Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Inteligencia Artificial

Trabajo Fin de Máster

Navegación Reactiva Aplicada a Agentes Fisicos en Entornos Domésticos usando Habitat Sim

Autor(a): Luna Jiménez Fernández
Tutores: Martín Molina Gómez
María Julia Flores Gallego

Madrid, Septiembre - 2021

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Máster
Máster Universitario en Inteligencia Artificial

Título: Navegación Reactiva Aplicada a Agentes Fisicos en Entornos Domésticos usando Habitat Sim

Septiembre - 2021

Autor(a): Luna Jiménez Fernández
Tutor(a): Martín Molina Gómez
Computer Vision and Aerial Robotics (CVAR)
ETSI Informáticos
Universidad Politécnica de Madrid
Co-autor(a): María Julia Flores Gallego
Sistemas Informáticos (SI)
Escuela Superior de Ingeniería Informática de Albacete (ESIIAB)
Universidad de Castilla-La Mancha

Resumen

«Aquí va el resumen del TFM. Extensión máxima 2 páginas.»

Abstract

«Abstract of the Master Project. Maximum length: 2 pages.»

Agradecimientos

Tabla de contenidos

Índice de figuras

Capítulo 1

Introducción

En este capítulo se realizará una breve introducción a los contenidos que serán expuestos posteriormente a lo largo de la memoria. Tras esta presentación, se expondrá la motivación que ha propiciado el desarrollo de este trabajo. Finalmente, se describirá la estructura seguida por la memoria.

1.1. Introducción

La **navegación autónoma** de robots en entornos desconocidos y complejos es un problema de gran interés en la actualidad para el que se ha propuesto una amplia gama de soluciones, buscando que éstas sean a la vez eficientes durante su entrenamiento y capaces de navegar entornos de forma exitosa. Una de las familias de algoritmos más relevantes para este propósito son los **algoritmos de aprendizaje por refuerzo**, capaz de aprender de forma autónoma a navegar entornos desconocidos a partir de experiencia previa, con gran éxito.

Además, la **simulación virtual** tanto de estos robots como de otros problemas es un campo en crecimiento, especialmente durante la pandemia del CoVID-19, al verse limitadas las capacidades de experimentación en entornos físicos. Por tanto el objetivo de este trabajo es aunar el **desarrollo de un algoritmo híbrido eficiente** para la navegación en entornos complejos (como el interior de un domicilio) con el **estudio y uso de *Habitat Sim***, un simulador novedoso para el entrenamiento y evaluación de agentes robóticos físicos.

Esta memoria comienza con una revisión de las principales técnicas usadas actualmente tanto en *Deep Learning* como en aprendizaje por refuerzo y en navegación autónoma de robots (centrándose en los algoritmos de *Artificial Potential Field*, o *APF* por sus siglas en inglés). Tras ésto, se realiza un estudio en detalle del simulador *Habitat Sim* y su principal librería / *API* para *Python*, *Habitat Lab*; centrándose en los principales componentes de la librería, su funcionamiento y su uso. Posteriormente, se describe el diseño e implementación que se ha realizado en el trabajo, haciendo hincapié en la representación del conocimiento (problema a resolver y definiciones de estado, acción y recompensa), las arquitecturas del agente propuestas y su funcionamiento tanto durante el entrenamiento como la actuación.

Se procede después a la explicación de los experimentos realizados (tanto los parámetros usados como los experimentos a realizar), analizando el rendimiento de las

variantes propuestas del agente durante el entrenamiento y en una evaluación posterior, analizando estos resultados y comparándolos con otros agentes usados como *benchmarks*. Finalmente, se interpretarán estos resultados, extrayendo unas conclusiones y ofreciendo futuras líneas de trabajo a partir del conocimiento adquirido.

1.2. Motivación

Este trabajo se puede entender como una continuación del trabajo realizado por C. Sampedro *et al.* en 2018 [1], en el que se desarrolla con buenos resultados un sistema de navegación autónomo para drones aéreos usando aprendizaje por refuerzo profundo con campos de potenciales artificiales y láseres para percibir el entorno. Una de las metas de este trabajo es estudiar si la implementación de un algoritmo de características similares pero aplicado a robots terrestres usando cámaras de profundidad en interiores (domicilios, fábricas...) sería igualmente efectivo.

Además, la situación de pandemia actual ha dejado en evidencia la necesidad del uso de simuladores, especialmente para algoritmos que necesiten un entrenamiento largo y que puedan necesitar equipamiento especializado para ello (como robots, drones, instalaciones especializadas...). Por eso, otra de las principales metas del trabajo es el estudio de la herramienta *Habitat Sim* [2] [3], viendo su viabilidad de cara a futuros trabajos.

Para acabar, otra razón no despreciable para la elección de esta temática de trabajo es el propio interés de la alumna por el campo del aprendizaje por refuerzo profundo. Ya se realizó un trabajo previo estudiando la aplicación de estas técnicas a juegos reales como Tetris [4], y este trabajo sirve para ampliar más el conocimiento y aplicarlo a tecnologías modernas y a otros campos de interés.

1.3. Estructura

Esta memoria está dividida en un total de 7 capítulos, que serán descritos brevemente a continuación.

- **Capítulo 1:** En este capítulo se introduce el trabajo desarrollado, la motivación que ha llevado a éste y la estructura general de la memoria.
- **Capítulo 2:** En este capítulo se describe en profundidad el problema a resolver, presentando los antecedentes previos al trabajo realizado y detallando los objetivos que se esperan cumplir.
- **Capítulo 3:** En este capítulo se realiza una revisión de las principales técnicas en los campos relacionados con el trabajo: *deep learning* y redes neuronales convolucionales, aprendizaje por refuerzo (estudiando tanto las técnicas clásicas como las técnicas de aprendizaje por refuerzo profundo) y algunos de los principales algoritmos de navegación automática.
- **Capítulo 4:** En este capítulo se presentan tanto *Habitat Sim* como *Habitat Lab*, las principales herramientas usadas durante el desarrollo del trabajo. Tras esto, se exponen los principales componentes de Habitat Lab, explicando su funcionamiento y uso. Finalmente, se habla sobre la instalación y las dependencias necesarias del simulador.

- **Capítulo 5:** En este capítulo se detalla el diseño del agente de navegación reactiva propuesto. Se describe tanto la representación del conocimiento (estado, acciones y recompensas) como la arquitectura, el método de actuación y el entrenamiento llevado a cabo por el agente. Finalmente, se realiza una breve explicación del funcionamiento y la arquitectura del resto de agentes usados como *benchmarks* y comparativas ofrecidos por *Habitat Lab*.
- **Capítulo 6:** En este capítulo se detalla la experimentación realizada, indicando los parametros utilizados. Además, se presentan los resultados y el rendimiento obtenido por los agentes tanto durante el entrenamiento como durante la evaluación posterior.
- **Capítulo 7:** Finalmente, en este capítulo se presentan las conclusiones alcanzadas tras el desarrollo del trabajo, proponiendo posibles líneas de trabajo futuro para continuarlo.

Además, al final de la memoria se incluye una bibliografía en la que se encuentra la lista de fuentes y referencias usadas a lo largo de ésta.

Capítulo 2

Descripción del problema

En este capítulo se planteará en detalle el problema a resolver. Tras esto, se comentarán soluciones previas propuestas al problema, y se describirán los objetivos que se esperan alcanzar con el desarrollo del trabajo.

2.1. Definición del problema

El problema a resolver es el diseño de un agente físico (un robot) capaz de navegar desde una posición inicial hasta una posición final (conocidas sus coordenadas) en un entorno de interior del que no se tiene conocimiento previo (como puede ser el interior de una casa) de forma eficiente. Este problema se encuadra en el campo de la navegación automática, concretamente en el de la **navegación a meta** (*Point Goal Navigation*) **sin exploración previa** del entorno [5].

Para lograr esto, es necesario entrenar al agente en un entorno controlado para que aprenda a navegar de forma autónoma en interiores desconocidos, usando alguna técnica de aprendizaje por refuerzo. Además. se busca que el agente sea capaz de navegar **sin exploración previa** (como se ha mencionado anteriormente), buscando un sistema reactivo frente a uno basado en exploración-navegación.

El agente físico cuenta con las siguientes características a tener en cuenta:

- **Movimiento:** El agente es terrestre (se desplaza con ruedas sobre el suelo), pudiendo moverse hacia adelante y rotar sobre si mismo. El agente no es capaz de realizar movimientos ortogonales.
- **Sensores:** El agente cuenta con dos sensores para recibir información del entorno: una **cámara de profundidad** para observar el espacio frente al robot y un **GPS**, indicando la distancia y ángulo hasta la meta.

2.2. Antecedentes

La navegación autónoma de robots en interiores es un campo de gran interés tanto para la investigación como para la industria, existiendo gran cantidad de grupos dedicados a la propuesta y desarrollo de agentes capaces de resolver estos problemas de forma eficaz.

Algunos ejemplos de este interés *Habitat Challenge*, un desafío anual organizado por *Facebook AI Research* (el grupo de investigación de inteligencia artificial de Facebook) parte de la *Conference on Computer Vision and Pattern Recognition (CVPR)* en el que se buscan los mejores algoritmos para resolver problemas de **navegación autónoma a metas** [6] (aunque también incluye problemas de **navegación autónoma a objetos** [7]) aplicados a agentes físicos en entornos de interiores. Algunas de las mejores propuestas de estos últimos años han sido:

- **Devendra Singh Chaplot *et al.* (2019)** [8]: Se propone un nuevo algoritmo, *Active Neural SLAM (Simultaneous Localization And Mapping)*, que combina las capacidades de planificadores de ruta clásicos como SLAM con la capacidad del aprendizaje por refuerzo profundo para generar políticas de acciones locales y globales. Esta propuesta obtuvo el primer puesto del *Habitat Challenge 2019*.
- **Santhosh K. Ramakrishnan *et al.* (2020)** [9]: Se propone un sistema de navegación entrenado con aprendizaje por refuerzo que, a partir de sus observaciones a través de una cámara RGB, es capaz de inferir la posición de los objetos más allá de su ángulo de visión para mejorar su rendimiento a la hora de generar un mapa de su entorno. Esta propuesta alcanzó el primer puesto del *Habitat Challenge 2020*.
- **Samyak Datta *et al.* (2020)** [10]: Se propone un sistema de navegación entrenado con aprendizaje por refuerzo (*PPO*) que tiene en cuenta el ruido existente en entornos reales (problemas durante la actuación, desviaciones entre la posición real y estimada...) y hace especial hincapié en solventar los problemas causados por éste. Esta propuesta alcanzó el segundo puesto del *Habitat Challenge 2020*.
- **Ruslan Partsey (2021)** [11]: Se desarrolla un agente que usa técnicas de odometría (usar datos de sensores de movimiento para estimar la posición real) visual con aprendizaje por refuerzo para conseguir una navegación eficiente en entornos con ruido. Esta propuesta alcanzó el primer puesto del *Habitat Challenge 2021*.

Si bien todas las propuestas anteriores utilizan técnicas de aprendizaje por refuerzo, la gran mayoría de éstas utilizan enfoques basados en el mapeado y navegación de los entornos, frente a una propuesta puramente reactiva (sin mapa) como la de éste trabajo.

El antecedente más directo al trabajo descrito en esta memoria es la propuesta realizada por **Carlos Sampedro *et al.* (2018)** [1], siendo éste trabajo una adaptación y continuación directa. El agente propuesto utiliza un método basado en campos de potenciales, entrenado con aprendizaje por refuerzo profundo para navegar un dron aéreo a través de entornos de interior, usando un conjunto de láseres (para percibir el entorno a su alrededor) y la posición relativa del propio dron respecto a la meta, con buenos resultados. Esta arquitectura se puede observar en la Figura 2.1.

Ahora bien, existen diferencias destacables entre la propuesta original y el trabajo descrito en esta memoria:

- **Movimientos del agente:** La propuesta original utiliza como agente a un dron multirotor, capaz de navegar por el aire (aunque se mantiene a una altura constante) y de realizar movimiento omnidireccional. En cambio, el trabajo desarrollado utiliza un robot terrestre (susceptible a obstáculos en el suelo) incapaz

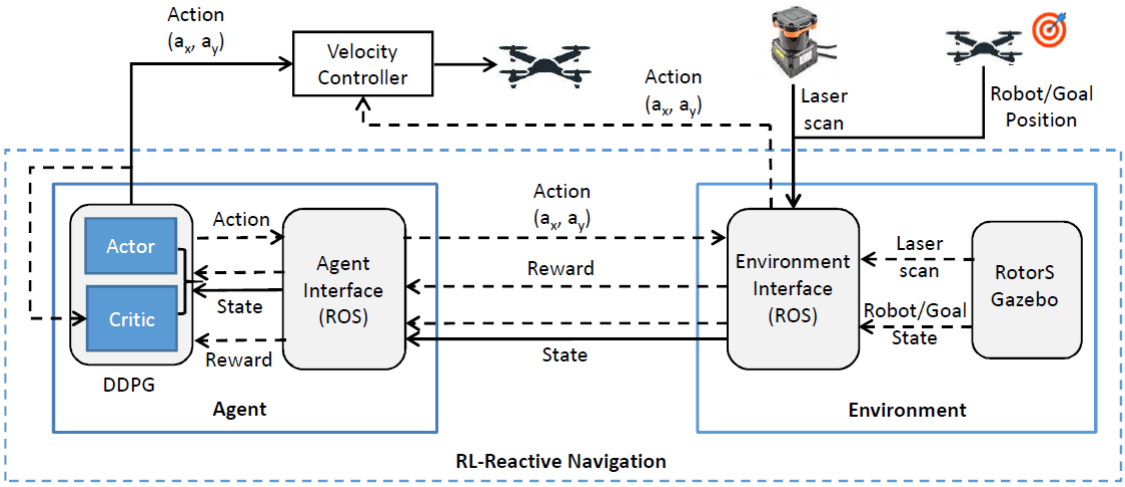

Figura 2.1: Arquitectura propuesta para el sistema de navegación reactivo original [1].

de movimiento omnidireccional, siendo necesario el giro del agente para poder esquivar obstáculos y desplazarse.

- **Sensores del agente:** Mientras que la propuesta original utiliza un conjunto de láseres para percibir su entorno, el trabajo desarrollado propone un agente con una única cámara de profundidad frontal. Si bien el conjunto de láseres resulta más caro que una cámara de profundidad, también ofrece un ángulo de visión mayor de los obstáculos del entorno.
- **Complejidad del entorno:** La propuesta original entrena al agente en entornos de interior simples (contando con espacios simples con obstáculos dispersos), frente a los entornos usados por el trabajo desarrollado (siendo recreaciones del interior de domicilios, incluyendo topografías más complejas con cuartos y una mayor cantidad de obstáculos y ruido).

2.3. Objetivos

El principal objetivo de este trabajo es el estudio y aplicación de técnicas de aprendizaje por refuerzo profundo y navegación autónoma basada en campos de potenciales para el desarrollo de un agente capaz de navegar entornos de interior, evaluando su viabilidad y eficacia.

Para cumplir este objetivo, es necesario a su vez cumplir una serie de objetivos parciales:

- Revisión de bibliografía para comprender plenamente las técnicas a usar durante el desarrollo.
- Búsqueda y evaluación de librerías y herramientas disponibles para el desarrollo del agente (incluyendo simuladores, entornos de trabajo...)
- Caracterización, formalización e implementación del agente y de posibles variaciones propuestas dentro del entorno elegido, para poder ser evaluado posteriormente.

- Realización de experimentos para estudiar el comportamiento del agente durante el entrenamiento y posteriormente al enfrentarse a problemas reales.
- Estudio y análisis de los resultados, realizando comparación con *benchmarks* para extraer observaciones y conclusiones que permitan valorar la viabilidad y eficacia del agente propuesto.

Este trabajo además aborda un segundo objetivo, el estudio y uso del simulador *Habitat Sim*, con el fin de evaluar su utilidad de cara a posteriores trabajos. Para esto, se plantean los siguientes objetivos parciales:

- Revisión y estudio de documentación oficial y ejemplos ofrecidos por el simulador.
- Desarrollo del agente descrito previamente en el marco del simulador, usando las herramientas ofrecidas.
- Creación de documentación sobre el uso adecuado del simulador para facilitar trabajos posteriores.
- Evaluación de la idoneidad del simulador para la resolución de problemas de navegación autónoma.

Capítulo 3

Revisión de técnicas

En este capítulo se describirán las técnicas y algoritmos que preceden y en los que se basa el trabajo descrito en esta memoria. Concretamente, se realizará una introducción a redes neuronales y *Deep Learning*, tras la cual se hablarán de los principales métodos de aprendizaje por refuerzo clásicos y actuales. Finalmente, se estudiarán algunos de los principales algoritmos de navegación autónoma existentes.

3.1. Redes neuronales y *Deep Learning*

En esta sección se describe el concepto de las redes neuronales y sus principales características. Tras esto, se describe *Deep Learning* y uno de sus modelos más característicos, las redes convolucionales.

3.1.1. Redes neuronales

En el campo de la inteligencia artificial, una **neurona** (también conocida como **nodo**) es la unidad lógica básica que forma una red neuronal [12]. En esencia, una neurona es una función matemática no lineal que, a partir de unas entradas, genera una salida. Además, las neuronas se conectan entre ellas a través de conexiones dirigidas, por lo que se puede propagar esta salida entre ellas.

La estructura básica de una neurona se puede observar en la Figura 3.1, siendo los componentes principales [12]:

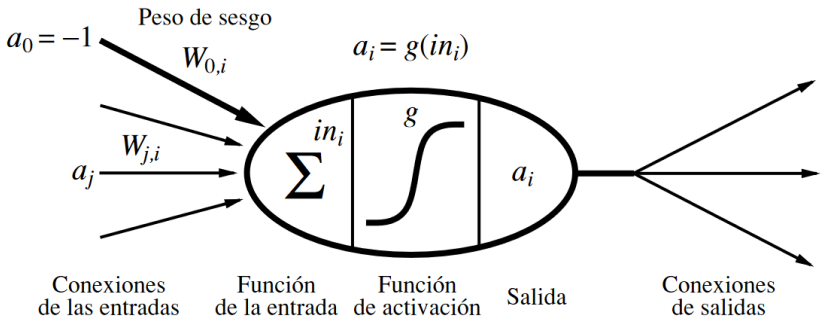

Figura 3.1: Estructura de una neurona artificial [12].

- **Conexiones de las entradas:** El conjunto de entradas recibidos por la neurona. Estas entradas a_j están ponderadas por pesos $w_{j,i}$ (siendo i la neurona actual y j el origen de la entrada) determinando el peso y el signo de cada entrada.
- **Función de entrada:** La función de entrada in_i no es más que el sumatorio de todas las entradas ponderadas que recibe la función, siendo éste:

$$in_i = \sum_{j=0}^{N} w_{j,i} a_j$$

  Donde N es el número total de entradas.
- **Función de activación:** La función de activación g es una función aplicada a la función de entrada para generar la salida a_i. Esta función debe cumplir que:
  - La función debe devolver valores adecuados (la neurona debe estar activa o con una salida cercana a $+1$ con las entradas correctas y apagada o con salida cercana a 0 en otro caso).
  - La función debe ser no lineal para evitar que toda la red neuronal se pueda simplificar a una función lineal simple [12].

  Hay varias funciones de activación usadas, siendo algunas de las más comunes la función umbral o la lineal. Ahora bien, la función más típica es la **función sigmoide**, teniendo ésta la fórmula:

$$g(in_i) = \frac{1}{1 + e^{-in_i}}$$

- **Salida:** La salida a_i no es más que la aplicación de la función de activación sobre la función de entrada, $a_i = g(in_i)$. Esta salida se puede propagar a otras neuronas (siendo ésta una de las entradas de dichas neuronas), o servir como salida de la red neuronal.

Como se puede ver en la Figura 3.1, las neuronas tienen una entrada constante a_0 conocida como la entrada de sesgo, de valor $a_0 = -1$, cuyo peso se conoce como el **peso de sesgo**. La utilidad de este peso es definir el umbral real de la neurona (es decir, una neurona se activa únicamente cuando la suma de todas las entradas supera a este peso de sesgo) [12]. El peso de sesgo permite desplazar la activación de la neurona sin alterar la pendiente, de forma similar al término independiente de una recta.

Definida una neurona, se puede definir una **red neuronal artificial** como un grupo de neuronas trabajando en conjunto, representando una función compleja no lineal con una gran cantidad de parámetros (los pesos de sus neuronas). [12]. Existen dos tipos de redes neuronales: redes cíclicas o **recurrentes** y redes acíclicas o de **propagación hacia adelante** [12], siendo éstas el foco de esta sección.

Una **red neuronal de propagación hacia adelante** es una red acíclica de neuronas representando una función a partir de las entradas de la red, sin ningún tipo de estado interno o memoria [12]. Este tipo de redes suele estar organizado en **capas**, siendo una capa un conjunto de neuronas de forma que una neurona recibe únicamente entradas de la capa anterior y propaga sus entradas únicamente a la capa posterior

[12]. Dependiendo del número de capas, se puede hablar de redes de una única capa (**perceptrones**) o de redes de varias capas (**perceptrones multicapa** o *MLPs*) [12].

Un **perceptrón multicapa** es la estructura más típica de red neuronal de propagación hacia adelante, siendo ésta una red neuronal con más de una capa [12], teniendo cada capa un número variable de neuronas. La principal ventaja de este tipo de estructura es su capacidad de (con un número suficiente de neuronas) aproximar cualquier función no lineal [13]. Se puede observar un ejemplo de perceptrón multicapa en la Figura 3.2, viendo que la estructura se divide en tres partes [14]:

- **Capa de entrada:** Representa las entradas de la red (las entradas de la función a simular). Es típico que estas neuronas no usen una función de activación.
- **Capa(s) oculta(s):** Una o más capas ubicadas entre las capas de entrada y salida. Estas capas se encargan de identificar las relaciones existentes entre las entradas durante el entrenamiento [14], para obtener la salida esperada.
- **Capa de salida:** Representa las salidas finales procesadas de la red (las salidas de la función a aproximar).

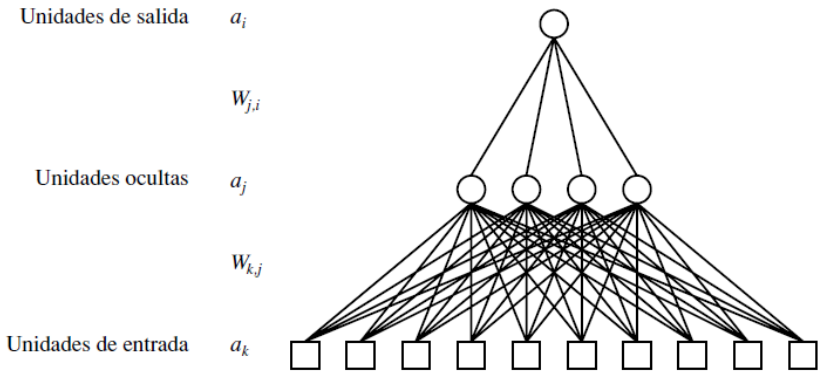

Figura 3.2: Perceptrón multicapa con una capa oculta [12].

El principal objetivo a la hora de desarrollar una red neuronal es encontrar el conjunto de pesos que logre que, para unos valores de entrada, la salida de la red neuronal se aproxime lo máximo posible al comportamiento esperado en el problema real [15]. Esto se consigue **entrenando** la red neuronal a partir de un conjunto de entrenamiento (cuyos valores de salida son conocidos), utilizando el algoritmo de **retropropagación** (propuesto originalmente por Rumelhart *et al.* en 1986 [16]). El objetivo del algoritmo es ajustar los pesos de la capa de salida para minimizar el error, y posteriormente propagar este ajuste a las capas anteriores para ajustar sus pesos usando gradiente descendiente, con el objetivo de minimizar el error de la red.

El pseudocódigo del algoritmo se puede ver en la Figura 3.3. Por lo general, la condición de parada del algoritmo de retropropagación consiste en alcanzar un umbral de error o realizar el proceso un número determinado de veces sobre el conjunto de entrenamiento entero. Cada ciclo del algoritmo sobre el conjunto de entrenamiento completo se conoce como una **época** (o *epoch* por su nombre en inglés) [12].

Algoritmo 1: Algoritmo de retropropagación

Variables iniciales: Conjunto de entrenamiento ent, formado por tuplas de elementos (x, y) (siendo x las entradas de la red y y las salidas esperadas de la red). Pesos de la red neuronal $w_{i,j}$ donde i, j indica un peso dirigido de la neurona i a la neurona j.

1. Inicializa los pesos de la red neuronal utilizando valores aleatorios.

2. Mientras no se cumpla la condición de parada:

2.1. Para cada ejemplo (x, y) contenido en el conjunto de entrenamiento ent:

2.1.1. Calcula la salida h_N para todas las neuronas N de la red neuronal, utilizando propagación hacia adelante.

2.1.2. Calcula el error $error$, las salidas obtenidas h_N y las salidas esperadas y_N para cada neurona N.

2.1.3. Para cada neurona N (empezando por las neuronas de la capa de salida, y retrocediendo en orden hasta la capa de entrada):

2.1.3.1. Calcula la influencia de la neurona N en el error final, δ_N:
Si N es una neurona de salida:

$$\delta_N = (y_N - h_N) * h_N * (1 - h_N)$$

En otro caso (N pertenece a una capa oculta):

$$\delta_N = \left(\sum_{X \in sucesores(N)} w_{N,X} * \delta_X \right) * h_N * (1 - h_N)$$

Donde $sucesores(N)$ es un conjunto de las neuronas en la capa posterior a la capa de la neurona N.

2.1.3.2. Para todas las neuronas R pertenecientes a la capa anterior a la capa de la neurona N, calcula la actualización $\Delta_{R,N}$:

$$\Delta_{R,N} = \eta * \delta_N * h_R$$

Donde η es el factor de aprendizaje de la red neuronal.

2.1.4. Para cada peso $w_{R,N}$ en la red neuronal, actualiza el peso:

$$w_{R,N} = w_{R,N} + \Delta_{R,N}$$

3. Devolver la red con los pesos entrenados.

Figura 3.3: Algoritmo de retropropagación para entrenamiento de perceptrones multicapa.

3.1.2. Aprendizaje por representación y *Deep Learning*

El **aprendizaje por representación** es un conjunto de métodos contenidos dentro del aprendizaje automático cuyo objetivo es, dada una entrada sin procesar, extraer las características más relevantes de la entrada para trabajar con ellas [13]. Esto permite resolver uno de los problemas más típicos del aprendizaje automático, determinar las características más relevantes de un problema. Ahora bien, los algoritmos pueden tener problemas para extraer directamente características abstractas de alto nivel (como los contornos de un objeto o los acentos de una voz).

Para solucionar este problema surge ***Deep Learning***, un subconjunto de las técnicas de aprendizaje por representación que utilizan varios niveles de abstracción, partiendo de conceptos básicos que van componiendo para alcanzar conceptos complejos y abstractos [13]. La principal ventaja frente a otros métodos es que estas características son identificadas automáticamente por los algoritmos, sin necesidad de un experto que guíe el aprendizaje.

Actualmente, las técnicas de *Deep Learning* se han vuelto el estado del arte en una gran cantidad de problemas que se consideraban inabordables hace apenas una década, como la visión artificial, el reconocimiento de voz o el procesamiento de lenguaje natural [13] entre otros.

La familia de modelos más característica de *Deep Learning* es las redes neuronales, siendo uno de los modelos más usados los perceptrones multicapa descritos previamente [13]. Estas redes se conocen también como **redes neuronales profundas**, teniendo éstas una gran cantidad de capas ocultas y neuronas por capa (donde cada capa se encarga de aprender un nivel de representación de la entrada).

Ahora bien, el aumento del tamaño de las redes neuronales profundas (tanto en neuronas como en capas) conlleva una serie de problemas:

- Conforme aumenta el tamaño de la red, aumenta el número de parámetros (pesos) a ajustar. Esto significa que se necesitan conjuntos de entrenamiento exponencialmente más grandes para alcanzar buenos resultados y, por tanto, un mayor coste computacional y de tiempo para el entrenamiento.

  Esto se ha solucionado principalmente mediante el uso de ordenadores más potentes, conjuntos de datos más grandes y técnicas para ampliar los conjuntos de datos ya existentes como el *data augmentation* [17].

- **Problema del gradiente desvaneciente / explosivo (*Vanishing / Exploding gradient problem):*** Al aumentar el número de capas, la eficiencia del algoritmo de retropropagación disminuye si se utilizan las funciones de activación tradicionales (como la función sigmoide) [18]. Esto se debe a que la propagación del error puede diluirse (provocando cambios demasiado lentos) o incrementar rapidamente (provocando inestabilidad) conforme se propaga de capa en capa.

  Una de las principales soluciones al problema es el uso de funciones de activación específicas como *ReLU* [13], siendo la fórmula:

$$g(in_i) = max\{0, in_i\}$$

Si bien las redes neuronales profundas son uno de los modelos más usados en *Deep Learning*, también es frecuente el uso de otros tipos de redes neuronales como redes

recurrentes, redes recursivas, redes residuales... o redes convolucionales, como se verán a continuación [13].

3.1.3. Redes neuronales convolucionales

Una **red neuronal convolucional** es una red neuronal que usa funciones de convolución, diseñada para trabajar con entradas en forma de matrices (como líneas de tiempo o imágenes) [13]. Éstas fueron originalmente propuestas por Yann LeCun *et al.* en 1989 [19], si bien cobraron importancia tras *AlexNet* [20] y sus resultados en la competición de *ImageNet*.

En el campo del aprendizaje automático, una **convolución** es una operación matemática lineal aplicada sobre dos funciones de números reales: una **entrada (*input*)** I, generalmente una matriz de datos como puede ser una imagen; y un **nucleo (*kernel*)** K, una matriz de parámetros entrenables. La salida de esta operación es otra matriz, típicamente conocida como el **mapa de características (*feature map*)** S [13]. Esta función normalmente se define como:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Donde m y n son las dimensiones del núcleo. Se puede ver un ejemplo de la función actuando sobre una entrada con un núcleo en la Figura 3.4.

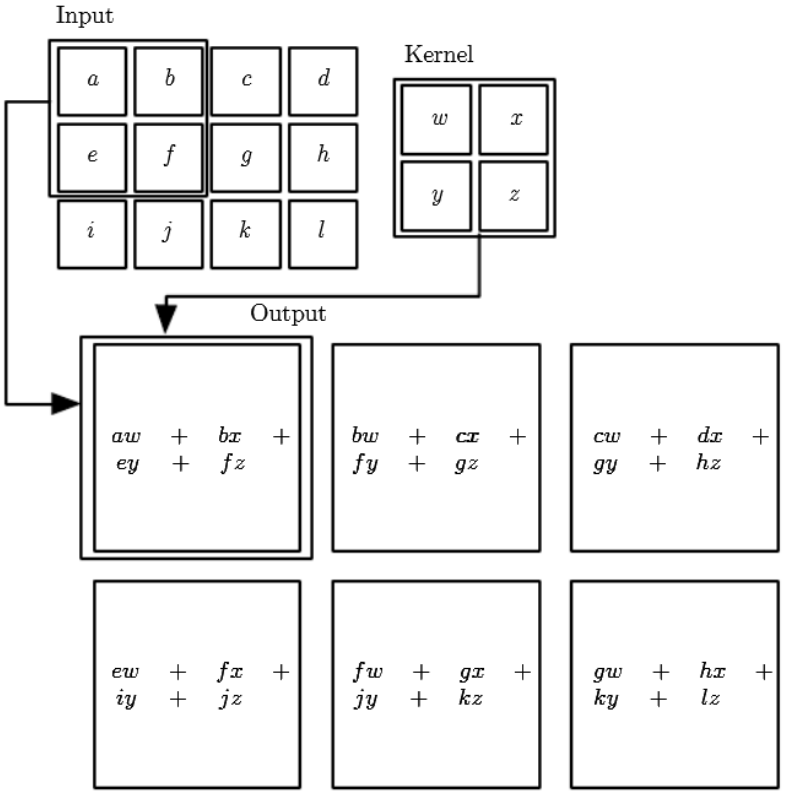

Figura 3.4: Ejemplo de operación de convolución [13].

Normalmente, se suelen utilizar **varios *kernels*** en vez de uno único, permitiendo detectar varias características a partir de una misma entrada [14]. Esto hace que el

mapa de características obtenido como resultado pase a ser una matriz tridimensional (donde cada capa representa los resultados de uno de los filtros).

El uso de convoluciones en las redes neuronales ofrece ventajas significativas respecto a las redes neuronales profundas tradicionales [13]:

- **Conectividad dispersa:** En las redes neuronales profundas, es normal que una neurona de una capa reciba entradas de todas las neuronas de la capa anterior, y propague su salida a todas las neuronas de la capa posterior. Esto puede provocar que el número de parámetros a ajustar crezca exponencialmente para entradas grandes y redes con muchas capas y neuronas [14].

  En cambio, el uso de convolución con *kernels* más pequeños que la entrada reduce drásticamente el número de conexiones, aumentando la eficiencia.

- **Parámetros compartidos:** En redes neuronales profundas estándares, cada parámetro (peso) interactúa únicamente en una propagación (el peso $w_{i,j}$ solo es relevante para la propagación entre las neuronas i y j).

  En cambio, cada parámetro del *kernel* interactúa con todos los valores de la entrada, reduciendo el número de parámetros que necesitan ser entrenados en la red, acelerando el entrenamiento.

- **Representación equivariante:** La convolución es **equivariante**, lo que significa que cualquier cambio en la entrada se refleja de forma idéntica en su representación en el mapa de características.

  Esto permite, por ejemplo, que si en una imagen de entrada se desplaza un objeto, su representación en la salida se verá desplazada de forma equivalente, permitiendo una mayor generalización a la hora de detectar características.

Por lo general, una capa de convolución en una red neuronal convolucional tiene la siguiente estructura [13]:

1. Una **convolución** con una entrada y varios núcleos, como se ha descrito previamente.
2. Una **función de activación** aplicada sobre el mapa de características obtenido de la convolución. Generalmente se utiliza *ReLU*.
3. Una función de ***pooling***. Una función de *pooling* es una función matemática similar a la convolución que sustituye una región de la red de tamaño (m, n) por un único estadístico de los valores de la región. Ésto se realiza principalmente para reducir el tamaño de la salida (permitiendo encadenar más convoluciones) y para mejorar la capacidad de invariancia de la convolución.

   Generalmente, el estadístico más utilizado para la función de *pooling* es el **máximo** [21], aunque también es común utilizar la **media**. Se puede ver un ejemplo de una operación de *pooling* por máximo en la Figura 3.5.

3.2. Aprendizaje por refuerzo

El **aprendizaje por refuerzo** es un subconjunto de métodos de aprendizaje automático consistentes en enseñar a un agente a actuar de forma óptima en cada situación para maximizar una recompensa numérica [23].

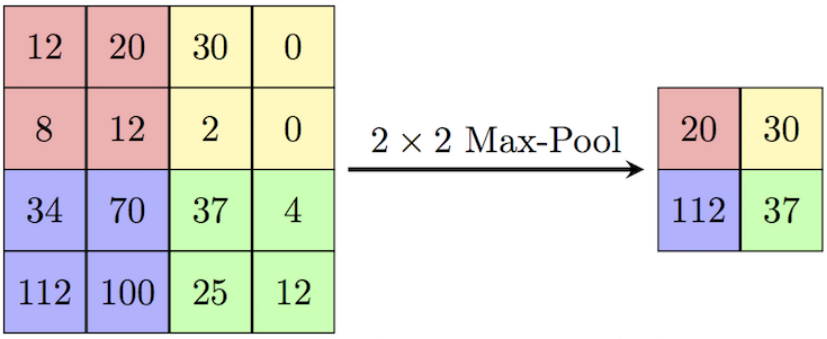

Figura 3.5: Ejemplo de *pooling* usando estadístico de máximo [22].

Existen dos características principales que distinguen al aprendizaje por refuerzo de otros métodos de aprendizaje automático (como el aprendizaje supervisado o el no supervisado) [23]:

- El agente no conoce de antemano las acciones óptimas, sino que debe descubrirlas mediante ensayo y error.
- Las acciones del agente no solo afectan a la recompensa inmediata, sino que pueden afectar a los estados y recompensas futuras.

Los dos componentes principales de un problema de aprendizaje por refuerzo son el **agente** (encargado de aprender y tomar decisiones) y el **entorno** (todo lo que no es el agente, con lo que interactúa) [23]. Además, en la relación entre estos dos componentes se pueden encontrar tres conceptos cuya definición adecuada es necesaria para la resolución correcta del problema [23]:

- **Estado (s):** Una representación del estado actual del entorno, tal cual lo interpreta el agente. Esta representación s_t pertenece a un conjunto de estados S.

  La definición del estado debe cumplir la **Propiedad de Markov:** el estado, por sí mismo, debe ser capaz de representar toda la información relevante [23]. Esto significa que el estado que se alcance tras realizar una acción debe depender únicamente del estado actual, y no de ningún estado o acción previa.

- **Acción (a):** Una actuación llevada a cabo por el agente según el estado actual s_t. Una acción a_t pertenece a un conjunto posible de acciones en un estado concreto, $A(s)$.

- **Recompensa (r):** Un valor numérico positivo (recompensa) o negativo (penalización) otorgada por el entorno tras la actuación del agente. El objetivo del agente a largo plazo es **maximizar la recompensa obtenida**.

  El principal objetivo de las recompensas es formalizar la meta del agente. Es decir, un agente que maximiza las recompensas debería ser un agente que alcanza la meta propuesta.

El agente y el entorno interactúan en un bucle continuo como se puede ver en la Figura 3.6, donde en cada paso t [23]:

1. El agente percibe un estado s_t del entorno.
2. El agente procesa el entorno y elige una acción a_t, que aplica sobre el entorno.

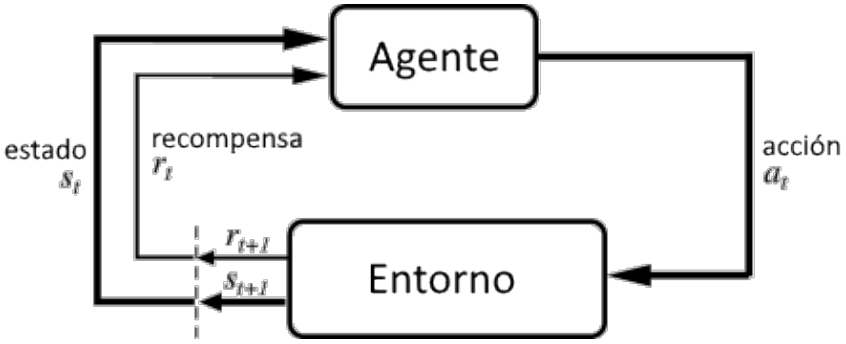

Figura 3.6: Interacción entre agente y entorno en el aprendizaje por refuerzo [23].

3. El entorno se modifica como respuesta a la acción a_t y devuelve al agente un nuevo estado s_{t+1}, junto a la recompensa obtenida r_{t+1}.

Además del agente y el entorno, hay otros cuatro conceptos de gran importancia en la definición de un problema de aprendizaje por refuerzo [23]:

- **Política** (π)**:** Método de actuación de un agente ante un estado s en un instante concreto, definiéndose como una función $\pi(s) = a$. La política es uno de los puntos claves del aprendizaje automático, siendo el objetivo a optimizar y pudiendo determinar la actuación del agente por sí misma.
- **Modelo:** Representación aproximada del entorno y su comportamiento usada por el agente para predecir estados y recompensas del entorno ante una acción. Este concepto es opcional, no siendo usado por todos los métodos de aprendizaje por refuerzo.
- **Modelo de recompensas (**$R(s)$ **/** $R(s, a)$**):** Función que, para un estado s (o un par estado-acción s, a) devuelve la recompensa adecuada al agente. Este modelo es parte del entorno y asocia a cada estado su deseabilidad inmediata (teniendo los estados deseables una recompensa mayor a los que no lo son).

  Este modelo no puede ser alterado por el agente, si bien la política del agente suele alterarse como respuesta al modelo de recompensas.
- **Modelo de utilidad (**$V(s)$ **/** $Q(s, a)$**):** Función que, para un estado s (o un par estado-acción s, a), devuelve la **utilidad** de ese estado. En aprendizaje por refuerzo se entiende la utilidad de un estado s como la recompensa total que el agente puede esperar conseguir si el estado inicial es s. A diferencia de las recompensas (que determinan la deseabilidad inmediata), la utilidad de un estado determina la deseabilidad a largo plazo.

En general, el objetivo principal de los algoritmos de aprendizaje por refuerzo es encontrar una política π^* (política óptima) que maximice la utilidad de todos los estados s del entorno, $V^*(s)$.

3.2.1. Métodos de aprendizaje por refuerzo clásicos

Los métodos de aprendizaje por refuerzo clásicos más usados suelen basarse en los principios del **aprendizaje por diferencia temporal** (*TD-Learning*) [23]. Los modelos de esta familia son capaces de aprender directamente de las experiencias sin necesidad de estimar un modelo. Además, pueden actualizar las estimaciones de la utilidad

de los estados tras cada paso realizado, pudiendo actualizar su política de forma incremental.

Se pueden distinguir dos aproximaciones a los modelos de aprendizaje por refuerzo de *TD-Learning* [23]:

- **Métodos *on-policy*:** Se optimiza la misma política que se usa para elegir la acción a realizar.
- **Métodos *off-policy*:** La política que se optimiza y la política que se usa para elegir las acciones a realizar no son la misma.

A continuación, se describen algunos de los principales métodos de aprendizaje por refuerzo usados.

3.2.1.1. Q-Learning

Q-Learning, propuesto originalmente por Christopher Watkins en 1989 [24] es un método de aprendizaje por refuerzo *off-policy*, siendo el método de aprendizaje por refuerzo más importante de la época [23].

En esencia, la actualización de la estimación de la utilidad de un par estado-acción $\hat{Q}(s_t, a_t)$ se calcula usando la siguiente fórmula [23]:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} \hat{Q}(s_{t+1}, a') - \hat{Q}(s_t, a_t)]$$

Donde α representa el peso del nuevo valor de $\hat{Q}$, y γ representa la depreciación realizada al estado futuro.

Q-Learning intenta aproximar directamente la política que optimiza la utilidad, Q^* independientemente de la política que se esté utilizando gracias al operador de máximo [23], obteniendo la máxima utilidad del estado alcanzado. Esto convierte a *Q-Learning* en un método *off-policy* como se ha comentado previamente.

Se puede observar el pseudocódigo del algoritmo en la Figura 3.7. En la práctica, es típico almacenar los valores $\hat{Q}(s, a)$ en una tabla que se va actualizando durante la ejecución del algoritmo.

Un problema que presenta el algoritmo de *Q-Learning* es que, por defecto, es un algoritmo voraz (es decir, el agente siempre elegirá la acción que maximice la utilidad). Esto puede provocar que el agente converja a un óptimo local, siendo incapaz de explorar estados nuevos que a la larga podrían ofrecer mejores utilidades.

Para solventar este problema se utiliza una política $\epsilon - greedy$ [23] (también conocida como exploración-explotación), introduciendo un parámetro adicional ϵ. Antes de elegir la acción a realizar, el agente calcula un número aleatorio en el rango $[0.0, 1.0]$. Si este número es mayor o igual que ϵ se elige la acción que maximiza la utilidad (explotación). En otro caso, se elige una acción aleatoria (exploración).

De esta forma se puede obtener un equilibrio entre la exploración de nuevos estados y la explotación de la política actual.

3.2.1.2. SARSA

SARSA (*State Action Reward State Action*), originalmente propuesto como *Modified Q-Learning* por Rummery y Niranjan en 1994 [25] es un algoritmo de aprendizaje

Algoritmo 2: Algoritmo de Q-Learning

1. Inicializar $\hat{Q}$ con valores aleatorios.
2. Mientras no se cumpla la condición de parada:
  2.1. Obtener estado inicial s
  2.2. Repetir:
    2.2.1. Seleccionar la acción a según la política π obtenida a partir de $\hat{Q}$.
    2.2.2. Ejecutar la acción a en el estado s.
    2.2.3. Obtener del entorno el nuevo estado s' y la recompensa r.
    2.2.4. Actualizar $\hat{Q}(s, a)$:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} \hat{Q}(s_{t+1}, a') - \hat{Q}(s_t, a_t)]$$

    2.2.5. $s \leftarrow s'$
  hasta que s **sea un estado final.**
3. Generar la política óptima π^* a partir de $\hat{Q}$.

Figura 3.7: Pseudocódigo del algoritmo de Q-Learning.

por refuerzo *on-policy* similar a *Q-Learning*. La actualización de la estimación de la utilidad de un par estado-acción $Q(s_t, a_t)$ se calcula usando la siguiente fórmula [23]:

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha[r_{t+1} + \gamma \hat{Q}(s_{t+1}, a_{t+1}) - \hat{Q}(s_t, a_t)]$$

Como se puede ver, la principal diferencia con *Q-Learning* es el cálculo de la utilidad del próximo estado s_{t+1}, utilizando la acción realizada en el próximo estado a_{t+1} en vez de la acción que maximiza la utilidad. Esto convierte a *SARSA* en un método *on-policy* [23].

El pseudocódigo del algoritmo es muy similar al de *Q-Learning* visto en la Figura 3.7, siendo la principal diferencia la actualización de la estimación de la utilidad en el punto **2.2.4.**, pasando a usar la fórmula descrita previamente. *SARSA* también puede utilizar la política $\epsilon - greedy$ para alcanzar un equilibrio de exploración-explotación.

Si se compara el rendimiento de *Q-Learning* y *SARSA*, *Q-Learning* tiende a ser un algoritmo más agresivo y arriesgado a la hora de optimizar su política, tomando riesgos para maximizar la utilidad; mientras que *SARSA* suele ser más conservador, buscando políticas buenas pero no necesariamente óptimas [23]. Por tanto, *SARSA* es más apropiado para aplicaciones en las que los errores sean muy costosos.

3.2.1.3. Métodos de actor-crítico

Los métodos de **actor-crítico** (siendo Witten uno de los primeros proponentes en 1977 [26]) son métodos que separan explícitamente la política (el **actor** que elige las acciones) de la estimación de la función de utilidad (el **crítico** que evalúa las acciones del actor) [23].

En general, el crítico sigue una función de utilidad-estado (estimando la utilidad de un estado), donde la evaluación del crítico sigue la siguiente fórmula [23]:

$$\sigma_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

Donde V es la función de utilidad usada por el crítico. Si esta evaluación es positiva (la utilidad ha mejorado), la tendencia del actor de elegir la acción a_t en el estado s_t debe reforzarse. En caso contrario (la utilidad ha empeorado), debe reducirse el uso de la acción [23].

Los métodos de actor-crítico presentan dos ventajas considerables frente a otros métodos [23]:

- El coste computacional de elegir una acción se reduce. Frente a otros métodos que deben explorar el espacio de acciones entero para elegir una acción adecuada, el actor puede usar simplemente la política almacenada.
- Se pueden aprender políticas explícitamente estocásticas (es decir, los actores pueden aprender las probabilidades óptimas de elegir varias acciones).

Ahora bien, los métodos de actor-crítico no fueron muy utilizados tradicionalmente [23], usandose más los métodos que estiman la política a partir de una función de utilidad como *Q-Learning* o *SARSA*.

3.2.2. Métodos de aprendizaje por refuerzo profundos

Los algoritmos descritos previamente tienen una serie de limitaciones, que se traducen en un conjunto de requisitos que deben cumplirse para garantizar que acaben convergiendo [27]:

- Los pares de estado-acción deben poder representarse de forma discreta.
- Todas las acciones tienen que realizarse en todos los estados varias veces (tiene que haber suficiente exploración como para que no sea necesario un modelo del entorno)

Esto significa que los métodos anteriores no son aplicables a problemas donde el conjunto de estados o de acciones sea continuo, o a problemas con una alta dimensionalidad [27]. En estas situaciones, es necesario utilizar una función de utilidad parametrizada $Q(s, a, \theta)$, donde θ son los parámetros de un modelo usado para **aproximar** los valores de Q [27].

Algunas propuestas iniciales para resolver estos problems fueron los métodos de *Fitted Q-Learning* (método propuesto por Geoffrey Gordon en 1995 [28] donde se almacenan las experiencias realizadas por el agente) o de *Neural Fitted Q-Learning* (método propuesto por Martin Riedmiller en 2005 [29] que adapta *Fitted Q-Learning* para usar una red neuronal como modelo estimador). Ahora bien, estos métodos presentan problemas de inestabilidad que limitan su funcionamiento a casos específicos [27].

Finalmente, la propuesta que más éxito tuvo fueron los métodos de **Deep Reinforcement Learning**, una combinación de las técnicas de *Deep Learning* y aprendizaje por refuerzo [27]. Estos métodos se popularizaron en 2015 con la propuesta de *Deep Q-Learning* por parte de DeepMind [30], consiguiendo diseñar agentes genéricos con rendimiento sobrehumano en un gran número de tareas.

Actualmente, existen principalmente dos grandes familias de métodos de *Deep Reinforcement Learning*:

- Métodos basados en **Deep Q-Learning** y sus variantes propuestas.

- Métodos basados en los principios de **actor-crítico**.

Estas familias son descritas a continuación.

3.2.2.1. Familia de métodos de *Deep Q-Learning*

El algoritmo de **Deep Q-Learning** fue propuesto en 2015 por DeepMind [30], siendo una adaptación del algoritmo de *Q-Learning* aplicando técnicas de *Deep Learning*, en el que se usa una red neuronal profunda como estimación de la función de utilidad $Q(s, a)$, siendo el objetivo del método entrenar los pesos de la red para maximizar la utilidad.

Además de la red neuronal profunda, Deep Q-Learning utiliza una serie de restricciones para limitar las posibles inestabilidades que presentaban las propuestas previas [27]:

- **Uso de red neuronal objetivo:** *Deep Q-Learning* utiliza dos redes neuronales, una **red Q** (que se usa para generar la política y que se actualiza tras cada paso) y una **red objetivo** (que se utiliza para calcular la máxima utilidad del estado alcanzado). La red objetivo se actualiza con menor frecuencia, siendo típico actualizarla cada N pasos o al final de cada episodio.

  Usar dos redes separadas evita que las inestabilidades del entrenamiento se propaguen rápidamente (no varía la utilidad objetivo tras cada paso), y reduce el riesgo de divergencia.

- ***Experience Replay*:** En vez de aprender inmediatamente de las actuaciones, el agente almacena la información tras cada paso (estado, acción, recompensa y estado alcanzado $< s, a, r, s' >$, conocido como **experiencia**) en una memoria, el *Replay Memory*. Tras cada paso, el agente obtiene una muestra aleatoria del *Replay Memory* y entrena su red Q utilizando las experiencias.

  De esta forma, el agente es capaz de cubrir un rango más amplio del espacio de estados-acciones durante su entrenamiento. Además, reduce la inestabilidad del entrenamiento evitando que el agente sobreajuste debido a correlaciones temporales (al muestrear experiencias antiguas y nuevas).

- **Beneficios de *Deep Learning* [30]:** Se pueden aplicar técnicas de *Deep Learning* para el entrenamiento como el uso de redes convolucionales para trabajar con imágenes como entrada o la paralelización del proceso para acelerar el entrenamiento.

El pseudocódigo del algoritmo se puede observar en la Figura 3.8.

Deep Q-Learning tuvo una gran popularidad por su buen funcionamiento al aplicarse a tareas complejas como videojuegos, obteniendo resultados superiores a los del jugador humano promedio en muchos de los juegos probados [30]. Ahora bien, también plantea una serie de carencias que limitan su rendimiento. Para solucionar ésto, se han propuesto una serie de ampliaciones y mejoras al algoritmo original, con el fin de suplir estas limitaciones [31], siendo algunas de las más importantes:

- ***Double Q-Learning*:** Propuesto originalmente por Hado Van Hasselt en 2010 [32] y aplicado a *Deep Q-Learning* en 2016 [33], consiste en entrenar por separado dos funciones de utilidad. Una de ellas es usada para elegir la acción a

Algoritmo 3: Algoritmo de Deep Q-Learning

1. Inicializar el *Replay Memory* D con un tamaño máximo N.
2. Inicializar la función aproximadora Q (la red neuronal profunda) con pesos aleatorios.
3. Para cada *episodio* desde 1 hasta M:
  3.1. Obtener estado inicial $s_1 = \{x_1\}$ (donde x_1 es la imagen del estado sin procesar) y preprocesarlo $\phi_1 = \phi(s_1)$
  3.1. Para cada momento t desde 1 hasta el final del *epoch*:
    3.1.1. Elegir acción:
      Con probabilidad ϵ, elegir acción aleatoria a_t.
      Si no, selecciona acción $a_t = max_a Q^*(\phi(s_t), a; \theta)$
    3.1.2. Ejecutar la acción a_t, obtener recompensa r_t e imagen del nuevo estado x_{t+1}.
    3.1.3. Fijar estado actual $s_{t+1} = s_t, a_t, x_{t+1}$ y preprocesar $\phi_{t+1} = \phi(s_{t+1})$.
    3.1.4. Almacenar experiencia $(\phi_t, a_t, r_t, \phi_{t+1})$ en el *Replay Memory* D.
    3.1.5. Tomar muestra aleatoria de experiencias $(\phi_j, a_j, r_j, \phi_{j+1})$ del *Replay Memory* D.
    3.1.6. Fijar el valor de y_j $\begin{cases} r_j & \text{si } \phi_{j+1} \text{ es terminal.} \\ r_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{en otro caso.} \end{cases}$
    3.1.7. Realizar gradiente descendiente usando $(y_j - Q(\phi_j, a_j; \theta))^2$ como error.

Figura 3.8: Pseudocódigo del algoritmo de *Deep Q-Learning*.

realizar, mientras que la otra sirve para estimar la utilidad de realizar esa acción en el estado actual.

De esta forma se reduce el sesgo que introduce usar la función de máximo durante la estimación del valor, mejorando el rendimiento.

- ***Prioritized Experience Replay*:** Propuesto por Tom Schaul *et al.* en 2015 [34], consiste en modificar la toma de muestras del *Replay Memory*, pasando de una distribución de probabilidades uniformes a una distribución donde las experiencias de mayor error y más nuevas (las experiencias que pueden resultar más útiles al agente) tienen mayor probabilidad de ser elegidas.
- ***Dueling Networks*:** Propuesto por Ziyu Want *et al.* en 2015 [35], consiste en dividir el funcionamiento de la red en dos flujos paralelos: un flujo encargado de estimar la utilidad del estado-acción, y otro flujo encargado de estimar la **ventaja** (utilidad adicional de usar una acción concreta en el estado actual). Ambos flujos son posteriormente unidos para obtener una salida única.
- ***Multi-step Learning*:** Mencionado originalmente por Richard Sutton en 1988 [36], consiste en considerar una secuencia de acciones en vez de una única acción a la hora de estimar utilidades y actualizar la función de estimación Q.
- ***Distributional RL*:** Propuesto por Marc Bellemare *et al.* en 2017 [37], consiste en cambiar el modelado y aprendizaje de una función que estima la acción a elegir por una función que estima la distribución de probabilidades de las acciones posibles para el estado.
- ***Noisy Nets*:** Propuesto por Meire Fortunato *et al.* en 2017 [38], consiste en in-

troducir una capa de ruido a la red neuronal profunda para aumentar la exploración realizada por el agente.

Estas mejoras conducen al algoritmo *Rainbow*, propuesto por DeepMind en 2017 [31], que implementa todas las mejoras discutidas previamente para obtener un rendimiento notablemente superior, siendo actualmente el estado del arte de esta familia de métodos.

Ahora bien, esta familia de métodos también presenta una serie de desventajas, principalmente un tiempo de entrenamiento muy largo para obtener buenos resultados [14] o problemas de rendimiento al trabajar con espacios de estados y acciones muy grandes o continuos [27].

3.2.2.2. Familia de métodos de actor-crítico

Los métodos de **actor-crítico**, como ya se vio anteriormente, están compuestos por dos elementos separados explícitamente: un **actor** (la política que elige las acciones) y un **crítico** (la función de utilidad que evalúa la actuación del actor) [23]. En general, la arquitectura de actor-crítico se utiliza junto a métodos de **gradiente de políticas** [39], consistentes en optimizar la política para maximizar la utilidad esperada de usar dicha política (generando trayectorias que maximizan las recompensas, y evitando trayectorias con penalizaciones).

Existen varias formas de plantear los métodos de gradiente de políticas, siendo una de ellas los métodos de **actor-crítico con ventaja** (*Advantage Actor-Critic*) [40]. Esta familia de métodos utiliza una función de utilidad $Q(s, a)$ para optimizar las políticas, de forma similar a Q-Learning. Ahora bien, para evitar la variabilidad inherente a estas funciones, se utiliza un *baseline* a la hora de usar los gradientes conocido como **ventaja**, siguiendo la fórmula:

$$A(s, a) = Q(s, a) - V(s)$$

Donde $A(s, a)$ es la ventaja de aplicar, $Q(s, a)$ la utilidad de un par estado-acción y $V(s)$ la valoración del crítico a un estado.

Los principales métodos de esta aproximación son los siguientes:

- ***Asynchronous Advantage Actor-Critic (A3C)*:** Propuesto por OpenAI en 2016 [40], el método consiste en un **crítico** global y varios **actores** actuando en paralelo de forma asíncrona.

  Cada actor completa episodios, almacenando las acciones realizadas durante el episodio. Cuando el episodio finaliza, el actor calcula los gradientes de su propia política y del crítico, actualizándolo de forma asíncrona. Estos agentes no tienen ninguna interacción entre sí durante su ejecución.

  El crítico es entrenado utilizando la **ventaja**, usando esta métrica para evaluar el rendimiento de los agentes.

- ***Advantage Actor-Critic (A2C)*:** Este método es una variante de *A3C* [40] cuya única diferencia es que los agentes pasan de ser asíncronos a estar sincronizados (los agentes y las actualizaciones se realizan una tras otra en orden). Pese a parecer peor, el método resulta más eficiente al poder aprovechar mejor la paralelización disponible.

Otro planteamiento son los métodos de **gradiente natural** [39]. Los métodos tradicionales de gradientes usados durante el entrenamiento de redes neuronales actualizan los pesos de la red en dirección opuesta al gradiente del error para alejarse de éste, buscando la modificación más pequeña de los parámetros que consiga la disminución más grande del error. Estos cambios grandes del error provocan que las estimaciones de la utilidad no sean certeras (estimando políticas desfasadas), introduciendo sesgos durante el entrenamiento [41].

En cambio, estos métodos buscan la mayor modificación de los parámetros que conlleve el menor cambio en la política. Un gran cambio en los parámetros conlleva un aprendizaje interno para el agente, mientras que las modificaciones pequeñas de la política permiten la reutilización de las experiencias previas del agente [41].

Los principales métodos de esta aproximación son los siguientes:

- ***Trust Region Policy Optimization (TRPO)*:** Propuesto por John Schulman *et al.* en 2015 [42], el método consiste en el uso de gradientes naturales para mejorar la utilidad esperada de la política de forma monótona. Para esto, se utiliza una función objetivo surrogada (una cota inferior de la función de utilidad esperada), que cambia los parámetros de la red neuronal de forma iterativa usando episodios largos con cambios pequeños en la política.

  Este método presenta varios problemas [41] como problemas de rendimiento al trabajar con redes neuronales convolucionales o con varias salidas, o una implementación compleja. Por esto, se ha dejado de usar el método a favor de otros como *PPO*.

- ***Proximal Policy Optimization (PPO)*:** Propuesto por John Schulman *et al.* en 2017 [43], el método fue propuesto para simplificar y solventar las limitaciones de *TRPO*. En esencia, la función objetivo surrogada se simplifica para ser compatible con procesos de gradiente descendiente estándares (simplificando notablemente la implementación). Además, el método se ejecuta por varios agentes de forma síncrona, de forma similar a *A2C*.

  El pseudocódigo del algoritmo se puede ver en la Figura 3.9 [41].

Actualmente, *PPO* es considerado el estado del arte para los algoritmos de actor-crítico y, en general, para los problemas de aprendizaje por refuerzo, obteniendo resultados mejores que el resto de métodos descritos en ambas secciones y siendo fácil de implementar y usar en la práctica [41].

3.3. Algoritmos de navegación autónoma

En el campo de la robótica, se entiende el problema de la **navegación** como el conseguir que un robot (una máquina que percibe, planifica y actúa) se desplace hasta una meta [44]. Este problema, a su vez, se puede descomponer en tres tareas entrelazadas que el robot debe resolver [45]:

- **Mapeado:** ¿Qué aspecto tiene el entorno alrededor del robot?

  Esta tarea incluye la interpretación de las entradas (sensores) del robot y la representación del entorno en el que se encuentra el robot.

- **Localización:** ¿Dónde está el robot?

Algoritmo 4: Algoritmo de Proximal Policy Optimization

1. Inicializa un **actor** π_{theta} y un crítico V con pesos aleatorios.
2. Mientras no se alcance la condición de parada:
  2.1. Para cada actor de N actores, en paralelo:
    2.1.1. Recoge la información de T transiciones usando la política vieja π_{old}.
    2.1.2. Calcula la ventaja general $A_{pi_{old}}(s, a)$ de cada transición usando al crítico.
  2.2. Repite durante K épocas:
    2.2.1. Muestrea M transiciones de las recogidas por los actores.
    2.2.2. Entrena al actor para maximizar la función de objetivo surrogada acotada:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t[\text{mín}(\rho_t(\theta)\, A_{\pi_{\theta_{\text{old}}}}(s_t, a_t), \text{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon)\, A_{\pi_{\theta_{\text{old}}}}(s_t, a_t)]$$

    2.2.3. Entrena al crítico para minimzar el error cuadrático medio.
  2.3. Actualiza los parámetros de la red neuronal.

Figura 3.9: Pseudocódigo del algoritmo de *Proximal Policy Optimization*.

Existen varias formas de plantear esta tarea, incluyendo *localización global* (en la que el robot no tiene conocimiento a priori sobre su posición) o *pose planning* (en la que el robot conoce su posición inicial)

- **Planificación de ruta:** ¿Cómo puede llegar el robot a la meta?

  Esta tarea busca encontrar una ruta eficiente para que el robot pueda desplazarse desde su posición inicial hasta la meta, y es la tarea en la que se centra esta revisión.

Se pueden hablar de dos grandes aproximaciones para resolver este problema [44]:

- **Navegación basada en mapas (planificación global):** El robot posee (o construye) un mapa de su entorno, que utiliza junto a estimaciones de su posición en el entorno para planificar una ruta óptima (ya sea en distancia o coste) entre su posición inicial y la meta a alcanzar.
- **Navegación reactiva (planificación local):** El robot no realiza tareas de mapeado ni tiene información sobre su localización, sino que únicamente reacciona a la información que recibe por sus sensores para planificar una ruta hacia la meta, actualizando esta ruta constantemente a partir de la información nueva para evadir los posibles obstáculos.

  Esta familia de algoritmos es más simple que la navegación basada en mapas, y es en la que se centra esta discusión.

3.3.1. Algoritmos de navegación reactiva clásicos

Uno de los primeros algoritmos de navegación reactiva usados es la familia de algoritmos ***BUG***, propuesta por Vladimir Lumelsky y Alexander Stepanov en 1987 [46]. Estos algoritmos trabajan con la suposición de que el robot es un punto en un espacio bidimensional con obstáculos desconocidos.

El robot cuenta con un sensor indicándole los contornos de los obstáculos en un radio cercano al agente, y conoce en todo momento la distancia y el ángulo hasta la meta. El objetivo del robot es desplazarse con éxito desde una posición inicial hasta dicha meta. Existen principalmente dos algoritmos *BUG* [47]:

- ***BUG1*:** El robot se mueve hacia la meta hasta encontrar un obstáculo. Cuando el robot se topa con un obstáculo recorre su contorno entero en una única dirección (generalmente en el sentido de las agujas del reloj) hasta volver a la posición inicial. Tras esto, vuelve a recorrer el contorno hasta llegar a la posición del contorno más cercana a la meta, siguiendo su recorrido hacia la meta desde ese punto.
- ***BUG2*:** El robot se mueve hacia la meta. Si hay obstáculos en su camino, el robot recorre el contorno hasta que deja de haber un obstáculo entre su posición y la meta.

Otra familia de algoritmos de navegación automática son los algoritmos de **banda elástica (*elastic band*)**, propuestos por Quinlan *et al.* en 1993 [48]. Estos algoritmos son híbridos entre algoritmos de planificación global y reactiva, planificando rutas globales (desde la posición inicial del robot hasta la meta) que se modifican de forma local ante los obstáculos no previstos que se encuentran.

Finalmente os algoritmos de **campos de potenciales** [49], sugeridos inicialmente por Andrews en 1983 [50], son una familia de algoritmos de navegación reactiva populares. Este algoritmo funciona mediante la creación de dos tipos de fuerzas artificiales en el entorno [49]:

- Una **fuerza atractiva** ejercida por la meta, atrayendo al robot hacia ella.
- Varias **fuerzas repulsivas** ejercidas por los obstáculos del entorno, apartando al robot de ellas.

La suma de todas estas fuerzas, R, indica al robot la dirección en la que debe desplazarse y la velocidad a la que debe hacerlo [49]. Inicialmente esta propuesta tuvo gran acogida debido a su simplicidad y elegancia, teniendo algunas implementaciones en robots reales como la de Arkin en 1989 [51].

Ahora bien, estas implementaciones iniciales presentaban velocidades de movimiento lentas, haciéndolos poco eficientes [49]. Además, el planteamiento original presenta algunos problemas como oscilaciones en el movimiento de los agentes, posibilidad de bucles o la dificultad del robot para navegar entre dos obstáculos [49].

3.3.2. Antecedentes al trabajo en navegación autónoma reactiva

Actualmente existe una amplia gama de robots con técnicas de navegación autónoma reactiva sin mapas, siendo frecuente el uso de técnicas de aprendizaje por refuerzo profundo para el entrenamiento del agente, permitiendo la resolución de problemas complejos de navegación y el uso de agentes aéreos (drones) con movimiento omnidireccional en tres dimensiones [1].

Ahora bien, hay dos propuestas concretas que sirven como antecedentes directos al trabajo realizado:

- La propuesta de Tai *et al.* en 2017 [52], consistente en el entrenamiento de un

robot terrestre con movimiento omnidireccional en dos dimensiones usando un algoritmo de aprendizaje por refuerzo profundo basado en actor-crítico.

El agente cuenta únicamente con la posición relativa de la meta y un conjunto de láseres como sensores de entrada, aprendiendo a partir de estos el movimiento continuo (la salida) que debe realizar el agente para desplazarse por el entorno y alcanzar la meta.

- La propuesta de Carlos Sampedro *et al.* en 2018 [1]. Esta propuesta es similar en concepto a la propuesta de Tai (un robot entrenado usando aprendizaje por refuerzo profundo a partir un conjunto de láseres como entrada, indicando la dirección en la que debe desplazarse el agente para alcanzar la meta).

  Las principales diferencias en este caso son la inclusión principios de **campos de potenciales** para guiar y mejorar el proceso de entrenamiento y para hacer la navegación más robusta frente a errores en la lectura de los sensores o casos límite; y el uso de un agente aéreo (navegando a una altura constante) frente a un agente terrestre.

Capítulo 4

Habitat: Simulador *Habitat Sim* y *Habitat Lab*

En este capítulo se describirá en profundidad el simulador utilizado, *Habitat Sim 2.0*, y su librería de Python *Habitat Lab*. Además, se presentarán los principales componentes usados por la librería, detallando su funcionamiento y su uso. Finalmente, se explicará la instalación del simulador y las dependencias necesarias.

4.1. *Habitat*

Habitat [2] es una plataforma para el desarrollo de inteligencia artificial con agentes físicos, diseñada con el fin de estandarizar el conjunto de herramientas necesarias para el entrenamiento en entornos hiperrealistas tridimensionales bajo una sola plataforma unificada e integrada.

Esta plataforma pretende resolver algunos de los problemas tradicionales que afectan a otros simuladores, impidiendo la generalización y comparación de resultados experimentales [2]:

- La dependencia entre componentes (como simuladores funcionando únicamente con conjuntos de datos o tareas especificas), dificultando el trabajo al necesitar varias herramientas.
- Los parámetros fijos (como las acciones disponibles, los sensores...), dificultando la comparativa de resultados.
- El rendimiento subóptimo de los simuladores (tanto en renderizado como en físicas), dificultando el entrenamiento de agentes a gran escala.

La plataforma *Habitat* cuenta con dos componentes principales [2]:

- ***Habitat Sim*:** Un simulador 3D modificable con agentes configurables, diversos sensores y manejo de conjuntos de datos 3D genéricos (con soporte de fábrica para conjuntos como *Matterport3D* o *Gibson*).
- ***Habitat Lab*:** Una librería modular de alto nivel desarrollada para Python, con el fin de facilitar el desarrollo de agentes físicos en *Habitat Sim*, ofreciendo herramientas para la definición, configuración, entrenamiento y evaluación de éstos.

4.1.1. Simulador *Habitat Sim*

Habitat Sim [2] es un simulador 3D ampliable, diseñado para el entrenamiento de agentes físicos en entornos hiperrealistas. Este simulador se encarga de representar escenarios tridimensionales en formatos estandarizados y de simular agentes físicos, tanto sensores como movimiento.

Este simulador fue diseñado con el propósito de solventar los problemas descritos previamente, siendo algunos de sus objetivos principales:

- **Soporte genérico a conjuntos de datos:** *Habitat Sim* es capaz de reconstruir y simular conjuntos de datos genéricos independientemente de su origen, usando un formato uniforme y estandarizado como son los *scene graphs* [53] (representaciones estructuradas de los escenarios). Esta estandarización permite al simulador trabajar de forma consistente con cualquier conjunto de datos.
- **Modularidad:** El simulador está diseñado ofreciendo *APIs* de todos sus componentes, permitiendo la ampliación del simulador con nuevos sensores, escenarios, tareas...
- **Rendimiento:** *Habitat Sim* está implementado en C++ usando la librería *Magnum Graphics* como *middleware* para el procesamiento de la imagen. Además, la tubería de creación y renderizado de imágenes está optimizada para evitar la repetición de procesos, siendo capaz de generar todas las imágenes de cada instante en una única pasada.

  Todo ésto permite al simulador generar miles de *frames* por segundo, siendo este valor al menos un orden de magnitud superior al que ofrecen otros simuladores como **Gibson** [54] o **MINOS** [55]. Esta velocidad mueve el cuello de botella de la simulación al entrenamiento del agente, permitiendo entrenamientos más profundos en menos tiempo.

Habitat 2.0 [3] es una versión posterior de *Habitat Sim* lanzada en Junio de 2021, diseñada con el fin de ser capaz de simular entornos interactivos con físicas complejas (frente a las simulaciones de físicas simples de *Habitat Sim*) y centrada en permitir la creación y evaluación de agentes físicos asistentes. Las principales características ofrecidas son:

- **Simulador *Habitat 2.0*:** Una segunda versión del simulador *Habitat Lab* con capacidad para simular movimientos y físicas de objetos rígidos (como puertas, cajones...), robots articulados, cinemáticas, dinámicas...

  El rendimiento del simulador es superior al de *Habitat Sim*, especialmente durante la simulación de físicas. Además, el rendimiento escala notablemente, pudiendo trabajar de forma distribuida.
- **Conjunto de datos *ReplicaCAD*:** *ReplicaCAD* es un conjunto de datos creado a partir del conjunto *Replica* [56]. Este conjunto de datos está diseñado para aprovechar las capacidades del nuevo simulador y pensado para evaluación de tareas de reorganización de elementos, implementando objetos articulados e interactivos.
- ***Home Assistant Benchmark (HAB)*:** Un *benchmark* implementando un conjunto de tareas típicas para robots asistentes (como limpiar un cuarto o preparar

una mesa), diseñado para evaluar el rendimiento de estos agentes de forma estandarizada.

4.1.2. Librería *Habitat Lab*

Habitat Lab (originalmente conocido como *Habitat-API*) [2] es una librería modular de alto nivel desarrollada para *Python*, con el fin de facilitar el uso de *Habitat Sim* y el desarrollo de agentes físicos. El principal objetivo de la librería es permitir a los usuarios:

- **Definir tareas para agentes físicos**: Se ofrecen tareas típicas (como navegación, seguimiento de orden, búsqueda de objetos...) y *APIs* para desarrollar tareas propias.
- **Configurar agentes físicos:** Se permite modificar al agente físico ajustando su forma física, los sensores disponibles, las acciones posibles...
- **Entrenar agentes físicos:** Se ofrece soporte para técnicas clásicas (como *SLAM*) además de entrenamiento por refuerzo y entrenamiento por imitación.
- **Evaluar agentes físicos:** Se ofrecen métricas estándares [5] usadas para evaluar el rendimiento de los agentes.

Además, la librería opcionalmente ofrece *Habitat Baselines*, un conjunto de ejemplos y ampliaciones a *Habitat Lab*. Entre estas ampliaciones se incluyen diversas utilidades para facilitar el desarrollo, agentes prediseñados y ejemplos para entender el funcionamiento del simulador y la librería.

4.2. Conceptos principales de *Habitat*

La arquitectura de *Habitat* (y, especialmente, la de la librería *Habitat Lab*) gira principalmente en torno a tres conceptos principales, como se puede ver en la Figura 4.1:

1. **Tarea (*Task*):** El problema concreto a resolver por el agente, gestiona los objetivos y el éxito de la tarea usando métricas del simulador.
2. **Episodio (*Episode*):** Especificación del episodio a realizar, conteniendo información como la posición y ángulo iniciales del agente, posición de la meta...
3. **Entorno (*Environment*):** Concepto principal del simulador, abstrae toda la información necesaria para permitir el trabajo con el simulador y el agente físico.

Ahora bien, la documentación oficial del simulador es pobre y ofrece poca información sobre los conceptos y, especialmente, sobre su uso. Además, hay otros conceptos (como entrenadores, ficheros de configuración...) de igual relevancia e importancia para el uso del simulador que no reciben ninguna explicación.

Por tanto, en esta sección se busca explicar en detalle los conceptos principales necesarios para trabajar con *Habitat*, detallando sus características y su uso.

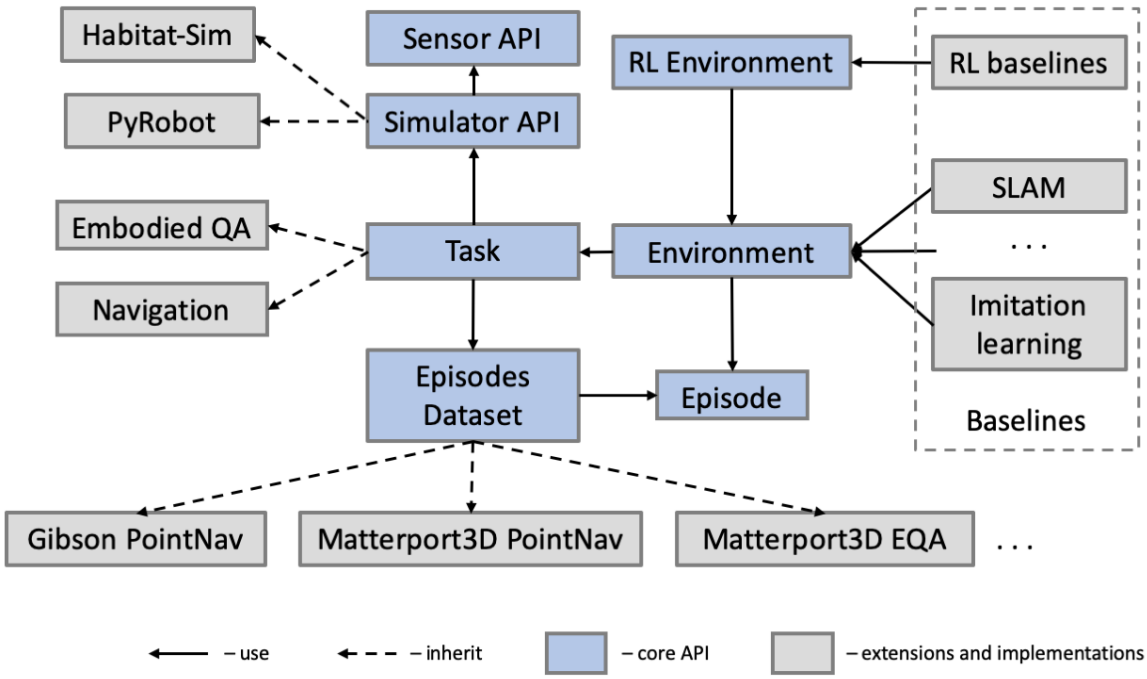

Figura 4.1: Arquitectura de *Habitat Lab* [2].

4.2.1. Entornos

Un **entorno (*environment*)** es el concepto principal de *Habitat Lab*. El entorno se encarga de abstraer las principales tareas necesarias para el trabajo con agentes físicos, siendo las principales tareas que realiza:

- **Carga y manejo del conjunto de datos:** El entorno se encarga de la lectura y carga de los ficheros, y de la conversión de los escenarios al formato de *scene graph*.

- **Control de las métricas y los objetivos de la tarea:** El entorno se encarga de la interacción con el simulador para el control de los objetivos y las métricas de la tarea. Además, se encarga de detener los episodios cuando se cumplen los objetivos.

- **Generación y manejo de los episodios:** El entorno genera la lista de episodios a partir del conjunto de datos y de la tarea a realizar. Además, se encarga de ordenar los episodios de forma óptima (juntando los episodios que se realizan en un mismo escenario) para agilizar el proceso.

- **Control del agente físico:** El entorno interactúa con el simulador en ambos sentidos, leyendo los sensores del agente físico y devolviendole la acción realizada para simular los resultados.

Por defecto, *Habitat Lab* ofrece dos tipos de entornos, a partir de los cuales se puede derivar cualquier entorno necesario.

4.2.1.1. *Env*

Env es el simulador básico ofrecido por *Habitat*, usado principalmente para agentes sin aprendizaje y para la evaluación de agentes entrenados, implementando todas las características descritas previamente. El entorno ofrece dos métodos básicos, usados para controlar la simulación:

- **reset():** Este método se encarga de preparar el entorno para el comienzo de un episodio, cargando el escenario y configurándolo para el episodio actual. Además, devuelve las primeras observaciones que recibe el agente con el sensor, para poder actuar a partir de ellas.
- **step(acción):** Dada una acción como argumento, el entorno aplica la acción al agente físico y devuelve las nuevas percepciones del agente a través de sus sensores.

Cualquier entorno personalizado que se quiera crear debe ser una subclase de *Env* (al ofrecer toda la funcionalidad básica).

4.2.1.2. *RLEnv*

RLEnv es la principal subclase de *Env* disponible, ofreciendo capacidades adicionales al entorno para permitir el entrenamiento usando aprendizaje por refuerzo. Además de los métodos anteriores, incluye varios más que deben ser implementados para su funcionamiento:

- **get_reward(observaciones):** A partir de las observaciones del agente, calcula una recompensa o penalización para éste. En general, el procesamiento de recompensas se implementa directamente en el entorno a través de este método.
- **get_reward_range():** Este método únicamente devuelve los límites de la recompensa (el rango $[min, max]$ en el que se encuentra).
- **get_done(observaciones):** A partir de las observaciones del agente, calcula si el episodio ha acabado (ya sea de forma satisfactoria o fallida). En este método se implementan comprobaciones como colisiones u otras condiciones de parada.
- **get_info(observaciones):** Devuelve información respecto al entorno a partir de la última observación. Es típico devolver información de sensores o métricas con este método.

Por defecto, *RLEnv* no implementa los métodos descritos previamente, por lo que es necesario realizar un entorno personalizado para el entrenamiento. Ahora bien, *Habitat Baselines* ofrece *NavRLEnv*, una implementación básica de entorno pensado para problemas de navegación a objetivo que sirve como punto de partida para otras implementaciones más complejas.

En este trabajo, se ha implementado un entorno personalizado (*ReactiveNavigationEnv*) como subclase de *NavRLEnv*, con definiciones propias de los métodos previos para usar el sistema de recompensas basado en potenciales propuesto (descrito en más detalle en el Capítulo 5).

4.2.2. Tareas

Una **tarea (*embodied task*)** es un contenedor de la información necesaria para que el simulador pueda definir y trabajar con una tarea. Concretamente, contiene información sobre:

- El **espacio de acciones** disponible.
- Los **sensores** disponibles para el agente.
- Las **métricas** a utilizar para valorar al agente.
- El **simulador** sobre el que se va a trabajar.

La clase *EmbodiedTask* implementa los métodos básicos para la definición de tareas, y todas las tareas diseñadas deben heredar de esta. Ahora bien, *Habitat Lab* ofrece varias tareas típicas por defecto:

- ***NavigationTask* (navegación a meta / objetivo):** El objetivo del agente es alcanzar una posición, dada sus coordenadas. La mayoría de tareas que incluyen navegación se definen como subclases de esta tarea. Es la tarea más típica, y la que se busca resolver en este trabajo.
- ***ObjectNavigationTask* (navegación a objeto):** Una variante de *NavigationTask*, en la que el agente debe desplazarse hasta un objeto concreto localizado en el entorno.
- ***VLNTask / Vision and Language Navigation Task* (navegación con visión y lenguaje):** Una variante de la *NavigationTask*, en la que el agente debe alcanzar una meta siguiendo instrucciones expresadas en lenguaje natural.
- ***EQATask / Embodied Question Answering Task* (respuesta física a preguntas) [57]:** El agente recibe una pregunta en lenguaje natural (como, por ejemplo, *"¿De qué color es el coche aparcado el garaje?"*) y debe explorar el entorno para ser capaz de responder.
- ***RearrangeTask* (re-organización):** Tareas introducidas con *Habitat 2.0*, *RearrangeTask* es una tarea genérica introducida para ofrecer soporte a tareas que necesitan agentes con articulaciones móviles. Actualmente existen dos tareas de reorganización, ***RearrangePickTask*** (agarrar un objeto determinado del entorno) y ***RearrangeReachTask*** (alcanzar un objeto determinado del entorno).

Estas tareas se instancian a partir del fichero de configuración, como se verá posteriormente. Es importante destacar que no todos los sensores y métricas son compatibles con todas las tareas. Algunos de ellos han sido diseñados para tareas específicas, y no pueden ser usados fuera de éstas.

4.2.3. Episodios

Un **episodio (*episode*)** es una especificación de una instancia específica de un agente en un escenario realizando una tarea, incluyendo la información relativa al estado inicial y a la meta a cumplir. Por defecto, un episodio (*Episode)* contiene información sobre:

- **Posición** y **rotación** inicial del agente.

- **Escenario** en el que se realiza el episodio.

Al igual que con las tareas, la clase *Episode* es una clase genérica que implementa únicamente los métodos básicos y de la que se deben derivar las definiciones de episodios, existiendo subclases de episodios para cada tarea específica:

- ***NavigationEpisode* (episodio de navegación):** Incluye información adicional sobre las coordenadas de la meta y la ruta más corta desde la posición inicial hasta la meta.
- ***ObjectGoalNavigationEpisode* (episodio de navegación a objeto):** Incluye información adicional sobre las coordenadas del objeto a encontrar y la ruta más corta desde la posición inicial hasta el objeto.
- ***VLNEpisode* (episodio de navegación con visión y lenguaje):** Incluye información adicional sobre las coordenadas de la meta y las instrucciones en lenguaje natural proporcionadas al agente.
- ***EQAEpisode* (episodio de respuesta física a preguntas):** Incluye información adicional sobre la pregunta y la respuesta esperada.
- ***RearrangeEpisode* (episodio de reorganización):** Incluye información adicional sobre las partes articuladas del escenario y los objetos a alcanzar o agarrar.

Los episodios son creados y gestionados automáticamente por los conjuntos de datos, como se verá a continuación.

4.2.4. Conjuntos de datos

Un **conjunto de datos *(dataset)*** es un contenedor y gestor de episodios a realizar, encargándose de:

- **Cargar los escenarios desde memoria**, convirtiendo los *meshes* (información sobre la estructura tridimensional del escenario) al formato de *state graph* esperado.
- **Crear los episodios** automáticamente a partir de la información del conjunto de datos.
- **Gestionar la iteración de episodios**, ordenándolos de forma que se reduzcan las lecturas de ficheros.
- Permitir al usuario **filtrar los episodios** en base a criterios especificados.

De forma similar a las tareas y los episodios, existe una clase base *Dataset* que implementa toda la lógica necesaria, existiendo subclases para cada tipo de tarea:

- ***PointNavDatasetV1*** para *NavigationTask* y *NavigationEpisode*.
- ***ObjectNavDatasetV1*** para *ObjectNavigationTask* y *ObjectGoalNavigationEpisode*
- ***VLNDatasetV1*** para *VLNTask* y *VLNEpisode*.
- ***Matterport3DDatasetV1*** para *EQATask* y *EQAEpisode*. Este conjunto de datos funciona exclusivamente con *Matterport3D*.

- ***RearrangeDatasetV0*** para tareas de reorganización (*RearrangePickTask* y *RearrangeReachTask*) y *RearrangeEpisode*.

Habitat ofrece por defecto soporte a cuatro conjuntos de datos. Además, espera que éstos (y cualquier otro conjunto de datos personalizado que se use) siga una estructura específica. Ambos puntos serán detallados a continuación.

4.2.4.1. Conjuntos de datos disponibles

La primera versión de *Habitat Lab* ofrece por defecto soporte a **dos** conjuntos de datos típicos para el entrenamiento de agentes físicos en interiores:

- ***Matterport3D* [58]:** *Matterport3D* es un conjunto de datos de gran escala formado por **199,400** imágenes *RGB-D* (imágenes obtenidas por cámaras de color y profundidad) tomadas por una cámara *Matterport*, formando un total de **10,800** vistas panorámicas de 90 edificios (incluyendo edificios públicos, oficinas e interiores de viviendas).

  Este conjunto de datos incluye información sobre:

  - Reconstrucciones 3D del escenario, con texturas a partir de las panorámicas.
  - Panorámicas de color (*RGB*).
  - Panorámicas de profundidad.
  - Anotaciones semánticas (tanto a nivel de objetos como de regiones de los escenarios)
  - *Normales* de las superficies.

  Se puede observar un ejemplo de un escenario en la Figura 4.2.

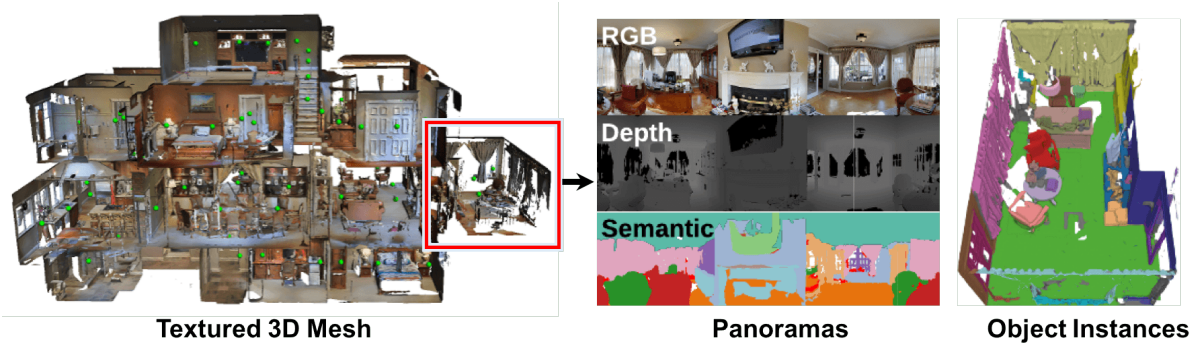

Figura 4.2: Modelos 3D de *Matterport3D*, y panorámicas asociadas [58].

- ***Gibson* [54]:** *Gibson* es un conjunto de datos originalmente diseñado para el simulador *Gibson*, generado a partir de escaneo 3D y reconstrucción de espacios del interior de edificios. El conjunto de datos cuenta con **572** edificios, llegando a un total de **1447** plantas con una superficie total de **211** kilómetros cuadrados.

  Este conjunto de datos incluye información sobre:

- Reconstrucciones 3D del escenario, con texturas a partir de las panorámicas.

- Panorámicas de color (*RGB*).
- Panorámicas de profundidad.
- *Normales* de las superficies.

  Además, una fracción de los escenarios incluyen anotaciones semánticas sobre los objetos contenidos. Se puede ver un ejemplo de un escenario en la Figura 4.3.

Figura 4.3: Imagen de color (izquierda), profundidad (centro) y normales (derecha) de un escenario de *Gibson* [54].

En general, tanto *Matterport3D* como *Gibson* son conjuntos de datos extensos con información similar, pudiendo cumplir las mismas funciones. Ahora bien, *Gibson* se considera un conjunto más *"fácil"* para el entrenamiento de agentes físicos [2], al estar formado por escenarios más pequeños.

Habitat 2.0 añadió soporte a **dos** conjuntos de datos adicionales por defecto:

- ***ReplicaCAD* [3]:** *ReplicaCAD* es una recreación del conjunto de datos *Replica* [56], adaptado para el motor de físicas de *Habitat 2.0*. Se puede ver un ejemplo de esta recreación en la Figura 4.4.

  El conjunto de datos cuenta con **6** cuartos (teniendo cada cuarto **5** variaciones y ruido añadido proceduralmente), donde todos los objetos incluyen simulaciones físicas y anotaciones semánticas.

  Este conjunto de datos está diseñado expresamente para tareas de reorganización, no estando preparado para otras tareas como navegación.

- ***Habitat-Matterport 3D Research Dataset* [59]:** *Habitat-Matterport 3D Research Dataset* (también conocido como *HM3D*) es un conjunto de datos diseñado por el *Facebook AI Research*, generado a partir de imágenes tomadas con cámaras *Matterport*.

  Actualmente cuenta con **1000** escenarios 3D realizados con imágenes de alta resolución de edificios residenciales, comerciales, públicos... Esto lo hace el conjunto de datos más grande actualmente, con alrededor de **365** kilómetros cuadrados de superficie.

  El conjunto de datos ofrece la siguiente información:

Figura 4.4: Escenario original de *Replica* (izquierda) y escenario reconstruido de *ReplicaCAD* (derecha) [3].

- Reconstrucciones 3D del escenario, con texturas a partir de las panorámicas.
- Panorámicas de color (*RGB*).
- Panorámicas de profundidad.
- Anotaciones de metadatos de cada escena (incluyendo información como la valoración, el numero de pisos y cuartos, la cantidad de ruido en el escenario...)

Se puede observar un ejemplo de un escenario con la información en la Figura 4.5.

Figura 4.5: Imagen de color (izquierda), profundidad (centro) y mapa (derecha) de una escena de *HM3D* [59].

El conjunto *HM3D* ofrece características y dificultad similar a *Matterport3D* y *Gibson*, siendo su principal diferencia la cantidad de escenarios disponibles.

4.2.4.2. Estructura de los conjuntos de datos

Para trabajar con conjuntos de datos, *Habitat* espera que se siga una estructura concreta en el árbol de directorios, como se puede ver en la Figura 4.6.

Especificamente, se espera una carpeta base *data* (o un enlace simbólico a la misma) en el mismo directorio que el fichero ejecutable, que debe tener en su interior a su vez dos carpetas:

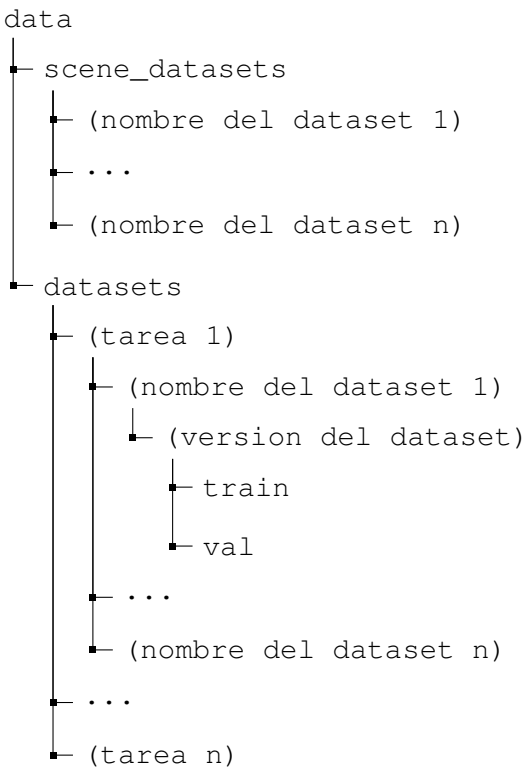

Figura 4.6: Ejemplo de árbol de directorios de la carpeta *data*.

- ***scene_datasets*:** Esta carpeta contiene los modelos 3D de las escenas de cada conjunto de datos, estando cada conjunto contenido en una carpeta con su nombre. Los escenarios en general se encuentran en formato *.glb*, aunque pueden usarse otros.
- ***datasets*:** Esta carpeta contiene, para cada tarea, la definición de todos los escenarios (posición inicial, meta...) para cada conjunto de datos. *Habitat Lab* ofrece estos conjuntos, pudiendo descargarse desde el repositorio de *GitHub*.

4.2.5. Acciones

Una **acción** *(action)* representa una acción que el agente puede realizar en el entorno mientras resuelve una tarea.

Las acciones derivan de la clase base *SimulatorTaskAction*. Cada tarea incluye un conjunto de acciones posibles, siendo las principales acciones relacionadas con navegación las siguientes:

- ***MOVE_FORWARD*:** Mueve el agente hacia adelante. Por defecto, el agente avanza $0.25m$.
- ***TURN_RIGHT / TURN_LEFT*:** Gira al agente sobre sí mismo hacia la derecha o la izquierda respectivamente. Por defecto, el agente rota 10 grados.
- ***LOOK_UP / LOOK_DOWN*:** Mueve el ángulo de visión del agente hacia arriba o hacia abajo respectivamente. Por defecto, el ángulo se mueve 15 grados.
- ***TELEPORT*:** Teletransporta al agente a las coordenadas indicadas.

- ***VELOCITY_CONTROL*:** Ajusta la velocidad lineal y angular del agente a la indicada.
- ***STOP*:** Finaliza el episodio actual, evaluando si ha sido un éxito o no dependiendo de las métricas usadas. Esta acción es importante para evaluar el rendimiento de los agentes en tareas de navegación [5].

Existen otras acciones relacionadas con otras tareas (como *VLN*, *EQA* o *reorganización*) que no han sido mencionadas al no ser relevantes para el problema de navegación a resolver.

4.2.6. Sensores

Un **sensor** proporciona información del entorno al agente. Esta información es recibida tras cada paso de simulación realizado (siendo el valor devuelto por el método *step* de los entornos).

La clase *Sensor* implementa la funcionalidad básica, existiendo gran cantidad de sensores para diversas tareas. A continuación se describen algunos de los sensores más importantes para navegación:

- **Cámaras:** Las cámaras ofrecen información visual (imágenes) del entorno, devuelta en forma de matriz de valores numéricos. Por defecto, las cámaras apuntan hacia el frente del agente con un ángulo de visión de 90 grados. Existen tres tipos básicos de cámaras disponibles:
  - **Cámara de color (*RGB_SENSOR*):** Devuelve una imagen en color (siguiendo el formato RGB).
  - **Cámara de profundidad (*DEPTH_SENSOR*):** Devuelve una imagen en escala de grises representando la profundidad percibida por la cámara. Los valores de esta imagen se representan en forma de números reales en el rango $[0, 1]$ siendo 0.0 lo más cercano a la cámara y 1.0 lo más lejano.
  - **Cámara semántica (*SEMANTIC_SENSOR*):** Devuelve una imagen seccionada en categorías semánticas (como suelo, techo...). Cada categoría semántica tiene un color asociado. Esta cámara solo puede usarse en conjuntos de datos compatibles con anotaciones semánticas.

  Se puede ver un ejemplo de la misma escena vista desde los tres tipos de cámaras en la Figura 4.7.
- **Indicadores de posición:** Los indicadores de posición dan información al agente sobre su propia posición o la de la meta. Estos sensores son exclusivos de tareas de navegación, siendo los principales:
  - ***GPS (GPS_SENSOR)*:** Indica las coordenadas actuales del agente.
  - **Brújula *(COMPASS_SENSOR)*:** Indica la orientación actual del agente.
  - **Meta *(POINTGOAL_SENSOR)*:** Indica las coordenadas de la meta.
  - **Combinado *(POINTGOAL_WITH_GPS_COMPASS_SENSOR)*:** Una combinación de los tres sensores anteriores, indicando las coordenadas de agente y meta y el ángulo del agente. Además, indica la distancia del agente a la meta.

Figura 4.7: Imagen de color (izquierda), profundidad (centro) y semántica (derecha) de una escena de *Gibson* [54].

- **Proximidad *(PROXIMITY_SENSOR)*:** Indica la distancia en metros del obstáculo más cercano al agente.

4.2.7. Métricas

Una **métrica** *(measure)* indica información sobre la tarea que se está realizando. Esta información no está disponible para el agente, sino que se usa para evaluar el éxito de los episodios y para obtener estadísticas del proceso.

La clase *Measure* implementa la funcionalidad básica de las métricas, de la que se debe derivar cualquier otra métrica definida. Además, las métricas se definen para tareas específicas, no pudiendo usarse métricas de una tarea en otra distinta. Las principales métricas usadas en navegación son:

- **Distancia a la meta *(DISTANCE_TO_GOAL)*:** Indica la distancia actual en metros entre el agente y la meta.
- **Éxito *(SUCCESS)*:** Valor booleano que indica si el agente actualmente ha alcanzado la meta o no. No es posible usar esta métrica sin incluir la distancia a la meta.
- ***Success weighted by Path Length (SPL)*:** Métrica originalmente propuesta por Peter Anderson *et al.* [5] y diseñada para ser usada en un conjunto de episodios, indica un valor en el rango $[0, 1]$ calculado a partir de la tasa de éxito ponderada por la distancia recorrida para alcanzar ésta. Esta métrica se calcula usando la siguiente fórmula:

$$\frac{1}{N}\sum_{i=1}^{N} S_i \frac{l_i}{max(p_i, l_i)}$$

  Donde N es el número total de episodios, S_i es un valor binario (1 si el episodio es un éxito, 0 en otro caso), l_i es la distancia más corta entre la posición inicial y la meta en el episodio i, y p_i es la distancia recorrida por el agente.

  Esta métrica es una mejor estimación del rendimiento de los agentes, al ser capaz de penalizar a los agentes por tomar rutas subóptimas. Ahora bien, es una métrica estricta, siendo un valor de 0.5 suficiente para esperar un buen rendimiento por parte del agente.

- ***SPL* suavizado (*SOFT_SPL*):** Variante de *SPL* diseñada para ser una métrica menos estricta. La diferencia es que el valor S_i pasa de ser binario a lineal. El nuevo valor de la métrica es:

$$\frac{1}{N}\sum_{i=1}^{N} S_i \frac{l_i}{max(p_i, l_i)}$$

  Donde N es el número total de episodios, l_i es la distancia más corta entre la posición inicial y la meta en el episodio i, p_i es la distancia recorrida por el agente y S_i es:

$$S_i = 1 - \frac{p_i^{euc}}{l_i^{euc}}$$

  Donde l_i^{euc} es la distancia euclidiana entre la posición inicial y la final, y p_i^{euc} es la distancia euclidiana entre el agente y la posición final. Esto es equivalente al porcentaje de la distancia que ha recorrido el agente (donde $S_i = 0$ equivale a un agente que no se ha acercado a la meta y $S_i = 1$ equivale a un agente que ha alcanzado la meta).

- **Colisiones *(COLLISIONS)*:** Indica el número total de colisiones hasta el momento y si el agente está colisionando actualmente.

- **Mapa del escenario *(TOP_DOWN_MAP)*:** Mapa en plano cenital donde se muestra el escenario entero. Además, se muestra la posición actual del agente, el camino recorrido hasta el momento por el agente (en azul) y la ruta óptima desde la posición inicial hasta la meta (en verde). Se puede ver un ejemplo de un mapa en la Figura 4.8.

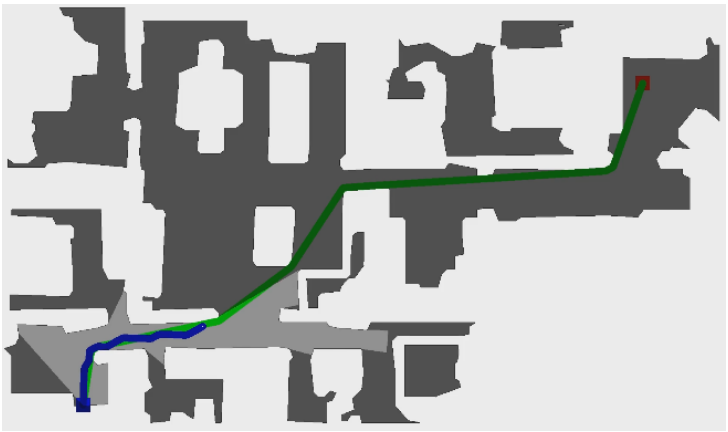

Figura 4.8: Mapa de un escenario de *HM3D* [59].

Las métricas generalmente son devueltas por el método *get_info* de los entornos, y son principalmente usadas durante tareas de aprendizaje por refuerzo.

4.2.8. Entrenadores

Un **entrenador** *(trainer)* es una clase conteniendo los algoritmos y métodos necesarios para el entrenamiento de un agente con aprendizaje (ya sea aprendizaje por refuerzo o aprendizaje por imitación). A pesar de no estar mencionados directamente en la documentación, los entrenadores son una de las partes más esenciales del uso del simulador, al ser la principal forma de realizar el proceso de entrenamiento.

Si bien *Habitat Lab* no ofrece por defecto ningún entrenador, *Habitat Baselines* incluye la clase *BaseTrainer* (con los métodos básicos que deben heredar los entrenadores) y *BaseRLTrainer* (una subclase con métodos usados por problemas de aprendizaje por refuerzo).

Los entrenadores de aprendizaje por refuerzo deben heredar de la clase *BaseRLTrainer* y ser desarrollados específicamente para el algoritmo usado, teniendo que implementar los siguientes métodos:

- ***save_checkpoint(nombre) / load_checkpoint(ruta)*:** Guardan un *checkpoint* con el nombre especificado o cargan un *checkpoint* con la ruta especificada, respectivamente.

  Para *Habitat*, se suele entender un *checkpoint* como un fichero comprimido conteniendo los **pesos de la red neuronal** usada durante el entrenamiento y una copia del **fichero de configuración** usado. De esta forma, es posible reanudar el entrenamiento en cualquier momento garantizando que no hay modificaciones en el entorno.

- ***_eval_checkpoint(ruta)*:** Evalúa el rendimiento de un *checkpoint* (indicado por la ruta) en el entorno. Este método es llamado desde un método superior ya implementado, *eval*, que evalúa todos los *checkpoints* realizados. El método se usa principalmente para elegir el *checkpoint* de mayor rendimiento en el conjunto de evaluación.

  La forma más típica de evaluar un *checkpoint* es simular uno o varios episodios enteros usando la información contenida en el *checkpoint*, y posteriormente valorar las métricas del entorno para realizar una comparativa.

- ***train()*:** El método más importante del entrenador, *train* se encarga de:

  - Inicializar los modelos y estructuras de datos usadas.
  - Simular los episodios del agente.
  - Aplicar el algoritmo de aprendizaje, actualizando los modelos conforme son entrenados.

  El entrenamiento se realiza en bucle hasta que se cumpla una de las dos condiciones de parada posibles: se alcanza el **número máximo de pasos** durante todos los episodios, o se realiza el **número de episodios** indicado. Solo se puede indicar una de las dos condiciones de parada a la vez.

  Por lo general, el método de entrenamiento sigue el pseudocódigo de la Figura 4.9. Este bucle se puede modificar para añadir otras acciones que sea necesario realizar (como actualizaciones durante el episodio o registro de información).

BaseRLTrainer además incluye varios métodos ya implementados para facilitar el desarrollo del entrenamiento y la comprobación de las condiciones:

- ***is_done()*:** Devuelve un valor booleano **verdadero** si se ha cumplido la condición de parada (ya sea número de pasos o de episodios), o **falso** en otro caso. Este método se usa para controlar el bucle general de entrenamiento.
- ***should_checkpoint()*:** Devuelve un valor booleano **verdadero** si es necesario realizar un *checkpoint*, o **falso** en otro caso.

Algoritmo 5: Bucle principal de entrenamiento

1. Inicializa las estructuras de datos necesarias para el entrenamiento, los contadores globales de pasos num_steps_done y actualizaciones $num\_updates\_done$, y los modelos a entrenar.
2. Mientras no se cumpla la condición de parada (número máximo de pasos o número de actualizaciones):
  2.1. Prepara el escenario (a través del método *reset* del entorno)
  2.2. Mientras no haya finalizado el episodio:
    2.2.1. El agente percibe el estado del escenario a través de los sensores.
    2.2.2. El agente procesa el estado, actualizando su modelo si es necesario, y eligiendo una acción.
    2.2.3. El agente aplica la acción (a través del método *step* del entorno).
    2.2.4. Incrementa el contador global de pasos num_steps_done.
  2.3. Actualiza el modelo.
  2.4. Si es necesario, guarda un *checkpoint* del estado actual del modelo.
  2.5. Incrementa el contador global de actualizaciones $num\_updates\_done$.

Figura 4.9: Pseudocódigo del bucle principal de entrenamiento.

Este método calcula automáticamente el porcentaje de entrenamiento realizado, y si es necesario realizar un *checkpoint* en base a dicho porcentaje (a partir del número de *checkpoints* que se ha especificado en el fichero de configuración).

En el trabajo se ha implementado un entrenador personalizado (*ReactiveNavigationTrainer*) como subclase de *BaseRLTrainer*, implementando un algoritmo de aprendizaje por refuerzo via *Deep Q-Learning*.

4.2.9. Agentes

Un **agente** (*agente*) es un contenedor para los modelos entrenados, diseñado para ser usado con los *benchmarks* de *Habitat*.

Los agentes deben heredar de la clase *Agent*, e implementar los dos siguientes métodos:

- ***reset()*:** Prepara al agente para el inicio de un nuevo episodio.
- ***act(observaciones)*:** A partir de las observaciones, el agente debe devolver una acción a realizar en forma de diccionario $\{action = "accion\_elegida"\}$.

Estos métodos son llamados automáticamente por el entorno contenido en el *benchmark*.

Habitat Baselines ofrece algunos agentes preconstruidos (principalmente agentes heurísticos para funcionar como *benchmark* y un agente de aprendizaje aplicando *PPO*). En este trabajo se ha diseñado un agente propio (*ReactiveNavigationAgent*) para evaluar el rendimiento de la propuesta.

En contra de lo que pueda dar a entender el nombre del concepto, los agentes no tienen ninguna relación con los agentes físicos simulados, y su uso se limita exclusivamente a *benchmarks*.

4.2.10. *Benchmarks*

Habitat ofrece por defecto un *benchmark* con el que se puede evaluar el rendimiento de los agentes ya entrenados. El benchmark recibe como entrada un agente (definido previamente) y el número de episodios que se quiere evaluar (por defecto todos los posibles), y se encarga de:

- Crear y gestionar el entorno (usando un entorno *Env*).
- Simular internamente los episodios, usando los métodos del entorno y del agente.
- Obtener las métricas de cada episodio realizado.

El *benchmark* devuelve como resultado final el valor medio de cada métrica tras todos los episodios evaluados, para poder ser analizado y comparado con otros agentes.

4.2.11. Ficheros de configuración

Habitat usa un **fichero de configuración** para cargar todos los parámetros necesarios para su funcionamiento. Este fichero es, junto a los entrenadores, una de las partes más importantes del uso del simulador, pese a no estar recogida en la documentación.

Los ficheros de configuración son documentos en formato *YAML* [60] divididos en bloques. Dentro de cada bloque hay pares de clave - valor que se corresponden al parámetro a configurar (clave) y el valor asignado a ese parámetro (valor).

Los principales bloques del fichero de configuración son:

- **ENVIRONMENT:** En este bloque se incluyen parámetros relativos al entorno, incluyendo la forma en la que se ordenan los episodios o las duraciones máximas de éstos. Algunas de las claves principales de éste bloque son:
  - **MAX_EPISODE_STEPS:** Pasos máximos que puede dar un agente en un episodio. Si se supera este valor, el episodio se finaliza inmediatamente.
  - **MAX_EPISODE_SECONDS:** Duración máxima del episodio en segundos. Si se supera este valor, el episodio se finaliza inmediatamente.
- **SIMULATOR:** En este bloque se incluyen parámetros relativos al simulador y al agente, como los sensores a usar o los parámetros de éstos. Algunas de las claves principales de este bloque son:
  - **AGENT_0:** Sub-bloque donde se encuentra la información sobre el agente. Su clave más importante es **SENSORS**, donde se indica en una lista (entre corchetes) los sensores a usar.
  - **RGB_SENSOR / DEPTH_SENSOR:** Sub-bloques donde se especifican los parámetros de los sensores, **WIDTH** (anchura en píxeles de la imagen devuelta por el sensor) y **HEIGHT** (altura en píxeles). El resto de sensores se pueden configurar de una forma similar.
- **DATASET:** Contiene información sobre el conjunto de datos a utilizar. Algunas de las claves principales de este bloque son:

  - **TYPE:** Tipo de conjunto de datos a utilizar. Este tipo se debe corresponder con el tipo de tarea a realizar.
  - **DATA_PATH:** Ruta a los ficheros del conjunto de datos a utilizar. Se puede usar la palabra $\{split\}$ para indicar una sección de la ruta que se sustituye por el valor de la variable **SPLIT**.
  - **SPLIT:** Partición del conjunto de datos a utilizar. Por defecto, las particiones son **train** (entrenamiento) y **val** (validación).
- **TASK:** Contiene información sobre la tarea a realizar y las métricas a utilizar. Algunas de las claves principales de este bloque son:
  - **SENSORS:** Lista de métricas a usar para la valoración de la tarea. Estas métricas pueden personalizarse en sub-bloques, de forma similar a la configuración de los sensores.
  - **SUCCESS_DISTANCE:** Distancia (en metros) a la que tiene que estar el agente de la meta para que se considere que el episodio se ha completado con éxito.
- **RL:** Contiene información relacionada con el aprendizaje por refuerzo, para ser usada por los entrenadores. Esta sección es opcional y no tiene una estructura fija, pudiendo añadir las claves deseadas para ser leídas posteriormente.
- **Claves sin bloque:** Se pueden indicar claves fuera del resto de bloques, quedando a la altura de la raíz. Estas claves en general están relacionadas con configuraciones para los procesos de entrenamiento y registro de datos, siendo algunas de las principales:
  - **BASE_TASK_CONFIG_PATH:** Ruta al fichero de configuración base. Si se especifica este valor, los valores contenidos en este fichero se fusionarán con los del fichero indicado. Permite crear una configuración base sobre la que crear variaciones.
  - **TRAINER_NAME / ENV_NAME:** Identificador del entrenador o del entorno a utilizar, respectivamente. Estos identificadores se usan para obtener la clase adecuada del registro (como se verá después).
  - **TOTAL_NUM_STEPS / NUM_UPDATES:** Número máximo de pasos o actualizaciones a realizar durante el entrenamiento. El entrenamiento dura hasta que se alcanza uno de estos valores. Solo puede aparecer una de las dos claves en el fichero.
  - **NUM_CHECKPOINTS:** Número total de *checkpoints* a realizar durante el proceso de entrenamiento.
  - **CHECKPOINT_FOLDER:** Carpeta en la que se almacenarán los *checkpoints* realizados.

  Existen más claves dentro de esta categoría, pudiendo observarse en los ficheros de configuración de ejemplo.

Se puede ver un ejemplo de un fichero de configuración para una tarea de navegación usando *Gibson* en la Figura 4.10. Además, se pueden ver los ficheros de configuración desarrollados (documentados en inglés) en el Anexo B (Ficheros de configuración).

```
ENVIRONMENT:
  MAX_EPISODE_STEPS: 500
SIMULATOR:
  AGENT_0:
    SENSORS: ['RGB_SENSOR']
  HABITAT_SIM_V0:
    GPU_DEVICE_ID: 0
  RGB_SENSOR:
    WIDTH: 256
    HEIGHT: 256
  DEPTH_SENSOR:
    WIDTH: 256
    HEIGHT: 256
TASK:
  TYPE: Nav-v0
  SUCCESS_DISTANCE: 0.2

  SENSORS: ['POINTGOAL_WITH_GPS_COMPASS_SENSOR']
  POINTGOAL_WITH_GPS_COMPASS_SENSOR:
    GOAL_FORMAT: "POLAR"
    DIMENSIONALITY: 2
  GOAL_SENSOR_UUID: pointgoal_with_gps_compass

  MEASUREMENTS: ['DISTANCE_TO_GOAL', 'SUCCESS', 'SPL']
  SUCCESS:
    SUCCESS_DISTANCE: 0.2

DATASET:
  TYPE: PointNav-v1
  SPLIT: train
  DATA_PATH: data/datasets/pointnav/gibson/v1/{split}/{split}.json.gz
```

Figura 4.10: Fichero de configuración por defecto para tareas de navegación usando *Gibson*.

El simulador no trabaja directamente con el fichero de configuración, sino que primero convierte el contenido del fichero a un objeto de la clase *Config*. Este objeto contiene todos los valores del fichero en forma de diccionario, y se puede obtener de la siguiente forma:

- **Método *get_config(ruta)* de *Habitat Lab*:** Este método (disponible en el fichero *habitat/config/default.py*) genera una instancia de *Config* a partir de una serie de valores por defecto y los valores indicados en el fichero de configuración. Es recomendable usar este método para configuraciones relacionadas con *benchmarks*.

- **Método *get_config(ruta)* de *Habitat Baselines*:** Este método se diferencia del anterior en tres puntos:

- Incluye valores por defecto de aprendizaje por refuerzo.
- Es capaz de cargar los valores de otro fichero de configuración (indicado con la clave *BASE_TASK_CONFIG_PATH*.
- Almacena los valores de los cuatro bloques principales (*ENVIRONMENT, SIMULATOR, DATASET* y *TASK*) bajo un nuevo bloque llamado *TASK_CONFIG*.

Es recomendable usar este método para configuraciones de entrenadores.

4.2.12. Registros

Un **registro** (*registry*) es un contenedor global y central de información, en el que se almacenan pares de clave y clase asociada a la clave. Estos registros sirven para poder instanciar las clases apropiadas (de simulador, tarea, entorno...) a partir de sus identificadores, siendo especialmente útil para cargar las clases indicadas en el fichero de configuración.

Los registros funcionan usando **decoradores**, funciones que se llaman durante la definición de las clases y que las registran para poder acceder a ellas posteriormente desde cualquier punto del código.

Por defecto, *Habitat* incluye dos registros, cada uno almacenando información distinta:

- **Registro *Registry* de *Habitat Lab*:** El registro principal en el que se incluye información sobre:
  - **Tareas**, usando el decorador *@registry.register_task*.
  - **Acciones**, usando el decorador *@registry.register_task_action*.
  - **Simuladores**, usando el decorador *@registry.register_simulator*.
  - **Sensores**, usando el decorador *@registry.register_sensor*.
  - **Métricas**, usando el decorador *@registry.register_measure*.
  - **Conjuntos de datos**, usando el decorador *@registry.register_dataset*.
- **Registro *BaselineRegistry* de *Habitat Baselines*:** Un registro adicional pensado para almacenar las clases creadas por el usuario, incluye información sobre:
  - **Entornos**, usando el decorador *@baseline_registry.register_env*.
  - **Entrenadores**, usando el decorador *@baseline_registry.register_trainer*.
  - **Políticas de acciones**, usando el decorador *@baseline_registry.register_policy*.

4.3. Instalación de *Habitat*

El proceso de instalación de *Habitat* puede resultar complicado debido a la gran cantidad de componentes y versiones específicas necesarias. Por eso, en esta sección se indican los pasos necesarios para instalar el entorno de *Habitat*, remarcando los requisitos de versiones necesarios para el funcionamiento adecuado.

4.3.1. Requisitos

El entorno de *Habitat* requiere versiones específicas de

- **Python:** Versión superior a *3.6.0* (recomendable usar *3.6.10*).
- **cmake:** Versión superior a *3.10* (recomendable usar *3.14.0*).
- **Sistema operativo:** Si bien *Habitat Sim* incluye soporte para Windows, es recomendable usar alguna distribución de *Linux* con soporte para *CUDA*. En este trabajo se ha utilizado **Ubuntu 20.04**.
- **Tarjeta gráfica:** En caso de ser necesaria, se necesita una tarjeta gráfica de *nVidia* que ofrezca soporte para *CUDA*.
- **CUDA:** Se ha usado la versión *11.0* en el trabajo realizado, aunque es posible que dependiendo de la configuración se necesite otra.

Además, la instalación de *Habitat Lab* y *Habitat Baselines* instala versiones concretas de diversas librerías de *Python*, por lo que es recomendable instalar *Habitat* en un entorno propio de *Conda* para evitar problemas de compatibilidad.

4.3.2. Proceso de instalación

Si bien es posible utilizar *dockers* para instalar el entorno del simulador, es recomendable realizar la instalación directamente para garantizar acceso a las actualizaciones. El proceso de instalación en distribuciones de *Linux* se puede dividir en tres pasos principales:

1. **Instalación de CUDA:** Para el uso de *Habitat* y para el entrenamiento adecuado de redes neuronales, es necesaria una instalación correcta y completa de CUDA. Para esto, es necesario:

   a) Tener *drivers* **oficiales** de *nVidia*, actualizados a la versión más moderna. No es necesaria una versión concreta del *driver*.

   b) Instalar **CUDA**. La guía oficial de instalación en *Linux*[1] incluye los pasos a seguir, mientras que la página de descarga[2] ofrece las instrucciones para descargar e instalar CUDA.

   Es recomendable seguir los pasos de pre-instalación y post-instalación indicados en la guía oficial, pero seguir los pasos de la página de descarga para la instalación.

   c) Instalar **cuDNN**. La guia oficial de instalación en *Linux*[3] incluye los pasos necesarios para instalar cuDNN. Es importante comprobar que las versiones de CUDA y cuDNN instaladas sean compatibles entre sí.

   Es necesaria una cuenta de desarrollador de *nVidia* para la descarga de cuDNN.

2. **Instalación de *Habitat Sim*:** La instalación de *Habitat Sim* se hace a través del repositorio de paquetes *Conda*. Además, como se ha comentado previamente,

[1] https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html
[2] https://developer.nvidia.com/cuda-downloads
[3] https://docs.nvidia.com/deeplearning/cudnn/install-guide/index.html

es recomendable crear un entorno propio de *Conda* para evitar problemas de compatibilidades.

Contando con una instalación válida de *Conda*, los siguientes comandos en una terminal crean un entorno con las versiones adecuadas de *Python* y *cMake* e instalan *Habitat Sim*:

```
# Crea el entorno de Conda con las versiones adecuadas de Python y cMake
conda create -n habitat python=3.6 cmake=3.14.0
# Inicializa el entorno creado
conda activate habitat
# Instala el simulador con soporte para fisicas (withbullet)
conda install habitat-sim withbullet -c conda-forge -c aihabitat
```

3. **Instalación de *Habitat Lab* y *Habitat Baselines*:** Ya con el simulador instalado, el último paso es instalar la librería *Habitat Lab*. Si bien *Habitat Baselines* es opcional, es muy recomendable su instalación por las utilidades adicionales que ofrece. Los siguientes comandos en una terminal instalan *Habitat Lab* y *Habitat Baselines*:

```
# Descarga la version mas reciente del repositorio de Habitat Lab
# El repositorio se descarga en la carpeta actual
git clone --branch stable https://github.com/facebookresearch/habitat-lab.git
# Accede al repositorio descargado
cd habitat-lab
# Instala todos los paquetes de Python necesarios
# Es recomendable realizar la instalacion en el mismo entorno que Habitat Sim
# para evitar incompatibilidades
pip install -r requirements.txt
# Instala Habitat Lab y Habitat Baselines
python setup.py develop --all
```

Para comprobar que la instalación se ha realizado correctamente, *Habitat Lab* ofrece el fichero *examples/example.py*. Si no ha habido ningún problema en la instalación, al ejecutar el fichero se realizará una simulación con un agente de ejemplo ofrecido por defecto por *Habitat*, imprimiendo en la terminal el número de pasos realizado.

Capítulo 5

Diseño del agente

En este capítulo se detallará el diseño del agente propuesto para la resolución del problema.

Se empezará presentando la formalización del conocimiento usada (estado, acción y recompensa) y las características físicas del agente. Tras esto, se describirán las dos arquitecturas propuestas para el agente: una primera basada en una red neuronal convolucional, y una segunda basada en una red profunda híbrida. Finalmente, se explicará el proceso de actuación y entrenamiento del agente.

5.1. Caracterización del conocimiento

En esta sección se describe la caracterización del conocimiento realizada, centrándose en los puntos principales del aprendizaje por refuerzo: el **estado**, las **acciones** disponibles y las **recompensas** que recibe el agente. Además, se describen las características del agente propuesto dentro del entorno *Habitat*.

5.1.1. Características del agente físico en *Habitat*

Las características del agente físico simulado son las siguientes:

- **Forma física del agente:** La forma del agente es un **cilindro** de 1.5 metros de altura y 0.1 metros de radio.

  Esta forma física no se corresponde a ningún robot real, siendo la del agente por defecto simulado por *Habitat*.

- **Capacidad de movimiento:** El agente únicamente es capaz de desplazarse hacia delante, pudiendo moverse 0.25 metros en cada paso simulado.

  Para cambiar la dirección de su movimiento, el agente necesita rotar sobre sí mismo. Ésto lo hace en intervalos de 10 grados en cada paso simulado. El agente no puede girar y desplazarse en el mismo paso.

- **Sensores disponibles:** El agente cuenta con acceso a los siguientes sensores:

  - **Cámara de profundidad:** Una cámara de profundidad (*DEPTH_SENSOR)* situada a una altura de 1.25 metros respecto al suelo, apuntando a la dirección en la que se desplaza el agente.

Esta cámara genera imágenes de profundidad de $(256x256)$ pixeles con valores en el rango $[0.0, 1.0]$. La cámara tiene un ángulo de visión de 90 grados, y es capaz de detectar objetos hasta una distancia de 10 metros.

- **Brújula y GPS (*POINTGOAL_WITH_GPS_COMPASS_SENSOR*):** Una brújula y un GPS que conocen la posición exacta de la meta en todo momento. Estos sensores ofrecen el **ángulo** y la **distancia euclidiana** hasta la meta en forma de valor decimal.

- **Métricas usadas:** El agente ofrece las siguientes métricas para su evaluación posterior:
  - **Distancia hasta la meta**: Un valor numérico que indica la distancia euclidiana hasta la meta (la distancia euclidiana entre el agente y la meta) en metros.
  - **Éxito:** Un valor booleano que indica si el agente está en la meta (verdadero) o falso. El agente se considera en la meta si se encuentra a menos de 0.3 metros de esta.
  - ***SPL* y *Soft SPL*:** Dos valores numéricos que indican la métrica de *Success weighted by Path Length* [5] y su variante suavizada. Ambas fórmulas fueron definidas en el Capítulo 4.

5.1.2. Estado

El **estado** (la percepción que tiene el agente del entorno) consta de tres elementos:

- **Imagen de profundidad:** Una imagen en escala de grises representando las observaciones de la cámara de profundidad. Esta imagen tiene un tamaño de $(256x256)$ pixeles, con los valores de cada celda en el rango $[0.0, 1.0]$ (donde 0.0 o negro significa cercanía a la cámara y 1.0 o blanco significa lejanía). Se puede ver un ejemplo de la imagen en la Figura 5.1.

Figura 5.1: Ejemplo de imagen de profundidad usada como parte del estado.

Se ha optado por añadir este valor debido al planteamiento del problema. Al buscar diseñar un agente reactivo frente a obstáculos, es importante que el agente

sea capaz de percibir cualquier objeto que se encuentre en su camino. Una cámara de profundidad nos permite estimar las distancias a estos obstáculos de forma rápida y simple.

- **Distancia a la meta:** Un valor decimal que representa la distancia actual entre el agente y la meta en metros.

  Este valor forma parte del estado al ser parte del cálculo de la recompensa (como se verá posteriormente). Además, es importante que el agente sea capaz de estimar la distancia hasta la meta para que tenga la posibilidad de aprender cuando detenerse.

- **Ángulo respecto a la meta:** Un valor decimal que representa el ángulo que debería girar el agente para enfocarse hacia la meta, en radianes. Un valor positivo significa que el agente debería girar hacia la derecha, mientras que un valor negativo significa que el agente debería girar hacia la izquierda.

  Si bien este valor no forma parte del cómputo de la recompensa, se ha optado por añadir el ángulo al estado para permitir al agente tener la posibilidad de aprender información respecto a su orientación que no podría aprender en otro caso.

Si bien se consideró añadir una **imagen en color** al estado, finalmente se ha optado por no incluirla. Esto se debe a que la información que añade resulta superflua, ya que la cámara de profundidad incluye toda la información necesaria para el cálculo de obstáculos. Además, los escenarios de interior son complejos con altos niveles de ruido, por lo que una cámara de color podría llevar a sobreajustes (al aprender el aspecto de los interiores frente a los obstáculos).

Un estado se considera **final** cuando el agente lo finaliza (ya sea por realizar la acción específica de terminar o por superar el número máximo de acciones permitidas). Este estado final puede ser **exitoso** si la distancia a la meta es menor a un umbral (por defecto 0.3 metros), o **fallido** en otro caso.

5.1.3. Acciones

El agente es capaz de realizar **cuatro** acciones en total:

- **Desplazarse** hacia delante. Por defecto, el desplazamiento es de 0.25 metros.
- **Girar** hacia la derecha o la izquierda. Por defecto, el giro es de 10 grados.
- **Finalizar el episodio**. La inclusión de una acción para finalizar el episodio es una de las principales sugerencias de Peter Anderson *et al.* [5] para la evaluación de agentes físicos.

Como se comentó en el Capítulo 2 y se puede observar, el agente no es capaz de un movimiento omnidireccional (como sí podría un dron volador). El agente únicamente puede desplazarse hacia adelante, necesitando rotar sobre sí mismo para cambiar la dirección de su movimiento.

5.1.4. Recompensas

El sistema de recompensas diseñado está basado en el sistema originalmente propuesto por Carlos Sampedro *et al.* [1], siendo éste un sistema de recompensas basado

en campos de potenciales artificiales, con un **atractor** que atrae al agente hacia la meta y **repulsores** que repelen al agente de los obstáculos. Ahora bien, este sistema ha sido adaptado a la arquitectura del agente desarrollado (con cámara de profundidad), y se han propuesto variantes para evaluar su rendimiento.

El cálculo de la recompensa de un estado se puede dividir en los siguientes pasos:

1. Preprocesamiento de la imagen del estado.
2. Identificación de obstáculos en la imagen (con dos posibles métodos)
3. Cálculo de los potenciales atractivos y repulsivos.
4. Cálculo de la recompensa final (con dos posibles métodos)

Estos pasos se describirán a continuación.

5.1.4.1. Preprocesamiento de la imagen de profundidad

Para calcular las recompensas posteriormente, es necesario identificar los obstáculos o los objetos que pueden suponer un riesgo en la imagen. Ahora bien, no se puede usar la imagen directamente al contener ruido e información necesaria. Por esto, se realiza un preprocesamiento antes de identificar los obstáculos en la imagen, siguiendo los siguientes pasos:

1. **Normalización:** Por defecto, la imagen obtenida por la cámara está formada por valores decimales en el rango $[0.0, 1.0]$. Ahora bien, para poder trabajar por la imagen es necesario que estos valores sean enteros en el rango $[0, 255]$ por compatibilidad con otras librerías. Por tanto, se transforman los valores de un rango al otro.
2. **Recorte de los extremos:** Se ha visto que los extremos superiores e inferiores de la imagen (el suelo y el techo) no aportan información útil a la hora de calcular la recompensa, pudiendo llegar a introducir ruido y obstáculos que no existen.

   Para evitar esto, se recortan los extremos superiores e inferiores de la imagen. Por defecto, se recortan 35 pixeles de cada extremo. Este valor se ha obtenido de forma empírica con pruebas y es heurístico.
3. **Eliminación de artefactos del simulador:** En ocasiones, el simulador introduce ruido en la imagen en forma de partes de color negro puro (con valor 0). Estos artefactos no son reales (al no ser capaz la cámara de devolver un valor tan bajo en la práctica) y pueden interferir con la detección de obstáculos, por lo que es necesario eliminarlos.

   Para eliminarlos, se sustituyen todos los valores de 0 (negro puro) por 255 (blanco puro). Esto hará que sean ignorados en los pasos posteriores.
4. **Umbralización (*Thresholding*):** Es necesario identificar los obstáculos en la imagen. Si bien se podría haber optado por alguna técnica de búsqueda de contornos (como *Canny*), se ha elegido realizar una umbralización, donde todos los valores de la imagen son reemplazados usando la siguiente fórmula:

$$imagen(x) = \begin{cases} 1, & \text{si } imagen(x) \leq suelo(255 * umbral) \\ 0, & \text{en cualquier otro caso} \end{cases}$$

Donde $umbral$ es el valor que se ha tomado para umbralización en el rango $[0.0, 1.0]$ (siendo por defecto 0.15, elegido de forma empírica).

En esencia, esta fórmula sustituye todos los valores menores a $suelo(255 * umbral)$ (cercanos a la cámara) por 1 (blanco), mientras que el resto de valores (lejanos) son sustituidos por 0. De esta forma, se obtiene una imagen binaria en la que solo se conservan los obstáculos más cercanos.

5. **Eliminación de ruido:** El proceso de umbralización puede crear ruido, como pueden ser regiones negras pequeñas dentro de contornos blancos más grandes, que pueden afectar al rendimiento.

   Para eliminar este ruido, se usa una técnica de *apertura morfológica*, que consiste en una dilatación (aumentar el volumen de los objetos en la imagen) seguida de una erosión (disminuir el volumen de los objetos en la imagen). Esto reduce el ruido incluido dentro de los contornos, sin afectar demasiado al volumen final.

6. **Dilatación:** El paso final consiste en dilatar (aumentar el volumen) de la imagen, para evitar pérdidas de información provocadas por la eliminación de ruido del paso previo. Esta dilatación final no afectará al proceso de identificación de obstáculos por su funcionamiento, que se verá posteriormente.

Se puede observar un ejemplo de este proceso en la Figura 5.2.

5.1.4.2. Identificación de los obstáculos y la distancia en la imagen

Tras el preprocesamiento de la imagen, es necesario identificar los obstáculos en la imagen y estimar la distancia a la que éstos se encuentran. Para eso, se han propuesto dos métodos:

- **Método de contornos:** Este método se basa en la propuesta original de Carlos Sampedro *et al.* [1].

  La idea principal del método es identificar los contornos que tengan un área mínima (experimentalmente determinado como 250 pixeles) en la imagen preprocesada. A partir de estos contornos, se crean máscaras en la imagen original para extraer los obstáculos de nuevo en escala de grises. Finalmente, se obtiene la cercanía de esos obstáculos (a partir del pixel de valor mínimo), usando la siguiente estimación:

  $$distancia = \frac{(pixel\_min/256) * distancia\_estimada}{umbral\_obstaculo}$$

  Donde $pixel\_min$ es el pixel de menor valor en el obstáculo (en el rango $0, 255$), $distancia\_estimada$ es un valor heurístico que indica la distancia a la que se encontraría un obstáculo en el umbral (estimado como 2 metros) y $umbral\_obstaculo$ es el umbral que se ha usado durante la umbralización (0.15).

  Se puede ver el pseudocódigo del proceso en la Figura 5.3.

(a) Imagen normalizada.

(b) Imagen recortada.

(c) Imagen sin artefactos.

(d) Imagen con umbral aplicado.

(e) Imagen sin ruido.

(f) Imagen dilatada.

Figura 5.2: Procesamiento realizado sobre la imagen de profundidad.

Algoritmo 6: Identificación de distancias con método de contornos

Variables: Imagen preprocesada $imagen\_p$, imagen recortada $imagen\_r$, umbral usado durante preprocesamiento $umbral$, distancia estimada hasta el umbral en metros $dist$, área mínima de los contornos en pixeles $area\_min$.

1. Inicializa una lista para almacenar las distancias obtenidas, $distancias$.

2. Extrae los contornos de la imagen $imagen\_p$ a una lista $contornos$.

3. Para cada contorno $cont$ de area $area\_contorno$ en $contornos$, con $area\_contorno \geq area\_min$:

- **3.1.** Aplica una máscara con la forma de $cont$ a $imagen\_r$, obteniendo el obstáculo obs (el obstáculo tal y como está representando en la imágen $imagen\_r$, en escala de grises).
- **3.2.** Obtén la distancia mínima en obs (el valor mínimo), $dist\_obs$.
- **3.3.** Convierte $dist\_obs$ de un valor entero en el rango $\{0, 255\}$ a una distancia en metros mediante una equivalencia usando $umbral$ y $dist$.
- **3.4.** Si $dist\_obs \leq dist$, almacena $dist\_obs$ en $distancias$.

4. Devuelve $distancias$.

Figura 5.3: Pseudocódigo del método de contornos para identificar distancias a obstáculos.

- **Método de columnas:** Un método original, consistente en dividir la imagen en columnas de misma anchura, siendo cada columna un posible obstáculo.

  El método consiste en dividir la imagen preprocesada en 8 (elegido experimentalmente) columnas de anchura idéntica. Tras esto, se cuenta el número de pixeles blancos (obstáculos) en cada columna, considerando las columnas que tengan una cantidad superior al área mínima (250 pixeles como se ha mencionado previamente) como obstáculos. Para estas columnas, se estima la distancia a partir de una columna equivalente de la imagen original, usando la fórmula descrita previamente:

  $$distancia = \frac{(pixel\_min/256) * distancia\_estimada}{umbral\_obstaculo}$$

  Donde $pixel\_min$ es el pixel de menor valor en el obstáculo (en el rango $0, 255$), $distancia\_estimada$ es un valor heurístico que indica la distancia a la que se encontraría un obstáculo en el umbral (estimado como 2 metros) y $umbral\_obstaculo$ es el umbral que se ha usado durante la umbralización (0.15).

  Se puede ver el pseudocódigo del proceso en la Figura 5.4.

  Este método se propone al considerar que el método anterior daría la misma importancia a un obstáculo grande (que ocupa gran parte de la pantalla) y a uno pequeño, siempre y cuando estuviesen a la misma distancia. La idea es remediar ese problema, dando más peso a los obstáculos más grandes. Además, al evitar tener que usar algoritmos de búsqueda de contornos, se espera que la velocidad de ejecución sea mayor.

5.1.4.3. Cálculo del potencial atractivo y repulsivo

Tras el cálculo de las distancias a los obstáculos, se calcula el valor de los potenciales atractivos y repulsivos. Este cálculo es equivalente al propuesto por Carlos Sampedro

Algoritmo 7: Identificación de distancias con método de columnas

Variables: Imagen preprocesada $imagen\_p$, imagen recortada $imagen\_r$, umbral usado durante preprocesamiento $umbral$, distancia estimada hasta el umbral en metros $dist$, área mínima de los contornos en pixeles $area\_min$.

1. Divide las imágenes $imagen\_p$ y $imagen\_r$ en columnas de anchuras iguales, $columnas\_p$ y $columnas\_r$.

2. Para cada columna col con $pixeles$ pixeles blancos en $columnas\_p$ y col_r en $columnas\_r$, cumpliendo que $pixeles \geq area\_min$:

  2.1. Obtén la distancia mínima en col_r (el valor mínimo), $dist\_obs$.

  2.2. Convierte $dist\_obs$ de un valor entero en el rango $\{0, 255\}$ a una distancia en metros mediante una equivalencia usando $umbral$ y $dist$.

  2.3. Si $dist\_obs \leq dist$, almacena $dist\_obs$ en $distancias$.

3. Devuelve $distancias$.

Figura 5.4: Pseudocódigo del método de columnas para identificar distancias a obstáculos.

et al. [1] originalmente.

El **potencial atractivo** es la fuerza con la que la meta atrae al agente. Cuanto más cerca esté el agente de la meta, mayor debe ser su influencia. Este potencial se obtiene con la siguiente fórmula:

$$U_{atr} = \alpha p_{goal}(t_r)$$

Donde α es una ganancia positiva usada para aumentar la influencia del potencial (estimada empíricamente como 100) y $p_{goal}(t_r)$ es la distancia euclidiana entre la posición actual del agente y la meta.

El **potencial repulsivo** es la suma de las fuerzas con las que los obstáculos repelen al agente. Cuantos más obstáculos perciba el agente y más cerca se encuentren, mayor debe ser su influencia. Este potencial se obtiene con la siguiente fórmula:

$$U_{rep} = \beta \sum_{i=1}^{N} (\frac{1}{k + l_i} - \frac{1}{k + l_{max}})$$

Donde N es el número total de obstáculos detectados, k es una constante usada para limitar la influencia de los obstáculos (con valor por defecto 0.04), l_i es la distancia al obstáculo i en metros, l_{max} es la distancia máxima a la que se detectan los obstáculos (con valor por defecto 2 metros) y β se obtiene con la siguiente fórmula:

$$\beta = \begin{cases} \delta, & \text{si } p_{goal}(t_r) > d_{infl} \\ \frac{\delta}{exp[4(d_{infl} - p_{goal})]}, & \text{si } p_{goal}(t_r) \leq d_{infl} \end{cases}$$

Donde δ es una ganancia positiva usada para aumentar la influencia del potencial (estimada empíricamente como 15) y $d_{infl} = 0.75 l_{max}$ es una distancia a partir de la cual la influencia del potencial repulsivo se disminuye, para fomentar al agente a acercarse a la meta cuando se encuentra próximo a ésta.

Tras el cálculo de ambos potenciales, es posible calcular el **valor del estado actual**. Cuanto mayor es el valor del estado, mejor estado se considera que es. Este valor se

comparará con el valor del estado previo para comprobar si ha mejorado o empeorado, y obtener una recompensa a partir de ello.

Este valor se calcula como:

$$valor_t = -U_{atr} - U_{rep}$$

5.1.4.4. Cálculo de la recompensa final

Tras haber calculado los potenciales y el valor del estado, es posible obtener la recompensa final a partir de las siguientes reglas:

- Si el episodio ha finalizado (ya sea por la acción correspondiente o por límite de acciones) y el agente no está en rango de la meta: **-100**.

  La penalización por finalizar un episodio sin éxito es muy alta para evitar que el agente finalice el episodio antes de tiempo con la intención de evitar penalizaciones por sus acciones.

- Si el episodio ha finalizado (ya sea por la acción correspondiente o por límite de acciones) y el agente está en rango de la meta: **+10**.

  La recompensa por finalizar un episodio con éxito es menor, pero sigue siendo elevada para fomentar al agente a llegar a la meta y finalizar en ella.

- En cualquier otro caso:

  Si el episodio no ha finalizado tras la acción del agente, se procede a calcular la recompensa a partir de los valores del estado actual y el previo:

  $$recompensa = (valor_t - valor_{t-1}) - 0.25, \text{acotado en el rango } [-100, +10]$$

  Si el valor del estado alcanzado tras realizar la acción es mayor (la acción lleva a un estado mejor) la recompensa será positiva, mientras que si el valor es menor (la acción lleva a un estado peor) la recompensa será negativa. De esta forma, se fomenta que el agente intente mejorar su posición continuamente.

  El término -0.25 es una penalización por paso, usada para evitar que el agente permanezca en bucles infinitos sin recompensa y acelerando su progreso.

Una alternativa propuesta es incluir una regla adicional al cálculo de recompensas:

- Si el agente ha colisionado con algún obstáculo tras la acción: **-100** y **finaliza el episodio**.

  Con esta regla adicional, se penaliza notablemente que el agente colisione con los obstáculos, fomentando que evite cualquier colisión con el entorno. Estas colisiones se comprueban usando la métrica *COLLISIONS*.

5.2. Arquitectura del agente

En esta sección se discuten las arquitecturas (redes neuronales) propuestas para el agente, su funcionamiento y su implementación. Se han propuesto dos arquitecturas, de las cuales se ha elegido una finalmente:

- Una primera aproximación basada en **redes neuronales convolucionales**.
- Una segunda aproximación basada en un **enfoque mixto**, con redes convolucionales y redes neuronales densas tradicionales.

5.2.1. Propuesta 1: Red convolucional (CNN)

La primera aproximación está basada en el uso de **redes convolucionales *(CNNs)*** para el procesamiento de la imagen, extrayendo las características profundas relevantes para trabajar posteriormente con ellas.

Ahora bien, los valores numéricos (parte del estado a procesar) no pueden ser usados directamente por las capas de convolución. Para solventar esto, los dos valores numéricos (distancia y ángulo) son concatenados directamente a la salida aplanada de las convoluciones, para ser procesados posteriormente por las capas densas de neuronas.

La arquitectura de la red neuronal se ha basado en la arquitectura original de *AlexNet* [20], adaptada de forma *ad-hoc* para las necesidades del trabajo y las limitaciones existentes de memoria y tiempo.

La red se puede dividir en varias secciones consecutivas:

1. **Entrada:** La entrada se obtiene de una muestra del *Experience Replay*. Cada elemento de esta muestra es un estado, correspondiendose con la definición dada previamente de estado:
   - Una imagen de profundidad (escala de grises), en forma de una matriz bidimensional de tamaño $256x256$ con una única capa.
   - Dos valores escalares: la distancia y el ángulo hasta la meta.
2. **Convolución:** El primer paso de la red es procesar la imagen a través de varios procesos de convolución, con el fin de obtener las características profundas de ésta. Para esto, se tienen **tres** procesos de convolución consecutivos, cada uno de ellos formado por, en orden:
   - Capa convolucional bidimensional de 16 filtros. El tamaño del *kernel* es de $5x5$, $3x3$ y $3x3$ en cada proceso respectivamente.
   - Función de activación ReLU.
   - Capa de *pooling* por función de máximo. El tamaño del *pooling* es de $3x3$, $3x3$ y $2x2$ en cada proceso respectivamente.

   A pesar de ser típico en redes convolucionales, no se ha incluido ninguna capa de *batch normalization* durante el proceso de convolución. Ésto se debe a que el proceso de normalización introduce ruido que puede alterar el proceso de aprendizaje por refuerzo [61].

3. **Aplanado (*Flatten*):** Tras la convolución de la imagen, el resultado del proceso (una matriz tridimensional profunda con las características de la imagen) es aplanado a un conjunto unidimensional de neuronas. Esto permite a las capas posteriores trabajar con la información obtenida. Esta capa no cuenta con ninguna función de activación.

   En este paso además se concatenan los dos valores escalares de la entrada (distancia y ángulo a la meta). Estos valores no han sido incluidos previamente al no poder ser procesados por las capas convolucionales. Al añadirlos ahora, las capas posteriores podrán trabajar con la información.

4. **Capas densas:** Tras el aplanamiento y concatenación de la información en el paso previo, se incluyen **dos** capas densas (capas de neuronas totalmente conectadas) para extraer relaciones entre la información obtenida.

   Estas capas densas cuentan con 256 neuronas cada una, utilizando *ReLU* como función de activación.

5. **Salida:** Finalmente, la capa de salida es una capa densa de cuatro neuronas, donde cada neurona se corresponde con el valor Q de una de las cuatro acciones disponibles para el agente. Estas neuronas cuentan con función de activación lineal.

Se puede observar un esquema de la arquitectura en la Figura 5.5.

La red neuronal ha sido implementada utilizando las librerías *TensorFlow* y *Keras*. Para las funciones usadas por la red para su entrenamiento, se ha optado por utilizar algunos algoritmos tradicionalmente usados en problemas de aprendizaje profundo, siendo éstos:

- **Función de inicialización de pesos:** Glorot y Bengio [62].
- **Función de optimización:** Adam [63].
- **Función de error:** Error cuadrático medio. Esta función de error es usada tradicionalmente en problemas de aprendizaje por refuerzo profundo.

Tras realizar pruebas iniciales (entrenamientos cortos) con la arquitectura propuesta, se observaron una serie de problemas con ésta:

- **Ineficiencia en memoria:** Debido al tamaño de la red y a problemas con la librería utilizada, el uso de memoria (tanto memoria *RAM* del ordenador como de la *GPU*) era excesivo. Además, este uso crecía tras cada episodio hasta llegar a un punto en el que se detenía el entrenamiento por falta de memoria.

  Esto provocaba que no fuese posible realizar entrenamientos largos, y que los entrenamientos realizados fuesen notablemente más lentos de lo esperado.

- **Malos resultados:** Durante el proceso de entrenamiento se observó que los resultados obtenidos por la red eran peores de lo que se podría esperar, sin mostrar signos de aprendizaje. Esto se puede deber al uso de los valores escalares, al ser menos relevantes (2 neuronas) frente a la información obtenida de la imagen (miles de neuronas).

Por estos problemas, se ha optado por **descartar** esta arquitectura en favor de la segunda propuesta, descrita a continuación. La implementación de esta propuesta se

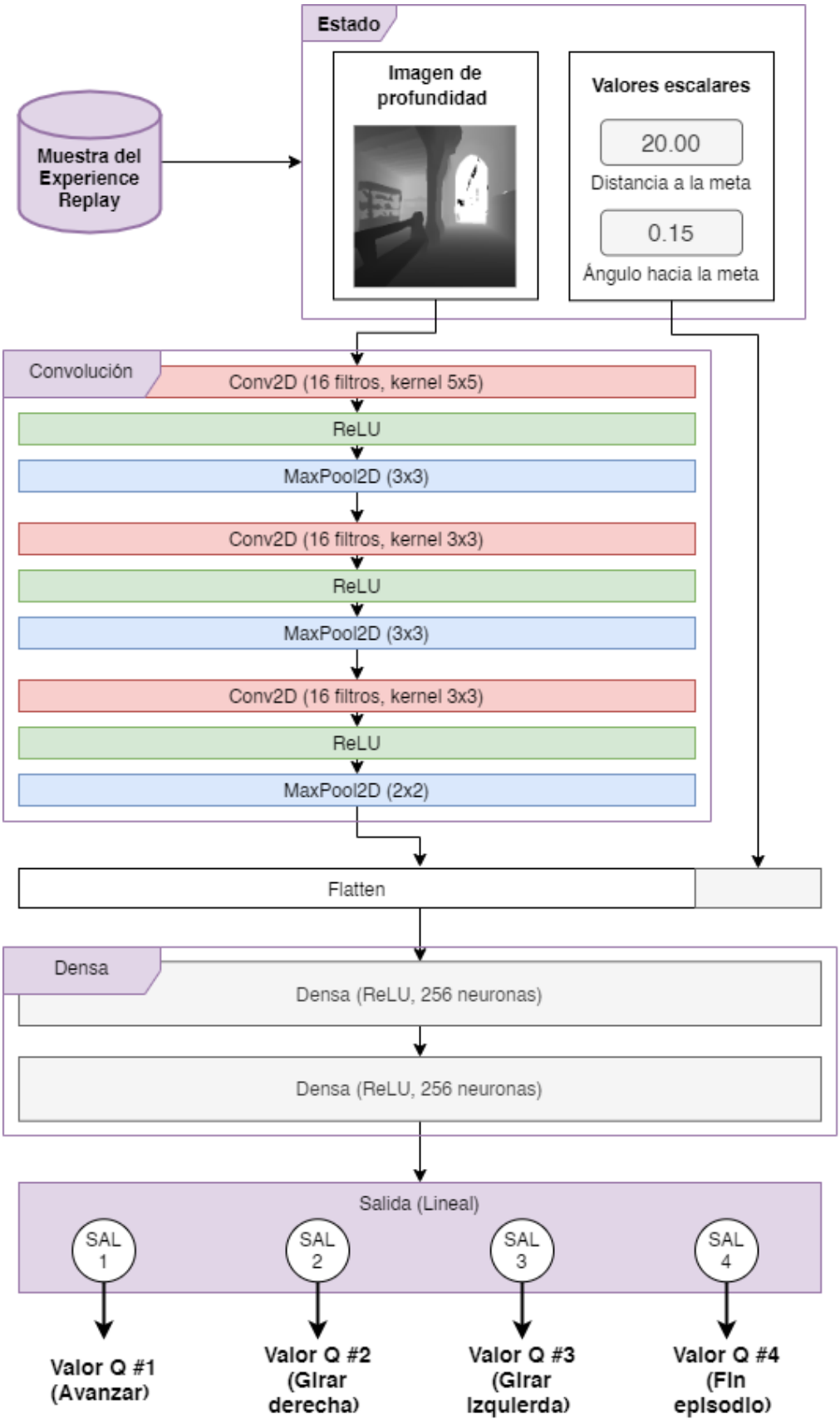

Figura 5.5: Arquitectura de la red neuronal - Propuesta 1 (CNN).

conserva en los ficheros *models/DEPRECATED_reactive_navigation_keras.py* y *trainers/DEPRECATED_reactive_navigation_trainer_keras.py*, por motivos de documentación

5.2.2. Propuesta 2: Red mixta (CNN + MLP)

La segunda aproximación está basada en el uso de **redes híbridas** (es decir, redes formadas por varias redes neuronales más pequeñas).

Una red convolucional no está preparada para trabajar con valores escalares numéricos. Ahora bien, un perceptrón multicapa (*MLP*) estándar no puede procesar imágenes con el mismo rendimiento que una red convolucional. Por tanto, una de las mejores opciones para trabajar con entradas mixtas de imágenes y valores escalares es procesar cada tipo de entrada en una red neuronal separada específica (las imágenes en redes convolucionales y los valores escalares en redes densas), obteniendo resultados que serán juntados y procesados posteriormente en una red neuronal final.

La arquitectura de esta propuesta está inspirada por las arquitecturas propuestas para resolver otros problemas con entrada mixta de imágenes y valores numéricos, como el trabajo de Md Manjurul Ahsan *et al.* para distinguir casos de COVID-19 [64] o el trabajo de Yanyu Zhang aplicando redes mixtas al aprendizaje por refuerzo profundo [65]. Se ha desarrollado la arquitectura de forma *ad-hoc* para las necesidades del proyecto.

La propuesta consta con dos redes neuronales (una red convolucional y un perceptrón multicapa) que procesan sus entradas de forma paralela. Tras esto, las salidas de ambas redes se pasan a una red final que obtiene el resultado. La arquitectura de las tres redes son las siguientes:

- **Red convolucional (CNN):** Esta red se encarga del procesamiento de la imagen de profundidad de la entrada, extrayendo las características profundas de ésta para ser aprovechadas posteriormente. Consta de las siguientes capas:

  1. **Entrada:** La entrada es una imagen de profundidad (escala de grises) en forma de matriz bidimensional de tamaño $256x256$, con una única capa. Esta imagen se obtiene de los estados muestreados del *Experience Replay*.

  2. **Convolución:** El primer paso de la red es procesar la imagen para extraer las características principales, reduciendo su tamaño. Para esto se usan **tres** procesos de convolución consecutivos, cada uno de ellos formado por, en orden:

     - Capa convolucional bidimensional. El número de filtros depende del proceso de convolución, siendo éste de 16, 32 y 16 filtros respectivamente. El tamaño del $kernel$ también varía dependiendo del proceso, siendo éste de $5x5$, $3x3$ y $3x3$ en cada proceso respectivamente.

     - Función de activación ReLU.

     - Capa de *pooling* con función de máximo. El tamaño de *pooling* es de $3x3$, $3x3$ y $2x2$ respectivamente dependiendo del proceso de convolución.

3. **Aplanado (*Flatten*):** Tras la convolución de la imagen, es necesario aplanar el resultado (la matriz tridimensional con las características identificadas) a un conjunto unidimensional de neuronas. Este aplanamiento permite a las capas densas posteriores trabajar de forma correcta con la información. Esta capa no cuenta con ninguna función de activación.

4. **Capa densa:** En este caso, tras el aplanado hay una única capa densa (capa de neuronas totalmente conectadas) de **64** neuronas usando **ReLU** como función de activación. Con esta capa se busca identificar relaciones existentes entre las características encontradas previamente con la convolución.

5. **Salida:** La capa de salida es una única capa densa con tres neuronas, con función de activación lineal. Se ha optado por utilizar una función de activación lineal para no modificar el valor de salida alcanzado de ninguna manera, buscando evitar sesgos.

   Estas neuronas posteriormente servirán como entrada para otra red neuronal.

- **Red neuronal profunda / Perceptrón multicapa (MLP):** Esta red se encarga del procesamiento de los valores numéricos de la entrada, preparándolos para su uso posterior. Consta de las siguientes capas:

  1. **Entrada:** La entrada son dos valores numéricos (la distancia y el ángulo hacia la meta), obtenidos de los estados muestreados del *Experience Replay*.

  2. **Capas ocultas:** Tras la entrada, la red cuenta con **dos** capas densas (neuronas totalmente conectadas) de **diez** neuronas cada una, con función de activación ReLU. Estas capas de neuronas se encargan de procesar las entradas numéricas.

     El número de neuronas se ha obtenido a partir del número de entradas y salidas, siendo $2(entrada + salida) = 10$.

  3. **Salida:** La capa de salida es una única capa densa con tres neuronas, con función de activación lineal. De nuevo, se utiliza una función de activación lineal para que el valor de las neuronas de salida no se vea modificado de ninguna forma.

     Estas neuronas serán usadas posteriormente como entrada para otra red neuronal.

- **Red mixta (CNN + MLP):** Esta red toma las salidas de las dos redes anteriores, juntándolas y procesándolas para obtener las salidas finales de la red neuronal. La arquitectura de esta red es una red neuronal estándar, con las siguientes capas:

  1. **Concatenación / Entrada:** La entrada de la red es una concatenación de las salidas de las dos redes neuronales anteriores: **3** neuronas de la imagen procesada y **3** neuronas de los valores numéricos procesados, para un total de **6** neuronas. Esta capa no tiene ninguna función de activación.

     Se ha decidido que ambas redes neuronales tengan el mismo número de neuronas en la salida para que ambas partes del estado (imagen y valores numéricos) tuviesen la misma relevancia a la hora de obtener una salida.

Además, se ha elegido usar **tres** neuronas en cada salida por tener un número pequeño de neuronas, pero suficientemente grande para que haya una expresión rica de características.

2. **Capas ocultas:** Tras la entrada, la red cuenta con **dos** capas densas de **32** neuronas, con funciones de activación ReLU.

   Se ha optado por utilizar dos capas con un número moderado de neuronas frente a una capa con una gran cantidad de neuronas por ofrecer mejor rendimiento, dando la posibilidad con más capas de encontrar más relaciones entre datos.

3. **Salida:** La capa de salida es una capa densa de cuatro neuronas, donde cada neurona se corresponde con el valor Q de una de las cuatro acciones disponibles para el agente. Esta capa utiliza una función de activación lineal.

Se puede observar un esquema de la arquitectura en la Figura 5.6.

A diferencia de la propuesta anterior, la red neuronal ha sido implementada utilizando la librería *PyTorch*, estando disponible la implementación en el fichero *models/reactive_navigation.py*. Para las funciones usadas por la red durante su entrenamiento, se han usado algoritmos más novedosos que los de la propuesta anterior, siendo estos:

- **Función de inicialización de pesos:** Kaiming [66].
- **Función de optimización:** Adam [63].
- **Función de error:** Error de Huber [67].

  Esta función es una variante del error cuadrático medio propuesta por Huber en 1964, siendo más resistente a los valores aislados. Concretamente, la función es cuadrática para valores residuales pequeños, mientras que se vuelve lineal para valores elevados.

Esta propuesta ha sido elegida como la arquitectura definitiva usada por el agente, al ofrecer mejores resultados en un tiempo de entrenamiento menor, sin ningún error ni problema durante la ejecución.

5.3. Actuación del agente

El proceso de actuación del agente está basado en el proceso de actuación estándar de un agente de *Deep Q-Learning* siguiendo el paradigma de exploración / explotación, pudiendo ser observado en la Figura 5.7.

Como ya se ha mencionado, la actuación del agente sigue el paradigma de exploración-explotación, usando ϵ como la variable que regula el proceso. ϵ se actualiza durante el entrenamiento tras cada episodio siguiendo la siguiente fórmula:

$$\epsilon = max\left(\frac{(\epsilon_{init} - \epsilon_{min}) * porcentaje}{\epsilon_{min\_porcentaje}} - \epsilon_{init}, \epsilon_{min}\right)$$

Donde ϵ_{init} es el valor inicial de ϵ (por defecto 1.0), ϵ_{min} es el valor mínimo alcanzado por ϵ (por defecto 0.05), $\epsilon_{min\_porcentaje}$ es el porcentaje de episodios tras el cual el

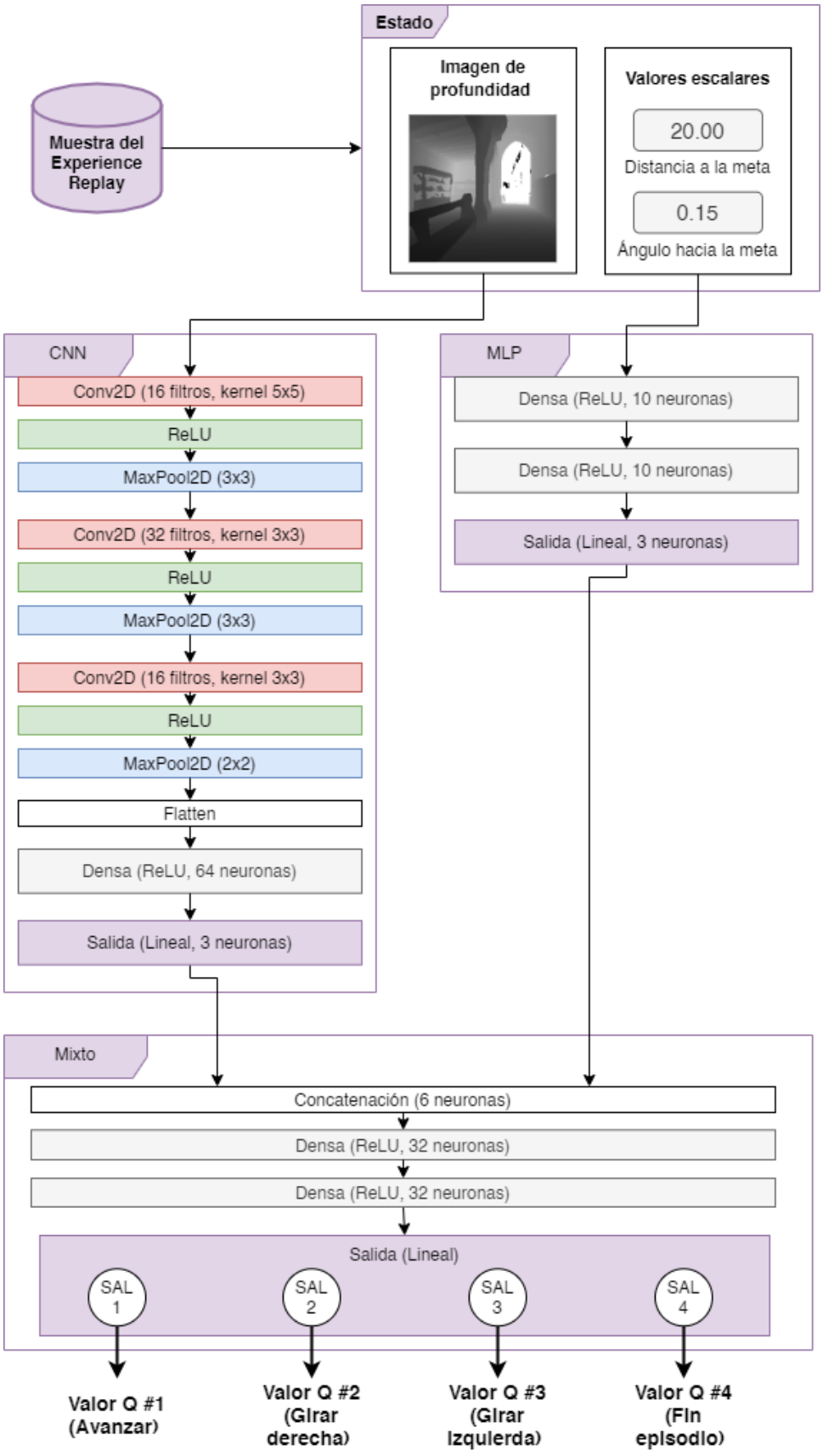

Figura 5.6: Arquitectura de la red neuronal - Propuesta 2 (Mixta).

Algoritmo 8: Actuación del agente

Entradas: Estado s, formado por una imagen de profundidad $depth$ y dos valores numéricos, la distancia a la meta $dist$ y el ángulo hacia la meta $angle$.

Variables internas del agente: Probabilidad de acción aleatoria ϵ, lista de acciones posibles $action\_list$. El agente conserva una variable ϵ para el ratio de exploración / explotación.

1. Genera un valor aleatorio $rand$ en el rango $[0.0, 1.0]$.

2. Si $rand < \epsilon$:

  2.1 Exploración. Se elige una acción $action$ de $action\_list$ aleatoriamente, siguiendo una distribución uniforme.

3. En otro caso ($rand \geq \epsilon$):

  3.1. Explotación. Procesa el estado s a través de la red neuronal, obteniendo la lista de valores Q para cada par estado-acción q_values.

  3.2. Elige la acción $action$ de $action\_list$ que tenga el máximo valor en q_values.

4. Devuelve $action$.

Figura 5.7: Proceso de actuación del agente.

valor de ϵ alcanzará ϵ_{min} (por defecto 0.8, el $80\,\%$ de los episodios) y $porcentaje$ es el porcentaje actual de episodios completados.

En esencia, el valor de ϵ decrece linealmente desde su valor inicial, ϵ_{init}, hasta su valor final, ϵ_{min}, conforme el agente completa episodios. El valor final será alcanzado tras completar el $\epsilon_{min\_porcentaje}\,\%$ de los episodios (por ejemplo, para un entrenamiento de 10.000 episodios ϵ_{min} se alcanzaría a los 8000 episodios). El valor de ϵ no puede bajar de ϵ_{min} en ningún momento.

El objetivo es que en los primeros episodios el agente explore una gran cantidad de estados (**exploración**) mientras no tiene suficiente conocimiento como para tener una política de acciones de calidad. Conforme el agente completa episodios y adquiere conocimiento, se busca que éste empiece a aprovechar las experiencias previas en los episodios finales (**explotación**), efectuando menos acciones aleatorias y más acciones acordes a la política aprendida.

Cuando el agente es usado fuera de un entorno de entrenamiento (como puede ser durante su evaluación), el valor de ϵ se queda fijado como $\epsilon = 0$, para explotar totalmente la política sin ninguna acción aleatoria.

5.4. Entrenamiento del agente

En esta sección se describe el proceso seguido por el agente para realizar su entrenamiento. El agente ha sido entrenado utilizando el algoritmo de *Deep Q-Learning*, siendo el pseudocódigo del algoritmo el expuesto en la Figura 5.8.

Se ha optado por utilizar *Deep Q-Learning* frente a otras técnicas de aprendizaje por refuerzo profundo como *PPO* principalmente por familiaridad con la técnica. Además, *Deep Q-Learning* sigue ofreciendo buenos resultados en problemas de aprendizaje por refuerzo profundo, especialmente si se aplican mejoras como *Prioritized Experience Replay*.

Algoritmo 9: Entrenamiento del agente

Variables iniciales: Dos agentes, un agente conteniendo la red Q, $agente\_q$, y un agente conteniendo la red objetivo, $agente\_obj$. *Experience Replay* exp_replay donde se almacenan las experiencias del agente. Número total de episodios a realizar durante el entrenamiento, ep_total. ϵ, probabilidad de realizar una acción aleatoria.

1. Inicializa un contador, $cont\_ep = 0$, para almacenar el numero de episodios realizados hasta el momento.

2. Mientras $cont\_ep < ep\_total$:

- **2.1** Inicializa el episodio $episodio$, obteniendo el estado inicial $estado$.
- **2.2** Mientras $episodio$ no haya finalizado:
  - **2.2.1.** $agente\_q$ elige la acción $accion$ a realizar para $estado$ dependiendo de ϵ, usando el método descrito previamente.
  - **2.2.2.** Se aplica $accion$ a $estado$, obteniendo una recompensa $recompensa$, un nuevo estado n_estado y un indicador de si n_estado es final, $final$.
  - **2.2.3.** Se almacena $estado$, $accion$, $recompensa$, n_estado y $final$ en exp_replay.
  - **2.2.4.** Se toma una muestra $batch$ de exp_replay, y se entrena al agente $agente\_q$ a partir de $batch$, usando los resultados de $agente\_q$ y $agente\_obj$.
  - **2.2.5.** $estado = n\_estado$.
- **2.3.** Actualiza $agente\_obj$ con los pesos de $agente\_q$.
- **2.4.** Actualiza ϵ.
- **2.5.** $cont\_ep++$.
- **2.6.** Documenta el proceso de entrenamiento.

3. Devuelve los pesos de $agente\_q$ como agente entrenado.

Figura 5.8: Proceso de entrenamiento del agente.

Se han planteado dos variantes para el entrenamiento, dependiendo de la técnica utilizada:

- Entrenamiento usando *Deep Q-Learning* estándar.
- Entrenamiento usando *Deep Q-Learning* con *Prioritized Experience Replay*.

A continuación, se describen los elementos principales del entrenamiento.

5.4.1. *Replay Memory* y memorización de la experiencia

El *Replay Memory* del agente se encarga de almacenar las experiencias previas del agente, para su muestreo posterior durante el entrenamiento. Estas experiencias son almacenadas con la forma $< s, a, r, s', f >$, siendo cada elemento:

- s: Estado inicial de la experiencia.
- a: Acción aplicada sobre el estado s.
- r: Recompensa obtenida tras aplicar la acción a al estado s.
- s': Estado nuevo, alcanzado tras aplicar la acción a al estado s.
- f: Valor booleano que indica si s' es un estado final (ha provocado el final del

episodio) o no.

Internamente, el *Replay Memory* es una cola FIFO estándar de tamaño M (por defecto, 20000 posiciones) donde se introducen las experiencias. Cuando la cola se llena, la introducción de una nueva experiencia provocará que la experiencia más antigua sea eliminada. De esta forma, se evita que el conocimiento del agente se estanque al ir renovando las experiencias conforme se van experimentando nuevas experiencias.

Tras cada actuación del agente, se introduce la experiencia (los valores descritos anteriormente) en la memoria y se realiza un proceso de aprendizaje como se verá posteriormente.

La variante usando *Prioritized Experience Replay* presenta las siguientes diferencias:

- El *Replay Memory* pasa de almacenar directamente la experiencia $< s, a, r, s', f >$ a una tupla $(experiencia, error)$. En esta tupla, $experiencia$ sigue teniendo la estructura $< s, a, r, s', f >$, pero $error$ simboliza el error que presenta la experiencia (siendo éste la diferencia entre los valores Q que se espera que devuelva la red para s y los valores Q realmente obtenidos).
- Cuando se inserta una experiencia en el *Replay Memory*, se inserta inicialmente como $(experiencia, \infty)$ (es decir, un valor de error infinito). Esto se debe a que inicialmente no se conoce el error, por lo que se busca el error más alto posible para fomentar que el agente aprenda la experiencia.

5.4.2. Aprendizaje a partir de las experiencias

Tras la memorización de una experiencia, se realiza aprendizaje a partir de una muestra tomada del *Experience Replay*, viéndose el proceso general en la Figura 5.9.

Hay algunos detalles importantes que remarcar sobre el proceso:

- Por defecto, el tamaño de la muestra es de 64 experiencias. Se ha optado por no entrenar al agente hasta que el *Replay Memory* contenga al menos 64 experiencias, como en la propuesta original de *Deep Q Learning*, para evitar problemas con la subdivisión de las muestras (como se verá a continuación).
- El aprendizaje se realiza en *batch* (en paralelo). Esto significa que todas las muestras son pasadas a través de la red neuronal de forma simultanea, aprovechando el paralelismo ofrecido por las GPUs y mejorando el rendimiento.
- Para mejorar el uso en memoria, cada muestra se divide en submuestras de menor tamaño (por defecto, muestras de 64 experiencias se dividen en submuestras de 32 experiencias), que son pasadas consecutivamente por la red neuronal.

  Esto se ha hecho para suplir los problemas de memoria que surgieron durante el entrenamiento (al no haber suficiente memoria para procesar todos los valores a través de la red neuronal al mismo tiempo), haciendo más eficiente el uso de memoria a costa del tiempo de entrenamiento. Aun así, el rendimiento es notablemente superior al de un entrenamiento no paralelo.

La variante usando *Prioritized Experience Replay* presenta las siguientes diferencias respecto a la Figura 5.9:

Algoritmo 10: Aprendizaje a partir de la experiencia (estándar)

Variables iniciales: *Experience Replay* exp_replay. Dos agentes, el agente con la red neuronal Q $agente\_q$ y el agente con la red neuronal objetivo $agente\_obj$. γ, el peso dado a las nuevas experiencias en *Deep Q-Learning*.

1. Obtén una muestra $muestra$ de exp_replay, de tamaño M. Si $tamao(exp\_replay) < M$ **finaliza el proceso sin aprender.**

2. Obtén los valores Q Q_s para los estados actuales contenidos en M usando al agente Q $agente\_q$.

3. Obtén los valores Q Q_s' para los estados alcanzados contenidos en M usando al agente objetivo $agente\_obj$.

4. Para cada experiencia exp de la muestra M, donde exp_a es la acción tomada en exp, exp_r es la recompensa obtenida en exp, exp_f es la indicación de si exp fue final y exp_q y exp_q' son los valores Q para el estado actual y alcanzado de exp (calculados previamente en Q_s y Q_s' respectivamente):

4.1. Actualiza el valor Q asociado a la acción exp_a en exp_s siguiendo la siguiente fórmula:

$$exp\_q(exp\_a) = \begin{cases} exp\_r, & \text{si } exp\_f \text{ (si la experiencia es final)} \\ exp\_r + \gamma * max(exp\_q'), & \text{en cualquier otro caso} \end{cases}$$

5. Actualiza las predicciones de $agente\_q$ para los estados actuales de M usando las nuevas predicciones exp_q (usando retropropagación).

Figura 5.9: Proceso de aprendizaje a partir de las experiencias (estándar).

- El muestreo de experiencias no sigue una distribución uniforme, sino que sigue la siguiente distribución, siendo la probabilidad de elegir una experiencia i:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

  Donde p_i es la prioridad de la experiencia i (siendo $p_i = \frac{1}{rango(i)}$, donde $rango(i)$ es la posición de la experiencia i en el *Replay Memory* si éste se ordena de mayor a menor error) y α es una constante que indica el el grado de priorización (por defecto 0.5).

  Esto significa que las experiencias con mayor error tienen más probabilidad de ser muestreadas.

- Los valores Q actualizados son normalizados con un peso w_i, siendo el peso de la experiencia i:

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta$$

  Donde N es el número total de experiencias en el *Replay Memory* y β es un factor para ajustar la influencia del peso (por defecto 0.5).

  Por tanto, la actualización de valores Q en el punto **4.1** pasa a ser:

$$exp\_q(exp\_a) = \begin{cases} exp\_r * w_i, & \text{si } exp\_f \text{ (si la experiencia es final)} \\ (exp\_r + \gamma * max(exp\_q')) * w_i, & \text{en cualquier otro caso} \end{cases}$$

- Tras la actualización de los pesos de la red neuronal Q, los errores de las experiencias muestreadas del *Replay Memory* son actualizados. Estos errores son calculados usando el error cuadrático medio, siendo la fórmula:

$$error = (q\_esperado - q\_obtenido)^2$$

5.4.3. Documentación del entrenamiento

Al final de cada episodio, el agente documenta el progreso durante el entrenamiento, imprimiendo en un fichero las siguientes métricas:

- ID del episodio.
- Duración del episodio (en segundos).
- Número de acciones realizadas durante el episodio.
- Distancia inicial y final hasta la meta.
- Distancia recorrida hasta la meta ($dist\_inicial - dist\_final$). Este valor puede ser negativo si la posición final del agente es más lejana que la inicial.
- Exitoso (**Verdadero** si el agente ha alcanzado la meta, **Falso** en cualquier otro caso).
- Recompensa media obtenida.

A partir de estas métricas se realizará un análisis del rendimiento del agente durante el entrenamiento en el Capítulo 6.

Además, el agente almacena registros de su progreso durante el entrenamiento (*checkpoints*), almacenando un total de **100** *checkpoints* a lo largo de todo el entrenamiento (aproximadamente uno cada 150 episodios). Estos *checkpoints* (ficheros de formato *.pt*) contienen la siguiente información:

- Los pesos de la red neuronal objetivo.
- El fichero de configuración que se estaba utilizando durante el entrenamiento.

A partir de estos *checkpoints* es posible reanudar el entrenamiento en cualquier momento, y usarlos como los pesos finales para el agente entrenado.

Capítulo 6

Experimentación

[PONEMOS EL ANALISIS EN UN CAPITULO APARTE?]

6.1. Experimentos realizados y parametros utilizados

6.1.1. Parametros utilizados

[EXPERIMENTOS A REALIZAR, PARAMETROS A USAR, ORDENADOR USADO, ETC]
[PARA REPRODUCIBILIDAD VAMOS]

6.1.2. Experimentos realizados

6.1.3. Elección del conjunto de datos

6.2. Resultados obtenidos

6.3. Comparativa y análisis de resultados

6.3.1. Comparativa durante el entrenamiento

6.3.2. Comparativa de los agentes entrenados

[TASA DE EXITO, CUAL ES MEJOR, ETC]

Capítulo 7

Conclusiones

7.1. Conclusiones

7.2. Trabajo futuro

Bibliografía

[1] C. Sampedro, H. Bavle, A. Rodriguez-Ramos, P. De La Puente y P. Campoy, "Laser-Based Reactive Navigation for Multirotor Aerial Robots using Deep Reinforcement Learning," *IEEE International Conference on Intelligent Robots and Systems*, págs. 1024-1031, 2018, ISSN: 21530866. DOI: 10.1109/IROS.2018.8593706.

[2] M. Savva, A. Kadian, O. Maksymets, Y. Zhao, E. Wijmans, B. Jain, J. Straub, J. Liu, V. Koltun, J. Malik, D. Parikh y D. Batra, "Habitat: A Platform for Embodied AI Research," en *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

[3] A. Szot, A. Clegg, E. Undersander, E. Wijmans, Y. Zhao, J. Turner, N. Maestre, M. Mukadam, D. Chaplot, O. Maksymets, A. Gokaslan, V. Vondrus, S. Dharur, F. Meier, W. Galuba, A. Chang, Z. Kira, V. Koltun, J. Malik, M. Savva y D. Batra, "Habitat 2.0: Training Home Assistants to Rearrange their Habitat," *arXiv preprint arXiv:2106.14405*, 2021.

[4] L. Jimenez, "Aplicación de Deep Reinforcement Learning a un juego real - Tetris," *Universidad de Castilla-La Mancha*, pág. 109, 2020. dirección: https://github.com/MoonDollLuna/dqlearning-tetris.

[5] P. Anderson, A. X. Chang, D. S. Chaplot, A. Dosovitskiy, S. Gupta, V. Koltun, J. Kosecka, J. Malik, R. Mottaghi, M. Savva y A. R. Zamir, "On Evaluation of Embodied Navigation Agents," *CoRR*, vol. abs/1807.06757, 2018. arXiv: 1807.06757. dirección: http://arxiv.org/abs/1807.06757.

[6] Abhishek Kadian*, Joanne Truong*, A. Gokaslan, A. Clegg, E. Wijmans, S. Lee, M. Savva, S. Chernova y D. Batra, "Sim2Real Predictivity: Does Evaluation in Simulation Predict Real-World Performance?," 4, vol. 5, 2020, págs. 6670-6677.

[7] D. Batra, A. Gokaslan, A. Kembhavi, O. Maksymets, R. Mottaghi, M. Savva, A. Toshev y E. Wijmans, "ObjectNav Revisited: On Evaluation of Embodied Agents Navigating to Objects," en *arXiv:2006.13171*, 2020.

[8] D. S. Chaplot, D. Gandhi, S. Gupta, A. Gupta y R. Salakhutdinov, "Learning to Explore using Active Neural SLAM," *CoRR*, vol. abs/2004.05155, 2020. arXiv: 2004.05155. dirección: https://arxiv.org/abs/2004.05155.

[9] S. K. Ramakrishnan, Z. Al-Halah y K. Grauman, "Occupancy Anticipation for Efficient Exploration and Navigation," *CoRR*, vol. abs/2008.09285, 2020. arXiv: 2008.09285. dirección: https://arxiv.org/abs/2008.09285.

[10] S. Datta, O. Maksymets, J. Hoffman, S. Lee, D. Batra y D. Parikh, "Integrating Egocentric Localization for More Realistic Point-Goal Navigation Agents," *CoRR*, vol. abs/2009.03231, 2020. arXiv: 2009.03231. dirección: https://arxiv.org/abs/2009.03231.

[11] R. Partsey, "Robust Visual Odometry for Realistic PointGoal Navigation," *Ukranian Catholic University*, pág. 87, 2021. dirección: https://er.ucu.edu.ua/handle/1/2703.

[12] S. Russell y P. Norvig, *Artificial Intelligence: A Modern Approach*, 3.ª ed. Prentice Hall, 2010.

[13] I. Goodfellow, Y. Bengio y A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[14] N. Buduma y N. Locascio, *Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms*, 1st. O'Reilly Media, Inc., 2017, ISBN: 1491925612.

[15] Q. jun Zhang, K. Gupta y V. Devabhaktuni, "Artificial neural networks for RF and microwave design - from theory to practice," *IEEE Transactions on Microwave Theory and Techniques*, vol. 51, págs. 1339-1350, 2003.

[16] D. E. Rumelhart, G. E. Hinton y R. J. Williams, "Learning Representations by Back-propagating Errors," *Nature*, vol. 323, n.º 6088, págs. 533-536, 1986. DOI: 10.1038/323533a0. dirección: http://www.nature.com/articles/323533a0.

[17] C. Shorten y T. Khoshgoftaar, "A survey on Image Data Augmentation for Deep Learning," *Journal of Big Data*, vol. 6, págs. 1-48, 2019.

[18] J. F. Kolen y S. C. Kremer, "Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies," en *A Field Guide to Dynamical Recurrent Networks*. 2001, págs. 237-243. DOI: 10.1109/9780470544037.ch14.

[19] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard y L. D. Jackel, "Backpropagation Applied to Handwritten Zip Code Recognition," *Neural Comput.*, vol. 1, n.º 4, 541–551, dic. de 1989, ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541. dirección: https://doi.org/10.1162/neco.1989.1.4.541.

[20] A Krizhevsky, I. Sutskever y G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, págs. 84-90, 2012.

[21] Y. Zhou y R. Chellappa, "Computation of optical flow using a neural network," *IEEE 1988 International Conference on Neural Networks*, 71-78 vol.2, 1988.

[22] C. S. Wiki, *Max-pooling / Pooling*, 2018. dirección: https://computersciencewiki.org/index.php/Max-pooling_/_Pooling.

[23] R. S. Sutton y A. G. Barto, *Reinforcement Learning: An Introduction*, Second Edition. The MIT Press, 2018. dirección: http://incompleteideas.net/book/the-book-2nd.html.

[24] C. J. C. H. Watkins, "Learning from Delayed Rewards," Tesis doct., King's College, Oxford, 1989.

[25] G. A. Rummery y M. Niranjan, "On-Line Q-Learning Using Connectionist Systems," Cambridge University Engineering Department, Cambridge, England, inf. téc. TR 166, 1994.

[26] I. Witten, "An Adaptive Optimal Controller for Discrete-Time Markov Environments," *Inf. Control.*, vol. 34, págs. 286-295, 1977.

[27] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare y J. Pineau, "An Introduction to Deep Reinforcement Learning," *CoRR*, vol. abs/1811.12560, 2018. arXiv: 1811.12560. dirección: http://arxiv.org/abs/1811.12560.

[28] G. J. Gordon, "Stable Fitted Reinforcement Learning," en *Advances in Neural Information Processing Systems*, D. Touretzky, M. C. Mozer y M. Hasselmo,

eds., vol. 8, MIT Press, 1996. dirección: https://proceedings.neurips.cc/paper/1995/file/fd06b8ea02fe5b1c2496fe1700e9d16c-Paper.pdf.

[29] M. Riedmiller, "Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method," en *Machine Learning: ECML 2005*, J. Gama, R. Camacho, P. B. Brazdil, A. M. Jorge y L. Torgo, eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, págs. 317-328, ISBN: 978-3-540-31692-3.

[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg y D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, págs. 529-533, 2015.

[31] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar y D. Silver, "Rainbow: Combining Improvements in Deep Reinforcement Learning," *CoRR*, vol. abs/1710.02298, 2017. arXiv: 1710.02298. dirección: http://arxiv.org/abs/1710.02298.

[32] H. Van Hasselt, "Double Q-learning.," ene. de 2010, págs. 2613-2621.

[33] H. van Hasselt, A. Guez y D. Silver, "Deep Reinforcement Learning with Double Q-learning," *CoRR*, vol. abs/1509.06461, 2015. arXiv: 1509.06461. dirección: http://arxiv.org/abs/1509.06461.

[34] T. Schaul, J. Quan, I. Antonoglou y D. Silver, *Prioritized Experience Replay*, cite arxiv:1511.05952Comment: Published at ICLR 2016, 2015. dirección: http://arxiv.org/abs/1511.05952.

[35] Z. Wang, N. de Freitas y M. Lanctot, "Dueling Network Architectures for Deep Reinforcement Learning," *CoRR*, vol. abs/1511.06581, 2015. arXiv: 1511.06581. dirección: http://arxiv.org/abs/1511.06581.

[36] R. S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Mach. Learn.*, vol. 3, n.º 1, 9–44, ago. de 1988, ISSN: 0885-6125. DOI: 10.1023/A:1022633531479. dirección: https://doi.org/10.1023/A:1022633531479.

[37] M. G. Bellemare, W. Dabney y R. Munos, "A Distributional Perspective on Reinforcement Learning," *CoRR*, vol. abs/1707.06887, 2017. arXiv: 1707.06887. dirección: http://arxiv.org/abs/1707.06887.

[38] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell y S. Legg, "Noisy Networks for Exploration," *CoRR*, vol. abs/1706.10295, 2017. arXiv: 1706.10295. dirección: http://arxiv.org/abs/1706.10295.

[39] J. Peters y S. Schaal, "Reinforcement learning of motor skills with policy gradients," *Neural Networks*, vol. 21, n.º 4, págs. 682-697, 2008, Robotics and Neuroscience, ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2008.02.003. dirección: https://www.sciencedirect.com/science/article/pii/S0893608008000701.

[40] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver y K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," *CoRR*, vol. abs/1602.01783, 2016. arXiv: 1602.01783. dirección: http://arxiv.org/abs/1602.01783.

[41] J. Vitay, 2020. dirección: https://julien-vitay.net/deeprl/DeepRL.html.

[42] J. Schulman, S. Levine, P. Abbeel, M. Jordan y P. Moritz, "Trust Region Policy Optimization," en *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach y D. Blei, eds., ép. Proceedings of Machine Learning Re-

search, vol. 37, Lille, France: PMLR, 2015, págs. 1889-1897. dirección: https://proceedings.mlr.press/v37/schulman15.html.

[43] J. Schulman, F. Wolski, P. Dhariwal, A. Radford y O. Klimov, "Proximal Policy Optimization Algorithms," *CoRR*, vol. abs/1707.06347, 2017. arXiv: 1707.06347. dirección: http://arxiv.org/abs/1707.06347.

[44] P. Corke, *Robotics, Vision and Control: Fundamental Algorithms In MATLAB, Second Edition*, 2nd. Springer Publishing Company, Incorporated, 2017, ISBN: 3319544128.

[45] C. Stachniss, *Robotic Mapping and Exploration*. ene. de 2009, vol. 55, ISBN: 978-3-642-01096-5. DOI: 10.1007/978-3-642-01097-2.

[46] V. J. Lumelsky y A. A. Stepanov, "Path-Planning Strategies for a Point Mobile Automaton Moving Amidst Unknown Obstacles of Arbitrary Shape.," *Algorithmica*, vol. 2, págs. 403-430, 1987. dirección: http://dblp.uni-trier.de/db/journals/algorithmica/algorithmica2.html#LumelskyS87.

[47] S. M. Lavalle, *Planning Algorithms*. Cambridge University Press, 2006, ISBN: 0521862051.

[48] S. Quinlan y O. Khatib, "Elastic bands: connecting path planning and control," en *[1993] Proceedings IEEE International Conference on Robotics and Automation*, 1993, 802-807 vol.2. DOI: 10.1109/ROBOT.1993.291936.

[49] Y. Koren y J. Borenstein, "Potential field methods and their inherent limitations for mobile robot navigation," en *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, 1991, 1398-1404 vol.2. DOI: 10.1109/ROBOT.1991.131810.

[50] J. Andrews, *Impedance Control as a Framework for Implementing Obstacle Avoidance in a Manipulator*. Massachusetts Institute of Technology, Department of Mechanical Engineering, 1983. dirección: https://books.google.es/books?id=OcmcNwAACAAJ.

[51] R. C. Arkin, "Motor Schema — Based Mobile Robot Navigation," *The International Journal of Robotics Research*, vol. 8, n.º 4, págs. 92-112, 1989. DOI: 10.1177/027836498900800406. dirección: https://doi.org/10.1177/027836498900800406.

[52] L. Tai, G. Paolo y M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, págs. 31-36, 2017.

[53] X. Chang, P. Ren, P. Xu, Z. Li, X. Chen y A. Hauptmann, "Scene Graphs: A Survey of Generations and Applications," *CoRR*, vol. abs/2104.01111, 2021. arXiv: 2104.01111. dirección: https://arxiv.org/abs/2104.01111.

[54] F. Xia, A. R. Zamir, Z.-Y. He, A. Sax, J. Malik y S. Savarese, "Gibson Env: real-world perception for embodied agents," en *Computer Vision and Pattern Recognition (CVPR), 2018 IEEE Conference on*, IEEE, 2018.

[55] M. Savva, A. X. Chang, A. Dosovitskiy, T. A. Funkhouser y V. Koltun, "MINOS: Multimodal Indoor Simulator for Navigation in Complex Environments," *CoRR*, vol. abs/1712.03931, 2017. arXiv: 1712.03931. dirección: http://arxiv.org/abs/1712.03931.

[56] J. Straub, T. Whelan, L. Ma, Y. Chen, E. Wijmans, S. Green, J. J. Engel, R. Mur-Artal, C. Ren, S. Verma, A. Clarkson, M. Yan, B. Budge, Y. Yan, X. Pan, J. Yon, Y. Zou, K. Leon, N. Carter, J. Briales, T. Gillingham, E. Mueggler, L. Pesqueira, M. Savva, D. Batra, H. M. Strasdat, R. D. Nardi, M. Goesele, S. Lovegrove y R. A.

Newcombe, "The Replica Dataset: A Digital Replica of Indoor Spaces," *CoRR*, vol. abs/1906.05797, 2019. arXiv: 1906.05797. dirección: http://arxiv.org/abs/1906.05797.

[57] E. Wijmans, S. Datta, O. Maksymets, A. Das, G. Gkioxari, S. Lee, I. Essa, D. Parikh y D. Batra, "Embodied Question Answering in Photorealistic Environments with Point Cloud Perception," en *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[58] A. Chang, A. Dai, T. Funkhouser, M. Halber, M. Niessner, M. Savva, S. Song, A. Zeng e Y. Zhang, "Matterport3D: Learning from RGB-D Data in Indoor Environments," *International Conference on 3D Vision (3DV)*, 2017.

[59] D. Batra, A. Chang, A. Clegg, A. Gokaslan, W. Goluba, O. Maksymets, M. Savva, S. Kumar, J. Turner, E. Undersander y et al., *Habitat Matterport Dataset*, 2021. dirección: https://aihabitat.org/datasets/hm3d/0002.html.

[60] C. Evans, O. Ben-Kiki e I. Döt Net, *The Official YAML Web Site*, 2001. dirección: https://yaml.org/.

[61] T. Salimans y D. P. Kingma, "Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks," *Advances in Neural Information Processing Systems*, págs. 901-909, 2016. arXiv: 1602.07868. dirección: https://arxiv.org/abs/1602.07868v3.

[62] X. Glorot e Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," en *AISTATS*, 2010.

[63] D. Kingma y J. Ba, "Adam: A Method for Stochastic Optimization," *International Conference on Learning Representations*, dic. de 2014.

[64] M. M. Ahsan, T. E. Alam, T. Trafalis y P. Huebner, "Deep MLP-CNN Model Using Mixed-Data to Distinguish between COVID-19 and Non-COVID-19 Patients," *Symmetry*, vol. 12, n.º 9, 2020, ISSN: 2073-8994. DOI: 10.3390/sym12091526. dirección: https://www.mdpi.com/2073-8994/12/9/1526.

[65] Y. Zhang, "Deep Reinforcement Learning with Mixed Convolutional Network," *CoRR*, vol. abs/2010.00717, 2020. arXiv: 2010.00717. dirección: https://arxiv.org/abs/2010.00717.

[66] K. He, X. Zhang, S. Ren y J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," *CoRR*, vol. abs/1502.01852, 2015. arXiv: 1502.01852. dirección: http://arxiv.org/abs/1502.01852.

[67] P. J. Huber, "Robust Estimation of a Location Parameter," *The Annals of Mathematical Statistics*, vol. 35, n.º 1, págs. 73 -101, 1964. DOI: 10.1214/aoms/1177703732. dirección: https://doi.org/10.1214/aoms/1177703732.

Apéndice A

Manual de usuario

[NO TENGO MUY CLARO COMO SE ESCRIBIAN LOS APENDICES NO TE VOY A ENGAÑAR] [EN LATEX ME REFIERO]

[APENDICES PROBABLES:] [MANUAL DE USUARIO, COMO GENERO GRAFICAS, CONTENIDOS DEL CD]

Apéndice B

Ficheros de configuración

Este anexo contiene los ficheros de configuración desarrollados durante el trabajo. En total, se han desarrollado un total de **12** ficheros de configuración, aunque se pueden dividir en **4** tipos básicos:

- **Fichero base:** Este fichero contiene la configuración básica y sirve de base para el resto de ficheros (siendo importados por éstos)
- **Ficheros de entrenamiento:** Este fichero contiene la configuración necesaria para el entrenamiento de los agentes, incluyendo los parámetros usados durante *Deep Q-Learning*. Hay **8** variantes de este fichero dependiendo de la combinación de variantes a utilizar.
- **Ficheros de *benchmark*:** Este fichero contiene la configuración usada para la evaluación de los agentes. Hay **2** variantes de este fichero, dependiendo del conjunto de datos a usar (*Matterport3D* o *Gibson*).
- **Fichero de generación de video:** Este fichero contiene la configuración usada para generar video del agente durante un episodio. Contiene un bloque con una clave *dummy* de *RL* para poder usar las utilidades de generación de video de *Habitat*.

Los ficheros creados (junto a su documentación en inglés) se muestran a continuación.

B.1. Fichero base

```
1 # REACTIVE NAVIGATION
2 # General evaluation / showcase configuration
3 # Developed by Luna Jimenez Fernandez
4 #
5 # This file contains all general parameters to be used while
    evaluating
6 # or showcasing the trained agents.
7 #
8 # In addition, the contents of this file are also used as a basis
    for the configuration
```

```
# used while training the agents. Training specific parameters can be found in the
# method-specific files (such as reactive_pointnav_train.yaml)
#
# Finally, some of the parameters are specified via arguments during program launch:
#       - Agent type
#       - Dataset and splits to be used
#       - Path to pre-trained weights
#

ENVIRONMENT:
  MAX_EPISODE_STEPS: 1000

SIMULATOR:
  AGENT_0:
    SENSORS: ['RGB_SENSOR', 'DEPTH_SENSOR']
  RGB_SENSOR:
    WIDTH: 256
    HEIGHT: 256
  DEPTH_SENSOR:
    WIDTH: 256
    HEIGHT: 256

  # Physics are explicitely disabled, to avoid segmentation faults
  HABITAT_SIM_V0:
    ENABLE_PHYSICS: False
    # PHYSICS_CONFIG_FILE: "None"


# DATASET: Data path, split and name are specified via argument
DATASET:
  TYPE: PointNav-v1
#  SPLIT:
#  DATA_PATH:
#  NAME:

TASK:
  TYPE: Nav-v0
  SUCCESS_DISTANCE: 0.3
  SENSORS: ['POINTGOAL_WITH_GPS_COMPASS_SENSOR']
  POSSIBLE_ACTIONS: ['STOP', 'MOVE_FORWARD', 'TURN_LEFT', 'TURN_RIGHT']
  POINTGOAL_WITH_GPS_COMPASS_SENSOR:
    GOAL_FORMAT: "POLAR"
    DIMENSIONALITY: 2
  GOAL_SENSOR_UUID: pointgoal_with_gps_compass
  MEASUREMENTS: ['DISTANCE_TO_GOAL', 'SUCCESS', 'SPL', 'SOFT_SPL', 'COLLISIONS']
```

```
54   SUCCESS:
55     SUCCESS_DISTANCE: 0.3
```

B.2. Ficheros de entrenamiento

```
1  # REACTIVE NAVIGATION
2  # Reactive Navigation Agent training configuration
3  # Developed by Luna Jimenez Fernandez
4  #
5  # This file contains all the specific parameters used by the agent
       training, including
6  # both the Deep Q Learning and the Reward computation parameters
7  #
8  # Note that this config is added on top of "base_config.yaml", so
       both files need to be configured
9  # The following arguments can be found in "base_config.yaml":
10 #       - Steps per episode (ENVIRONMENT->MAX_EPISODE_STEPS)
11 #       - Goal radius (TASK->SUCCESS_DISTANCE)
12 #
13 # NOTE: Not all parameters are configured via config, the following
       parameters can be specified
14 # as arguments when launching the script:
15 #       - Agent type
16 #       - Dataset and splits to be used
17 #       - Path to pre-trained weights
18 #
19
20 VERBOSE: False
21
22 BASE_TASK_CONFIG_PATH: "configs/base_config.yaml"
23 TRAINER_NAME: "reactive"
24 ENV_NAME: "ReactiveNavEnv"
25 SIMULATOR_GPU_ID: 0
26 TORCH_GPU_ID: 0
27 VIDEO_OPTION: []
28 # Can be uncommented to generate videos during training.
29 # VIDEO_OPTION: ["disk", "tensorboard"]
30 TENSORBOARD_DIR: "Evaluation/Reactive/Tensorboard"
31 VIDEO_DIR: "Evaluation/Reactive/Video"
32 # Evaluate on all episodes
33 TEST_EPISODE_COUNT: -1
34
35 SENSORS: ['DEPTH_SENSOR', 'POINTGOAL_WITH_GPS_COMPASS_SENSOR']
36
37 CHECKPOINT_FOLDER: "Training/Reactive/Checkpoints"
38 TRAINING_LOG_FOLDER: "Training/Reactive/Log"
39 # If True, the log will not output messages to the console screen
       during training
40 LOG_SILENT: False
```

```
41 EVAL_CKPT_PATH_DIR: "Training/Reactive/Checkpoints"
42
43 # One of these two parameters must be present:
44 #       TOTAL_NUM_STEPS: Maximum number of steps the agent will take
       across all epochs
45 #       NUM_UPDATES: Number of updates (completed episodes) that
      have been performed
46 # Training stops when either of these values are reached
47
48 # TOTAL_NUM_STEPS: 2000.0
49 NUM_UPDATES: 15000
50 LOG_INTERVAL: 25
51 NUM_CHECKPOINTS: 100
52
53 # Reinforcement Learning specific configs
54 RL:
55   # Seed used for all experiments. Can be commented to use a random
        seed
56   seed: 0
57
58   # Deep Q-Learning parameters
59   DQL:
60     # Learning rate of the neural network
61     learning_rate: 0.001
62     # Maximum size of the Experience Replay (once full, older
         experiences will be removed)
63     er_size: 20000
64     # Batch size when sampling the Experience Replay
65     batch_size: 64
66     # Batches of experiences are split into chunks of this size
         during training
67     # Reduces training speed, but improves memory usage
68     training_batch_size: 32
69     # Gamma value (learning rate of DQL)
70     gamma: 0.99
71     # Epsilon value (initial chance to perform a random action due
         to exploration-exploitation)
72     epsilon: 1.00
73     # Minimum epsilon value, achieved after a percentage of epochs (
         min_epsilon_percentage)
74     min_epsilon: 0.05
75     # Percentage of epochs (between 0 and 1) after which epsilon
         will reach min_epsilon.
76     # The value of epsilon will decrease linearly from epsilon to
         min_epsilon
77     min_epsilon_percentage: 0.8
78     # Chooses between standard DQL (False) or Prioritized DQL (True)
79     prioritized: False
80     # (PRIORITIZED ONLY) Alpha value (priority degree). The higher
```

```
        alpha is, the higher the probability of choosing higher error
         experiences is
81    prioritized_alpha: 0.5
82    # (PRIORITIZED ONLY) Beta value (bias degree). The higher the
        value is, the less weight variations have (to avoid big
        oscillations)
83    prioritized_beta: 0.5
84
85  # Image pre-processing parameters (used to compute the rewards)
86  IMAGE:
87    # Pixels to be trimmed from both bottom and top of the image (
        the depth view seen by the camera)
88    # This parameter is relevant since the robot is an embodied
        agent that will always see the floor (and, in the case of
        houses,
89    # the roof) at a constant height
90    # Therefore, it can be trimmed without problem
91    trim: 35
92    # Threshold to consider a part of the image an obstacle. Note
        that the image is a grayscale image from 0.0 to 1.0, where 0
         (black) means the closest and 1.0 (white) means the furthest
93    # This also doubles as the maximum distance to an obstacle.
94    obstacle_threshold: 0.15
95    # Minimum area (in pixels) for contours / columns. Contours /
        columns smaller than this size will be ignored
96    min_contour_area: 250
97    # (COLUMN REWARDS ONLY) Total columns to be used when using the
        column reward method. Ignored when using the contour
        reward_method
98    reward_columns: 8
99
100 # Reward parameters
101 REWARD:
102   # Reward method to be used. There are two possibilities:
103   #   - contour: Contour based approach, imitating the original
        laser-based proposal
104   #   - column: Column based approach, dividing the image into
        smaller columns and computing each column as obstacle / no
        obstacle.
105   reward_method: contour
106   # Approximate distance (in simulator units) at which obstacles
        are when they are at the threshold.
107   # Can also be understood as the maximum distance the camera will
         detect obstacles
108   obstacle_distance: 2
109   # Positive gain applied to the attractive field, to increase its
         weight
110   attraction_gain: 100
111   # Positive gain applied to the repulsive field, to increase its
```

```
      weight
112   repulsive_gain: 15
113   # Value used to limit the repulsive field's maximum value
114   repulsive_limit: 0.04
115   # Percentage (between 0 and 1). When the goal is closer than
         repulsive_goal_influence * obstacle_distance, the effect of
         the repulsive field gets decreased
116   repulsive_goal_influence: 0.75
117   # Success reward. Note that positive rewards will also be
         clipped to this value
118   success_reward: 10
119   # Slack penalty, added to the reward each non-final episode to
         ensure that the agent doesn't end in a loop of doing actions
         without reward to avoid a penalty
120   slack_penalty: -0.25
121   # Failure penalty. Note that negative rewards will also be
         clipped to this value
122   failure_penalty: -100
123   # If True, the episode will end immediately if a collision is
         detected
124   collisions: False
```

B.3. Ficheros de *benchmark*

```
1  # REACTIVE NAVIGATION
2  # Benchmark configuration
3  # Developed by Luna Jimenez Fernandez
4  #
5  # This file contains all general parameters to be used while
      benchmarking
6  # the agents
7  #
8  # In essence, this is a variation of the base file to be used
9  # when evaluating the agents
10
11 # BENCHMARKING OPTIONS
12 VIDEO_OPTION: ["disk"]
13 VIDEO_DIR: "Video/Reactive"
14 SEED: 0
15
16 ENVIRONMENT:
17   MAX_EPISODE_STEPS: 500
18
19 SIMULATOR:
20   AGENT_0:
21     SENSORS: ['RGB_SENSOR', 'DEPTH_SENSOR']
22   RGB_SENSOR:
23     WIDTH: 256
24     HEIGHT: 256
```

```
  DEPTH_SENSOR:
    WIDTH: 256
    HEIGHT: 256

  # Physics are explicitely disabled, to avoid segmentation faults
  HABITAT_SIM_V0:
    ENABLE_PHYSICS: False

# DATASET: Dataset path, name and split to be used must be specified here
DATASET:
  TYPE: PointNav-v1
  SPLIT: "val"
  DATA_PATH: "./data/datasets/pointnav/gibson/v1/{split}/{split}.json.gz"
  NAME: "gibson"

TASK:
  TYPE: Nav-v0
  SUCCESS_DISTANCE: 0.3
  SENSORS: ['POINTGOAL_WITH_GPS_COMPASS_SENSOR']
  POSSIBLE_ACTIONS: ['STOP', 'MOVE_FORWARD', 'TURN_LEFT', 'TURN_RIGHT']
  POINTGOAL_WITH_GPS_COMPASS_SENSOR:
    GOAL_FORMAT: "POLAR"
    DIMENSIONALITY: 2
  GOAL_SENSOR_UUID: pointgoal_with_gps_compass
  MEASUREMENTS: ['DISTANCE_TO_GOAL', 'SUCCESS', 'SPL', 'SOFT_SPL', "COLLISIONS"]
  SUCCESS:
    SUCCESS_DISTANCE: 0.3

# Dummy RL config, to allow usage with video
RL:
  dummy: True
```

B.4. Fichero de generación de video

```
# REACTIVE NAVIGATION
# Video
# Developed by Luna Jimenez Fernandez
#
# This file contains all general parameters to be used
# in order to generate video of the agents
#

BASE_TASK_CONFIG_PATH: "configs/base_config.yaml"
VIDEO_OPTION: ["disk"]
VIDEO_DIR: "Video"
```

```
SEED: 0

# Dataset is still specified via console

# Add a dummy RL section, to avoid errors
# (Habitat Lab expects video to only be created for RL agents)
RL:
  dummy: True
```

Apéndice C

Generación de gráficas

[NO TENGO MUY CLARO COMO SE ESCRIBIAN LOS APENDICES NO TE VOY A ENGAÑAR] [EN LATEX ME REFIERO]

[APENDICES PROBABLES:] [MANUAL DE USUARIO, COMO GENERO GRAFICAS, CONTENIDOS DEL CD]

Apéndice D

Contenidos del entregable

[NO TENGO MUY CLARO COMO SE ESCRIBIAN LOS APENDICES NO TE VOY A ENGAÑAR] [EN LATEX ME REFIERO]

[APENDICES PROBABLES:] [MANUAL DE USUARIO, COMO GENERO GRAFICAS, CONTENIDOS DEL CD]