
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
Tutor(a): Martín Molina Gómez



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

# Resumen

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

# Abstract

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

# 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 lineas 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 lineas 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

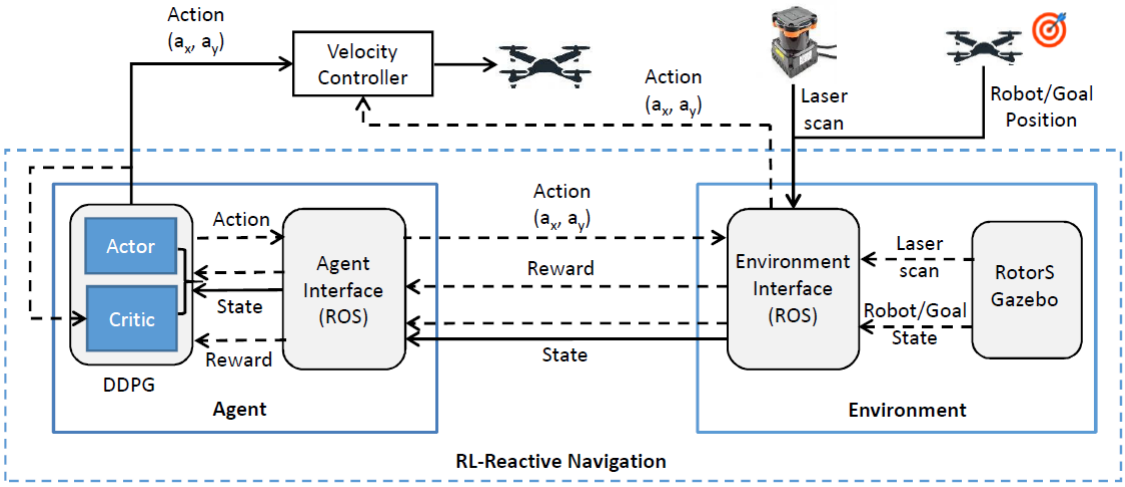


Figura 2.1: Arquitectura propuesta para el sistema de navegación reactivo original [1].

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

## 3.1. Redes neuronales y *Deep Learning*

### 3.1.1. Redes neuronales superficiales

### 3.1.2. Redes neuronales profundas

## 3.2. Aprendizaje por refuerzo

### 3.2.1. Algoritmos de aprendizaje por refuerzo clásicos

### 3.2.2. Algoritmos de aprendizaje por refuerzo profundos

[DQL y VARIANTES] [AGENTE / CRITICO, mas mencionado que otra cosa]

## 3.3. Algoritmos de navegación automática

[PROBABLEMENTE CENTRARSE MAS EN ALGORITMOS DE ATRACCION REPULSION]

# 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* [12] (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** [13] o **MINOS** [14]. 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* [15]. 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.

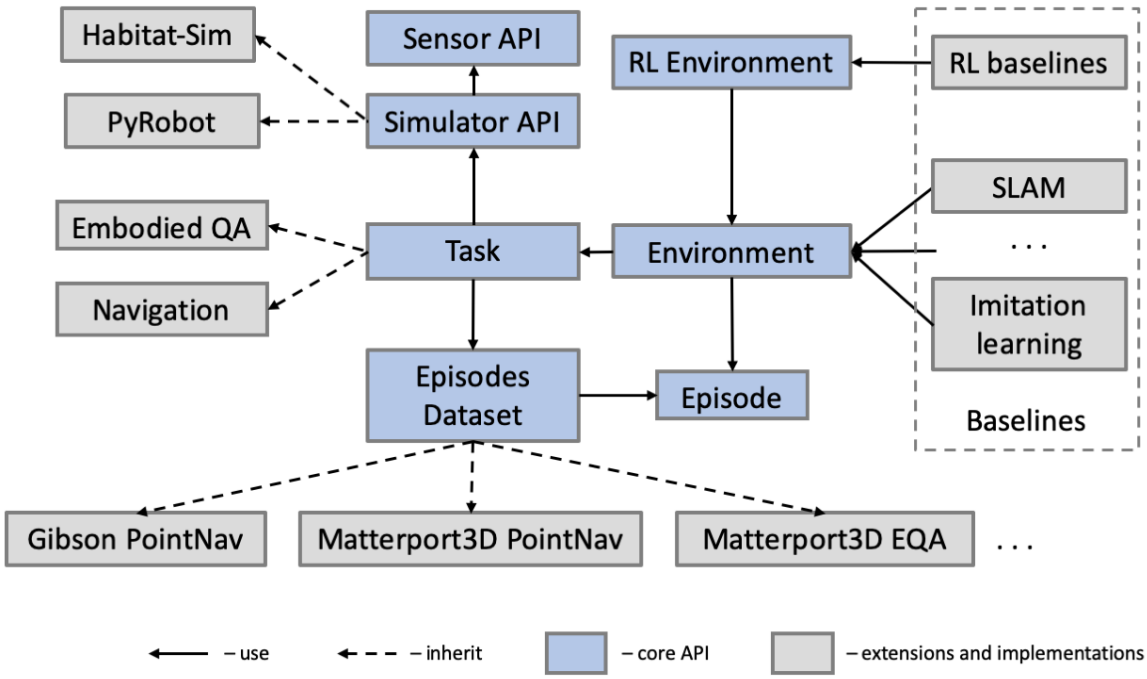


Figura 4.1: Arquitectura de *Habitat Lab* [2].

### 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) [16]:** 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* [17]:** *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.

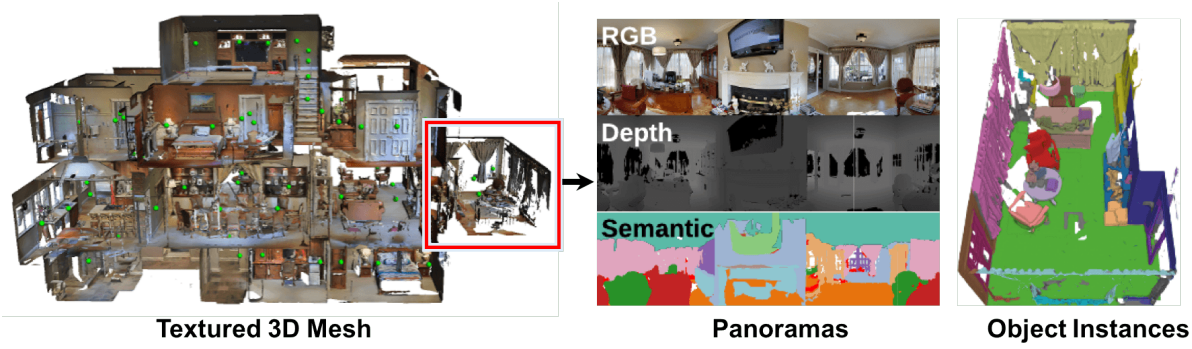


Figura 4.2: Modelos 3D de *Matterport3D*, y panorámicas asociadas [17].

- ***Gibson* [13]:** *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* [13].

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* [15], 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* [18]:** *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* [18].

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:

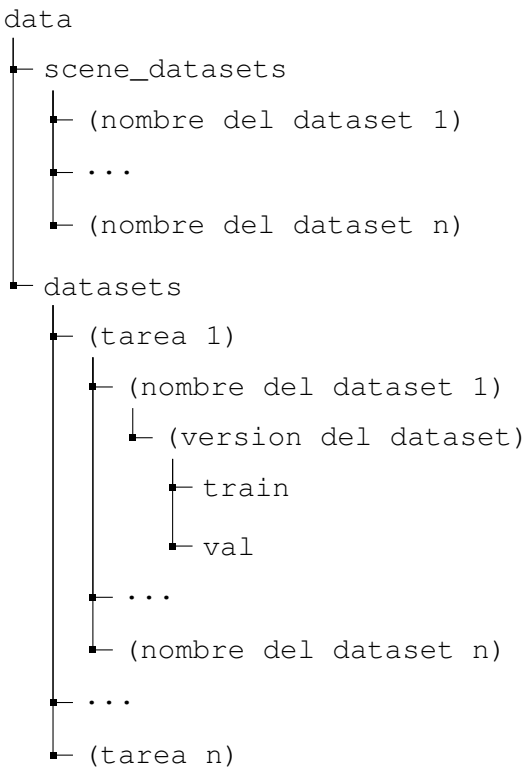


Figura 4.6: Ejemplo de árbol de directorios de la carpeta *data*.

- ***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* [13].

- **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.

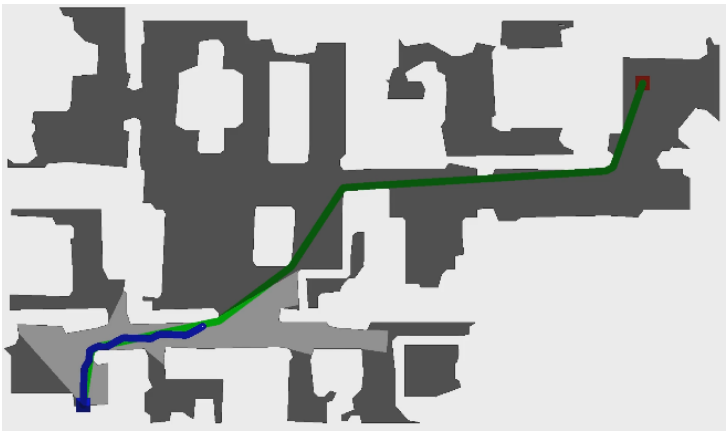

Figura 4.8: Mapa de un escenario de *HM3D* [18].

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 1:** 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* [19] 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 si 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 $(256x256)$ píxeles 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 $(256x256)$ píxeles, 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 perdidas 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 mínimo (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 2:** 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 píxeles $area\_min$.

**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 píxeles blancos (obstáculos) en cada columna, considerando las columnas que tengan una cantidad superior al área mínima ($250$ píxeles 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 3:** 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 píxeles $area\_min$.

**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 $\alpha$ es una ganancia positiva usada para aumentar la influencia del potencial (estimada empíricamente como $100$) y $p_{goal}(t_r)$ es la distancia euclídea 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 $\beta$ 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 $\delta$ 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 limite 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 [21].

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 [22].
- **Función de optimización:** Adam [23].
- **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

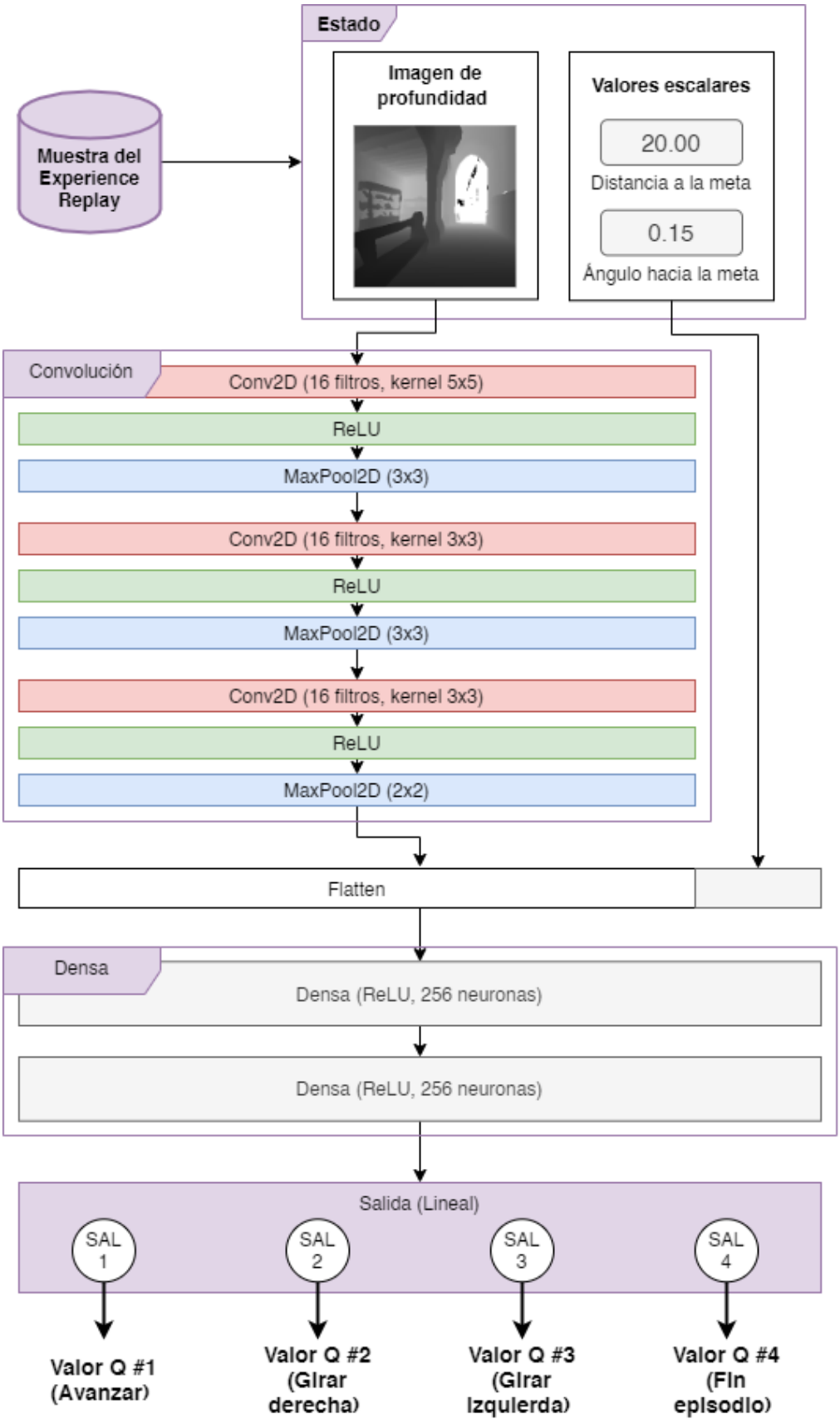


Figura 5.5: Arquitectura de la red neuronal - Propuesta 1 (CNN).

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 [24] o el trabajo de Yanyu Zhang aplicando redes mixtas al aprendizaje por refuerzo profundo [25]. 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 [26].
- **Función de optimización:** Adam [23].
- **Función de error:** Error de Huber [27].

  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 $\epsilon$ como la variable que regula el proceso. $\epsilon$ 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 $\epsilon_{init}$ es el valor inicial de $\epsilon$ (por defecto $1.0$), $\epsilon_{min}$ es el valor mínimo alcanzado por $\epsilon$ (por defecto $0.05$), $\epsilon_{min\_porcentaje}$ es el porcentaje de episodios tras el cual el

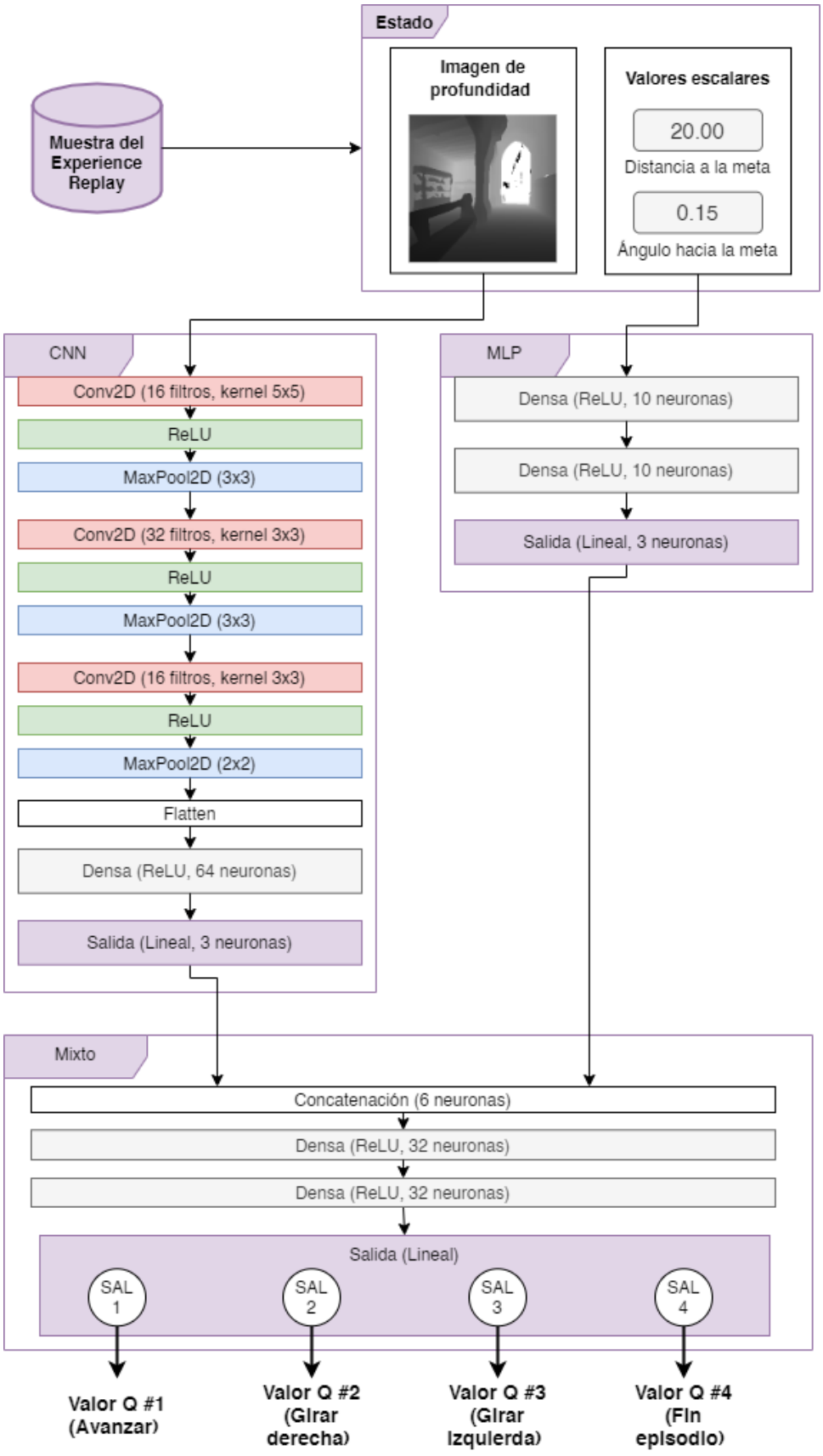


Figura 5.6: Arquitectura de la red neuronal - Propuesta 2 (Mixta).

**Algoritmo 4:** 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 $\epsilon$, lista de acciones posibles $action\_list$. el agente conserva una variable $\epsilon$ 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 $\epsilon$ alcanzará $\epsilon_{min}$ (por defecto $0.8$, el $80\,\%$ de los episodios) y $porcentaje$ es el porcentaje actual de episodios completados.

En esencia, el valor de $\epsilon$ decrece linealmente desde su valor inicial, $\epsilon_{init}$, hasta su valor final, $\epsilon_{min}$, 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 $\epsilon_{min}$ se alcanzaría a los 8000 episodios). El valor de $\epsilon$ no puede bajar de $\epsilon_{min}$ 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 $\epsilon$ 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 5:** 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, $ep\_total$. $\epsilon$, 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 $\epsilon$, 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 $\epsilon$.
- **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 6:** 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 $agente\_obj$. $\gamma$, 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 $\alpha$ 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 $\beta$ 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

## 7.3. Agradecimientos

[YA QUE NO LO TIENEN ANTES...]

# 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 col., "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 y col., "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 y col., "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 y col., "Sim2Real Predictivity: Does Evaluation in Simulation Predict Real-World Performance?," 4, vol. 5, 2020, págs. 6670-6677.

[7] D. Batra, A. Gokaslan, A. Kembhavi y col., "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] 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.

[13] 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.

[14] 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.

[15] J. Straub, T. Whelan, L. Ma y col., "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.

[16] E. Wijmans, S. Datta, O. Maksymets y col., "Embodied Question Answering in Photorealistic Environments with Point Cloud Perception," en *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[17] A. Chang, A. Dai, T. Funkhouser y col., "Matterport3D: Learning from RGB-D Data in Indoor Environments," *International Conference on 3D Vision (3DV)*, 2017.

[18] D. Batra, A. Chang, A. Clegg y col., *Habitat Matterport Dataset*, 2021. dirección: https://aihabitat.org/datasets/hm3d/0002.html.

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

[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] 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.

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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
```

```yaml
LOG_SILENT: False
EVAL_CKPT_PATH_DIR: "Training/Reactive/Checkpoints"

# One of these two parameters must be present:
#       TOTAL_NUM_STEPS: Maximum number of steps the agent will take across all epochs
#       NUM_UPDATES: Number of updates (completed episodes) that have been performed
# Training stops when either of these values are reached

# TOTAL_NUM_STEPS: 2000.0
NUM_UPDATES: 15000
LOG_INTERVAL: 25
NUM_CHECKPOINTS: 100

# Reinforcement Learning specific configs
RL:
  # Seed used for all experiments. Can be commented to use a random seed
  seed: 0

  # Deep Q-Learning parameters
  DQL:
    # Learning rate of the neural network
    learning_rate: 0.001
    # Maximum size of the Experience Replay (once full, older experiences will be removed)
    er_size: 20000
    # Batch size when sampling the Experience Replay
    batch_size: 64
    # Batches of experiences are split into chunks of this size during training
    # Reduces training speed, but improves memory usage
    training_batch_size: 32
    # Gamma value (learning rate of DQL)
    gamma: 0.99
    # Epsilon value (initial chance to perform a random action due to exploration-exploitation)
    epsilon: 1.00
    # Minimum epsilon value, achieved after a percentage of epochs (min_epsilon_percentage)
    min_epsilon: 0.05
    # Percentage of epochs (between 0 and 1) after which epsilon will reach min_epsilon.
    # The value of epsilon will decrease linearly from epsilon to min_epsilon
    min_epsilon_percentage: 0.8
    # Chooses between standard DQL (False) or Prioritized DQL (True)
    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:
```

```
    WIDTH: 256
    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]