# *Robot con navegación autonoma basada en redes neuronales*

Verónica Tornero Écija
Grado en Ingenieria de robótica Software
Universidad Rey Juan Carlos
Fuenlabrada, Madrid, España
v.tornero.27@gmail.com

Javier Martínez Madruga
Grado en Ingeniería de Robótica Software
Universidad Rey Juan Carlos
Fuenlabrada, Madrid, España
javimartinezmma@gmail.com

Carlos Caminero Abad
Grado en Ingeniería de Robótica Software
Universidad Rey Juan Carlos
Fuenlabrada, Madrid, España
carlos2caminero@gmail.com

Irene Bandera Moreno
Grado en Ingeniería de Robótica Software
Universidad Rey Juan Carlos
Fuenlabrada, Madrid, España
ibanderam@gmail.com

***Resumen*—El presente documento explica el proyecto realizado por los alumnos de Ingeniería de Robótica Software de la Universidad Rey Juan Carlos para el trabajo grupal de la asignatura de Aprendizaje Automàtico en el cuarto curso.**

***Palabras clave*—Agente, Red Neuronal, Robot, Obstáculos, Aprendizaje Automatico, Matlab, Clase, Láser, Intersección**

## I. INTRODUCCIÓN

Durante los últimos años hemos sido testigos del incremento en el uso de los sistemas robóticos. Estos están sirviendo de ayuda en la mayor parte de los ámbitos de la vida de las personas. Todo esto desencadena la necesidad de sistemas robóticos mas robustos e inteligentes. Con las redes neuronales conseguimos la simulación de un sistema nervioso para los sistemas robóticos. Una red neuronal es un modelo computacional inspirado en el funcionamiento del cerebro humano que se utiliza como técnica de aprendizaje automático para resolver problemas de aprendizaje supervisado y no supervisado. Gracias a ellas se consigue la capacidad de aprendizaje en base a diversas experiencias, lo cual permite una reducción a la hora de desarrollar el software, ya que, no es necesario definir todos los casos o actuaciones posibles.

Las redes neuronales se usan en problemas de Aprendizaje Supervisado donde el modelo de red neuronal se entrena a partir de un **conjunto de entrenamiento** que consta de entradas y sus correspondientes salidas para poder clasificar o predecir a posteriori con nuevos datos. También se utilizan, pero menos frecuente, en problemas de Aprendizaje No Supervisado, donde se proporciona sólo las entradas y el modelo debe ser capaz de encontrar patrones para su clasificación o **clustering**.

Este trabajo presenta el desarrollo de una red neuronal que permite a un **robot** (agente) llegar de un extremo a otro de la carretera (entorno) evitando todos los obstáculos que encuentre por su camino.

En la robótica, cuando se desarrolla aplicaciones de Aprendizaje Automático (Machine Learning) para su uso en la vida real, se llevan a cabo unos pasos previos para su correcto funcionamiento con el mundo exterior. Primero se lleva a una simulación, donde el agente simulado es entrenado en un entorno controlado para posteriormente ser capaz de responder en otro entorno completamente distinto. A esta técnica se le conoce como **sim2sim**. La segunda técnica empleada es **sim2real**: se entrena al robot en un entorno simulado para enfrentarse posteriormente en un entorno real. Un ejemplo de sim2real son los vehículos autónomos, ya que previamente se prueba su software en simuladores para luego probarlos en polígonos industriales. Otra técnica es **real2real**. Tal como su nombre indica dejamos atrás las simulaciones y solo nos enfrentamos con el mundo real.

Con este proyecto, pretendemos aplicar la técnica **sim2sim** para que nuestro robot sea capaz de enfrentarse a otros entornos simulados distintos.

Para el desarrollo de la simulación hemos usado Matlab como lenguaje de programación y varias funciones de alto nivel para la creación de la red neuronal.

## II. CLASES IMPLEMENTADAS

Este proyecto se ha realizado usando Programación Orientada a Objetos en Matlab. En POO, las **clases** son bloques de código que definen propiedades, métodos y eventos. La idea es desarrollar un mundo simulado que en cuanto a comportamiento se asemeje lo máximo posible a la realidad de manera que sea fácil de programar y entender. Gracias a la creación de una serie de clases, se ha podido realizar el ejercicio con el objetivo propuesto:

### *A. Clase Entorno*

Esta clase se ha construido con la idea de ser el receptor de información proporcionado por otras clases. No hereda de los demás métodos, los cuales se explicarán a continuación en los siguientes apartados, pero llama a la función show() de “mostrar” definido en cada uno de ellos.

Esta clase constará de constructor, sin el cual no sería posible la su existencia; una función para añadir al robot en la posición que se desee; otra función para poder incorporar obstáculos, los cuales el robot tiene que saber y poder esquivar; y por último, una función para mostrarlo todo.

Esta última función es la responsable de la visualización de la simulación. Eso se debe a que desde ella, se llama a los correspondientes métodos de visualización de las demás clases.

### *B. Clase Robot.*

La siguiente clase representa al **agente**, una entidad que percibe estímulos del entorno con la ayuda de sus sensores (el láser) y actuá en ese medio utilizando sus **actuadores.**

En la simulación es identificado como un círculo que consta de 2 atributos (x, y) que determinan su posición en el entorno bidimensional.

Posee un método público llamado show() que al ser invocado representa al robot en una gráfica XY. Este método será utilizado por la clase Entorno cuando necesite mostrar el entorno completo.

### *C. Clase Obstáculo*

Esta clase representa los objetos que esquivará el robot. Se representan como un cuadrado de un tamaño específico y su posición está determinado por 2 atributos (x, y).

Al igual que la clase Robot, posee un método público show() para ser utilizado por la clase Entorno

### *D. Clase Láser*

Es la clase clave para el sorteamiento de obstáculos. Se trata de una simulación que pretende acercarse al comportamiento real de un láser de 180 grados.

Su objetivo es proporcionar información de lo que le rodea al robot y esté dentro del alcance de los rayos láser. Más concretamente el método get_values() devuelve un array de distancias, mientras que update() actualiza estas. Por último la clase ofrece un método show() que dibuja los rayos. En el apartado III de este documento encontramos más información sobre la implementación de este láser simulado.

## III. IMPLEMENTACIÓN DEL LÁSER SIMULADO. ECUACIONES .

Para proporcionar información del entorno a nuestro robot, decidimos crear un láser simulado. Este tendrá la finalidad (al igual que un láser típico) de retornar un array de distancias. Al crear nosotros dicha simulación, hemos dado al usuario la oportunidad de personalizar los grados del láser (max_angle) y el número de rayos (max_angle/increment). En nuestro caso: 180º y 90 rayos (max_angle = 180, increment = 2).

El diseño del láser está basado en el cálculo de la intersección de rectas. Conocemos:

- La ecuación de la recta de cada rayo del láser

$$Y - y0 = m(X - x0)$$

Donde **(x0, y0)** es el origen del láser y **m** es la pendiente.

- La posición de los obstáculos en el mundo (figura 1)

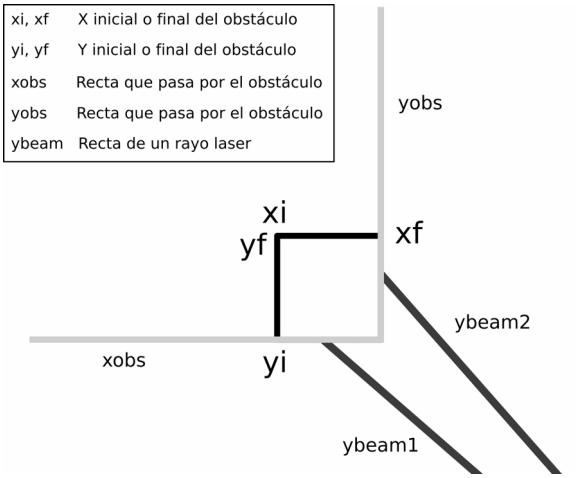


*Figura 1. Representación de un obstáculo y dos rayos láser*

- Las ecuaciones que delimitan al obstáculo en los ejes X e Y (figura 1)

$$xobs = yf$$

$$yobs = xf$$

Sabiendo todo lo anterior mencionado podemos calcular la intersección de la recta de un rayo láser con xobs o yobs (figura 1).

Para cada rayo de nuestro láser, comprobamos si este intersecciona con algún lado del obstáculo. En caso de que lo haga comprobamos si el punto de intersección se encuentra en la intervalo [xi, xf] en el caso de ybeam1 ∩ xobs o en el intervalo [yi, yf] en el caso de ybeam2 ∩ yobs. Con esto nos aseguramos de que el rayo está chocando con el obstáculo ya que como sabemos todas las rectas tienen valores infinitos para X e Y, por lo tanto podrían intersectar en cualquier otro lugar del espacio y dar una "falsa detección" de obstáculo.

Para calcular ybeam1 ∩ xobs debemos igualar la ecuación del rayo del láser con la ecuación del obstáculo xobs. Despejando obtenemos la X del punto de corte:

$$X = \frac{(yobs + \tan(\alpha)\, x0 - y0)}{\tan(\alpha)}$$

Y la Y tendrá el siguiente valor:

$$Y = yobs$$

Para calcular ybeam2 ∩ yobs sustituimos en la ecuación del rayo la X por xobs dándonos como resultado la Y del punto de corte

$$Y = \tan(\alpha)(xobs - x0) + y0$$

Y la X tendrá el siguiente valor:

$$X = xobs$$

Una vez que tenemos el punto de intersección (X, Y) calculamos la distancia euclídea desde el origen del láser (x0, y0) hasta dicho punto. Esta finalmente será la distancia que tendrá el láser en concreto.

Por último a la hora de dibujarlo (queremos sólo dibujar una parte de la recta que describe el láser, la distancia obtenida anteriormente) lo que hacemos es buscar un intervalo de valores en X (**x interval**) que satisfaga nuestro deseo (figura 2).

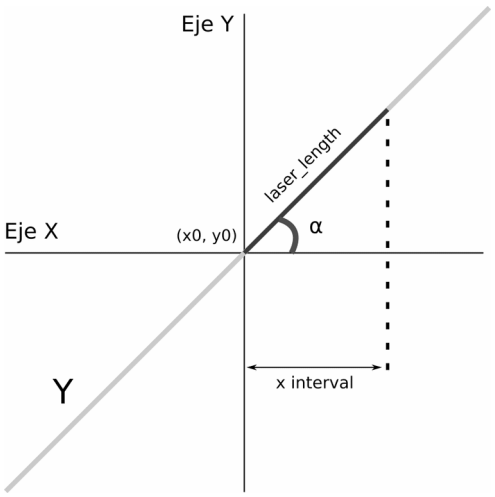


*Figura 2. Intervalo de valores en X necesarios para dibujar un rayo*

Donde **(x0, y0)** es el punto de origen del láser, **laser_length** la longitud del rayo y **α** el ángulo del rayo.

Dicho cálculo se realizará de forma sencilla aplicando trigonometría básica.

$$[x0, laserlength * \cos(\alpha) + x0]$$

Por último cabe destacar que todos los cálculos descritos en este apartado III están destinados a un lado específico del láser. Los valores de 0 a 90 grados tienen unas ecuaciones específicas y los de 90 a 180 grados tienen las opuestas.

## IV. FUNCIONAMIENTO DE LA RED NEURONAL

¿Qué es aprender? En Inteligencia Artificial, el aprendizaje no es más que la capacidad de adquirir conocimiento para a posteriori emplearlo en desarrollar una tarea encomendada. Se basa en optimizaciones iterativas hasta que se alcanza una meta o **límite**. En el caso de las redes neuronales, lo que internamente se desea es alcanzar un mínimo error a base de **épocas** o **iteraciones**.

Abstraemos una red neuronal como una **caja negra** que consta de un número N de entradas y un número M de salidas. Internamente, se basa en una red de neuronas interconectadas formando estructuras basadas en capas.

Como entradas para la red neuronal se ha usado un dataset generado a partir de un script (gendataset.m) que se encarga de ejecutar 3000 simulaciones donde dado un escenario preconfigurado, se toma las medidas del láser de un robot que por cada simulación se encuentra en una zona aleatoria y diferente.

La lectura del láser nos proporciona un vector de 90 valores en coma flotante. Dado el vector, calculamos las medias aritméticas del sector izquierdo (0-45) y del sector derecho (45-90) del láser. A continuación generamos la salida como una suma de las 2 medias (-left + right) multiplicado por un factor que podemos ajustar. El resultado es la velocidad lateral que usará el robot para esquivar los obstáculos. Nótese que cuando el láser detecta un obstáculo la media de las lecturas del sector correspondiente disminuyen aumentando el peso en el sector contrario, es decir, sucede una repulsión.

La idea de la repulsión está inspirada en el algoritmo de navegación local en Robótica denominado VFF (Virtual Force Field) en el cual la dirección del robot para alcanzar varios subobjetivos está determinado por 2 vectores: 1 vector de atracción hacia el siguiente subobjetivo y 1 vector de repulsión generado por el láser. La dirección se obtiene con la suma geométrica de los 2 vectores

Por lo tanto, nuestra red neuronal recibirá 2 entradas (media izquierda y derecha de las lecturas del láser) y devolverá 1 salida: velocidad lateral. En nuestro caso, la red neuronal ha sido óptima con 5 neuronas en la capa oculta y 1 neurona en la capa de salida (ya que sólo hemos tenido en cuenta la velocidad lateral como elemento de salida).

De todos los datos del dataset, el 75 % se ha empleado con datos de entrenamiento, el 15 % como datos de validación y otro 15 % como datos de test. Hemos establecido como parámetro de máximo fallo, 100 épocas consecutivas sin que se haya alcanzado un nuevo mínimo en la etapa de entrenamiento para evitar **overfitting**.

Al tratarse de un problema de Regresión (la salida se corresponde a un valor real), Matlab posee una función para generar este tipo de redes neuronales denominada **fitnet**(). Como parámetros le indiciamos que iba a tener 5 neuronas en la capa oculta y que el algoritmo de optimización usado sería el de Lavenberg-Marquardt (algoritmo por defecto de la función).

## REFERENCIAS


[1] *Inteligencia Artificial: Un Enfoque Moderno 2da Edición. Peter Norvig y Stuart J. Russell*

[2] Documentación MATLAB