Desarrollo de un Sistema Autónomo de Seguimiento Visual de Objetos Mediante Deep Q-Learning

Álvaro López García (alvaro.lopezgar@alumnos.upm.es)

***Resumen*—Este proyecto tiene como máximo objetivo el desarrollo de un sistema de seguimiento automático para un robot *Pioner 3-DX*, utilizando un controlador basado en *Deep Q-Learning*. El desarrollo se ha dividido en varias etapas, comenzando con la programación de un entorno *sofware* que nos permita tanto la interacción con el entorno de simulación, como la automatización del entrenamiento del modelo basado en inteligencia artificial. Después se ha procedido a la implementación de un sistema de control autónomo empleando métodos plenamente tradicionales capaz de desplazarse para seguir un objeto. Y por último, se ha realizado el diseño y entrenamiento de un modelo de *Deep Q-Learning* diseñado para realizar el seguimiento automático de una marca visual.**

En este documento se presenta un sumario del trabajo desempeñado abordando las técnicas empleadas, el progreso realizado y los resultados obtenidos durante el desarrollo. De este modo, se comentan detalles de la implementación, con el objetivo de su comprensión y replicación por parte de otros interesados en tecnologías similares.

***Palabras Clave*—Robótica, *CoppeliaSim*, *Pioneer 3-DX*, Aprendizaje Profundo, Redes Neuronales, Aprendizaje por Refuerzo, Seguimiento Automático, *Python*, *TensorFlow*, *Keras*, *TensorBoard*, *OpenCV*.**

I. Introducción

LOS recientes hitos en el mundo de la inteligencia artificial han propiciado esta se contemple como una posible solución solución a prácticamente cualquier problema. Esto ha permitido hallar nuevas fortalezas de este tipo de métodos así como el estudio de su viabilidad en diferentes campos.

Precisamente es ahí donde reside el valor de los modelos basados en inteligencia artificial, en su versatilidad. Ya que en muchos casos un modelo puede ser entrenado para otro propósito con tan solo realizar unos ligeros cambios en su especificación. Esto es infinitamente más sencillo, y por tanto más eficiente, que el desarrollo *ad-hoc* de cualquier tipo de solución algorítmica con métodos tradicionales. Normalmente este tipo de soluciones requerirían de un estudio del entorno en el que el modelo va a interaccionar, además de un conocimiento muy específico de este. Este tipo de metodologías son rígidas y difícilmente extrapolables, ya que desarrollar una nueva solución requeriría empezar de cero cada vez. El desarrollo de un modelo basado en inteligencia artificial no esta exento de esto, pues siempre se necesita un mínimo entendimiento del campo sobre el que se está trabajando (por ejemplo para extraer las características más significativas que van a conducir al modelo resultante). Pero la especialización que requieren las soluciones basadas en aprendizaje automático aumenta a menor razón de lo que lo hace la especialización que requieren las técnicas tradicionales conforme aumenta la complejidad de la tarea.

Precisamente a motivos como este se les puede atribuir el éxito de la inteligencia artificial en el mundo de la robótica. Hablamos de un conjunto de dispositivos (los robots) que interaccionan con el mismo mundo real de formas muy distintas: cada uno tiene diferentes capacidades, diferentes actuadores, diferentes sensores, etc. Pero precisamente, una solución única puede valer para muchos de ellos si esta es capaz de, (como en el caso de la inteligencia artificial) mediante entrenamiento, adaptarse a la idiosincrasia de cada aparato.

Es por esto que en este proyecto se investiga la aplicación de este tipo de técnicas a la tarea del seguimiento de un objeto mediante una marca visual, pues en este caso estudiamos la solución para un tipo de robot en concreto (el *Pioneer 3-DX*), pero dicha solución es potencialmente valiosa ya que es posible adaptarla a otro tipo de aparato con ligeros cambios y entrenamiento.

II. Estado del Arte

Independientemente de si hablamos de aprendizaje supervisado o no supervisado, cuando se habla del entrenamiento más clásico de redes neuronales para la realización de ciertos tipos de tareas, se suele hablar como consecuencia de algún tipo de banco de datos. De esta forma, a partir de dichos datos nuestro modelo aprende a realizar una tarea a partir de algún tipo de experiencia y que podremos medir de acuerdo a una métrica de rendimiento [1]. Para conseguir esto, se suele emplear dicha métrica de rendimiento para construir a su vez una función de coste que nos permita mejorar paulatinamente los parámetros de nuestro modelo (mediante algoritmos como el gradiente descendente) y, de este modo, alcanzar una solución que satisfaga nuestros requisitos.

Este tipo de planteamiento se ajusta a los paradigmas mencionados de aprendizaje supervisado o no supervisado, pues ambos requieren en su más mínima definición de algún tipo de *dataset* que permita realizar lo descrito en el anterior párrafo. El aprendizaje por refuerzo nos permite precisamente, entrenar un modelo mediante la interacción con un entorno en lugar de depender únicamente de *dataset* fijo. El aprendizaje por refuerzo [2] es un método estocástico que nos permite entrenar modelos que aproximan una distribución desconocida, pero que podemos construir en base a muestreos ya que podemos, de alguna forma, dar un valor a la bondad de las

acciones que desempeña este modelo. De este modo, un agente toma acciones en un entorno y recibe retroalimentación en forma de recompensas o penalizaciones. Este paradigma es particularmente relevante cuando se aborda el problema de la toma de decisiones secuenciales, como en el caso de juegos, robótica (nuestro caso) y control de sistemas.

III. Herramientas Software Desarrolladas

El entorno de simulación empleado para el desarrollo de este proyecto ha sido *CoppeliaSim*[1], controlado desde *Python*[2] en su versión 3.9 a través de la API que implementa. Y si bien el propósito de este proyecto no es la construcción de una arquitectura *software* en específico, la creación de los componentes *software* que han permitido desarrollarlo ha tomado una parte significativa del tiempo total, por lo que procede hacer algún ligero apunte sobre ello.

La filosofía de diseño empleada es la de DDD (*Domain-Driven Design*) [3]. Dicha filosofía se centra en que el dominio sea el que en última instancia guíe la toma de decisiones para el diseño del modelo, de tal modo que este se presente un comportamiento fidedigno a cómo lo haría el modelo en el entorno real. Algunos de los aspectos más relevantes de la implementación realizada son la presencia de distintos patrones por los que DDD aboga, como los servicios, las factorías o los repositorios. También cabe destacar que para el acceso a las funciones de *CoppeliaSim* a través de la API se ha desarrollado la clase *CoppeliaSimConnector*[3] y que a su vez implementa un patrón *singleton*, con el fin de evitar condiciones de carrera por el acceso a través de diferentes puntos.

También se han creado una serie de clases a modo de interfaces para operar con los elementos de la simulación de *CoppeliaSim* como para por ejemplo el *Pioneer 3-DX*, la esfera que sigue, el camino de puntos que recorre la esfera, etc. Toda la implementación software así como documentación más detallada sobre su uso se puede consultar en el repositorio adjunto a la entrega[4].

IV. Implementación del Sistema de Seguimiento

Con el fin de conocer el coste real de implementar una solución puramente algorítmica frente a diseñar un modelo de inteligencia artificial se ha desarrollado un algoritmo para controlar la velocidad de los dos motores del *Pioneer*.

Antes de describir el cómputo realizado por el algoritmo de control, hay que mencionar que tanto para esta solución como para la desarrollada empleando técnicas de inteligencia artificial se ha realizado la misma extracción de información a partir de la imagen. De este modo ambos algoritmos parten de la misma información y además se mantiene desacoplado el módulo de visión y procesado de imagen, por lo que en un futuro se podría trabajar con otras marcas visuales siempre y cuando se desarrolle un módulo de procesado de imagen que extraiga características análogas a aquellas con las que trabajan los algoritmos que se presentan en este documento.

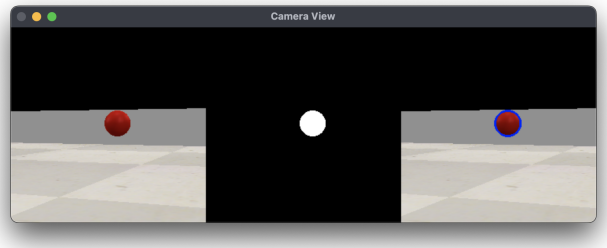

Figura 1. Pasos en el procesado llevado a cabo para la identificación de la marca visual.

En la figura anterior se pueden ver los pasos realizados en el procesado de imagen: a la izquierda, la visión "en bruto" que captura la cámara del *Pioneer*; en el centro, el resultado de aplicar una máscara de color que separa el color rojo del resto; y a derecha, los contornos identificados tras aplicar la máscara de color. Todo este *pipeline* de extracción de características a partir de la imagen se ha realizado invocando las pertinentes llamadas a las funciones de la librería *OpenCV*[5]. En la siguiente figura se muestra un esquema con las cotas de las variables extraídas de una imagen:

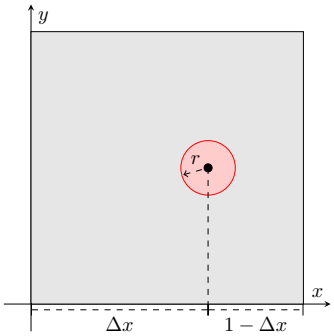

Figura 2. Esquema de la información obtenida a partir de una imagen de la cámara incorporada en el robot.

Y a partir de dichas variables se calcula $\vec{v} = (v_l, v_r)$ tal que:

$$v_l = v_{MAX} \cdot \sqrt{\left(1 - \frac{2\pi r}{x \cdot y}\right)} \cdot \text{mín}\left(\frac{\Delta x}{0{,}5}, 1\right)$$

$$v_r = v_{MAX} \cdot \sqrt{\left(1 - \frac{2\pi r}{x \cdot y}\right)} \cdot \text{mín}\left(\frac{1 - \Delta x}{0{,}5}, 1\right)$$

Podemos ver cómo en el cálculo de estas velocidades influyen tres factores: la velocidad máxima, que es una constante definida en nuestro código; una interpolación cuadrática, que nos sirve para disminuir la velocidad máxima conforme la pelota ocupa el total de la pantalla (noción de proximidad); y una interpolación lineal para calcular la velocidad de cada

[1] https://www.coppeliarobotics.com
[2] https://www.python.org
[3] Ruta hasta el archivo: *src/simulations/infrastructure/ coppelia_sim_connector.py*
[4] https://github.com/alvaro-lg/Robotica
[5] https://opencv.org

motor de tal forma que ambas son máximas cuando esta esta en el centro y cuando esta está en uno de los bordes laterales alternadamente una es máxima y la otra mínima.

La implementación del controlador que realiza este cómputo se provee en la clase *VisualAgent*[6]. Este controlador ha demostrado un gran rendimiento cuando la velocidad máxima que se presuponía para el cómputo era algo moderada. Sin embargo, cuando se incrementaba el valor de esta constante el algorimo fallaba en mantener la pelota centrada.

V. Implementación del Sistema de Seguimiento Basado en Deep Q-Learning

Para la implementación del sistema de control basado en inteligencia artificial se ha optado por un modelo DQN (*Deep Q-Network*) "clásico". Este tipo de modelos operan sobre un conjunto discreto de acciones que hay que predefinir en la fase de diseño del mismo. En nuestro caso se han escogido tres acciones, que se muestran en la siguiente figura:

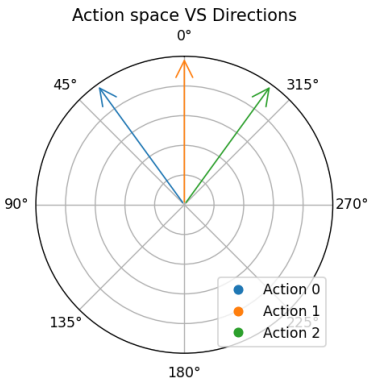

Figura 3. Direcciones escogidas en el diseño del modelo.

De este modo, nuestra red será un aproximador $f : \mathbb{R}^2 \longrightarrow \mathbb{R}^3$ con las siguientes características:

- Tamaño de memoria de repetición: 3000 elementos.
- Tamaño de lote de entrenamiento: 128 elementos.
- Se usa un modelo *target* (objetivo): Este modelo se emplea para realizar las predicciones que el robot hace sobre el entorno mientras una copia de la red se entrena. Al término de un número de iteraciones (20 en nuestro caso), se realiza una copia de este y se actualiza el modelo empleado para las predicciones. De este modo la convergencia es más estable y se evitan los mínimos locales.
- Número de episodios entrenados para el entrenamiento: 400.
- Se emplea la técnica de ϵ-*greedy* ("Epsilon-*greedy*"), para favorecer de forma estocástica y condicionada al número de entrenamientos realizados, la exploración de nuevas acciones para determinados estados sobre la explotación de las ya conocidas. Para obtener más información acerca de cómo fluctúa ϵ a lo largo del entrenamiento, ver la clase *TrainService*[7].

[6]Ruta hasta el archivo: *src/simulations/domain/controllers/visual_agent.py*
[7]Ruta hasta el archivo: *src/simulations/application/train_service.py*

- Para la arquitectura de la red se han empleado un total de 4 capas con las siguientes dimensiones: $(2 \times 1), (16 \times 1), (8 \times 1), (3 \times 1)$. Dicha arquitectura se ilustra en el siguiente esquema:

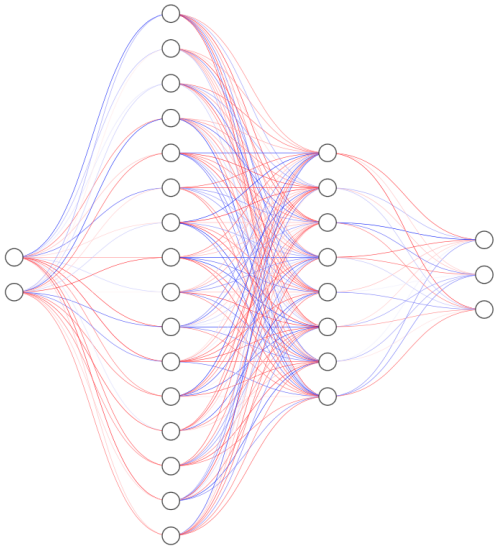

Figura 4. Arquitectura de red empleada[8].

  Llegados a este punto hay que mencionar que, con el fin de evitar problemas de *overfitting* (sobreajuste) y procurar una mayor robusted del modelo se ha incluido una capa de *Dropout* entre las capas segunda y tercera de la Figura 4.

El proceso de entrenamiento se ha realizado colocando el robot en frente de la pelota y procurando que durante cada episodio este se dirija a ella. La distancia a la que se coloca el robot de la pelota varía en cada inicialización del episodio, de nuevo para evitar problemas de *overfitting*.

La asignación de recompensa se realiza de forma analítica y de acuerdo a la siguiente función que toma como argumento la coordenada x normalizada (ver Figura 2):

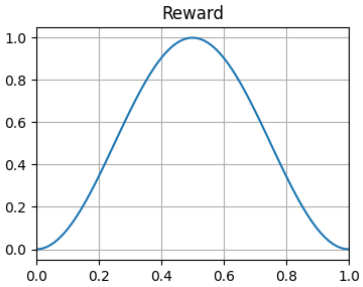

Figura 5. Función de recompensa empleada

Dicha función es una sinusoide con longitud de onda $\lambda = 1$, amplitud $A = 0{,}5$ y que toma su valor máximo en $x = 0{,}5$. De este modo, la recompensa máxima se asigna cuando la pelota se mantiene centrada. Y dado el carácter episódico de

[8]Figura realizada con la herramienta http://alexlenail.me/NN-SVG/index.html

la tarea (el seguimiento de la esfera), el episodio se termina forzosamente cuando el robot la pierde de vista además de penalizarlo con una recompensa negativa. También se ha probado la asignación de recompensa de forma relativa (por ejemplo: si la esfera esta más centrada en un instante que en el anterior), pero se han observado peores resultados en el entrenamiento, además de el incremento en complejidad de la función de recompensa, pues hay que tener en cuenta más casuísticas en su diseño.

VI. Resultados y Trabajo Futuro

Tras el entrenamiento del modelo durante los 400 episodios mencionados, se han obtenido los siguientes valores de recompensa total:

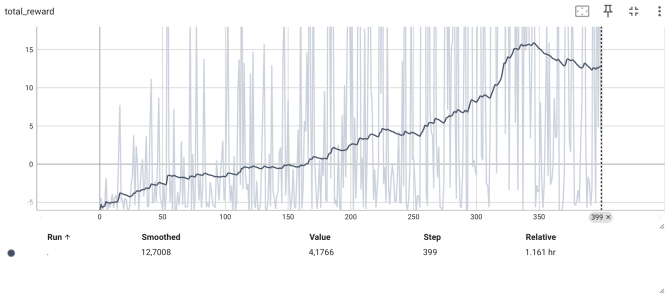

Figura 6. Recompensa total obtenida por episodio[9].

A los valores de dicha evolución se les ha aplicado un suavizado para poder ver mejor su tendencia, pues la naturaleza estocástica del entrenamiento hace que su duración sea variable (de acuerdo a las normas que se han establecido) y por tanto que la gráfica sea muy ruidosa.

A continuación, se muestra también una gráfica con las superficies generadas por la probabilidad (aplicando una distribución *Softmax* a los *Q*-valores) de escoger cada acción de acuerdo a pares coordenada x - área ocupada de entrada:

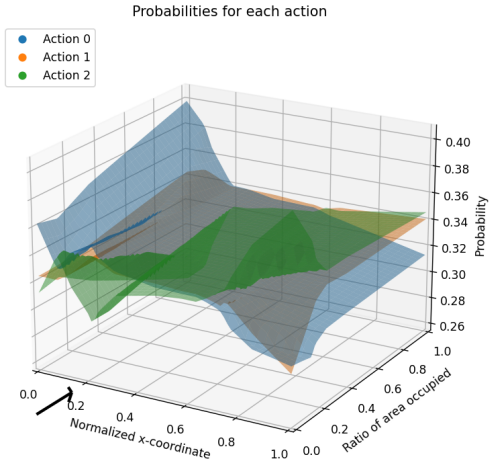

Figura 7. Probabilidad de predicción de cada acción de acuerdo a pares de valores de entrada aplicando normalización *Softmax*.

[9]Figura obtenida mediante la herramienta *TensorBoard* (https://www.tensorflow.org/tensorboard).

En estas superficies podemos ver cómo el modelo reacciona correctamente a las posiciones de la esfera en la pantalla.

En conclusión, el modelo *DQN* ha presentado mejor escalabilidad de que el desarrollado con técnicas clásicas cuando se establece una v_{MAX} mayor. Y a partir de aquí se plantean las siguientes líneas de trabajo futuro:

- Estudiar la inclusión de técnicas de inteligencia artificial en el proceso de reconocimiento de la marca visual.
  - Precisamente esto presenta un gran potencial en entornos de entrenamiento no ideales, donde el modelo debería de encargarse además de otras tareas como podría ser evitar obstáculos mientras sigue el objeto.
- Procurar extrapolar el entrenamiento del modelo a otros robots cuya forma de desplazarse por el entorno sea más compleja precisamente para ilustrar la versatilidad de las técnicas de inteligencia artificial.
- Plantear la incorporación de arquitecturas de red más complejas en el modelo *DQN*. Entre las distintas alternativas posibles, se consideran las más interesantes aquellas que tienen memoria (como por ejemplo las redes neuronales autorregresivas o los archiconocidos *transformers*) pues estos tienen capacidad (además de comparar con el instante que tiene lugar en el momento de la predicción) de tener en cuenta n momentos anteriores, lo que les puede dar potencialmente la capacidad de corregir la trayectoria.

Referencias

[1] T. M. Mitchell, *Machine learning*. McGraw-hill New York, 1997, vol. 1.

[2] R. S. Sutton y A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[3] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.