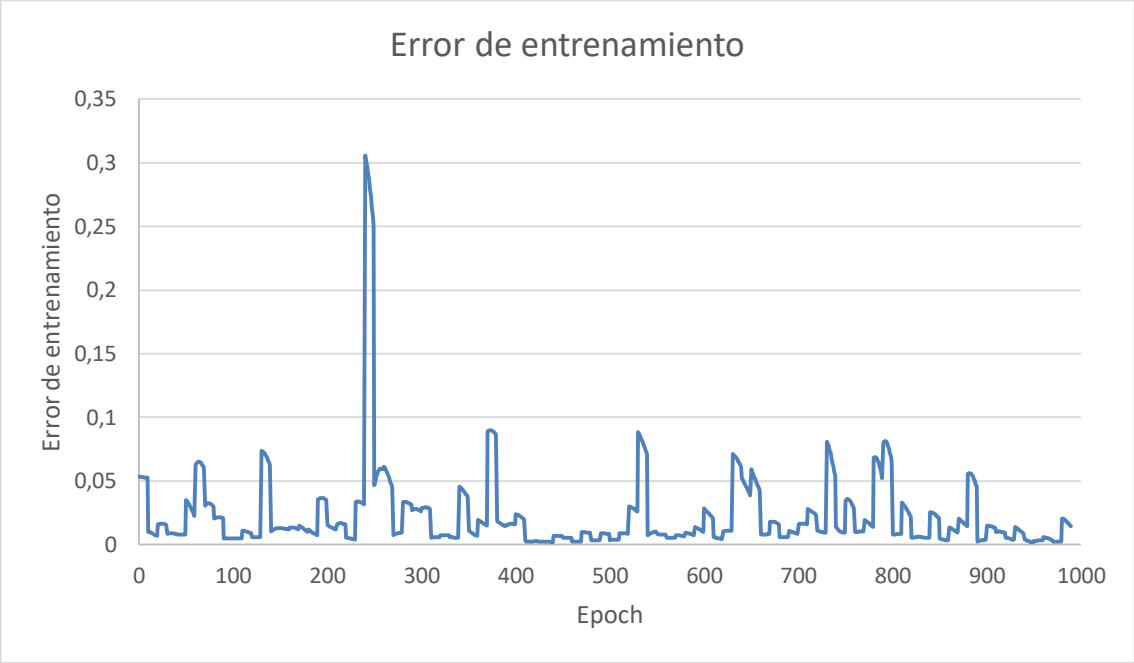
3.2.1 Tamaño de traza de dos y de cinco respectivamente

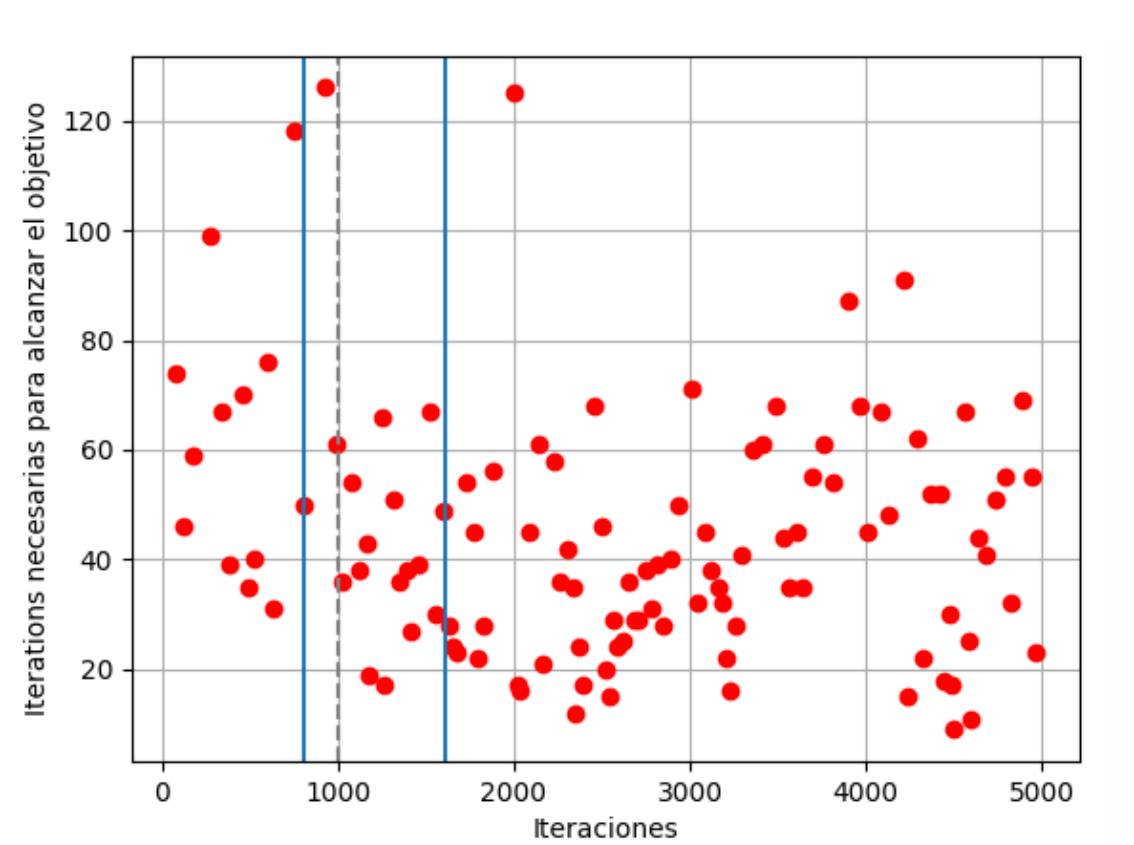
Nº SIM	2-6-6-1 (tamaño traza 2)	2-6-6-1 (tamaño traza 5)
1	0,05346	0,05346
2	0,02742	0,02661
3	0,05346	0,02945
4	0,05326	0,02662
5	0,03345	0,02995
6	0,02775	0,02588
7	0,04606	0,03040
8	0,04350	0,05346
9	0,05326	0,04547
10	0,02952	0,02661
Media	**0,04212**	**0,03479**
Desviación	**0,00118**	**0,00116**

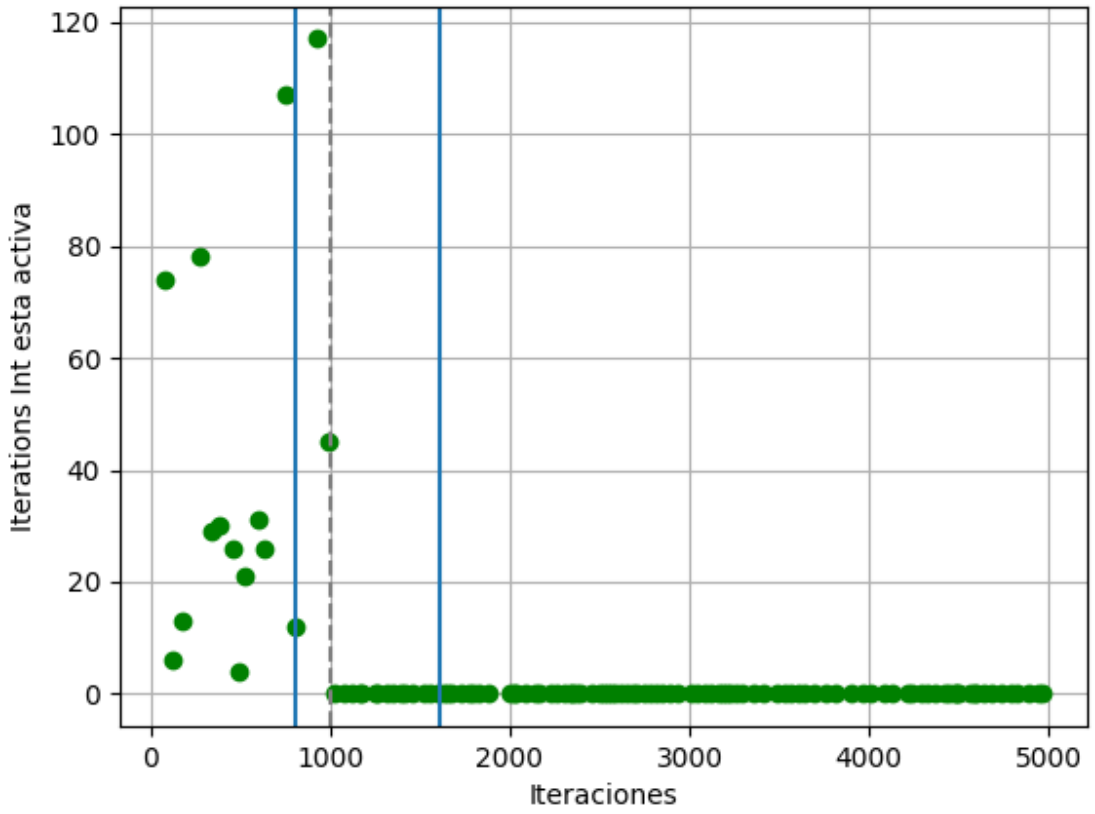
4 RESULTADOS PRUEBAS ON-LINE

4.1 En función del número de iteraciones

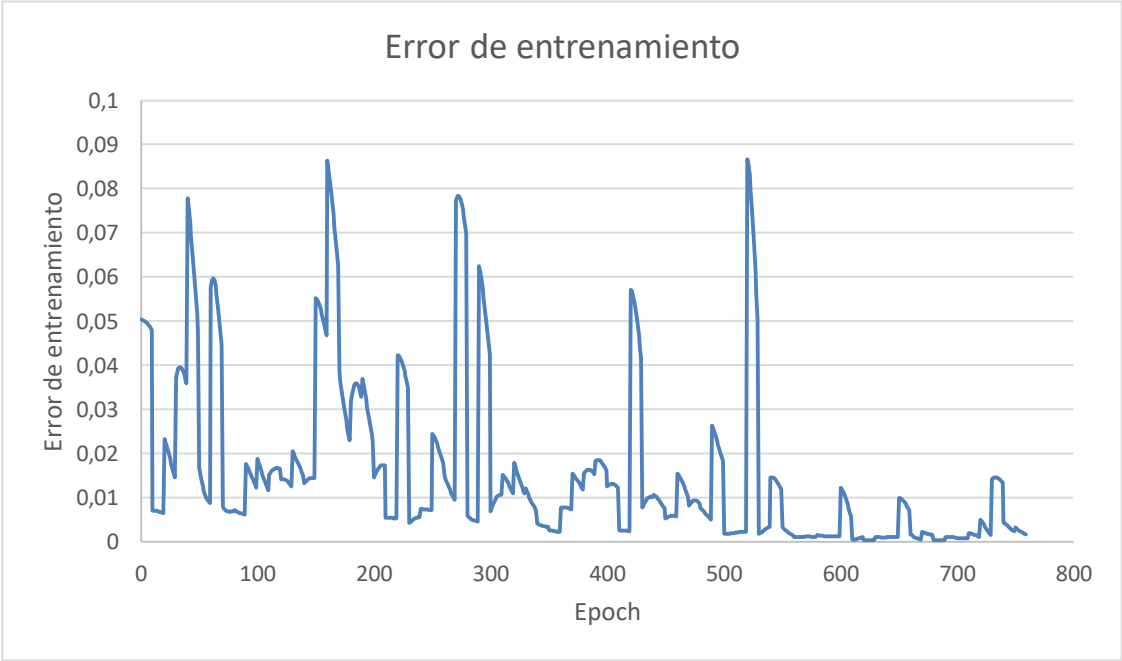
4.1.1 1000 iteraciones

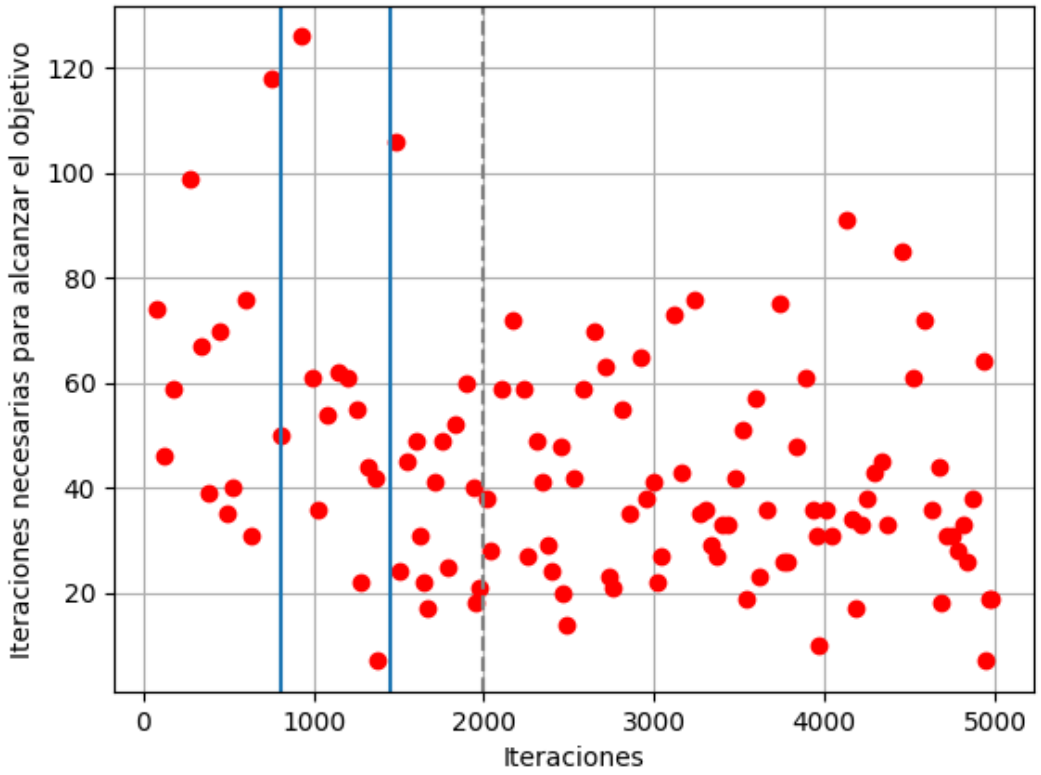


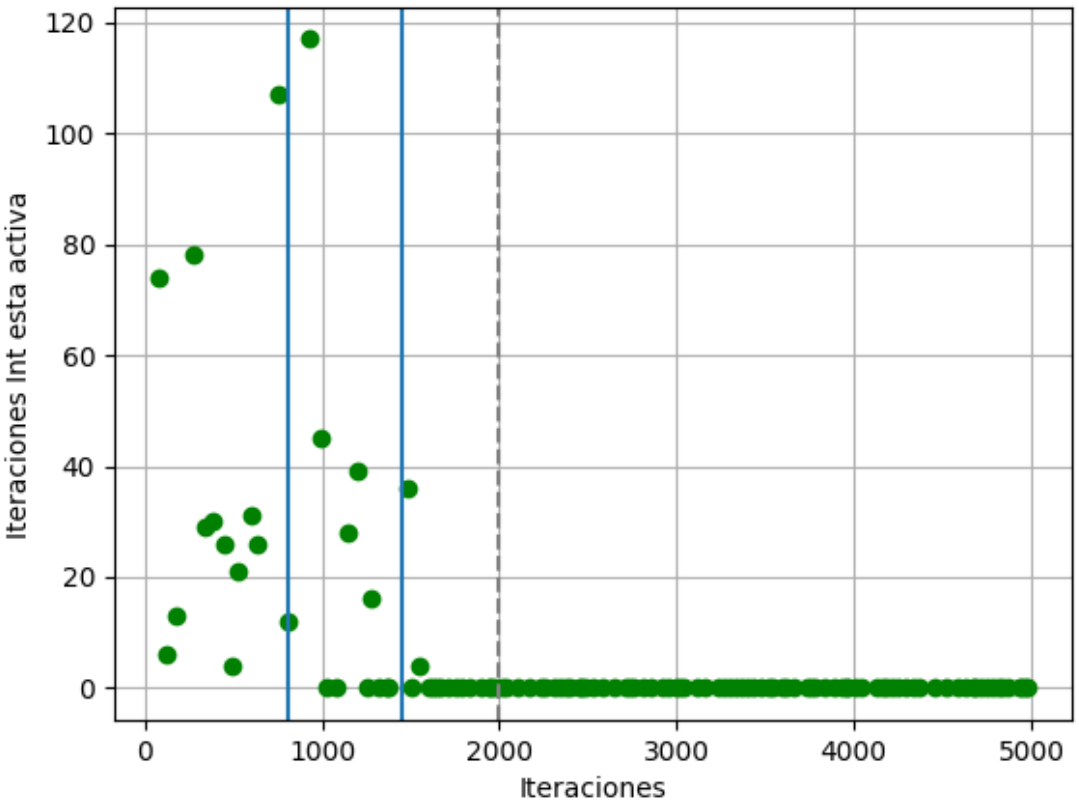




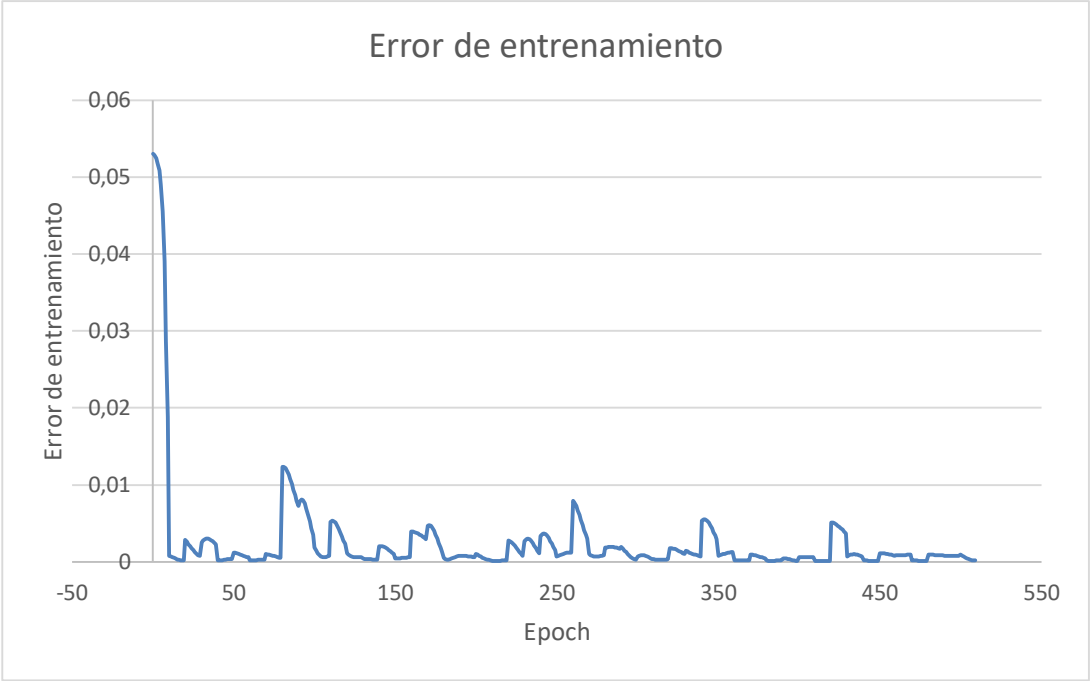
4.1.2 2000 iteraciones

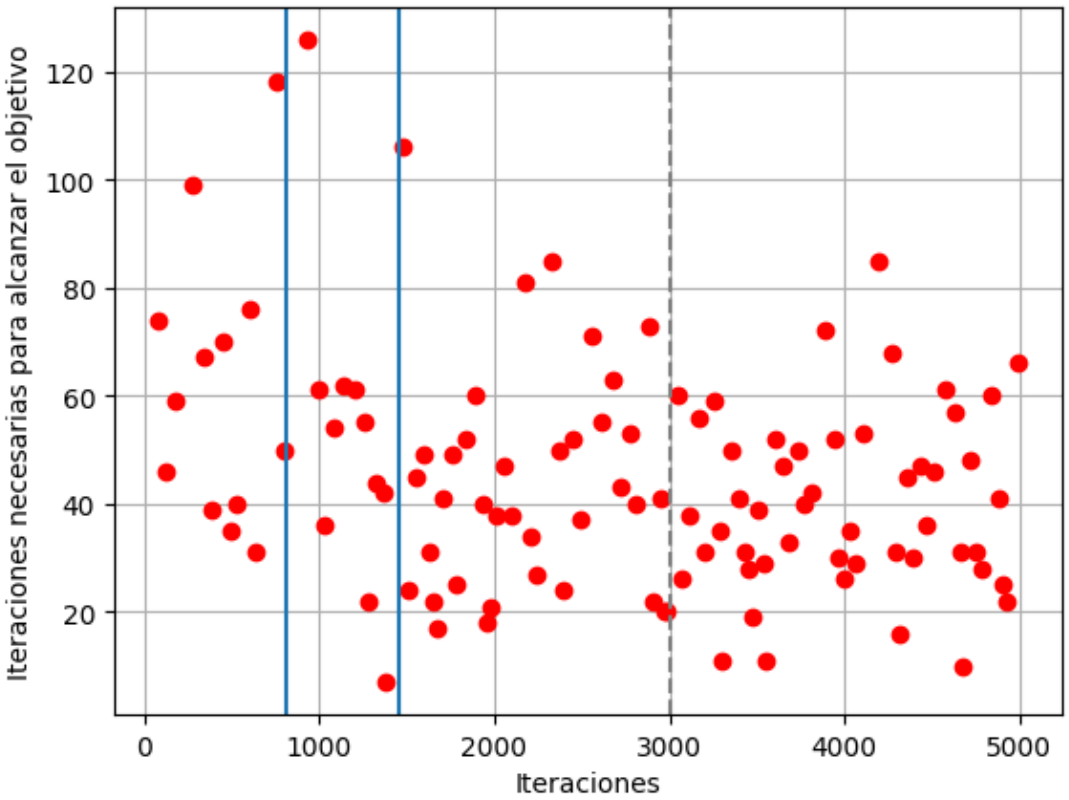


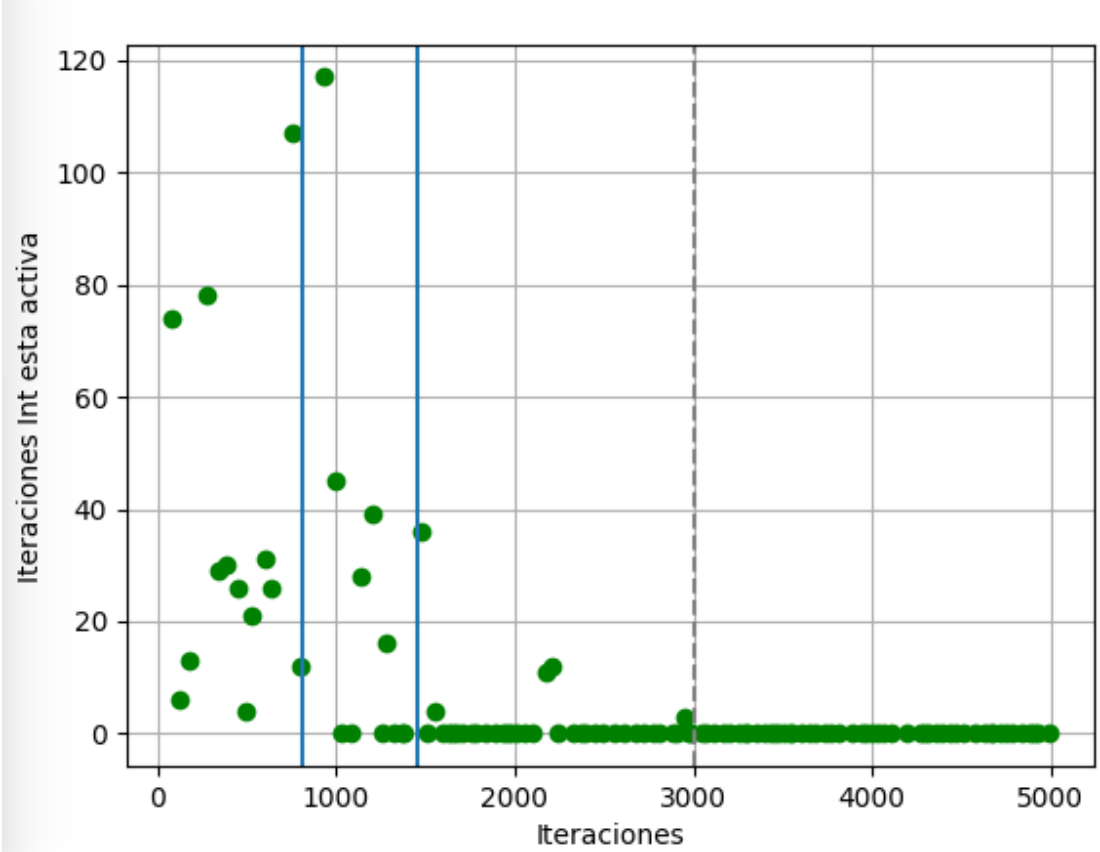

Iteraciones necesarias para alcanzar el objetivo
120
100
80
60
40
20
0
1000
2000
3000
4000
5000
Iteraciones


Iteraciones Int esta activa
120
100
80
60
40
20
0
0
1000
2000
3000
4000
5000
Iteraciones

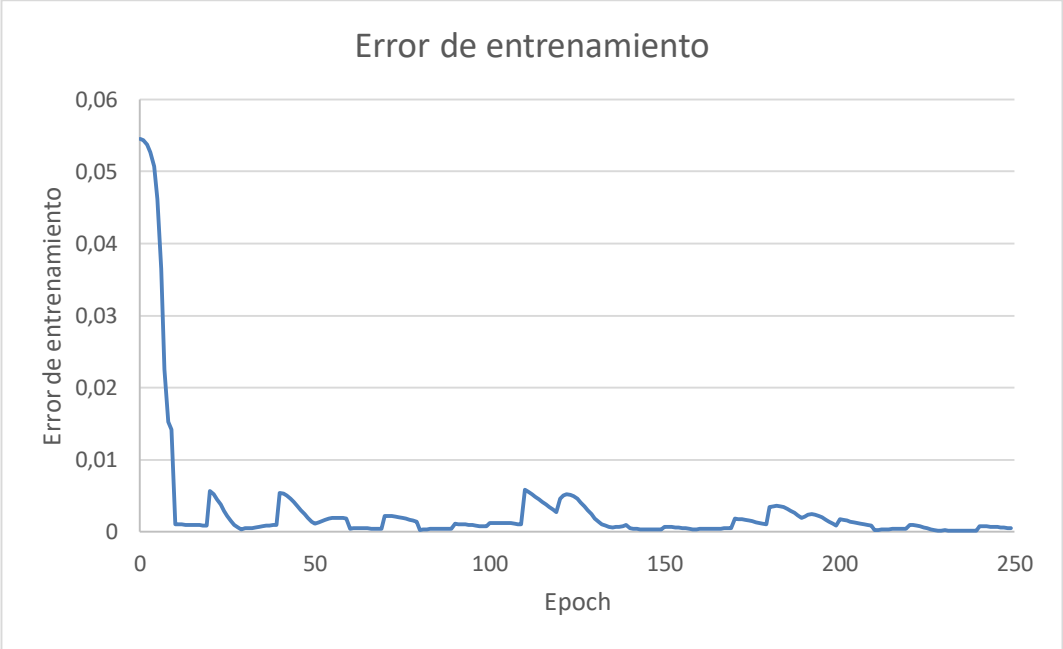
4.1.3 3000 iteraciones

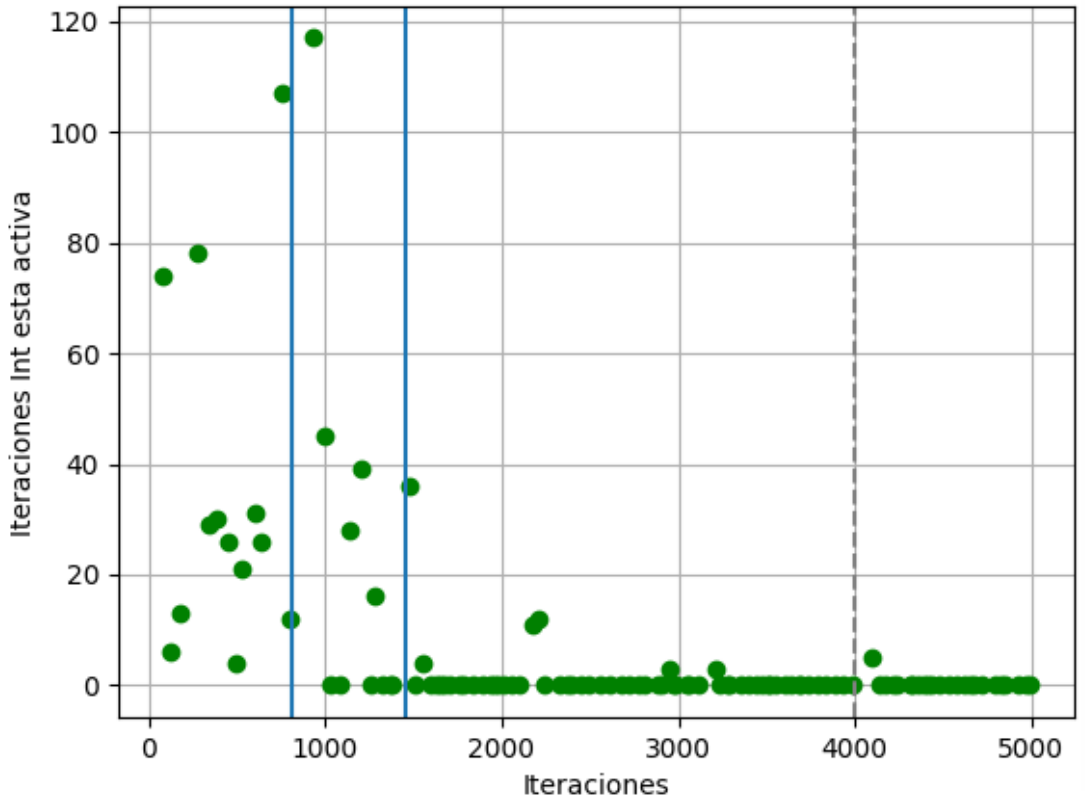


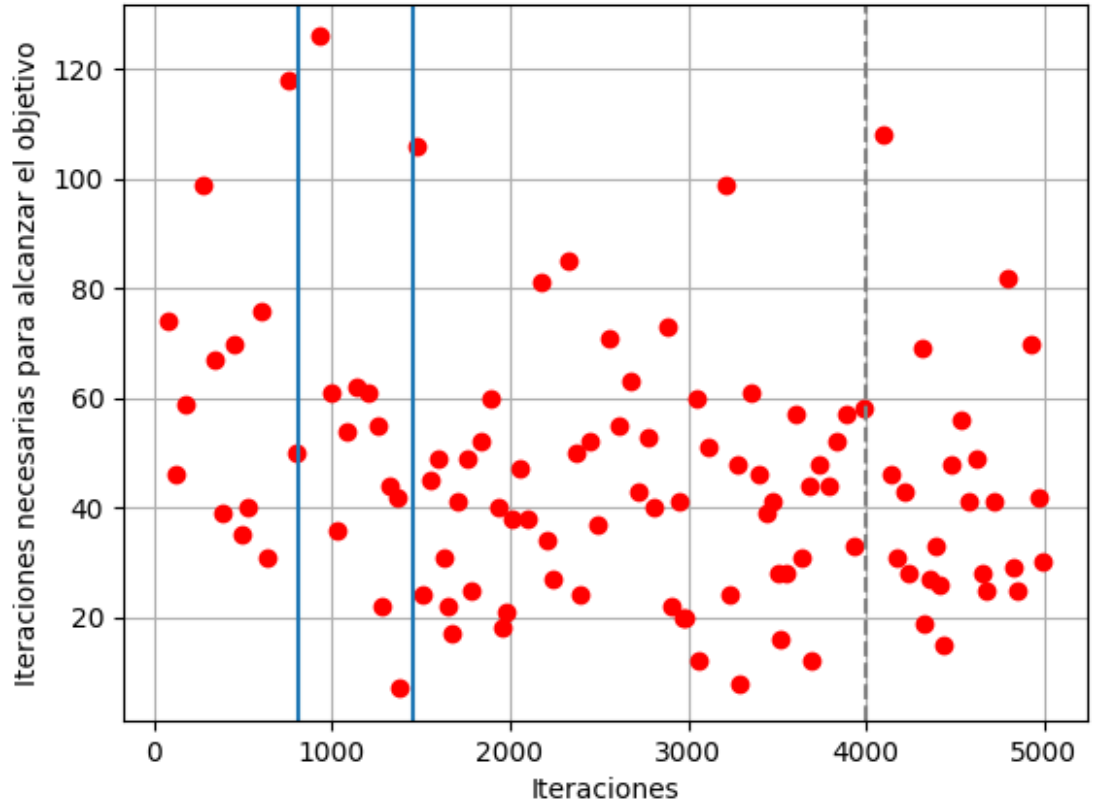




4.1.4 4000 iteraciones

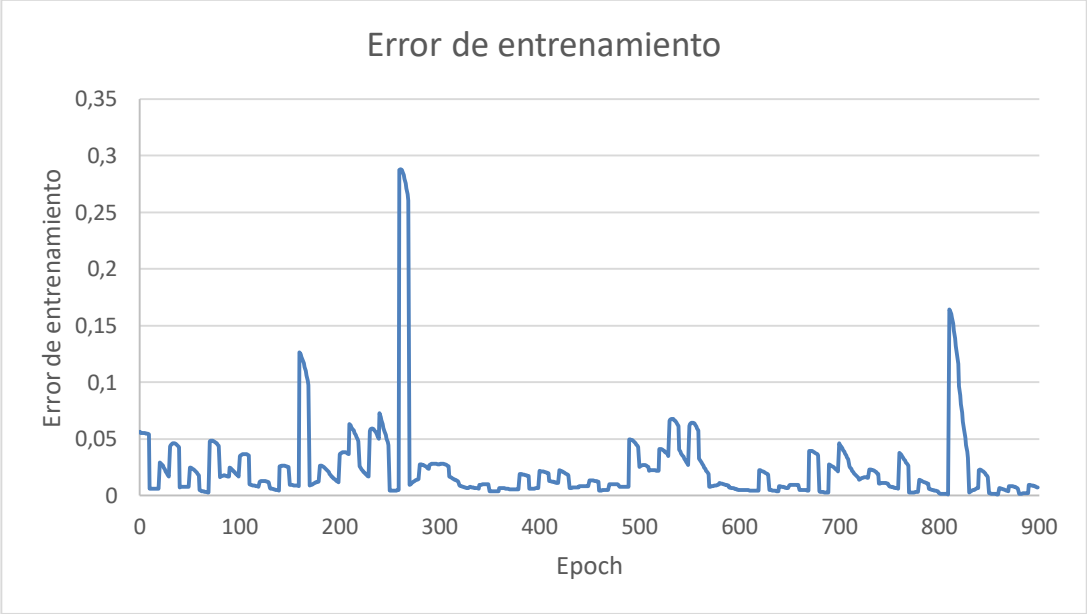


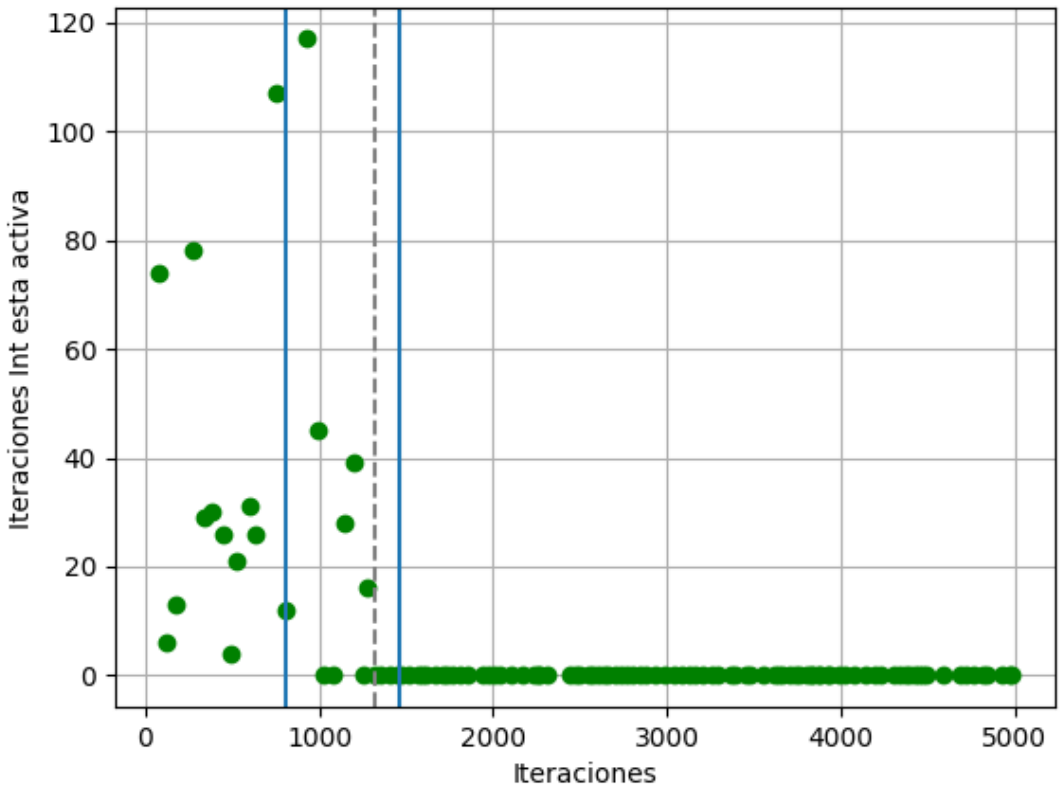

Iteraciones Int esta activa
120
100
80
60
40
20
0
0
1000
2000
3000
4000
5000
Iteraciones

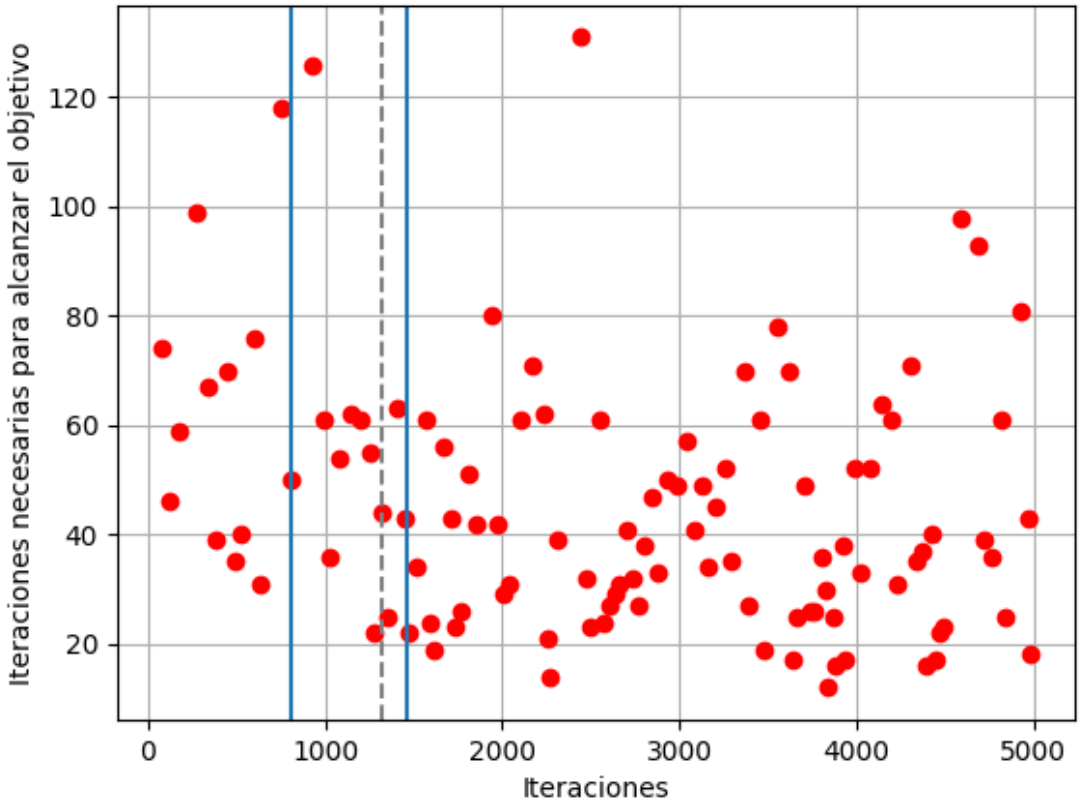

Iteraciones necesarias para alcanzar el objetivo
120
100
80
60
40
20
0
1000
2000
3000
4000
5000
Iteraciones

4.2 En función del número de veces que se ha llegado al objetivo con SURs

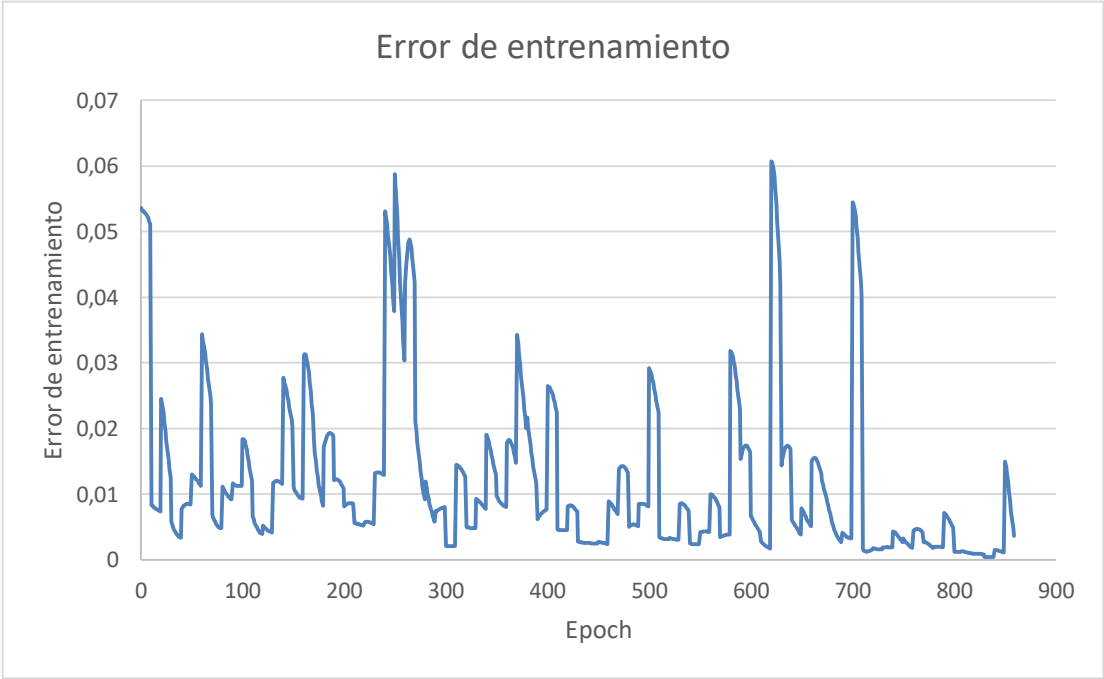
4.2.1 20 veces

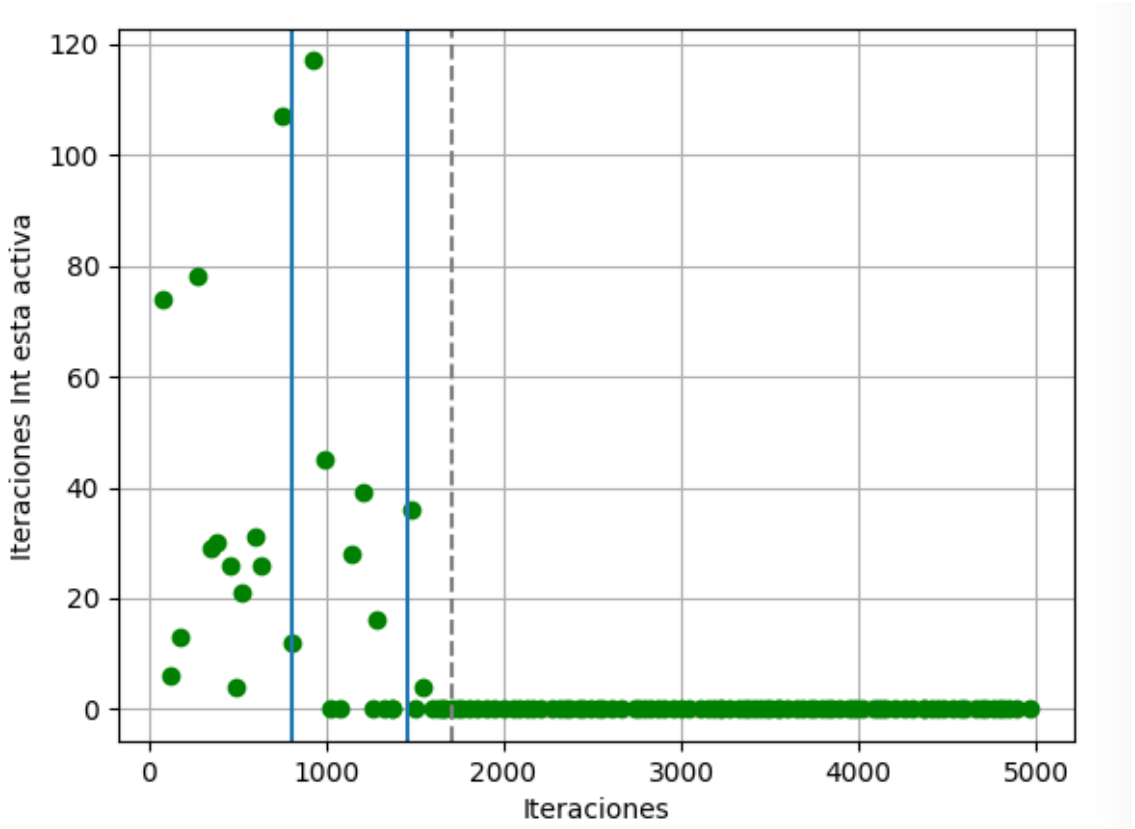


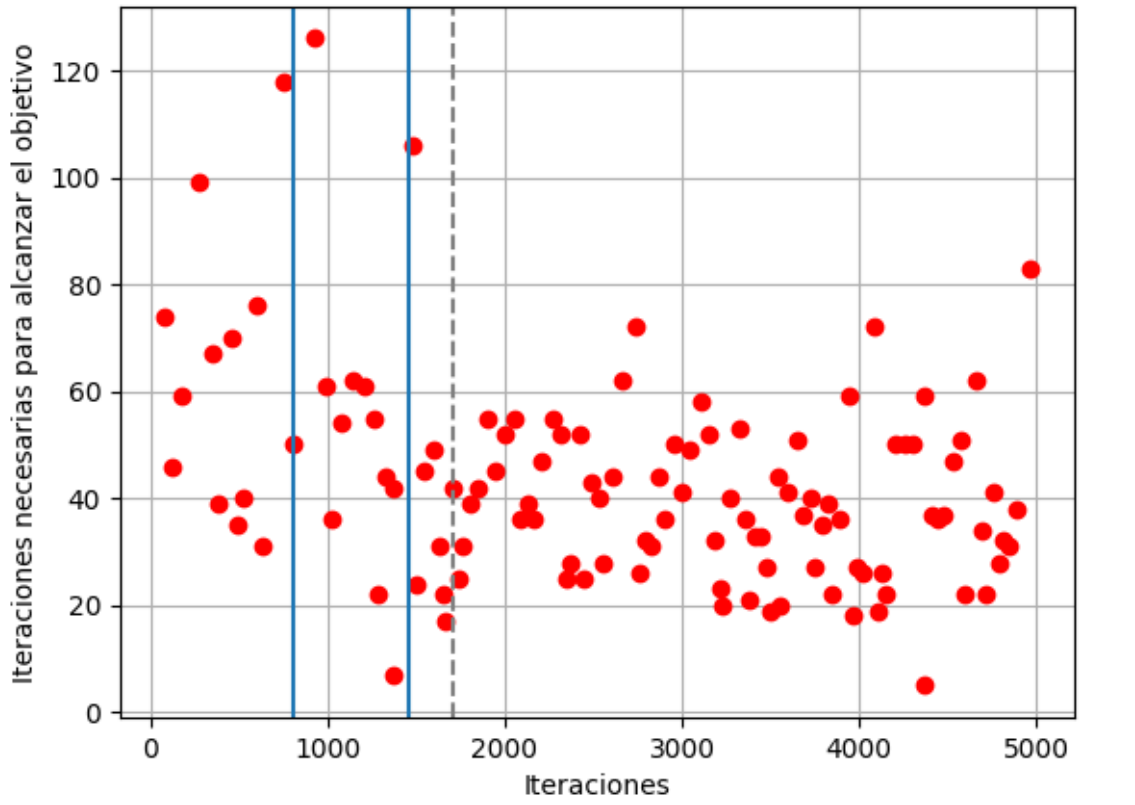



Iteraciones necesarias para alcanzar el objetivo
120
100
80
60
40
20
0
1000
2000
3000
4000
5000
Iteraciones

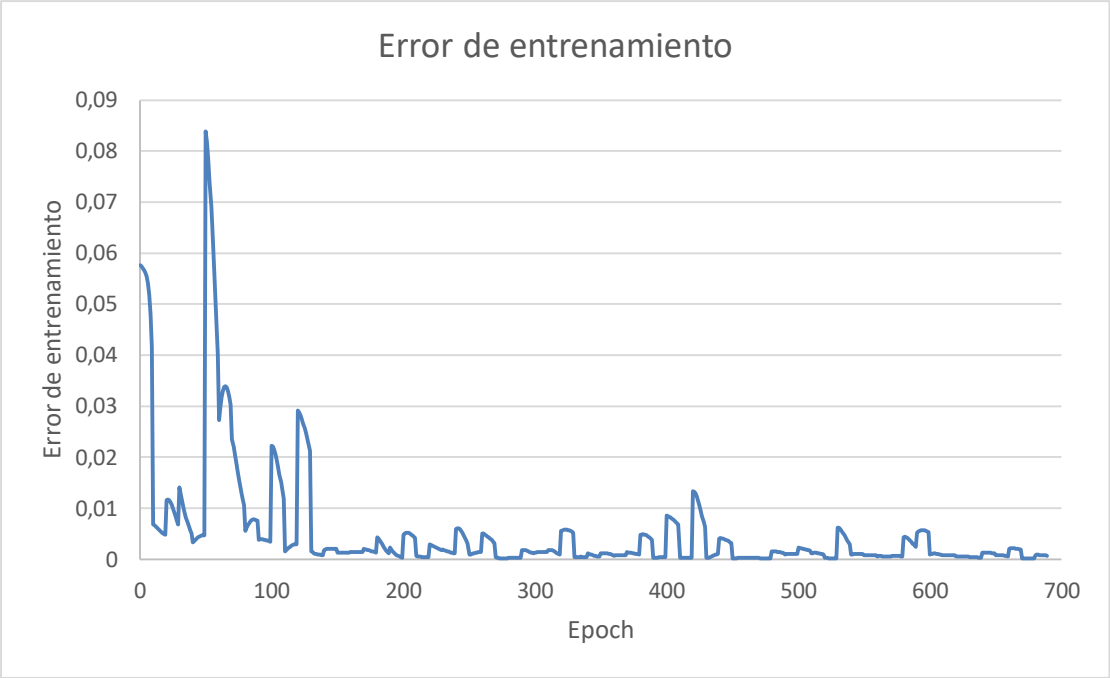
4.2.2 30 veces

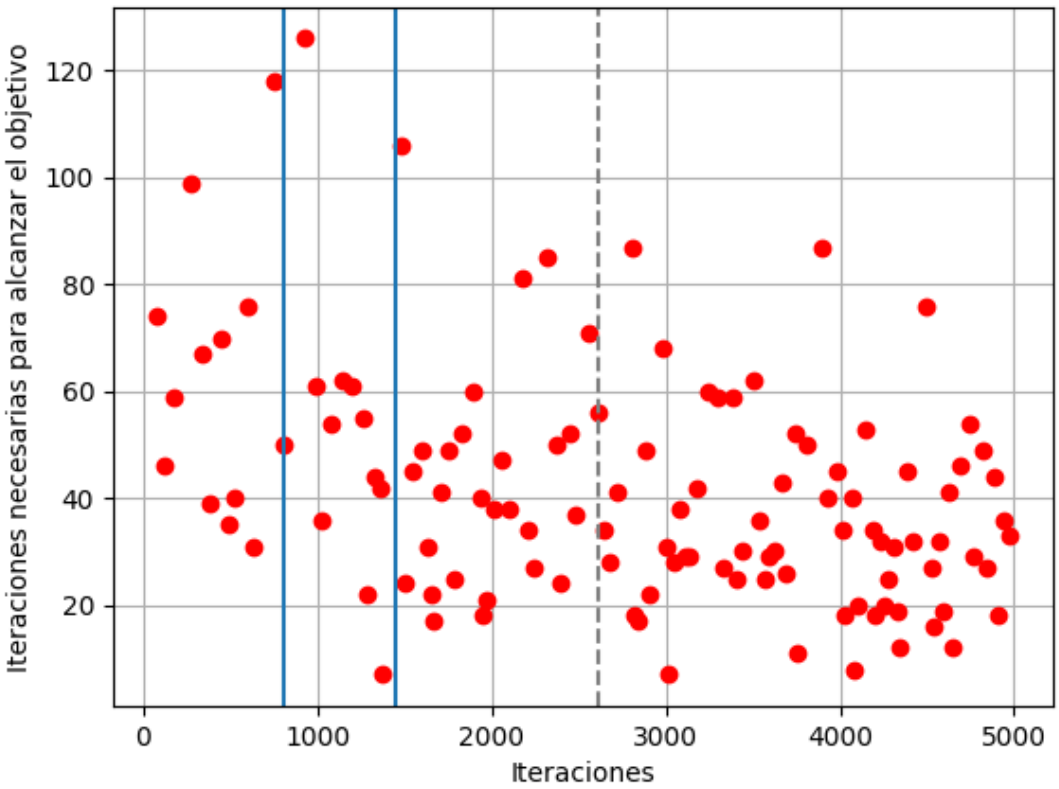

Error de entrenamiento
Error de entrenamiento
0,07
0,06
0,05
0,04
0,03
0,02
0,01
0
0
100
200
300
400
500
600
700
800
900
Epoch

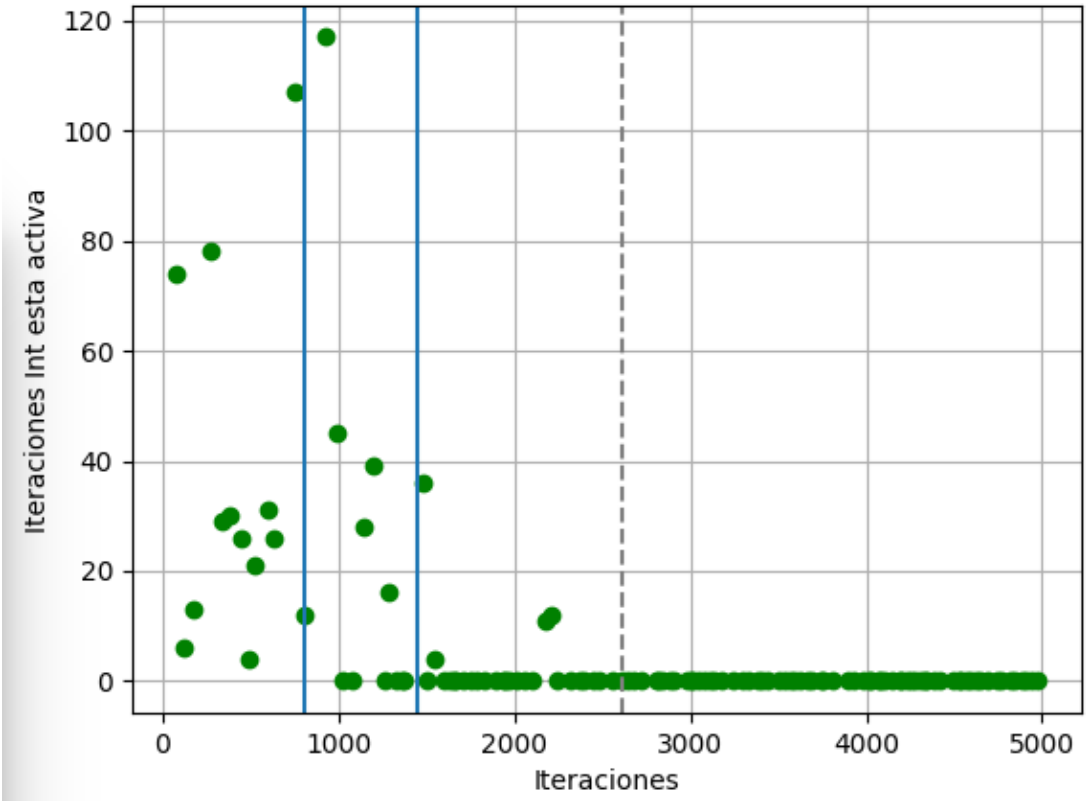

Iteraciones Int esta activa
120
100
80
60
40
20
0
0
1000
2000
3000
4000
5000
Iteraciones


Iteraciones necesarias para alcanzar el objetivo
120
100
80
60
40
20
0
0
1000
2000
3000
4000
5000
Iteraciones

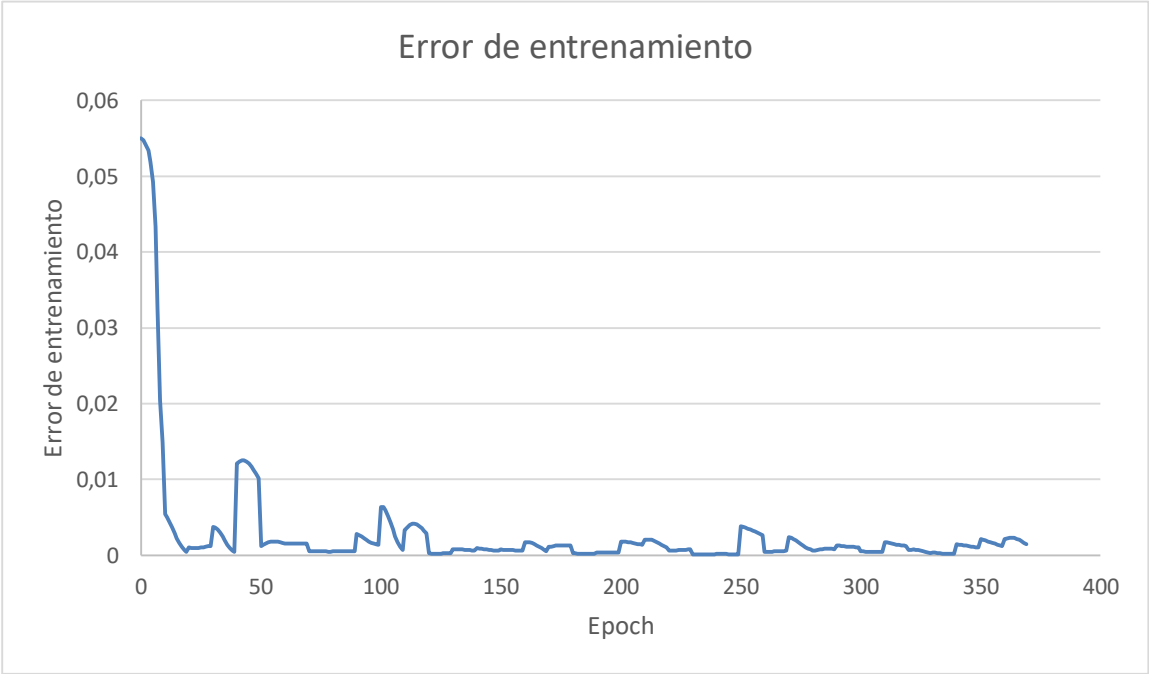
4.2.3 50 veces

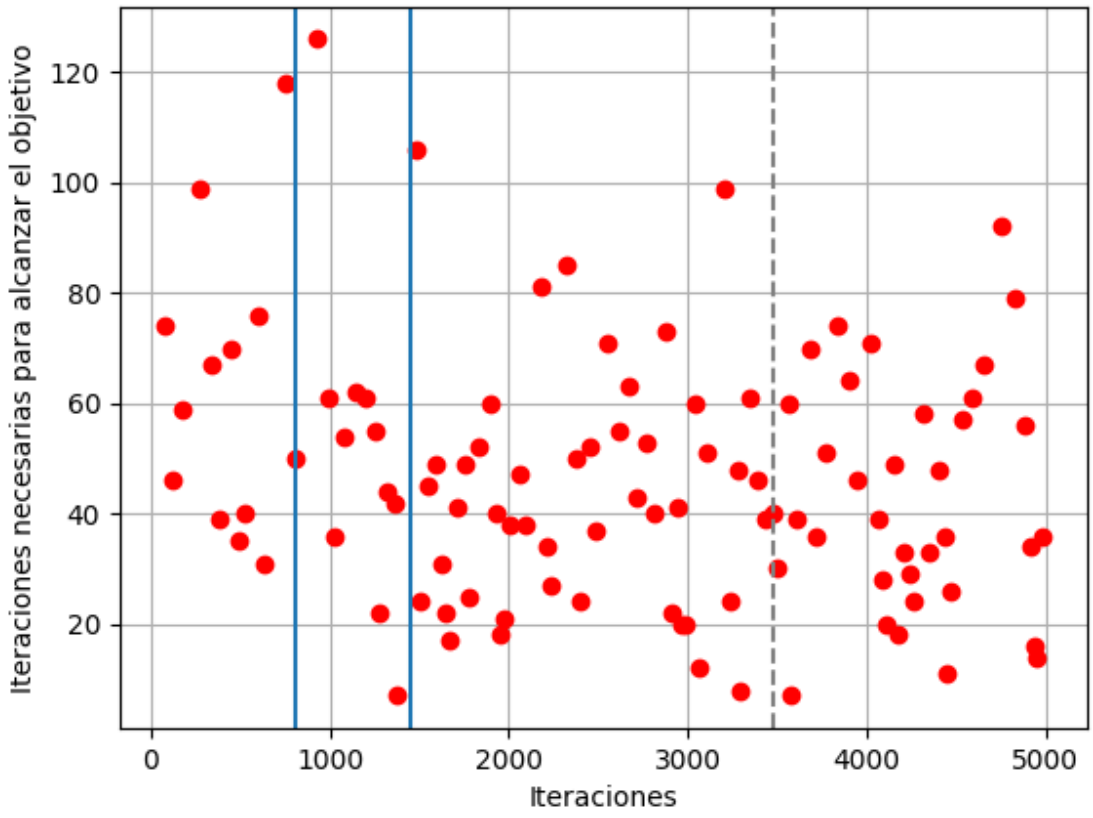

Error de entrenamiento
Error de entrenamiento
0,09
0,08
0,07
0,06
0,05
0,04
0,03
0,02
0,01
0
0
100
200
300
400
500
600
700
Epoch

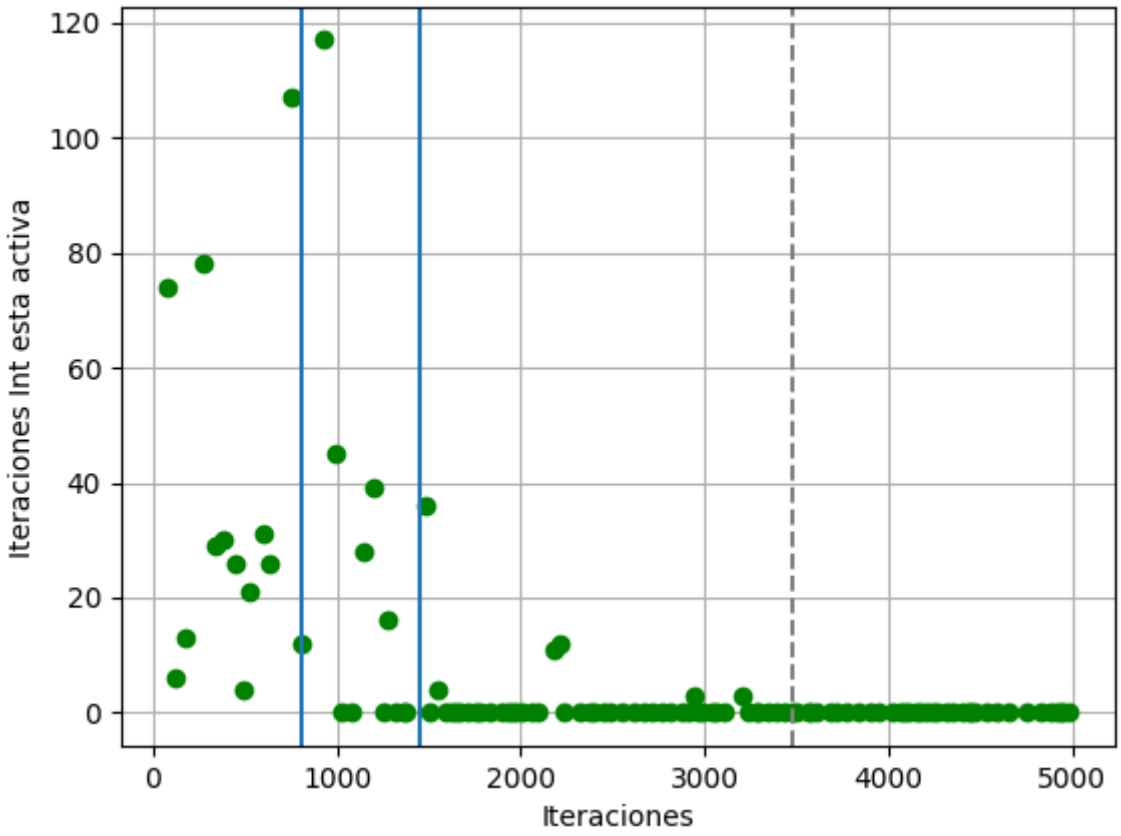

Iteraciones necesarias para alcanzar el objetivo
120
100
80
60
40
20
0
1000
2000
3000
4000
5000
Iteraciones


Iteraciones Int esta activa
120
100
80
60
40
20
0
0
1000
2000
3000
4000
5000
Iteraciones

4.2.4 70 veces

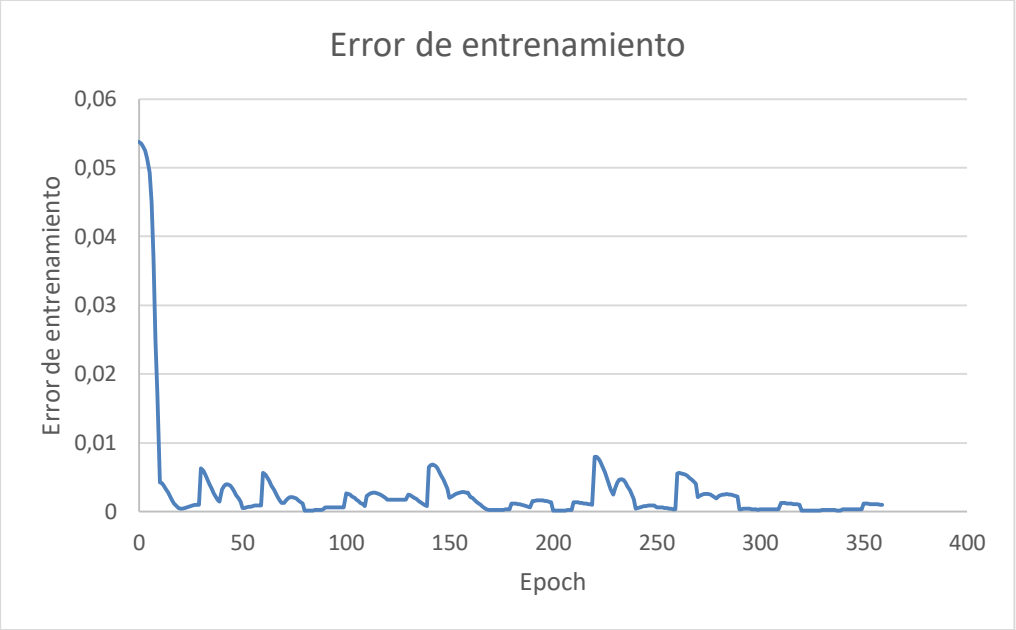

Error de entrenamiento
Error de entrenamiento
0,06
0,05
0,04
0,03
0,02
0,01
0
0
50
100
150
200
250
300
350
400
Epoch

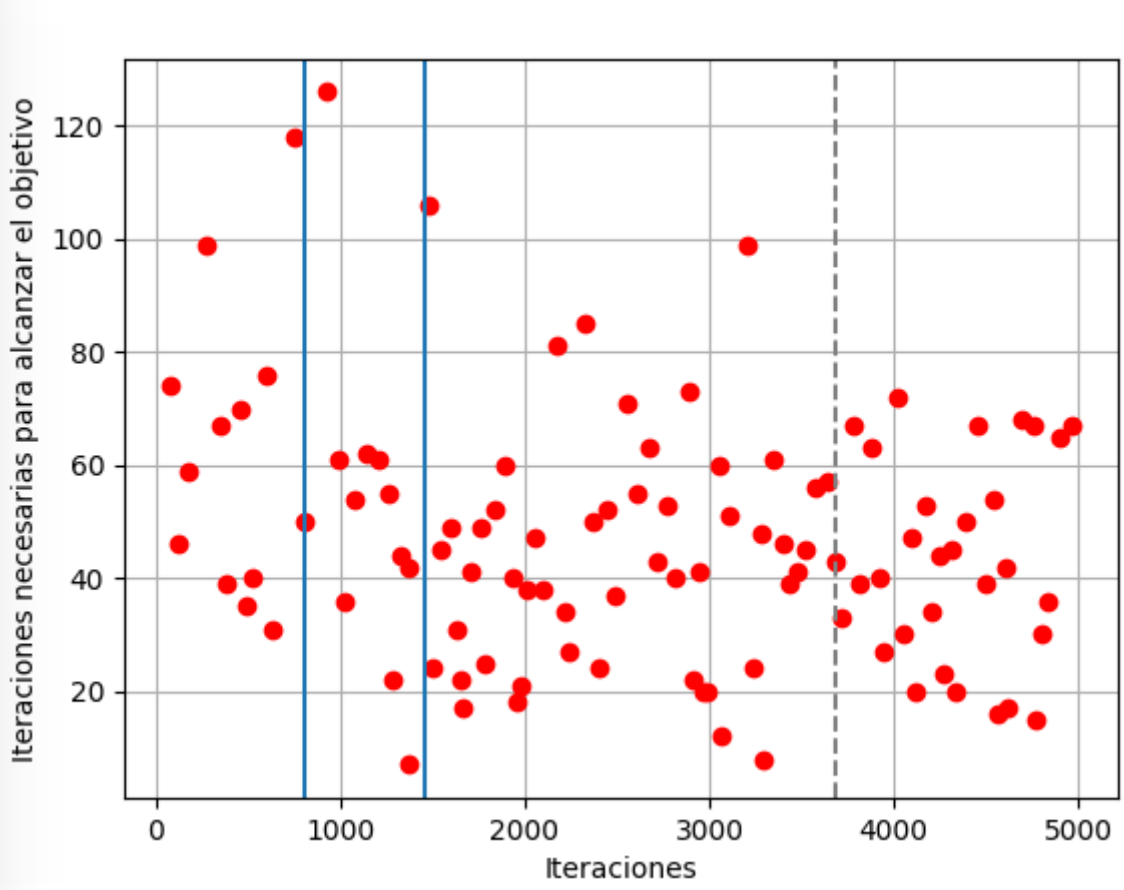

Iteraciones necesarias para alcanzar el objetivo
120
100
80
60
40
20
0
1000
2000
3000
4000
5000
Iteraciones

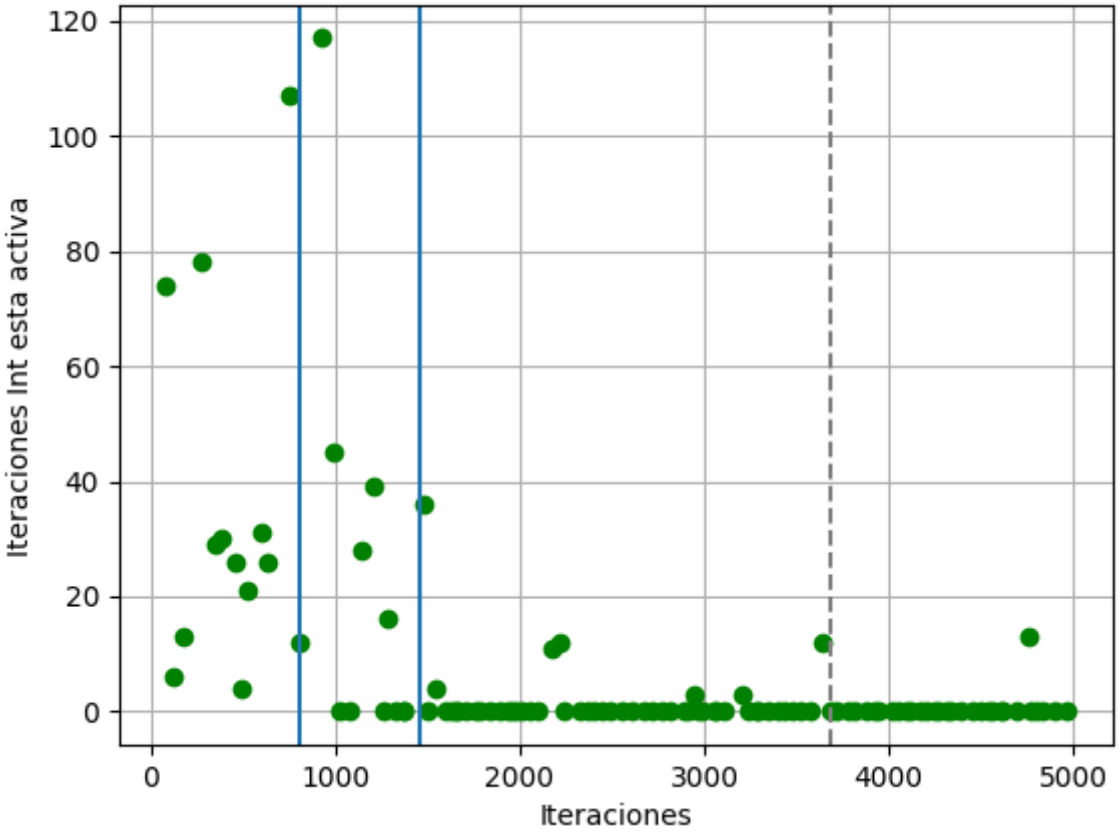

Iteraciones Int esta activa
120
100
80
60
40
20
0
0
1000
2000
3000
4000
5000
Iteraciones

4.3 En función del error de entrenamiento de la VF

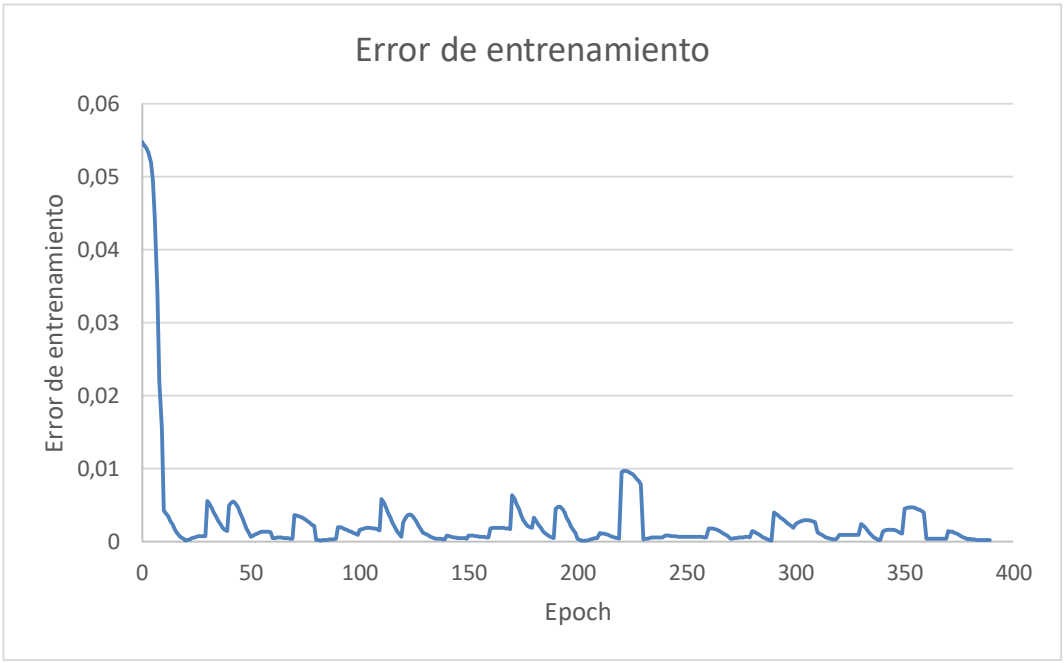
4.3.1 Umbral de error de entrenamiento de 0,01

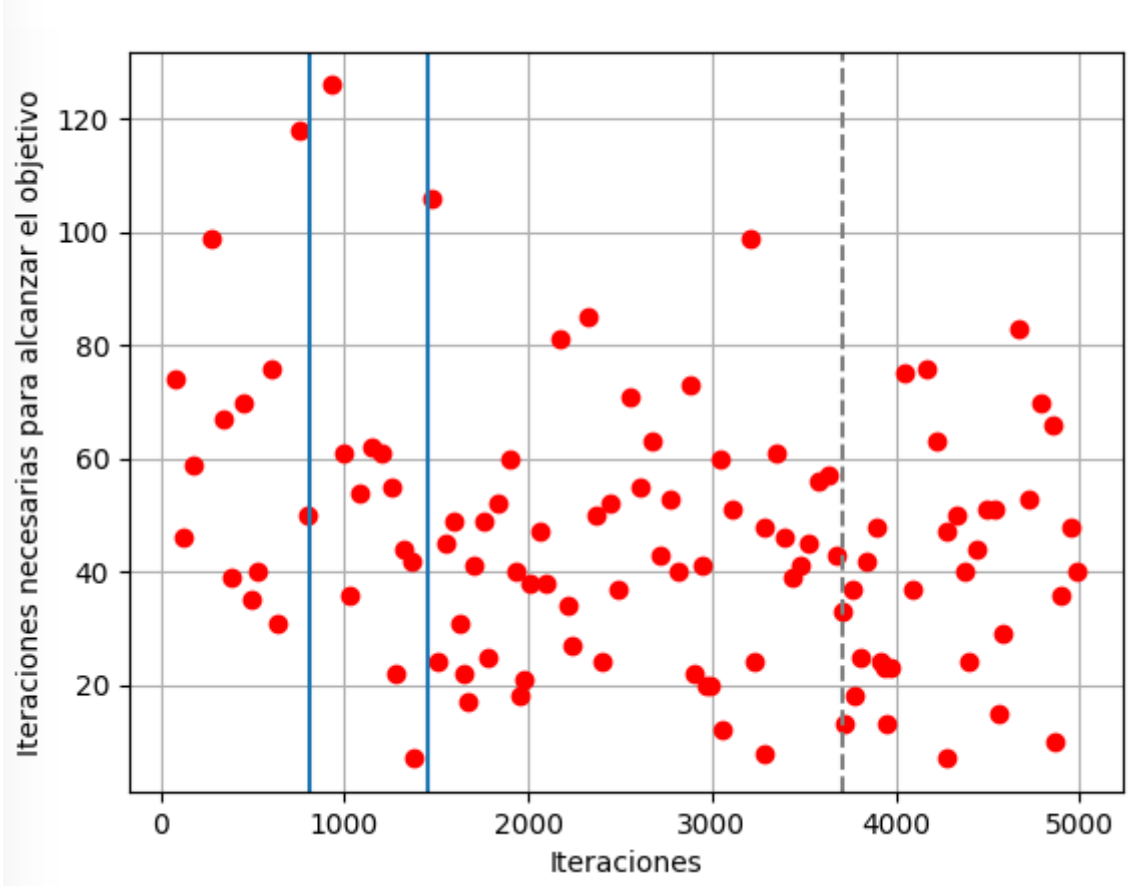


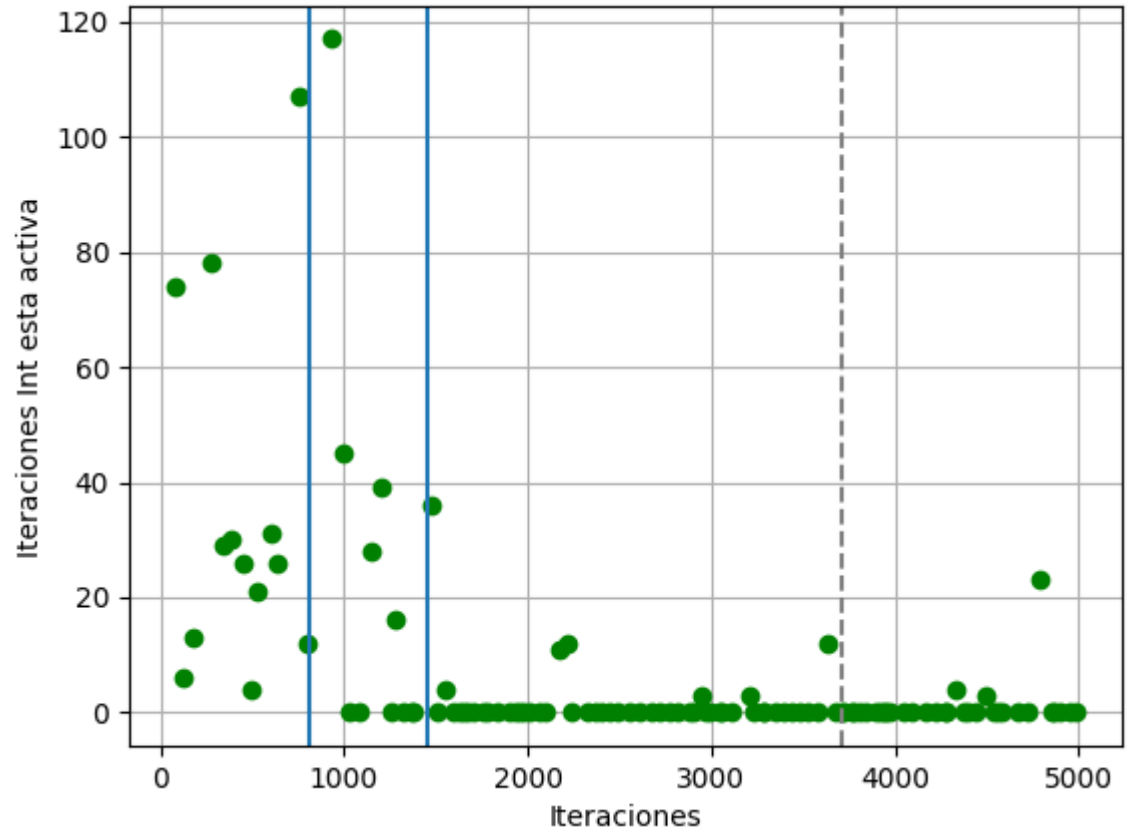




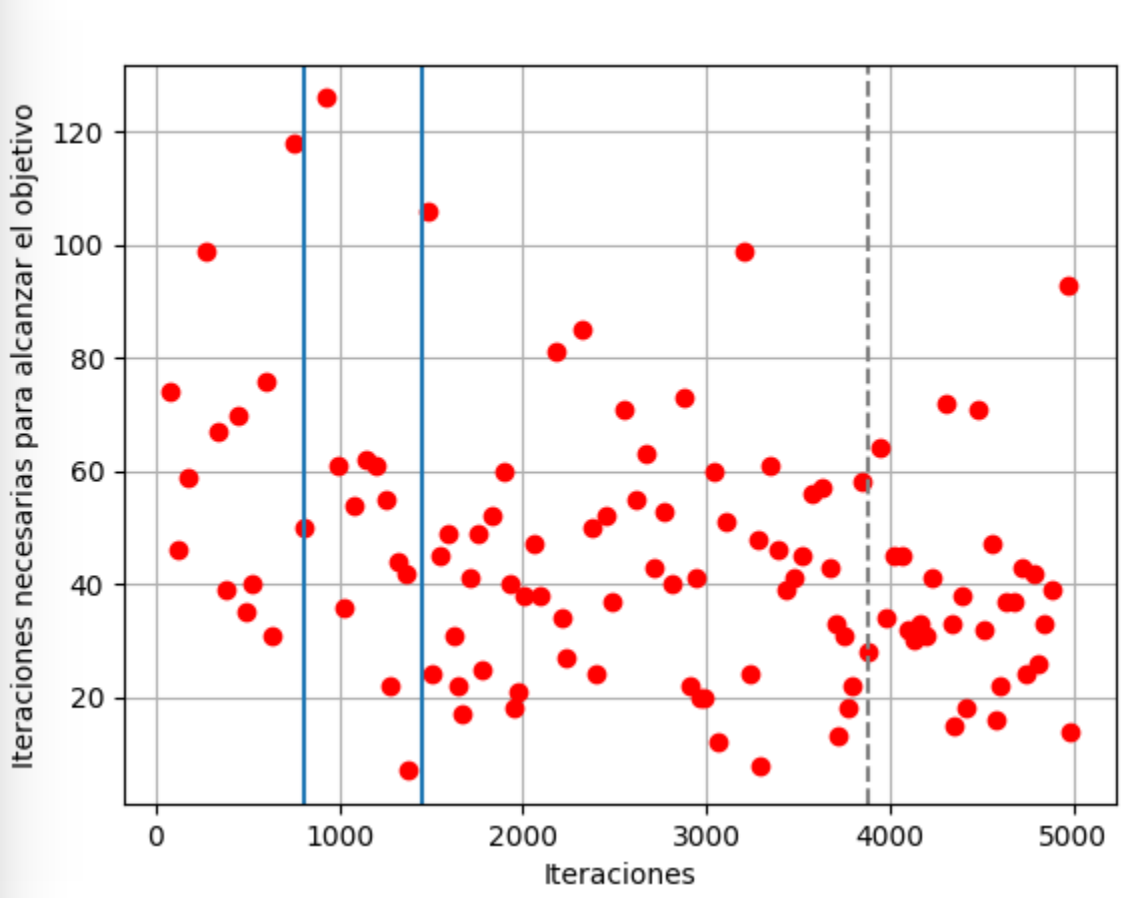
4.3.2 Umbral de error de entrenamiento de 0,008

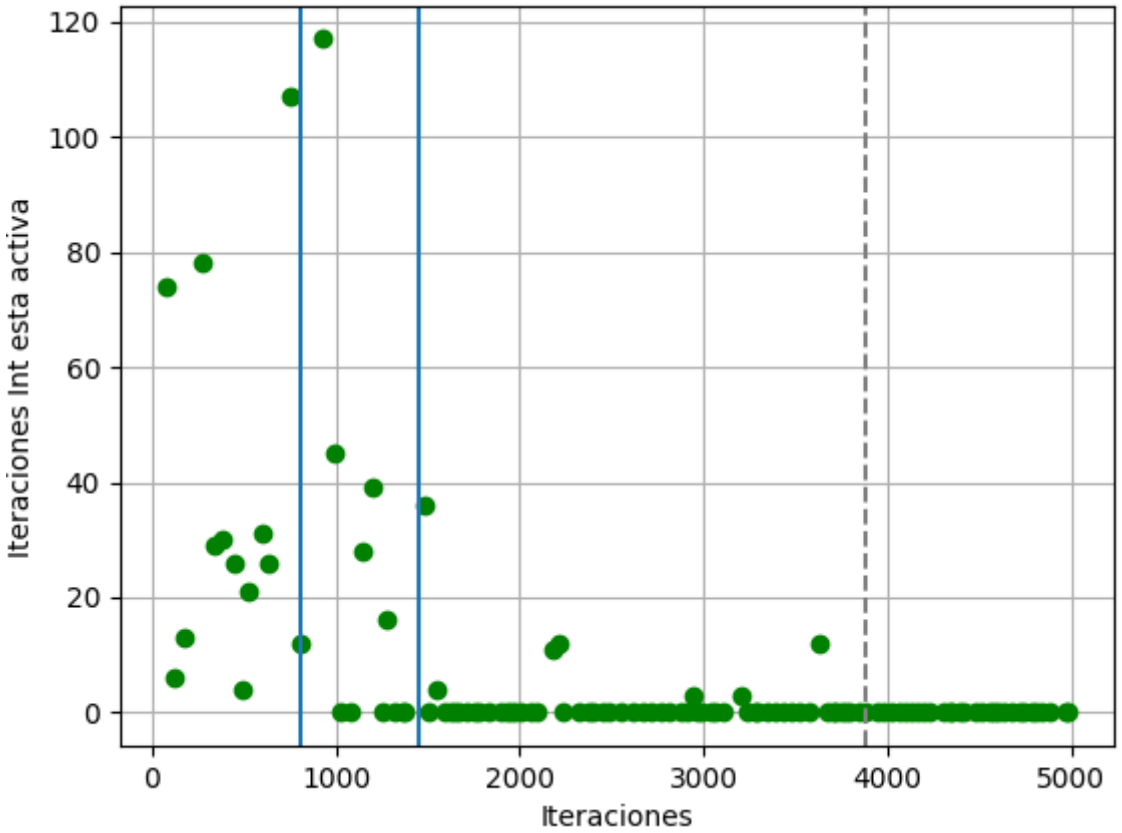



Iteraciones necesarias para alcanzar el objetivo
120
100
80
60
40
20
0
1000
2000
3000
4000
5000
Iteraciones

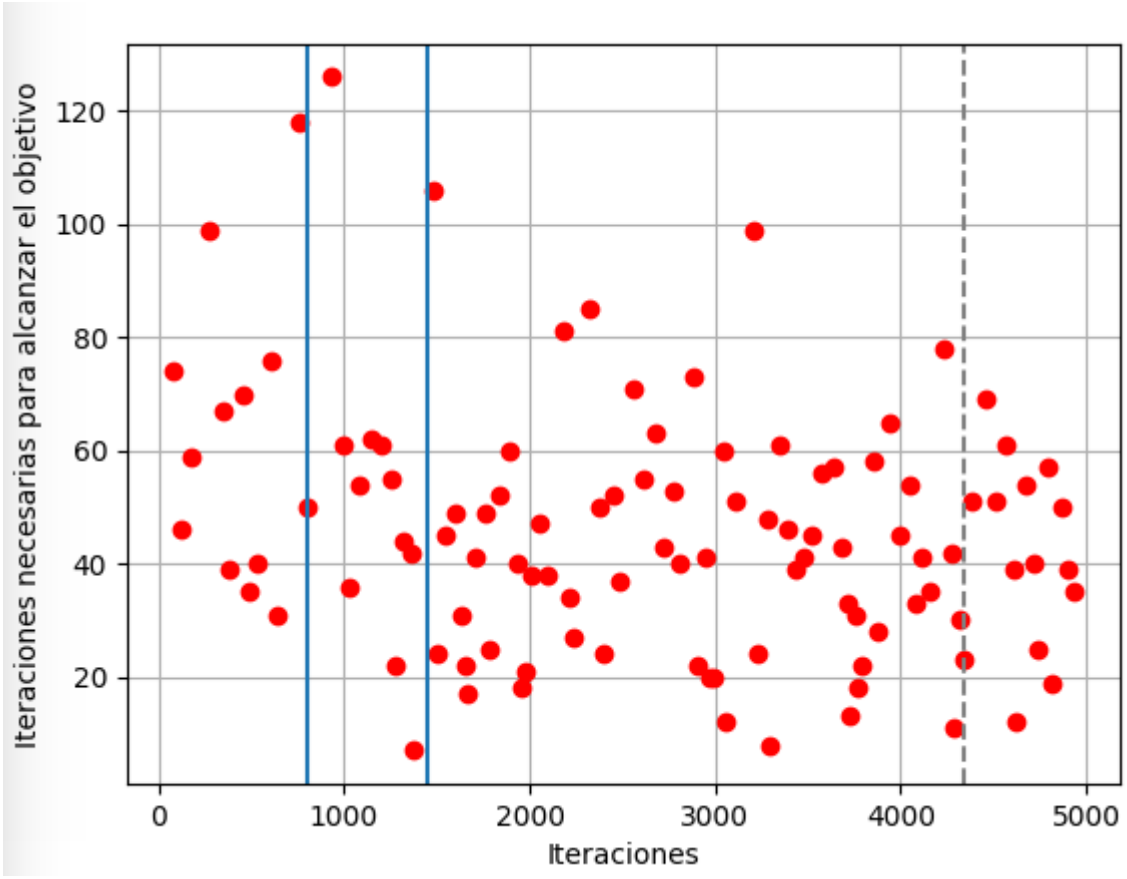

Iteraciones Int esta activa
120
100
80
60
40
20
0
0
1000
2000
3000
4000
5000
Iteraciones

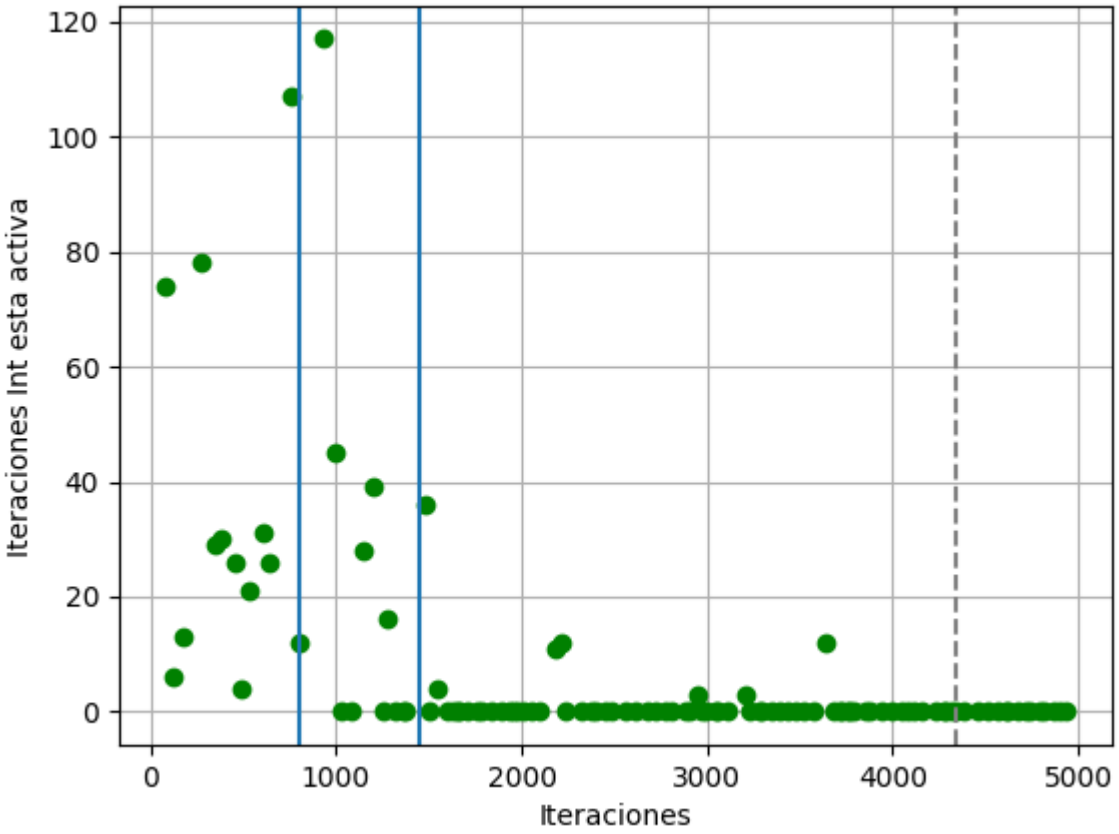
4.3.3 Umbral de error de entrenamiento de 0,005





4.3.4 Umbral de error de entrenamiento de 0,003


Iteraciones necesarias para alcanzar el objetivo
120
100
80
60
40
20
0
1000
2000
3000
4000
5000
Iteraciones


Iteraciones Int esta activa
120
100
80
60
40
20
0
0
1000
2000
3000
4000
5000
Iteraciones