UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Simulación basada en agentes de vehículos autónomos

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Carbonell Granados, Alberto

Tutor/a: Julian Inglada, Vicente Javier

Director/a Experimental: RINCON ARANGO, JAIME ANDRES

CURSO ACADÉMICO: 2021/2022

Resumen

En años recientes el desarrollo de vehículos de conducción autónoma para el transporte de personas a nivel personal o de servicio como taxis ha avanzado considerablemente. Esto crea la necesidad de implementar un sistema capaz de aprender a conducir un coche igual o mejor que un ser humano.

Es por ello que en este proyecto se ha desarrollado un entorno de simulación que permite el control de un coche de forma automática por parte de un agente inteligente. En esta herramienta se han unido las posibilidades que ofrece un sistema de agentes inteligentes junto con un entorno de simulación y un modelo de aprendizaje por refuerzo.

El sistema desarrollado se ha realizado usando una arquitectura que facilita utilizar una unión del sistema de agentes inteligentes SPADE, el simulador de vehículos CARLA y un modelo de aprendizaje por refuerzo. Gracias a esta estructura es posible llevar a cabo el desarrollo de aplicaciones con modelos de agentes inteligentes y modelos de aprendizaje por refuerzo adaptado a la conducción de vehículos.

Respecto al modelo de aprendizaje por refuerzo implementado, este trata de tomar acciones que resultan en las maniobras de un vehículo y recibir una recompensa en consecuencia. Esta recompensa representa el grado de beneficio proporcionado para alcanzar el objetivo final. A medida que se realizan las acciones, el algoritmo aprende qué acciones aportan mejores recompensas en determinados estados.

Palabras clave: sistemas multi-agente; simulación; aprendizaje por refuerzo.

Resum

En anys recents el desenvolupament de vehicles de conducció autònoma per al transport de persones a nivell personal o de servei com a taxis ha avançat considerablement. Això crea la necessitat d'implementar un sistema capaç d'aprendre a conduir un cotxe igual o millor que un ésser humà.

És per això que en aquest projecte s'ha desenvolupat un entorn de simulació que permet el control d'un cotxe de manera automàtica per part d'un agent intel·ligent. En aquesta eina s'han unit les possibilitats que ofereix un sistema d'agents intel·ligents juntament amb un entorn de simulació i un model d'aprenentatge per reforç.

El sistema desenvolupat s'ha realitzat usant una arquitectura que facilita utilitzar una unió del sistema d'agents intel·ligents SPADE, el simulador de vehicles CARLA i un model d'aprenentatge per reforç. Gràcies a aquesta estructura és possible dur a terme el desenvolupament d'aplicacions amb models d'agents intel·ligents i models d'aprenentatge per reforç adaptat a la conducció de vehicles.

Respecte al model d'aprenentatge per reforç implementat, aquest tracta de prendre accions que resulten en les maniobres d'un vehicle i rebre una recompensa en conseqüència. Aquesta recompensa representa el grau de benefici proporcionat per a aconseguir l'objectiu final. A mesura que es realitzen les accions, l'algorisme aprén quines accions aporten millors recompenses en determinats estats.

Paraules clau: sistemes multi-agent; simulació; aprenentatge per reforç.

Abstract

In recent years the development of autonomous driving vehicles for transporting people on a personal or service level such as cabs has advanced considerably. This creates the need to implement a system capable of learning to drive a car as well as or better than a human being.

That is why in this project a simulation environment has been developed that allows the control of a car automatically by an intelligent agent. In this tool, the possibilities offered by an intelligent agent system have been combined with a simulation environment and a reinforcement learning model.

The developed system has been developed using an architecture that facilitates the use of a union of the SPADE intelligent agent system, the CARLA vehicle simulator and a reinforcement learning model. Thanks to this structure it is possible to carry out the development of applications with intelligent agent models and reinforcement learning models adapted to vehicle driving.

Regarding the reinforcement learning model implemented, it deals with taking actions that result in the maneuvers of a vehicle and receiving a reward accordingly. This reward represents the degree of benefit provided to achieve the final goal. As actions are taken, the algorithm learns which actions provide better rewards in certain states.

Keywords: multi-agent systems; simulation; reinforcement learning.

Tabla de contenidos

1. Introducción

1.1 Motivación

Este proyecto se impulsa en la motivación de crear un entorno de simulación que disponga de la integración de un sistema de agentes inteligentes junto con la simulación de entornos de conducción para vehículos autónomos. La unión de estas dos tecnologías ofrece un amplio campo de investigación y desarrollo de aplicaciones de distintos ámbitos. Para este proyecto se decidió plantear modelos de conducción autónoma mediante el uso de agentes inteligentes y modelos de aprendizaje por refuerzo.

La comunicación entre agente y vehículo es una parte muy importante a la hora de realizar una conducción autónoma. De este modo, facilita una interacción correcta entre elementos de la simulación como son el acelerador, el freno y el volante. Por otra parte, los modelos de aprendizaje por refuerzo recopilan información ventajosa para la toma de decisión en la ejecución de tareas. En este tipo de aprendizaje no es necesario la supervisión del aprendizaje, basta con definir el beneficio (o recompensas) de manera adecuada.

De este modo, nació la necesidad y motivación de desarrollar un entorno de simulación que fuera capaz de albergar modelos de comunicación entre vehículo y agente y modelos de aprendizaje por refuerzo entre otros.

1.2 Objetivos

Los objetivos de este proyecto se basan en una serie de puntos obtenidos a partir de la inquietud de desarrollar un entorno que permita el control de agentes inteligentes sobre un vehículo plenamente funcional y modelos de aprendizaje por refuerzo aplicados a la conducción de vehículos de forma autónoma. A continuación se detallan los objetivos del proyecto:

- Estudiar las posibles opciones para interconectar un sistema multiagente con un simulador de vehículos.
    - Seleccionar una plataforma de sistemas multiagente para el desarrollo del proyecto.
    - Seleccionar un simulador de vehículos para el desarrollo del proyecto.
    - Implementar una conexión entre el sistema multiagente y el simulador de vehículos.

- Diseñar e implementar un entorno de simulación de vehículos para poder realizar pruebas con diferentes modelos de aprendizaje con un algoritmo de Reinforcement Learning con DQN para desarrollar un sistema de conducción autónoma de vehículos basado en aprendizaje por refuerzo.

- Analizar los resultados de las pruebas realizadas para analizar y seleccionar el modelo que genera el mejor rendimiento a partir de una serie de mediciones como la proximidad al objetivo, la precisión y la pérdida del modelo.

- Validar el entorno mediante el entrenamiento del modelo seleccionado que mejores resultados haya proporcionado durante las pruebas dejando en ejecución el modelo durante el máximo tiempo posible y comprobar si el agente ha aprendido a conducir analizando los resultados de dicho entrenamiento.

1.3 Estructura de la memoria

Este documento se encuentra dividido en cuatro partes. El primer bloque describe el estado del arte, incluyendo un análisis de simuladores de conducción autónoma. En concreto, se detalla de forma más extensa el simulador seleccionado para el proyecto y se hace un pequeño reconocimiento de unos simuladores de conducción autónoma para el desarrollo de aplicaciones que han parecido más interesantes para este proyecto.

Seguidamente se detalla de forma breve en qué consiste el aprendizaje por refuerzo en especial el algoritmo que se ha implementado para la solución del proyecto, comparándolo con otras metodologías de aprendizaje por refuerzo también explicadas.

Para finalizar el bloque se realiza una revisión de la plataforma SPADE para el desarrollo de sistemas de multiagente. En concreto se detallan los aspectos más importantes de dicha plataforma.

El segundo bloque describe el proceso de desarrollo que se ha seguido para obtener el sistema en el que realizaremos nuestros experimentos. Este bloque se divide en secciones, la primera detalla como instalar y configurar el sistema de agentes inteligentes. La segunda sección explica cómo interconectar el sistema de agentes inteligentes con nuestro simulador de conducción autónoma.

Para finalizar el bloque, la última sección trata sobre la estructura del sistema general, y cómo está organizado el código. Esto es esencial para conocer la estructura y poder realizar futuros experimentos.

El tercer bloque describe la evolución de nuestro proyecto en diferentes versiones, cada una siendo una ampliación de la versión anterior con un grado de mayor complejidad. En el cuarto bloque se comienza exponiendo los resultados obtenidos tras experimentar con las diferentes versiones descritas en el bloque anterior y se concluye con la elección de una configuración para la prueba de validación.

En el quinto bloque se exponen y analizan los resultados de la prueba realizada con la configuración elegida en el bloque previo. Finalmente, el último bloque plantea unas conclusiones globales acerca de todo el proyecto junto con una serie de posibles trabajos futuros. El último punto del documento es una recopilación de la bibliografía consultada en todos los bloques de la memoria y empleada para desarrollar el proyecto.

2. Estado del arte

En el presente capítulo expondremos varios de los principales simuladores de conducción autónoma, analizando sus cualidades y principales características. También comentaremos los métodos más avanzados de aprendizaje por refuerzo y los compararemos. Por último, hablaremos sobre la plataforma de agentes SPADE la cual está basada en el uso de servidores XMPP para el canal de comunicación.

2.1 SIMULADORES DE CONDUCCIÓN AUTÓNOMA

Debido a la dificultad que supone la implementación de este tipo de sistemas en un entorno real, hemos decidido utilizar un simulador que nos preste la capacidad de controlar un vehículo y monitorizar el estado del entorno con una precisión que se asemeje a la que tendríamos en una carretera real.

AirSim

AirSim es una plataforma de código abierto que pretende reducir la distancia entre simulación y la realidad para ayudar al desarrollo de vehículos autónomos [1]. La plataforma pretende influir positivamente en el desarrollo y la prueba de técnicas de inteligencia artificial basadas en datos, como el aprendizaje por refuerzo y el aprendizaje profundo.

Sigue un diseño modular con énfasis en la extensibilidad. Los componentes principales incluyen un modelo de entorno, un modelo de vehículo, un motor de física, modelos de sensores, una interfaz de representación, una capa de API pública y una capa de interfaz para el firmware del vehículo.

Este simulador también proporciona una interfaz para definir el vehículo como un cuerpo rígido que puede tener un número arbitrario de actuadores que generan fuerzas y pares. El modelo del vehículo incluye parámetros como la masa, la inercia, los coeficientes de resistencia lineal y angular, los coeficientes de fricción y restitución que son utilizados por el motor de física para calcular la dinámica del cuerpo rígido.

Flow

Flow [3] es un framework computacional para el aprendizaje por refuerzo profundo y experimentos de control para la microsimulación de tráfico. Flow integra otro microsimulador de tráfico con una biblioteca estándar de aprendizaje de refuerzo profundo lo que permite el entrenamiento de experimentos de aprendizaje por refuerzo a gran escala en Amazon Web Services (AWS) y Elastic Compute Cloud (EC2) para tareas de control de tráfico.

Aprovechando recientes avances en el aprendizaje por refuerzo (RL), Flow permite el uso de métodos de RL como el gradiente de políticas para el control del tráfico y permite comparar el rendimiento de los controladores clásicos (incluyendo los diseñados a mano) con políticas aprendidas (leyes de control).

SUMMIT

SUMMIT es un simulador que genera datos interactivos de alta fidelidad para el tráfico urbano denso y no regulado en mapas complejos del mundo real[2]. Utiliza mapas del mundo real obtenidos de fuentes en línea para proporcionar una fuente prácticamente ilimitada de entornos complejos.

A partir de ubicaciones arbitrarias, el simulador genera automáticamente multitudes de agentes de tráfico heterogéneos con comportamientos sofisticados y no regulados. El simulador aprovecha los contextos viales de los mapas del mundo real para guiar los comportamientos de los agentes de tráfico topológica y geométricamente con el fin de construir condiciones de tráfico realistas.

SUMMIT está basado en CARLA [4] para aprovechar la física de alta fidelidad, el renderizado y los sensores. A través de una API basada en python, este simulador revela cuantiosos datos de sensores, información semántica y contextos viales a algoritmos externos, permitiendo la aplicación en una amplia gama de campos como la percepción, el control y la planificación de vehículos, el aprendizaje de extremo a extremo, etc. Proporciona resultados cualitativos y cuantitativos para demostrar que SUMMIT puede generar tráfico mixto complejo y realista en entornos urbanos del mundo real.

CARLA

Entre los diversos simuladores vistos hemos elegido CARLA, mostrado en la figura 2.1, popular entre los usuarios por su flexibilidad, potencia y escalabilidad. Su API permite al usuario controlar todos los aspectos relacionados con la simulación como el tráfico, el comportamiento de los peatones, el clima y los sensores. Las condiciones del clima abarcan hasta catorce configuraciones a elegir de un conjunto prefijado. Su arquitectura cliente-servidor posibilita ejecutar varios agentes donde cada uno controla diferentes actores.

Figura 2.1: Visualización del simulador CARLA

CARLA dispone de una gran variedad de sensores entre los que destacan:

- Cámara RGB: actúa como una cámara normal que captura imágenes de la escena. Se puede aplicar a la imagen un conjunto de efectos de postprocesamiento en aras del realismo:
    - Viñeta: oscurece el borde de la pantalla.
    - Granulado: añade algo de ruido al renderizado.
    - Bloom: las luces intensas queman el área que las rodea.
    - Exposición automática: modifica el gamma de la imagen para simular la adaptación del ojo a las zonas más oscuras o más brillantes.
    - Destellos de lente: simula el reflejo de objetos brillantes en la lente.
    - Profundidad de campo: desenfoca los objetos cercanos o muy alejados de la cámara.

- Cámara de profundidad: esta cámara proporciona un dato bruto de la escena codificando la distancia de cada píxel a la cámara (también conocido como buffer de profundidad o z-buffer) para crear un mapa de profundidad de los elementos.

- Cámara de segmentación semántica: esta cámara clasifica cada objeto a la vista mostrándolo en un color diferente según sus etiquetas (por ejemplo, los peatones en un color diferente al de los vehículos). Cuando se inicia la simulación, cada elemento de la escena se crea con una etiqueta. Así ocurre cuando se genera un actor. Los objetos se clasifican por su ruta de archivo relativa en el proyecto.

- Cámara de segmentación por instancia: esta cámara clasifica cada objeto en el campo de visión tanto por clase como por ID de instancia. Cuando se inicia la simulación, cada elemento de la escena se crea con una etiqueta. Lo mismo ocurre cuando se genera un actor. Los objetos se clasifican por su ruta de archivo relativa en el proyecto.

- Sensor de obstáculos: este sensor registra un evento cada vez que el actor padre tiene un obstáculo delante. Para anticiparse a los obstáculos, el sensor crea una forma capsular por delante del vehículo padre y la utiliza para comprobar si hay colisiones. Para asegurarse de que se detectan las colisiones con cualquier tipo de objeto, el servidor crea actores "falsos" para elementos como edificios o arbustos, de modo que se pueda recuperar la etiqueta semántica para identificarlo.

- Sensor de colisión: este sensor registra un evento cada vez que su actor principal colisiona contra algo en el mundo. Pueden detectarse varias colisiones durante un solo paso de simulación. Para asegurarse de que se detectan las colisiones con cualquier tipo de objeto, el servidor crea actores "falsos" para elementos como edificios o arbustos, de modo que se pueda recuperar la etiqueta semántica para identificarlo.

Estos sensores son parametrizables, con lo que podemos establecer el nivel de detalle de los datos recogidos del entorno, con qué frecuencia los obtenemos o el rango de detección de los sensores. Otro rasgo a destacar de los sensores es su capacidad de personalización al acoplarlos a un vehículo ya que podemos especificar la posición y rotación del sensor respecto al vehículo al que irá adherido.

Por último, cuenta con una comunidad activa en la que los usuarios comparten sus problemas y soluciones en foros donde los desarrolladores publican los cambios del proyecto a medida que los propios usuarios encuentran errores y proponen arreglos. Además de la propia comunidad de usuarios que utilizan este simulador, la página oficial de CARLA proporciona documentación que especifica todas las clases que componen los objetos de la simulación, así como sus atributos que pueden pertenecer a ese conjunto de clases o a un tipo de dato primitivo como un número o una cadena de texto.

Su API utiliza el lenguaje de programación Python y en su documentación también se utilizan ejemplos de código. Los datos que recogen los sensores también están supeditados a este lenguaje puesto que utilizan sus estructuras de datos características como los diccionarios.

Para que los sensores que recopilan información visual funcionen se necesita ejecutar CARLA en modo visualización, esto consume gran cantidad de memoria gráfica, en nuestro caso utiliza el 90% de la memoria de nuestra máquina. El modo de visualización emplea el motor gráfico Unreal Engine, encargado de procesar las físicas de la simulación, en concreto de los vehículos, cuyos modelos 3D están almacenados en un subdirectorio de la carpeta donde se encuentra el simulador, permitiendo modificarlos o añadir nuevos.

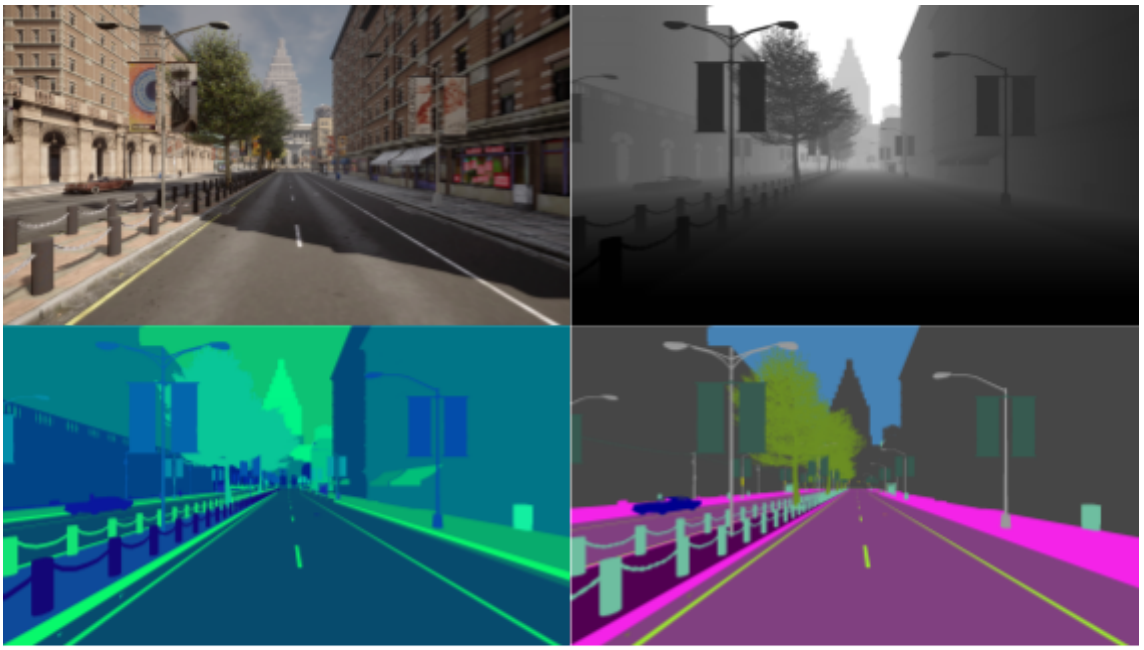

Figura 2.2: Visualización de las cámaras

Gracias al modo de visualización podemos emplear diversos filtros y efectos en las cámaras para un procesado distinto de las imágenes dependiendo de la finalidad para la que se utilicen estas cámaras como se puede apreciar en la figura 2.2.

La visualización se puede configurar desde uno de sus archivos de configuración con la capacidad de cambiar la calidad gráfica, la distancia de renderizado y otras propiedades visuales que, además de cambiar el cómo se ve la simulación desde la aplicación de Unreal Engine, afecta a los datos que obtendrán los sensores. Estos mismos archivos nos permiten modificar propiedades de la simulación como la gravedad, el clima, la velocidad angular máxima y otras físicas del mundo simulado.

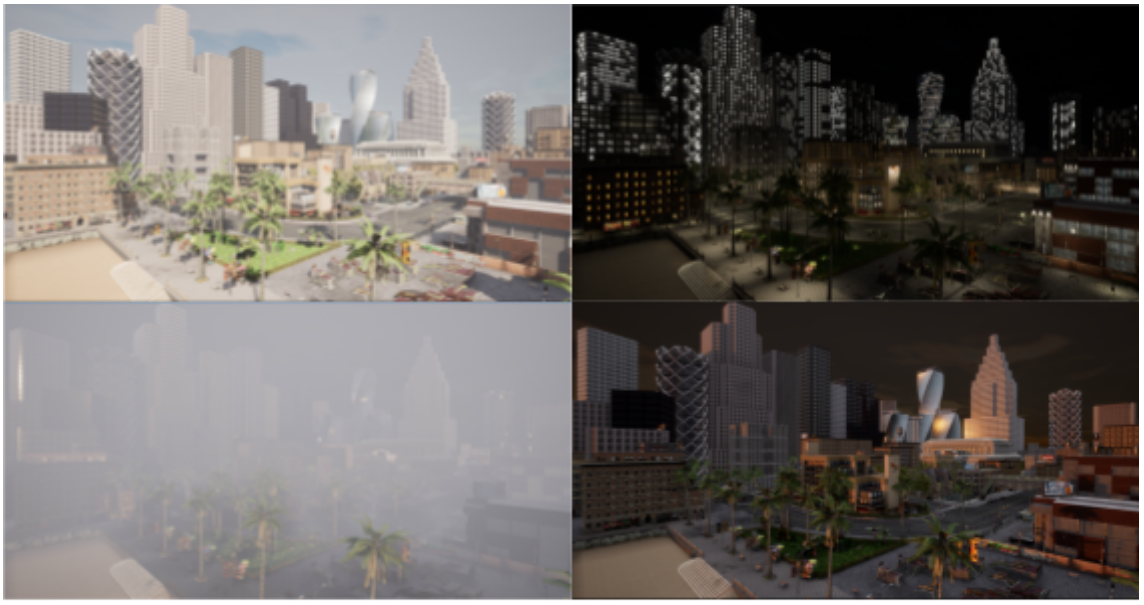

Figura 2.3: Cambio de clima en CARLA

Gracias a esta capacidad de renderizado y potencia que nos proporciona CARLA podemos realizar simulaciones muy diversas con distintos tipos de clima como se aprecia en la figura 2.3, con lo que se podría generar una gran diversidad de situaciones con las que un agente podría aprender.

Para poder utilizar CARLA necesitaremos tener ambos componentes de su funcionalidad, el servidor y el cliente. El servidor se puede obtener de su página de *GitHub*, descargando el proyecto y simplemente ejecutando el archivo *CarlaUE4.exe* se pondrá en funcionamiento el servidor.

Por defecto este hace uso del puerto 2000 del equipo donde se ejecute en *localhost*, esto permite que el servidor pueda lanzarse en un equipo distinto al que use el cliente y usar una conexión remota, seguiría usando el puerto 2000 pero la dirección ya no sería *localhost* sino otra dirección IP que corresponda con el equipo que ejecuta el servidor de CARLA.

Por parte del cliente, este será un programa de Python y por tanto se necesita importar la librería mediante el gestor de paquetes de este lenguaje, Pip. Una vez importada se puede utilizar para conectarse al servidor y empezar a usar el simulador. Esta facilidad es otra de las principales ventajas que nos proporciona este simulador frente a los demás.

2.2 APRENDIZAJE POR REFUERZO

El aprendizaje por refuerzo es el problema al que se enfrenta un agente que debe aprender un comportamiento a través de interacciones de ensayo y error con un entorno dinámico [5]. Hay dos estrategias principales para resolver los problemas de aprendizaje por refuerzo. La primera consiste en buscar en el espacio de los comportamientos para encontrar uno que funcione bien en el entorno. La segunda consiste en utilizar técnicas estadísticas y métodos de programación dinámica para estimar la utilidad de tomar acciones en los estados del mundo.

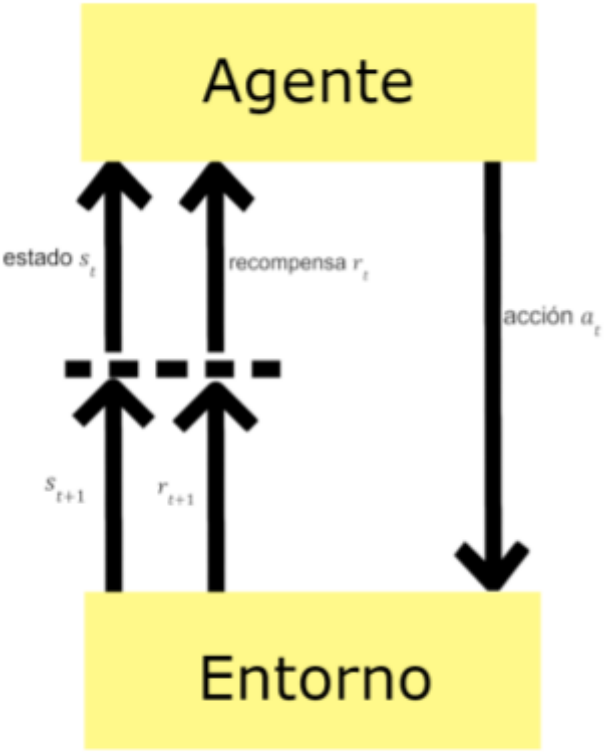

Figura 2.4: Aprendizaje por refuerzo

En el modelo estándar de aprendizaje por refuerzo, un agente está conectado a su entorno a través de la percepción y la acción. En cada paso de la interacción, el agente recibe como entrada, *i*, alguna indicación del estado actual, *s*, del entorno; a continuación, el agente elige una acción, *a*, para generar como salida.

La acción cambia el estado del entorno, y el valor de la transición de este estado se comunica al agente a través de una recompensa *r;* como se observa en la figura 2.4. El comportamiento del agente, B, debe elegir acciones que tiendan a aumentar la suma de la recompensa a largo plazo. Puede aprender a hacerlo a lo largo del tiempo mediante un proceso sistemático de ensayo y error, guiado por una gran variedad de algoritmos.

Según [5] se espera, en general, que el entorno sea no determinista, es decir, que realizar la misma acción en el mismo estado en dos ocasiones distintas puede dar lugar a estados siguientes diferentes y/o a recompensas diferentes. Sin embargo, se supone que el entorno es estacionario, es decir, que las probabilidades de realizar transiciones de estado o de recibir señales de refuerzo no cambian con el tiempo. Esta suposición puede ser decepcionante; después de todo, el funcionamiento en entornos no estacionarios es una de las motivaciones para construir sistemas de aprendizaje, pero hay muy poco análisis teórico en este ámbito.

Algunos aspectos del aprendizaje por refuerzo están estrechamente relacionados con los problemas de búsqueda y planificación en la inteligencia artificial. Los algoritmos de búsqueda de IA generan una trayectoria satisfactoria a través de un gráfico de estados. La planificación funciona de forma similar, pero normalmente dentro de un constructo de mayor complejidad que un grafo, en el que los estados se representan mediante composiciones de expresiones lógicas en lugar de símbolos atómicos.

Estos algoritmos de IA son menos generales que los métodos de aprendizaje de refuerzo, en el sentido de que requieren un modelo predeterminado de transiciones de estado, y, salvo algunas excepciones, asumen el determinismo. Por otra parte, el aprendizaje por refuerzo, al

menos en el tipo de casos discretos para los que se ha desarrollado la teoría, supone que todo el espacio de estados puede ser enumerado y almacenado en memoria, una suposición a la que los algoritmos de búsqueda convencionales no están vinculados.

En [6] se identifican cuatro subelementos principales de un sistema de aprendizaje por refuerzo: una política, una recompensa, una función de valor y, opcionalmente, un modelo del entorno.

Una política define la forma de actuar del agente de aprendizaje en un momento dado. A grandes rasgos, una política es un mapeo entre los estados percibidos del entorno y las acciones que deben realizarse cuando se encuentran en esos estados. Corresponde a lo que en psicología se llamaría un conjunto de reglas o asociaciones estímulo-respuesta (siempre que los estímulos incluyan los que pueden venir del interior del animal).

En algunos casos, la política puede ser una simple función o tabla de búsqueda, mientras que en otros puede implicar un cálculo extenso, como un proceso de búsqueda. La política es el núcleo de un agente de aprendizaje por refuerzo en el sentido de que sólo es suficiente para determinar el comportamiento. En general, las políticas pueden ser estocásticas.

Una recompensa define el objetivo en un problema de aprendizaje por refuerzo. En cada paso de tiempo, el entorno envía al agente de aprendizaje por refuerzo un único número, una recompensa. El único objetivo del agente es maximizar la recompensa total que recibe a largo plazo. La recompensa define, por tanto, cuáles son los eventos buenos y malos para el agente. En un sistema biológico, podríamos pensar en las recompensas como algo análogo a las experiencias de placer o dolor. Son las características inmediatas y definitorias del problema al que se enfrenta el agente.

La recompensa enviada al agente en cualquier momento depende de la acción actual del agente y del estado actual del entorno del agente. El agente no puede alterar el proceso que lo hace. La única forma en que el agente puede influir en la recompensa es a través de sus acciones, que pueden tener un efecto directo sobre la recompensa, o un efecto indirecto al cambiar el estado del entorno.

La recompensa es la base principal para alterar la política. Si una acción seleccionada por la política es seguida por una baja recompensa, entonces la política puede ser cambiada para seleccionar alguna otra acción en esa situación en el futuro. En general, las recompensas pueden ser funciones estocásticas del estado del entorno y de las acciones realizadas.

Mientras que la recompensa indica lo que es bueno en un sentido inmediato una función de valor especifica lo que es bueno a largo plazo. A grandes rasgos, el valor de un estado es la cantidad total de recompensa que un agente puede esperar acumular en el futuro, partiendo de ese estado. Mientras que las recompensas determinan la conveniencia inmediata e intrínseca de los estados del entorno, los valores indican la conveniencia a largo plazo de los estados después de tener en cuenta los estados que son probables a aparecer a continuación y las recompensas disponibles en esos estados.

Por ejemplo, un estado puede producir siempre una recompensa inmediata baja, pero seguir teniendo un valor alto porque le siguen regularmente otros estados que producen recompensas elevadas. Para hacer una analogía humana, las recompensas son algo así como el placer (si son altas) y el dolor (si son bajas), mientras que los valores corresponden a un juicio más refinado y clarividente de cómo nos complace o disgusta que nuestro entorno se encuentre en un estado determinado.

Las recompensas son en cierto sentido primarias, mientras que los valores, como predicciones de las recompensas, son secundarios. Sin recompensas no podría haber valores, y el único propósito de estimar los valores es conseguir más recompensas. Sin embargo, son los

valores los que más nos preocupan a la hora de tomar y evaluar decisiones. Las elecciones de acción se hacen en base a juicios de valor.

En general se buscan acciones que provoquen mayor valor, no mayor recompensa, porque estas acciones obtienen la mayor recompensa para nosotros a largo plazo. En la toma de decisiones y en la planificación, la cantidad derivada llamada valor es la que más nos preocupa. Por desgracia, es mucho más difícil determinar los valores que las recompensas.

Las recompensas vienen dadas básicamente de forma directa por el entorno, pero los valores deben estimarse y reestimarse a partir de las secuencias de observaciones que realiza un agente a lo largo de su vida. De hecho, el componente más importante de casi todos los algoritmos de aprendizaje por refuerzo que consideramos es un método para estimar eficazmente los valores. El papel central de la estimación de valores es posiblemente uno de los aspectos más importantes en los que se ha avanzado sobre el aprendizaje por refuerzo en las últimas décadas.

El cuarto y último elemento de algunos sistemas de aprendizaje por refuerzo es la representación de un modelo del entorno. Se trata de algo que imita el comportamiento del entorno, o más generalmente, que permite hacer inferencias sobre cómo se comportará el entorno. Por ejemplo, dado un estado y una acción el modelo podría predecir el siguiente estado y la siguiente recompensa resultantes. Los modelos se utilizan para la planificación, es decir, cualquier forma de decidir un curso de acción teniendo en cuenta posibles situaciones futuras antes de que se produzcan.

Los métodos para resolver problemas de aprendizaje por refuerzo que utilizan modelos y la planificación se denominan métodos basados en modelos, en contraposición a los métodos más sencillos sin modelos que son explícitamente aprendices de prueba y error, vistos casi como lo contrario de la planificación. El aprendizaje por refuerzo moderno abarca el espectro que va desde el aprendizaje por ensayo y error de bajo nivel hasta la planificación deliberativa de alto nivel.

Q-Learning

El Q-Learning es una forma de aprendizaje por refuerzo sin modelo [7]. También se puede ver como un método de programación dinámica (PD) asíncrona. Proporciona a los agentes la capacidad de aprender a actuar de forma óptima en dominios markovianos experimentando las consecuencias de las acciones, sin necesidad de construir mapas de los dominios. El aprendizaje procede de la siguiente forma: un agente intenta una acción en un estado particular y evalúa sus consecuencias en términos de la recompensa o penalización inmediata que recibe y su estimación del valor del estado al que se lleva.

Al probar repetidamente todas las acciones en todos los estados, aprende cuáles son las mejores en general, juzgadas por la recompensa descontada a largo plazo. El Q-Learning es una forma primitiva[8] de aprendizaje, pero, como tal, puede funcionar como base de dispositivos mucho más sofisticados.

[7] considera un agente computacional que se desplaza por un mundo discreto y finito, eligiendo una de una colección finita de acciones en cada paso de tiempo. El mundo constituye un proceso de Markov controlado con el agente como controlador. En el paso n, el agente está equipado para registrar el estado $x_n \in X$ del mundo, y puede elegir su acción $a_n \in A$ en consecuencia. El agente recibe una recompensa probabilística r_n, cuyo valor medio $R_{x_n}(a_n)$ depende sólo del estado y la acción, y el estado del mundo cambia probabilísticamente a y_n según la ley:

$$Prob\,[y_n = y|x_n, a_n] = P_{x_n y}[a_n]\,. \qquad (1)$$

La tarea a la que se enfrenta el agente es la de determinar una política óptima, una que maximice la recompensa descontada esperada. Por recompensa descontada, queremos decir que las recompensas recibidas *s* pasos antes valen menos que las recompensas recibidas ahora, por un factor de $\gamma^s (0 < \gamma < 1)$. Bajo una política π, el valor del estado *x* es

$$V^{\pi}(x) \equiv R_x(\pi(x)) + \gamma \sum_y P_{xy}[\pi(x)] V^{\pi}(y) \qquad (2)$$

porque el agente espera recibir $R_x(\pi(x))$ inmediatamente por realizar la acción que π recomienda, y entonces se desplaza a un estado que le "vale" $V^{\pi}(y)$, con probabilidad $P_{xy}[\pi(x)]$ .Se asegura que existe existe al menos una política óptima π^* tal que

$$V^*(x) \equiv V^{\pi^*}(x) = max_a \left\{ R_x(a) + \gamma \sum_y P_{xy}[a] V^{\pi^*}(y) \right\} \qquad (3)$$

es lo mejor que puede hacer un agente a partir del estado x. Aunque esto puede parecer circular, en realidad está bien definido, y se proporcionan una serie de métodos para calcular V* y un π^*(suponiendo que se conocen $R_x(a)$ y $P_{xy}[a]$. La tarea a la que se enfrenta un agente, es la de determinar un π^* sin conocer inicialmente estos valores.

Existen métodos tradicionales para aprender $R_x(a)$ y $P_{xy}[a]$, pero cualquier suposición de equivalencia de certeza, es decir, calcular las acciones como si el modelo actual fuera exacto, cuesta mucho en las primeras etapas del aprendizaje. [7] clasifica el Q-Learning como programación dinámica incremental, por la forma en que determina la política óptima paso a paso. Para una política π, define los valores Q (o valores de acción) como:

$$Q^{\pi}(x, a) = R_x(a) + \gamma \sum_y P_{xy}[\pi(x)] V^{\pi}(y) \qquad (4)$$

En otras palabras, el valor Q es la recompensa descontada esperada por ejecutar la acción *a* en el estado *x* y seguir la política π a partir de entonces. El objetivo de Q-Learning es estimar los valores Q para una política óptima. Por conveniencia, los definiremos como $Q^*(x, a) \equiv Q^{\pi^*}(x, a), \forall x, a$. Es sencillo demostrar que $V^*(x) = max_a Q^*(x, a)$ y que si a^* es una acción en la que se alcanza el máximo, entonces se puede formar una política óptima como $\pi^*(x) = a^*$. En esto yace la utilidad de los valores Q, si un agente puede aprenderlos, puede decidir fácilmente lo que es óptimo hacer. Aunque puede haber más de una política óptima o a^*, los valores Q^* son únicos.

En Q-Learning, la experiencia del agente consiste en una secuencia de etapas o episodios distintos[7]. En el enésimo episodio, el agente:

- observa su estado actual x_n,

- selecciona y realiza una acción a_n,

- observa el estado posterior y_n,

- recibe una recompensa inmediata r_n, y

- ajusta sus valores Q_{n-1} utilizando un factor de aprendizaje α_n, según:

$$Q_n(x,a) = \begin{cases} (1-\alpha_n)Q_{n-1}(x,a) + \alpha_n[r_n + \gamma V_{n-1}(y_n)] & si\ a_n = a\ y\ x_n = x \\ Q_{n-1}(x,a) & sino \end{cases} \quad (5)$$

donde

$$V_{n-1}(y) \equiv max_b\{Q_{n-1}(y,b)\} \quad (6)$$

es lo mejor que el agente cree que puede hacer desde el estado *y*. Por supuesto, en las primeras etapas del aprendizaje, los valores Q, pueden no reflejar con precisión la política que definen implícitamente. Los valores Q iniciales, $Q_0(x,a)$, para todos los estados y acciones se suponen dados.

DQN

Una red Q profunda (DQN) es una red neuronal multicapa que, para un estado *s* dado, produce un vector de valores de acción $Q(s,\cdot;\theta)$, donde θ son los parámetros de la red [9]. En un espacio de estados de n dimensiones y un espacio de acciones que contiene m acciones, la red neuronal es una función de $\mathbb{R}^n$ a $\mathbb{R}^m$. Dos ingredientes importantes del algoritmo DQN son el uso de una red objetivo y el uso de la memoria de recuerdo.

La red objetivo, con parámetros $\bar{\theta}$, es la misma que la red en línea, salvo que sus parámetros se copian cada τ pasos de la red red en línea, de modo que $\bar{\theta}_t = \theta_t$, y se mantienen fijos en todos los demás pasos. Para la memoria de recuerdo [10], las transiciones observadas se almacenan durante algún tiempo y se muestrean uniformemente desde este banco de memoria para actualizar la red. Tanto la red objetivo como la memoria de recuerdo mejoran notablemente el rendimiento del algoritmo [11].

DQN mantiene un amplio historial de las experiencias más recientes [12], donde cada experiencia es una quíntuple $(s,\ a,\ s',\ r,\ T)$, que corresponde a un agente que realiza la acción a en el estado s, llega al estado s' y recibe la recompensa r; y T es un booleano que indica si s' es un estado terminal. Después de cada paso en el entorno, el agente añade la experiencia a su memoria de recuerdo.

Después de un pequeño número de pasos, el agente muestrea aleatoriamente un mini lote de su memoria de recuerdo sobre el que realizar sus actualizaciones de la función Q. La reutilización de experiencias anteriores en la actualización de una función Q se conoce como repetición de experiencias [10]. Sin embargo, mientras que la repetición de experiencias en RL se utilizaba normalmente para acelerar la copia de seguridad de recompensas, el enfoque de DQN de tomar muestras totalmente aleatorias de su memoria para usarlas en actualizaciones de mini lotes ayuda a decorrelacionar las muestras del entorno que, de otro modo, pueden causar un sesgo en la estimación de la función.

La última contribución importante es el uso de parámetros de red más antiguos, o "viejos", al estimar el valor Q para el siguiente estado en una experiencia y sólo actualizar los parámetros de red antiguos en intervalos discretos de muchos pasos. Este enfoque es útil para DQN, porque proporciona un objetivo de entrenamiento estable para la función de la red, y le da un tiempo

razonable (en número de muestras de entrenamiento) para hacerlo. En consecuencia, los errores en la estimación están mejor controlados.

Deep Q-Learning

Según [13] DQN puede considerarse una extensión del algoritmo clásico de Q-Learning [7] que utiliza una red neuronal profunda para aproximar la función acción-valor. Aunque las propiedades algorítmicas y estadísticas del algoritmo clásico de Q-Learning están bien estudiadas, el análisis teórico de DQN es un gran reto debido a sus diferencias en los dos aspectos siguientes.

En primer lugar, en los algoritmos de aprendizaje por refuerzo en línea basados en el gradiente temporal, la aproximación de la función acción-valor a menudo conduce a la inestabilidad. [14] demuestra que este es el caso incluso con la aproximación de la función lineal. La técnica clave para lograr la estabilidad en DQN es la memoria de recuerdo [11][10]. En concreto, se utiliza una memoria de repetición para almacenar la trayectoria del proceso de decisión de Markov (MDP).

En cada iteración de DQN, un mini lote de estados, acciones, recompensas y próximos estados se muestrean de la memoria de repetición como observaciones para entrenar la red Q, que aproxima la función acción-valor. La intuición detrás de la repetición de experiencias es lograr la estabilidad rompiendo la dependencia temporal entre las observaciones utilizadas al entrenar la red neuronal profunda.

En segundo lugar, además de la mencionada red Q, DQN utiliza otra red neuronal denominada la red objetivo para obtener un estimador insesgado del error cuadrático medio de Bellman utilizado en el entrenamiento de la red Q. La red objetivo se sincroniza con la red Q después de cada periodo de iteraciones, lo que conduce a un acoplamiento entre las dos redes. Además, aunque fijemos la red objetivo objetivo y nos centramos en la actualización de la red Q, el subproblema del entrenamiento de una red neuronal sigue siendo sigue siendo menos comprendido en la teoría.

De acuerdo a [15], este enfoque presenta varias mejoras con respecto al Q-Learning estándar. En primer lugar, cada paso de la experiencia se utiliza potencialmente en muchas actualizaciones de peso, lo que permite una mayor eficiencia de los datos. En segundo lugar, el aprendizaje directo a partir de muestras consecutivas es ineficiente debido a las fuertes correlaciones entre las muestras.

La aleatorización de las muestras rompe estas correlaciones y reduce la varianza de las actualizaciones. Por último, cuando se aprende, los parámetros actuales determinan la siguiente muestra de datos con la que se entrenan los parámetros. Por ejemplo, si la acción maximizadora es moverse a la izquierda, las muestras de entrenamiento estarán dominadas por muestras del lado izquierdo; si la acción maximizadora cambia a la derecha, la distribución de entrenamiento también cambiará. Por tanto, pueden surgir bucles de retroalimentación no deseados y el método podría quedarse atascado en un mínimo local o incluso divergir.

Con la repetición de la experiencia, la distribución del comportamiento se promedia en muchos de sus estados, lo que suaviza el aprendizaje y evita oscilaciones o divergencias en los parámetros. Obsérvese que cuando se aprende mediante la repetición de la experiencia, es necesario aprender fuera de la política porque nuestros parámetros actuales son diferentes a los utilizados para generar la muestra, lo que motiva la elección del Deep Q-Learning sobre el Q-Learning cuya comparación se muestra en la figura 2.5.

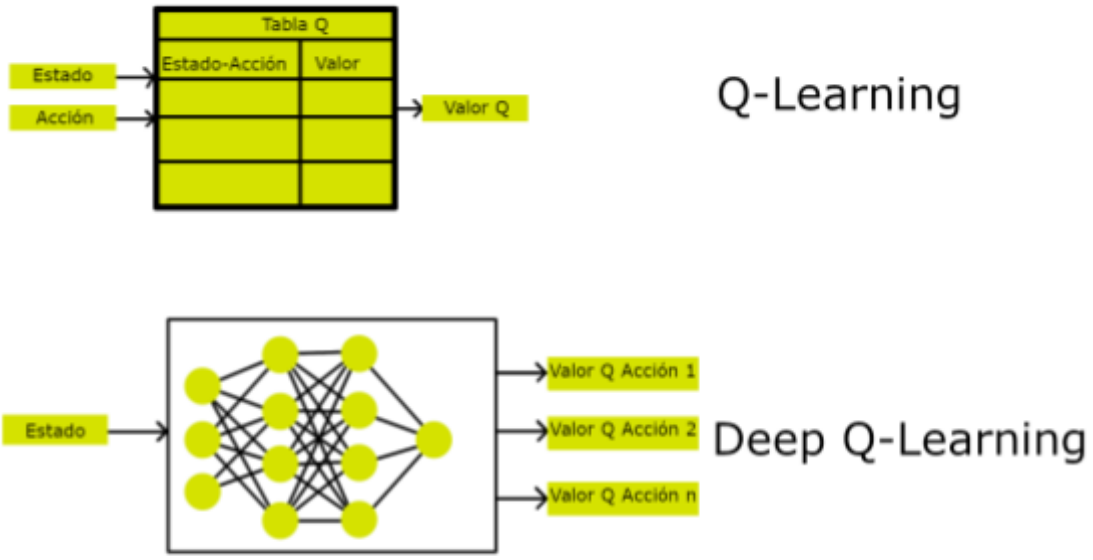

Figura 2.5: Comparación entre Q-Learning y Deep Q-Learning

2.3 PLATAFORMAS DE AGENTES: SPADE

SPADE es un middleware para sistemas multiagente que representa una evolución de los sistemas multiagente tradicionales mediante la incorporación de una cuidadosa selección de conceptos y tecnologías modernas en los ámbitos de los sistemas distribuidos, mensajería instantánea, sistemas asíncronos y sistemas sistemas abiertos [16]. Su modelo de agente es similar a los utilizados en otras plataformas. El modelo se basa internamente en algunas abstracciones simples y mecanismos.

El primero es el mecanismo de conexión, por el que cada agente se registra en SPADE utilizando un identificador único (cuyo formato es "nombre de usuario@servidor") y una contraseña. Después de registrarse, los agentes pueden crear uno o varios comportamientos, que son tareas independientes que ejecutan las acciones del agente.

Los comportamientos pueden ser de varios tipos, y cada tipo produce un patrón de ejecución particular diseñado para soportar un requisito de ejecución típico de los agentes en un sistema multiagente. En particular, SPADE soporta cinco tipos de comportamiento: cíclico, de una sola vez, periódico, de tiempo muerto y de máquina de estados finitos.

El tercer mecanismo principal es el despachador de mensajes que SPADE asocia a cada agente registrado. Este componente actúa como un cartero, redirigiendo cualquier mensaje entrante para el agente al comportamiento o comportamientos particulares que puedan estar esperando el mensaje, y retransmitiendo los mensajes salientes de cualquier comportamiento al sistema de comunicación de SPADE.

Entre las diversas tecnologías existentes, SPADE se basa en la selección de un determinado protocolo de comunicación como componente primordial del sistema multiagente, ya que puede aportar características muy valiosas a las entidades inteligentes, autónomas y sociales como los agentes. En concreto, SPADE incorpora XMPP (eXtensible Messaging and Presence Protocol), que es un protocolo abierto para la mensajería instantánea y la notificación de presencia.

XMPP es un protocolo basado en XML que permite a las entidades intercambiar mensajes (siendo estas entidades seres humanos, agentes, artefactos, etc.) y que también proporciona un mecanismo de notificación de presencia mediante el cual cualquier entidad puede tener una lista de otras entidades como contactos, y ser notificada cuando alguno de esos contactos cambia su estado (por ejemplo, cuando un contacto se conecta o desconecta, cuando está ocupado, etc.).

Además, XMPP se define como un protocolo extensible, lo que significa que muchas de sus características se proponen como extensiones (llamadas XEP); actualmente ofrece muchas extensiones para diferentes propósitos, y está abierto a las propuestas de la comunidad para hacer que el protocolo sea más flexible y útil. Por ello, XMPP está considerado como el protocolo estándar universal para la mensajería instantánea por entidades como el IETF o el W3C, y tiene un uso extendido en la industria. Whatsapp, Google Talk, Facebook Messenger o iMessage de Apple son algunos ejemplos de aplicaciones que utilizan XMPP, o alguna variante de este protocolo.

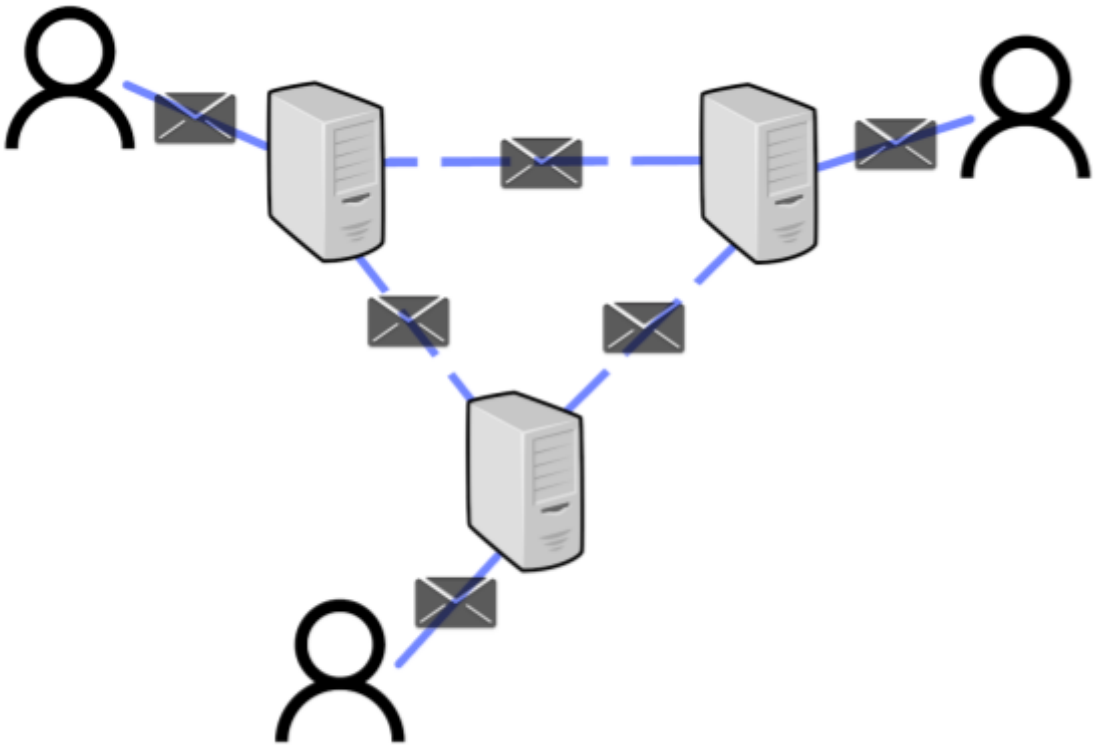

Figura 2.6: Arquitectura federada de servidores XMPP

XMPP proporciona una arquitectura de servidores abierta y federada, como se muestra en la figura 2.6 , mediante la cual cualquier servidor XMPP puede comunicarse con cualquier otro servidor XMPP en funcionamiento en Internet (del mismo modo que los servidores de correo SMTP) [16]. Al conectarse a dicha federación de servidores, SPADE permite que cualquier agente se comunique con cualquier otro agente, artefacto o humano en el mundo, llevando el concepto de un sistema multiagente abierto a un nuevo nivel. En particular, el soporte que SPADE requiere de un servidor XMPP puede configurarse ampliamente de tres maneras diferentes, dependiendo de los requisitos de la aplicación.

En primer lugar, una aplicación de sistema multiagente que se ejecute en SPADE puede configurarse para desplegar su propio servidor XMPP público (existen varias implementaciones de software de servidor XMPP de código abierto). Esta configuración hace que la aplicación sea autosuficiente al tiempo que permite a sus agentes comunicarse con otras entidades (agentes o no) conectadas a cualquier otro servidor XMPP en Internet.

Una segunda configuración, más sencilla, puede ser la de utilizar cualquiera de los servidores XMPP públicos existentes y de libre acceso en Internet. En este caso, los agentes de SPADE se registrarían directamente en este servidor público y luego se ejecutarían de forma natural sin ninguna otra infraestructura. Finalmente, una tercera configuración posible es desplegar un servidor XMPP privado, sin conexiones entre servidores, junto con la aplicación SPADE. En este caso, la aplicación se ejecutaría de forma privada, sin posibles conexiones de entidades externas.

El uso de este tipo de arquitectura para soportar sistemas multiagente ofrece varias ventajas. Probablemente la más importante es la flexibilidad: con la misma arquitectura, SPADE puede soportar desde sistemas multiagente muy grandes con varios servidores interconectados capaces de distribuir la carga de mensajería, hasta sistemas multiagente mucho más pequeños y ad hoc en los que se puede desplegar un único servidor adaptado para proporcionar la funcionalidad mínima requerida.

Además, aunque la arquitectura es por naturaleza distribuida, lo que proporciona las ventajas relacionadas de la distribución de la carga, la tolerancia a los fallos, etc., también puede proporcionar algunas ventajas típicas de los sistemas centralizados, como la notificación de presencia, el almacenamiento persistente en el lado del servidor, mecanismos de autenticación fuertes, etc.

Esta flexibilidad de soportar sistemas grandes o pequeños, centralizados o descentralizados, abiertos o privados, hace que SPADE pueda soportar sistemas multiagente que se adaptan mejor a los problemas abiertos comentados en la sección anterior. Entre otras consideraciones, esta arquitectura permite que el sistema multiagente sea elástico, ya que puede aumentar o disminuir su tamaño añadiendo nuevos servidores o eliminándolos bajo demanda, mientras está en funcionamiento.

Por último, una ventaja adicional derivada de esta arquitectura descentralizada es que SPADE proporciona a los agentes la capacidad de ser independientes de su ubicación física, es decir, de la dirección del ordenador donde se ejecutan. En muchas plataformas multiagente, la dirección física (IP) del ordenador donde se ejecuta un agente debe ser conocida por el resto de agentes para poder entregar con éxito los mensajes que envían a ese agente.

Esto se suele resolver incluyendo la dirección del ordenador donde se ejecuta el agente en el nombre o identificador del agente. Pero entonces, si el mismo agente quiere ejecutarse en un ordenador diferente, necesita un nuevo identificador, que también debe ser difundido a todos sus posibles interlocutores para ser accesible de nuevo. Esto se aborda en SPADE de una forma mucho más cómoda, ya que los agentes se identifican mediante el servidor XMPP en el que están registrados, no por el ordenador donde se ejecutan.

Así, mientras el servidor XMPP no cambie su dirección, los agentes pueden migrar de ejecutarse en un ordenador a otro, de forma totalmente transparente. Esto es similar, por ejemplo, al correo electrónico que permite a las personas recibir mensajes independientemente del ordenador en el que lean su correo electrónico. Esta independencia de la ubicación es una característica muy interesante en los sistemas sistemas multiagente de gran escala, en los que es habitual que cada vez que se ejecuta una aplicación multiagente, sus agentes pueden ejecutarse en un ordenador diferente.

Para finalizar se concluye con la determinación de la adecuación de todas las opciones seleccionadas, tanto por la parte del simulador como por parte de la plataforma de agentes inteligentes y el tipo de aprendizaje por refuerzo. Las tres opciones seleccionadas (CARLA, Spade y DQN) tienen en común la utilización del lenguaje de programación Python. Además de una API muy útil en cada caso que simplifica el proceso de integración en una única solución.

3. Desarrollo

En el presente capítulo se explican los pasos seguidos para la realización de este proyecto en el lenguaje de programación Python, incluyendo la instalación e importación de las librerías necesarias.

Empezaremos explicando como instalar un servidor XMPP y configurarlo correctamente para dar de alta a un agente de la librería de Python SPADE en él. A continuación detallaremos los pasos necesarios para que nuestro agente se conecte al servidor del simulador de CARLA como cliente. Por último, se expondrán los diferentes módulos que componen nuestro sistema, detallando las funciones que realiza cada uno de ellos.

3.1 CREACIÓN DE AGENTES EN SERVIDOR XMPP CON SPADE

Como eje principal de nuestro sistema tendremos un módulo agente que utilizará el resto de módulos en los que dividiremos las distintas herramientas que componen este proyecto. Para utilizar este módulo agente primero necesitaremos un servidor XMPP en el que pueda ejecutarse, en nuestro caso usaremos el software libre OpenFire.

La descarga e instalación de este software resulta sencilla y siguiendo los pasos del instalador eligiendo las opciones por defecto podemos conseguir un servidor XMPP perfectamente funcional. Una vez instalado debemos configurarlo correctamente. Nuestro agente necesita ser capaz de registrarse en el servidor pero el servidor no reconocerá a nuestro agente la primera vez que se conecte, por ello tenemos que habilitar el registro de cuenta por parte de los usuarios y la conexión anónima como se muestra en la figura 3.1.

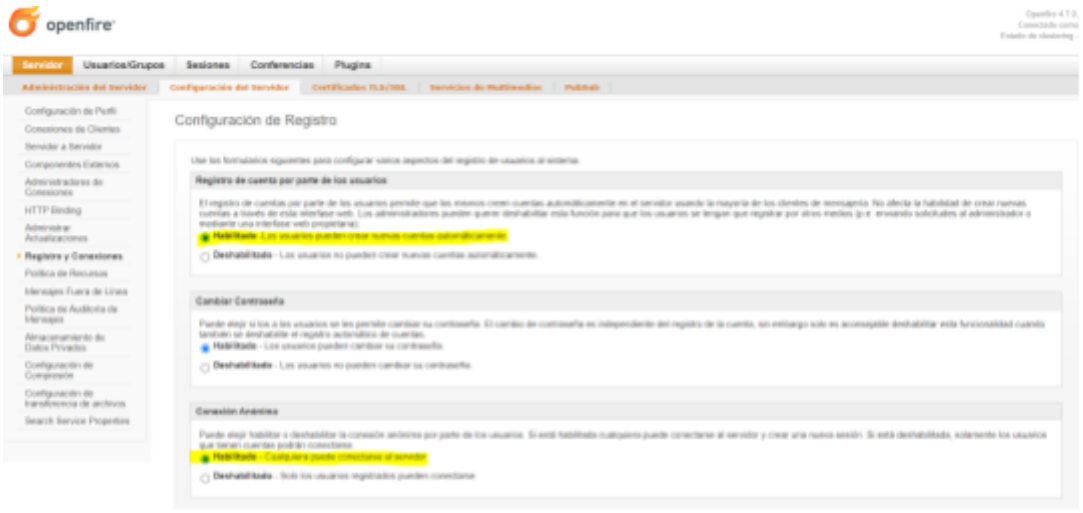

Figura 3.1: Panel de configuración de OpenFire

Una vez preparada la plataforma donde nuestro agente trabajará, el siguiente paso será crear el propio agente con SPADE, asignándole un nombre y contraseña para que pueda registrarse e iniciar sesión en nuestro servidor XMPP. En primer lugar necesitaremos importar la librería de SPADE. Para no cargar todas las utilidades de esta librería y ahorrar memoria solo importaremos el módulo *agent* y la clase *CyclicBehaviour* del módulo *spade.behaviour*. Después crearemos una clase a la que llamaremos *CarAgent* que heredará de la clase *Agent* perteneciente al módulo *agent*.

3.2 CONEXIÓN CON CARLA

Una vez creado el agente definiremos la función *setup* que se encargará de crear un comportamiento para nuestro agente. En este caso nos interesa que el comportamiento se repita mientras el agente esté activo por lo que optaremos por utilizar un comportamiento cíclico. Para crear este comportamiento crearemos una clase que herede de la clase *CyclicBehaviour* de SPADE y en el método *setup* le pasaremos una instancia de esta clase como parámetro a la función *add_behaviour* con lo que nuestro agente adquirirá este comportamiento.

```
class MyBehav(CyclicBehaviour):
```

```
async def setup(self):
    self.my_behav = self.MyBehav()
    self.add_behaviour(self.my_behav)
```

A continuación estableceremos qué acciones se tomarán al empezar el comportamiento. Aquí nuestro agente se conectará con el módulo CARLA previamente importado y creará una instancia de la clase CarlaEnv, cuyo constructor realizará la conexión con el servidor de CARLA. Para conectarnos llamaremos al constructor de la clase *Client* del módulo Python *carla* pasándole la dirección IP y puerto que utilice el servidor del simulador.

Al tener el servidor en local utilizaremos la dirección IP 127.0.0.1 y el puerto utilizado será el 2000. En adición pondremos un límite de tiempo para que se establezca de la conexión de modo que si en dos segundos no se ha logrado se entenderá que ha habido un error o no es posible conectar con el servidor por lo que la ejecución se abortará.

```
self.client = carla.Client('127.0.0.1', 2000)
self.client.set_timeout(2.0)
```

3.3 ESTRUCTURA DEL SISTEMA

Para construir nuestro sistema lo hemos dividido en cuatro módulos: el agente, el simulador CARLA, la red neuronal de entrenamiento por refuerzo DQN y la configuración. El módulo agente se encarga de coordinar al simulador y la red neuronal utilizando los parámetros del módulo de configuración, de esta manera podemos centrarnos en implementar las funcionalidades de cada módulo por separado para después utilizarlas en el agente.

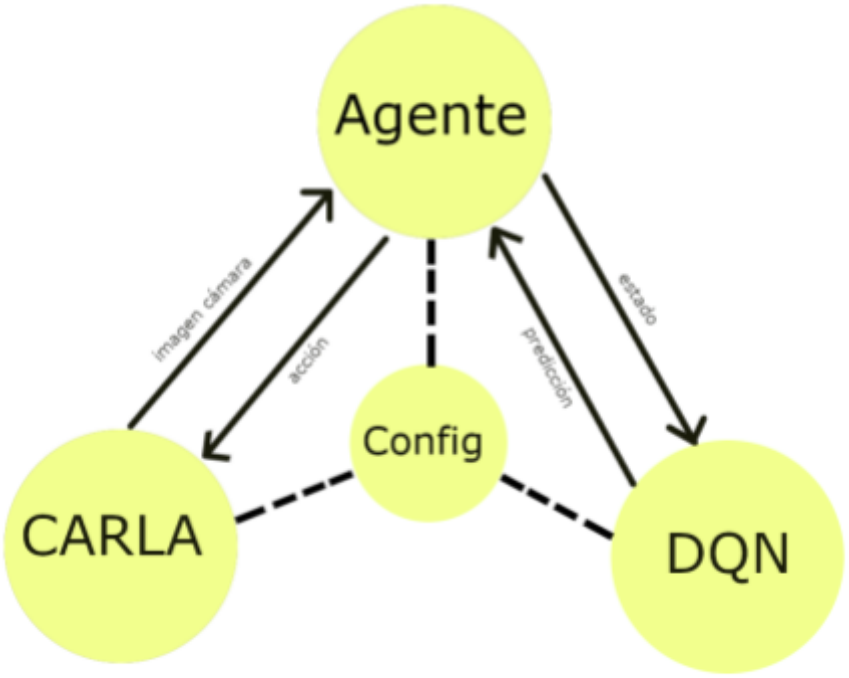

Figura 3.2: Esquema del sistema

Módulo Config

Este será un archivo donde almacenaremos el valor de las variables más relevantes de nuestro sistema como el número de acciones de las que dispondrá nuestro vehículo, el tamaño de imagen que servirá de entrada para nuestra red neuronal, el número de episodios y los segundos que durará cada uno. De este modo podremos cambiar las características de cada ejecución fácilmente y de forma centralizada.

Módulo CarlaEnv

Este módulo se encargará de controlar todos los aspectos de la simulación relacionada con los vehículos, sensores y el mundo. Para ello tenemos una clase CarlaEnv cuyo constructor realizará la conexión con el servidor local donde se ejecuta el simulador CARLA y almacenará la instancia del mundo simulado y la biblioteca de planos para su posterior uso junto con una lista de los actores que genere. Además cuenta con diversas funciones entre las que destacan *reset* y *step*.

La función *reset* hace uso de las funciones *destroy_all_actors*, la cual borra todos los objetos de la simulación que nuestro agente haya generado aplicando el comando *DestroyActor* a cada uno de los actores creados, la función *gen_vehicle*, que genera el vehículo que utilizaremos, y la función *add_sensor*, que crea un sensor y lo acopla a nuestro vehículo previamente generado, el sensor que se genere dependerá del nombre que se le pase como único parámetro a esta función. De esta forma la función *reset* nos ayudará a que el entorno de simulación vuelva a un estado inicial para poder repetir las pruebas y realizar el entrenamiento por refuerzo.

La función *step* recibe dos argumentos: la acción a tomar y la distancia entre nuestro vehículo y el objetivo. El primero será un número entre cero y el número de acciones que tenga nuestro agente menos uno, que servirá para indicarle al vehículo que movimiento debe realizar especificando la aceleración, el frenado y el giro de las ruedas. El caso estándar contempla cuatro posibles acciones que permiten acelerar en línea recta, frenar y acelerar girando las ruedas a derecha e izquierda. El giro de las ruedas es representado por un número entero en el

rango [-1, 1] de manera que el 1 indica giro máximo a la derecha y -1 giro máximo a la izquierda donde el ángulo de giro máximo depende de cada coche.

Una vez ordenado al vehículo que movimiento realizar en el presente ciclo de ejecución, pasaremos a analizar el estado del coche y a calcular la recompensa en función de varios factores. Por último la función devolverá la imagen de la cámara RGB como un array de numpy, la velocidad del vehículo, la recompensa final, el estado del episodio(si ha terminado o no) y el tipo de reinicio de la simulación, este último nos ayudará a analizar los resultados del entrenamiento para observar si el episodio terminó por límite de tiempo, colisión o éxito al alcanzar el objetivo.

Módulo DQNEnv

En este módulo gestionaremos la red neuronal que utilizaremos para realizar el entrenamiento por refuerzo. Para crearla partiremos de un modelo con la arquitectura Xception que nos proporciona la librería de keras e iniciaremos los pesos de forma aleatoria. Como tamaño de entrada usaremos una matriz de tamaño T donde $T = Ancho * Alto * 3$ y Ancho y Alto hacen referencia al tamaño de imagen de nuestra cámara RGB mientras que el 3 se debe a que cada píxel de nuestra imagen se codifica en tres valores numéricos que representan el color que tiene ese píxel a partir de la combinación de los tres colores básicos: el rojo, el verde y el azul. El valor de cada uno de los colores escala del 0 al 255, del más oscuro al más brillante.

A continuación agregaremos el resto de las capas a nuestra red neuronal, para este caso utilizaremos GlobalAveragePooling2D, Flatten y por último Dense como capa de salida, la cual tendrá un número de neuronas de salida que deberán coincidir con las posibles acciones que tenga nuestro vehículo en el simulador y una función de activación lineal. Finalmente configuraremos nuestra red con una función de pérdida de error cuadrático medio y un optimizador Adam con tasa de aprendizaje de 0.001.

Una vez creado nuestro modelo de red neuronal lo duplicaremos para obtener lo que llamaremos nuestro modelo objetivo, este servirá para comparar salidas con el modelo original y entrenarlo con las mejores predicciones de ambos modelos. Por otro lado necesitaremos una función que dado un estado de nuestro simulador, es decir, una imagen de la cámara RGB, nos devuelva la acción a tomar.

Esta acción será la predicción que realice nuestra red sobre las neuronas que tenga en su capa de salida, para esto la librería de keras con la que diseñamos nuestra red nos proporciona un conveniente método al que solo debemos pasarle como parámetro la imagen de nuestra cámara RGB y nos devuelve un conjunto de predicciones ordenadas de mayor a menor probabilidad de acierto, por lo que nuestra función se quedará con la primera de este conjunto y la devolverá.

La salida que produzca nuestra red neuronal tendrá consecuencias en el estado del entorno en el que la usaremos, estas consecuencias pueden ser buenas o malas en mayor o menor medida, esta medida será la recompensa que obtendrá nuestra red por cada predicción. Dependiendo de la recompensa producida por la última decisión tomada por nuestra red deberemos modificar sus pesos para adaptarla, de forma que con la misma entrada produzca una "mejor" decisión, siguiendo el concepto de aprendizaje por refuerzo.

Para este fin la librería de keras nos proporciona la función *fit*, la cual se encargará de entrenar nuestra red neuronal en función de la entrada que le proporcionaremos y la salida que debería obtener. Este entrenamiento lo realizaremos cada 20 episodios para que tengamos suficientes datos en la "memoria de recuerdo", de forma que extraeremos una muestra aleatoria de esta memoria de *n* transiciones, siendo *n* un valor establecido en nuestro módulo de

configuración, y obtendremos las predicciones que realicen nuestro modelo y el modelo objetivo sobre los estados actuales y nuevos de cada transición respectivamente.

Nuestra memoria de recuerdo es una lista de transiciones, donde cada transición es una tupla de la forma $(estado\_actual, acción, recompensa, nuevo\_estado, hecho)$, de modo que representa el pasar de un estado a otro mediante una acción que, dependiendo del nuevo estado, genera una recompensa y también se indica si ha terminado el episodio. Recorreremos la lista de transiciones del modelo objetivo y para cada una de ellas, calcularemos el nuevo valor Q[7] que asignaremos a la predicción de nuestro modelo cuya acción coincida con la de la transición del modelo objetivo.

Utilizando una lista formada por el estado_actual de cada transición del modelo objetivo y otra lista con las predicciones de nuestro modelo modificadas como entrada y salida esperada, entrenaremos nuestro modelo pasándole estas dos listas como argumentos a la función *fit*, junto con tres argumentos más para que no baraje los datos, no muestre el estado de la ejecución por pantalla e indicarle el tamaño de batch a utilizar en el entrenamiento.

El tamaño de batch puede influir en el resultado del entrenamiento al ser el número de "recuerdos" que la red usará para aprender, pero no tiene sentido barajar los datos porqué aunque cambies el orden o estructura de este conjunto de experiencias al final se quedará igual, solo supone un gastó de recursos computacionales. Lo mismo se aplica para la visualización por pantalla del estado del entrenamiento, la duración de estos no será tan alta como para saber si algo ha ido mal o en qué momento ha ocurrido. Si usáramos tamaños de batch más grandes la duración del entrenamiento incrementaría considerablemente y entonces sería recomendable tener un seguimiento para comprobar que se está realizando correctamente, pero en nuestro caso solo restará recursos a nuestro agente.

Esta función nos devolverá un objeto History cuyo atributo History.history almacenará métricas del resultado del entrenamiento como la precisión o la pérdida. El último paso del entrenamiento será actualizar los pesos del modelo objetivo con los pesos de nuestro modelo entrenado.

Por último, este módulo contará con una función que permitirá guardar nuestro modelo entrenado junto con sus pesos y otra para cargarlos. Los archivos donde se almacenarán utilizarán el formato HDF5 con el que se puede guardar la configuración del modelo, sus capas y los pesos de cada capa.

Módulo Agente

Aquí se lanzará nuestro agente y servirá como programa principal el cual ejecutaremos a la hora de poner en funcionamiento nuestro sistema. Primordialmente necesitaremos importar las librerías de Python spade y numpy junto con nuestros módulos Config, CarlaEnv y DQNEnv. También importaremos librerías como csv y matplotlib para guardar los resultados del entrenamiento y graficarlos, junto con tqdm y termcolor para mejorar la visualización durante el entrenamiento. La ejecución de este módulo se resumirá en lanzar el agente y esperar en un bucle infinito del que solo saldrá si el comportamiento del agente finaliza o realizamos una interrupción por teclado con la combinación de teclas Ctrl + C.

Una vez lanzado el agente, este inciará su comportamiento cuyos pasos iniciales hemos explicado anteriormente, pero falta añadir que al comienzo del comportamiento nuestro agente también se conectará con el módulo DQNEnv y se inicializará el estado de la simulación. En el método *run* del comportamiento prepararemos la ejecución principal del agente, primero prepararemos los archivos donde guardaremos los resultados del entrenamiento y, a continuación, entraremos en un bucle que tendrá tantos ciclos como episodios se vayan a realizar.

Cada episodio empezará reiniciando el entorno del simulador y acto seguido entrará en un bucle infinito, en este segundo bucle el agente obtendrá la acción a enviar al módulo CarlaEnv, esta acción podrá ser la que devuelva el módulo DQNEnv o una aleatoria dependiendo de un factor de aleatoriedad que se irá reduciendo en cada episodio de modo que cuantos más episodios haga menor será la probabilidad de que la acción elegida sea aleatoria.

Esta aleatoriedad evita que el modelo pueda quedarse estancado en un óptimo local dado que los pesos de la red neuronal sufren modificaciones muy leves y se tardaría cientos de episodios en lograr que un mismo estado produzca una salida distinta. Al tener decisiones aleatorias la memoria de recuerdo almacenará transiciones más diversas de las que podrá aprender nuestro modelo.

Después de conseguir la acción, el agente obtendrá la distancia que le falta al vehículo para alcanzar el objetivo y le pasará ambos datos al módulo del simulador invocando a la función *step*. Con el nuevo estado del entorno, la recompensa y la condición de finalización del episodio, se actualizará la memoria de recuerdo y el nuevo estado pasará a ser el estado actual.

Este ciclo se repetirá hasta que la función *step* devuelva como verdadera la condición de finalización del episodio, en cuyo caso saldremos del bucle mediante un *break* y el agente pasará a registrar los resultados del entrenamiento de ese episodio y multiplicará nuestro *epsilon* o probabilidad de tomar una acción aleatoria por una tasa de decrecimiento, ambos valores tomados del módulo de configuración.

Finalmente, si el número del episodio es un múltiplo de veinte efectuaremos el entrenamiento del modelo a partir de muestras aleatorias de la memoria de recuerdo descrito previamente. De esta forma finalizaría el episodio y se repetiría el proceso para los episodios restantes, una vez alcanzado el último episodio guardaremos el modelo y sus pesos en un fichero y procederemos a matar el comportamiento del agente, haciendo que la ejecución del programa finalice.

4. Implementación

A partir de la arquitectura básica se han desarrollado diferentes versiones en función del tipo de sensores que emplea el sistema.

La primera versión o versión 0 servirá como una prueba sencilla a modo de aproximación al aprendizaje por refuerzo para comprobar su funcionamiento, en esta versión el agente sólo podrá avanzar o frenar su vehículo y dispondrá de la cámara RGB como único sensor.

En la segunda versión o versión 1 el agente tendrá mayor capacidad de maniobrabilidad con la posibilidad de girar las ruedas del vehículo a derecha o izquierda, junto con la adición de un sensor de colisión y dos sensores de detección de obstáculos.

La tercera versión o versión 2 incluirá el sensor LiDAR y será la versión con la que experimentaremos para obtener el modelo de aprendizaje por refuerzo óptimo. Por último, la cuarta versión o versión final utilizará el modelo obtenido de la tercera versión para realizar un entrenamiento prolongado.

4.1 VERSIÓN 0

En esta primera versión del sistema pretendemos que nuestro modelo aprenda a ir hacia delante, para ello la red neuronal solo tendrá dos neuronas en su capa de salida, de modo que en la función *step* del módulo de CARLA contemplaremos dos posibles valores para el parámetro de la acción elegida: cero y uno. Si la acción vale cero le ordenaremos al vehículo que acelere y en caso contrario frenará. Puesto que el movimiento que realizará el coche será tan simple como desplazarse en una sola dirección a lo largo de una carretera, solo le agregaremos la cámara RGB como único sensor y configurada para capturar imágenes de 320x240 píxeles cada segundo con un campo de visión de 110 grados.

Figura 4.1: Nuestro vehículo

El objetivo que deberá cumplir nuestro agente en esta etapa del proyecto es conducir el vehículo que se muestra en la figura 4.1 y alcanzar un punto a diez metros en un tiempo igual o inferior a diez segundos. Cada llamada que nuestro agente haga a la función *step* será un paso

del episodio, en cada paso calcularemos la velocidad del vehículo en kilómetros por hora siguiendo la fórmula:

$$velocidad = 3,6 * \sqrt{v_x^2 + v_y^2 + v_z^2} \quad (7)$$

donde v es el vector velocidad que nos devuelve el método *get_velocity* del actor que corresponde a nuestro vehículo. Una vez obtenida la velocidad le daremos un valor inicial a la recompensa que devolverá este paso. Los valores que podrá tomar la recompensa o modificarla tienen tres categorías: mínimo, intermedio y máximo con valores de -200, 100 y 200 respectivamente. Si la velocidad se sitúa entre cinco y veinte kilómetros por hora la recompensa tomará el valor intermedio, en caso contrario se le asignará el valor mínimo.

A continuación comprobaremos a qué distancia se encuentra nuestro vehículo del objetivo y si esta es mayor a un metro le sumaremos a la recompensa actual la recompensa mínima dividida por la distancia al objetivo de modo que cuanto más cerca esté de alcanzarlo mayor será la recompensa, en caso contrario consideramos que ha llegado al punto objetivo y le sumaremos a la recompensa actual el valor máximo además de indicar que el episodio ha finalizado. Por último observaremos el tiempo que ha transcurrido desde el inicio del episodio gracias a la librería time de Python, si este tiempo es igual o superior a diez segundos finalizaremos el episodio y le sumaremos a la recompensa actual el valor de recompensa mínimo.

4.2 VERSIÓN 1

Para la segunda versión de nuestro sistema partiremos de la primera versión agregándole mayor complejidad, esto implica que el comportamiento de esta nueva versión contendrá el de la anterior y lo ampliará. Para empezar ahora nuestra red neuronal tendrá cuatro neuronas en su capa de salida lo que supone cuatro posibles acciones, estas serán las contempladas en el caso estándar, es decir, acelerar al 70% con las ruedas rectas, frenar al 70% con las ruedas rectas, acelerar al 70% con las ruedas giradas un ángulo α, que será el 50% de su ángulo de giro máximo, a la derecha y acelerar al 70% con las ruedas giradas α grados a la izquierda.

```
if action == 0:
    self.vehicle.apply_control(carla.VehicleControl(throttle=0.7, brake=0))
elif action == 1:
    self.vehicle.apply_control(carla.VehicleControl(throttle=0, brake=0.7))
elif action == 2:
    self.vehicle.apply_control(carla.VehicleControl(throttle=0.70, steer=0.5, brake=0))
elif action == 3:
    self.vehicle.apply_control(carla.VehicleControl(throttle=0.70, steer=-0.5, brake=0))
```

Figura 4.2: Movimiento del coche en función de la acción

Puesto que ha aumentado la complejidad de los movimientos que puede realizar nuestro vehículo mostrados en la figura 4.2, es posible que ahora colisione con otro objeto de la simulación, por ello agregaremos un detector de colisión y cuando en la función *step* se detecte una colisión finalizaremos el episodio y asignaremos a la recompensa el valor mínimo multiplicado por tres, así le indicaremos a la red neuronal que la mayor penalización es debida a las colisiones y que priorice evitarlas.

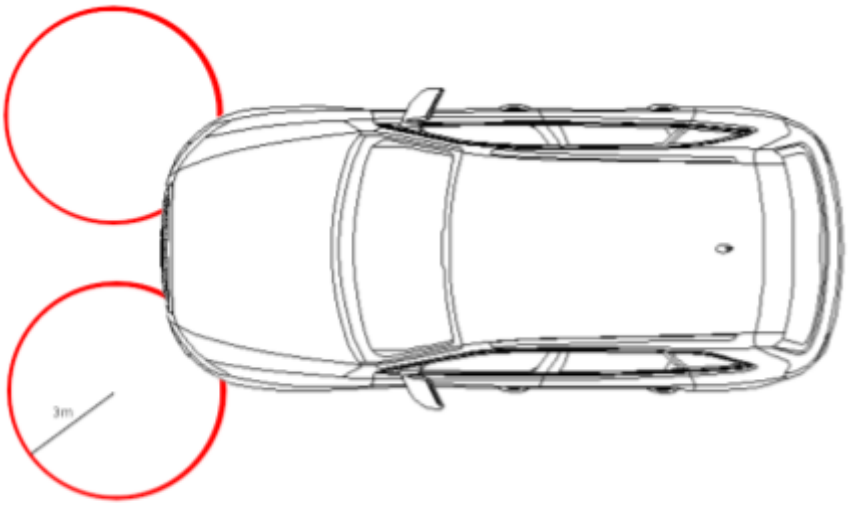

Figura 4.3: Sensores de obstáculos acoplados a nuestro coche

Además le acoplaremos a nuestro coche dos sensores de obstáculos en la parte delantera, uno a cada lado como se muestra en la figura 4.3. Estos sensores detectarán cualquier objeto dentro de un radio de tres metros cada segundo. Si se detecta algún obstáculo penalizaremos al agente sumándole a la recompensa actual el valor mínimo dividido por la distancia a la que se encuentre el obstáculo, de forma que cuanto más cerca esté el obstáculo mayor será la penalización que sufra el agente.

De esta forma, ahora nuestro agente tiene la posibilidad de cambiar de dirección y debe aprender a llegar al objetivo sin chocar con ningún obstáculo, es decir, a ir en línea recta, a diferencia de la versión anterior, en la cual no podía girar y su objetivo era simplemente aprender a acelerar para alcanzar la meta.

4.3 VERSIÓN 2

En esta tercera y última versión le agregamos a nuestro sistema el elemento final, un sensor LiDAR acoplado a nuestro coche. Los sensores LiDAR, también conocidos como escáneres láser, permiten la adquisición de datos de alta resolución (densidad de puntos de hasta 104 puntos/m^2) y una alta precisión (desviación estándar de 1 cm a 100 m) de la superficie del suelo[17].

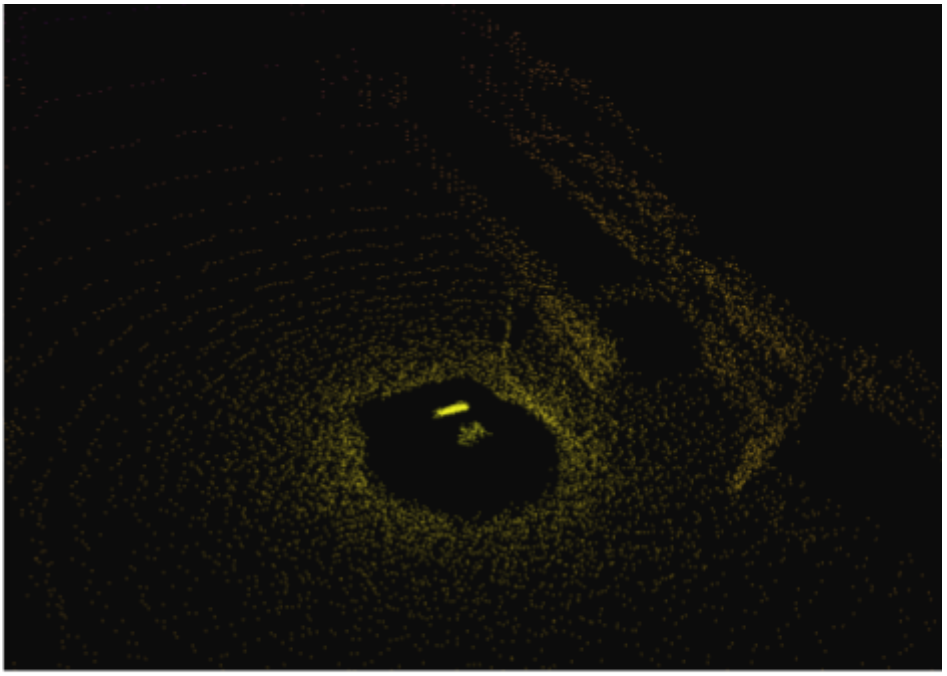

Figura 4.4: Nube de puntos del LiDAR

Estos sistemas permiten obtener las coordenadas (X, Y y Z) de los puntos de una superficie a gran velocidad (más de 222.000 mediciones por segundo) desde una distancia de adquisición considerable (hasta 6.000 m) formando una nube de puntos como se aprecia en la figura 4.4. Este sensor ha revolucionado la adquisición de los parámetros de los taludes rocosos que desempeñan un papel clave en la estabilidad global y local, incluyendo la orientación el espaciado, la persistencia y la rugosidad de las discontinuidades.

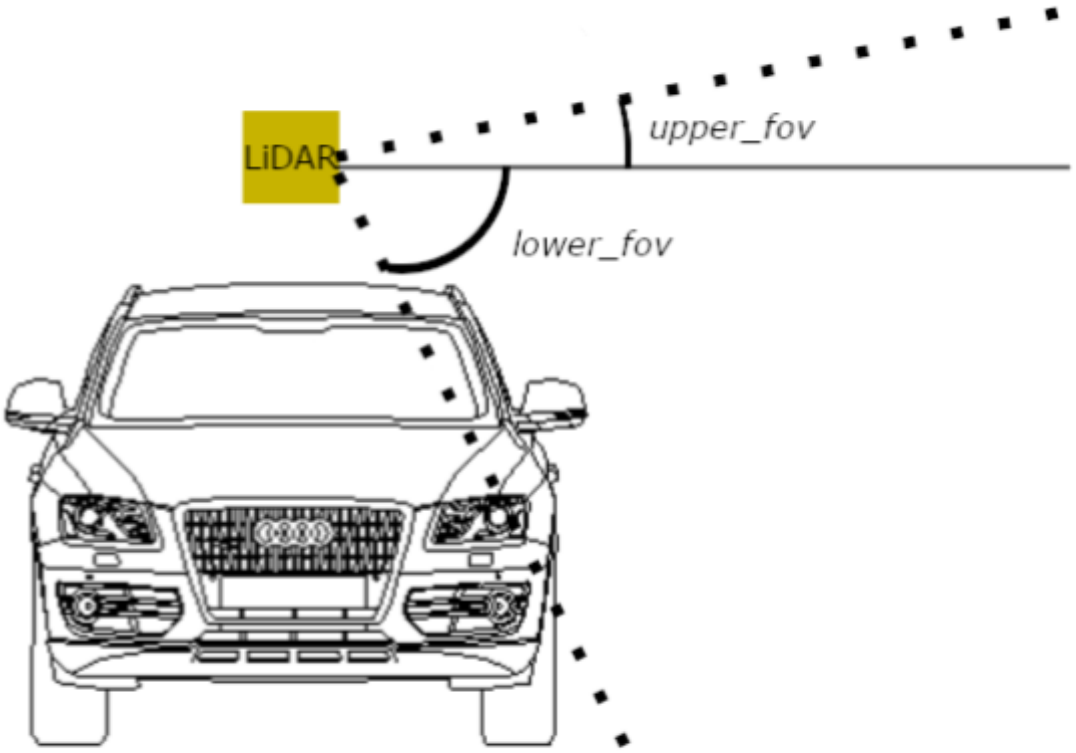

Figura 4.5: *upper_fov* y *lower_fov* del LiDAR

Acoplaremos este nuevo sensor a nuestro coche y lo configuraremos para que realice 40 giros por segundo con un rango de detección de 20 metros. En cada segundo escaneará 100.000 puntos y tendrá un campo de visión de 180 grados, no nos interesa escanear lo que haya detrás de nuestro vehículo puesto que no hemos implementado una acción para ir marcha atrás. También podemos configurar los ángulos máximo (*upper_fov*) y mínimo (*lower_fov*) que pueden formar los rayos del LiDAR respecto al plano horizontal como se muestra en la figura 4.5.

Los datos que queremos obtener del LiDAR servirán para detectar obstáculos a la altura de nuestro coche, por ello estableceremos el *upper_fov* a 15 grados y el *lower_fov* a -30 grados de forma que los puntos quedarán más concentrados en la mitad inferior del sensor que estará unos 50 centímetros por encima del techo de nuestro coche.

Para gestionar los datos que recopilemos del LiDAR utilizaremos una función *callback* que se ejecutará cada vez que el sensor detecte algo. Esta función recibirá los puntos detectados como argumento. Estos puntos vendrán como un objeto medición, que se compone de objetos detección. Cada detección tiene dos atributos, el punto y la intensidad. La intensidad representa la intensidad de la luz que devuelve la superficie al recibir el rayo del LiDAR. El punto es un objeto posición cuyos atributos son las coordenadas X, Y, y Z de ese punto respecto al sensor, es decir, el origen de coordenadas o punto (0,0,0) está en el centro del LiDAR.

Dicho esto, a la hora de analizar los puntos obtenidos por el LiDAR nos enfrentamos a tres problemas. Primero, el suelo también es escaneado. Esto significa que nuestro agente interpretará que hay un objeto próximo a nuestro coche y sufrirá una penalización en consecuencia aunque ese objeto resulte ser la propia carretera y esto no debería pasar. Para evitarlo eliminaremos de cada detección los puntos cuyo valor en su coordenada Z sea inferior o igual a -1.

Debido a que el LiDAR se sitúa a unos 150 centímetros del suelo lo que detectamos como carretera tiene una Z de -1,5 aproximadamente, por ello consideraremos un obstáculo todo punto por encima de los 50 centímetros de la superficie de la carretera dando como resultado una nube de puntos como se observa en la figura 4.6.

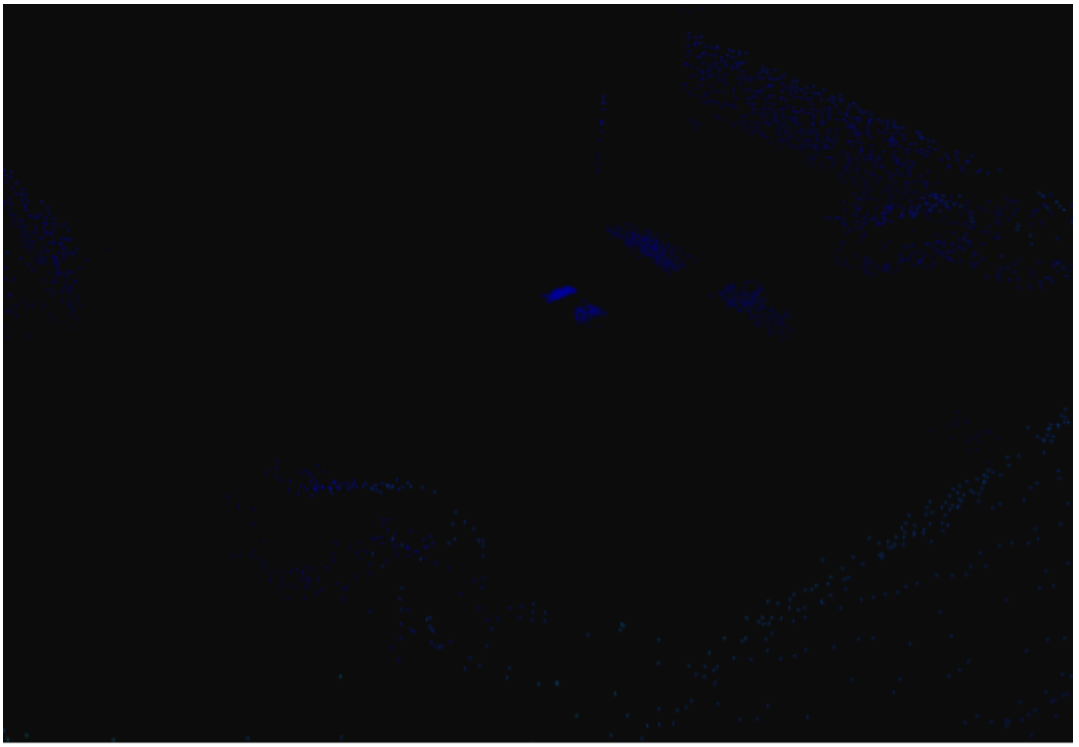

Figura 4.6: Nube de puntos del LiDAR tras eliminar el suelo

El segundo problema se debe a los rayos del LiDAR con un ángulo cercano al *lower_fov*, estos rayos detectan al propio coche y nos enfrentamos al mismo riesgo del primer problema de ser penalizados por haber una detección muy próxima al sensor. Para solucionarlo crearemos un cilindro invisible de altura infinita alrededor del coche dentro del cual los puntos detectados serán descartados.

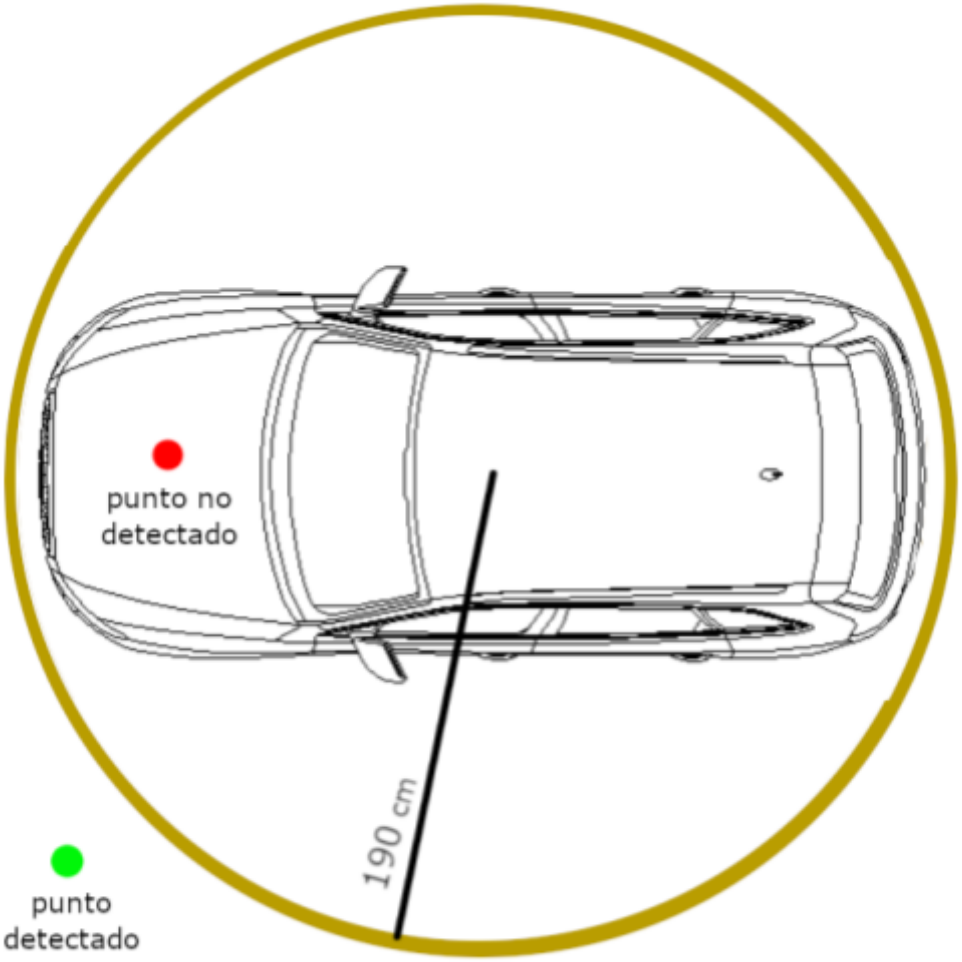

Figura 4.7: Radio de detección del LiDAR

Al tener altura infinita cualquier punto del cilindro será descartado sin importar su coordenada Z por lo que solo deberemos calcular la distancia entre el punto detectado y el centro de nuestro LiDAR usando sus coordenadas X e Y como se observa en la figura 4.7. Tras realizar varias pruebas con el coche que utilizamos en las simulaciones hemos concluido que el radio mínimo que debe tener este cilindro para que no detecte al propio coche es de 190 centímetros con lo que obtenemos una nube de puntos por parte del LiDAR como la mostrada en la figura 4.8.

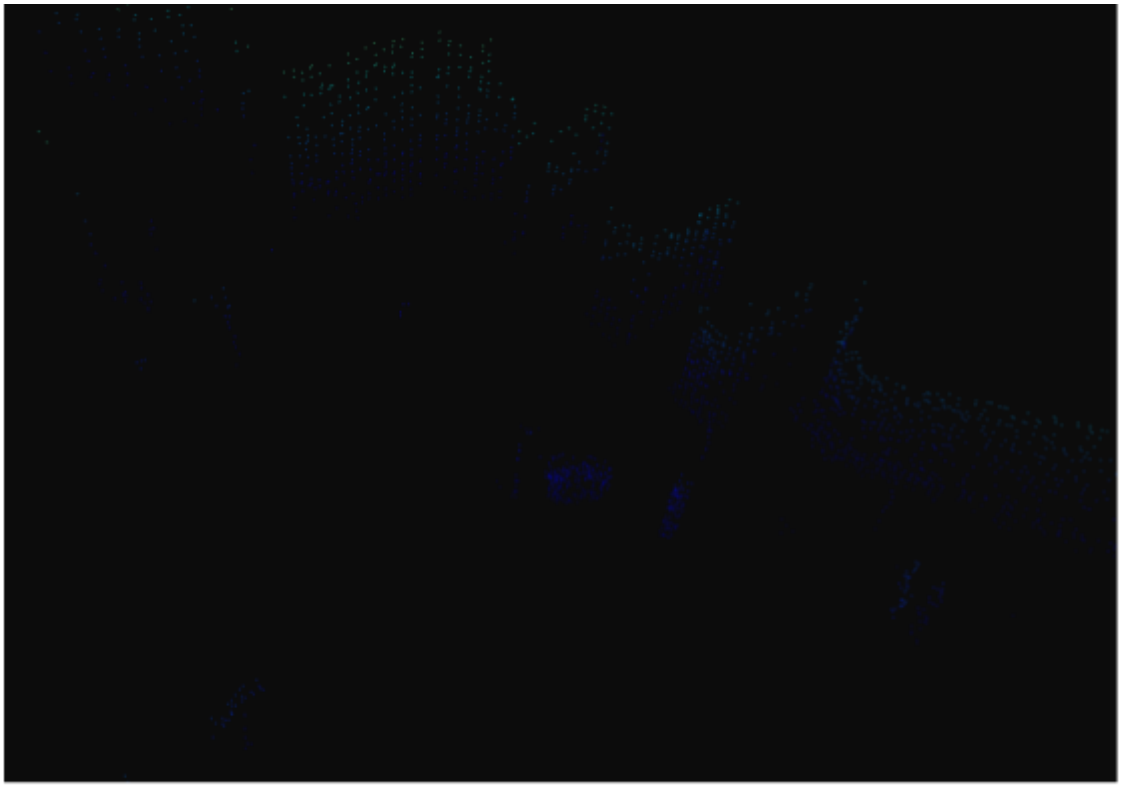

Figura 4.8: Nube de puntos del LiDAR tras eliminar el suelo y el coche

Por último, el tercer problema recae en el hecho de que los puntos sean relativos a la posición del LiDAR, esto supone que las distancias calculadas serán entre el punto detectado y el sensor y no entre el punto y nuestro coche.

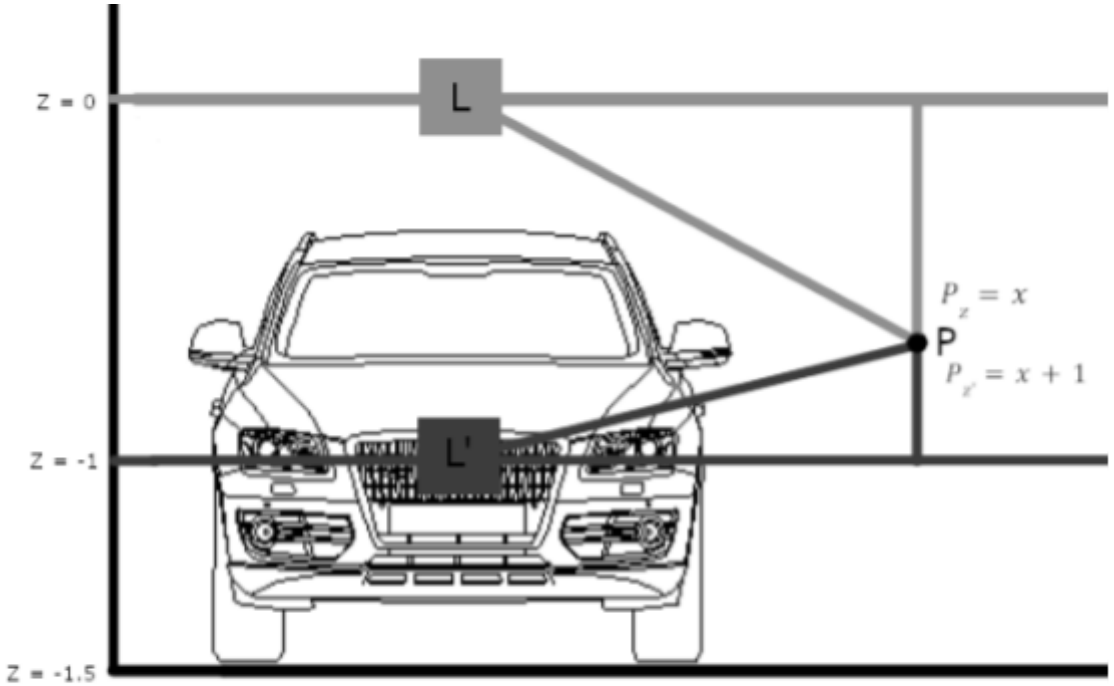

Figura 4.9: Distancia relativa al LiDAR

Para mejorar la precisión de los cálculos de las distancias entre nuestro vehículo y los posibles obstáculos y así que nuestro agente obtenga recompensas y penalizaciones justas, simularemos que el lidar se encuentra en el centro del coche como se muestra en la figura 4.9. Para ello tomaremos ventaja del propio problema, esto es, como la distancia entre dos puntos se basa en la diferencia de las coordenadas del origen y el destino y el origen es el punto (0,0,0) la distancia resultante queda de la siguiente forma:

$$Distancia = \sqrt{(x-0)^2 + (y-0)^2 + (z-0)^2} = \sqrt{x^2 + y^2 + z^2} \quad (8)$$

donde *x, y* y *z* son las coordenadas del punto detectado. Para simular que el sensor se encuentra en el centro del coche y así que las distancias sean entre el punto y el vehículo, bastará con poner la coordenada Z del LiDAR en la posición -1 con lo que la fórmula para calcular las distancias quedará de la siguiente forma:

$$Distancia = \sqrt{(x^2 - 0) + (y^2 - 0) + (z-(-1))^2} = \sqrt{x^2 + y^2 + (z+1)^2} \quad (9)$$

Una vez calculadas las distancias entre el coche y los puntos detectados las filtraremos para quedarnos con aquellas inferiores a cuatro metros y si hay una o más nos quedaremos con la menor de esas distancias como se muestra en la figura 4.10 y modificaremos la recompensa actual sumándole el valor de recompensa intermedio multiplicado por la de esa distancia.

```
# Eliminamos puntos pertenecientes al suelo y al propio coche
points = points[points[:, 2] > -1, :]
points = points[np.sqrt(points[:, 0]**2 + points[:, 1]**2) > 1.9, :]
# Calculamos vector de distancias
X = points[:, 0]
Y = points[:, 1]
Z = points[:, 2]
D = np.sqrt(X**2 + Y**2 + (Z + 1)**2)
# Nos quedamos con distancias menores a 4 metros
D = D[D[:] < 4]

if D.shape[0] > 0:
    self.obj_prox = np.amin(D)
```

Figura 4.10: Código para tratar los datos del LiDAR

5. Validación

Una vez preparado el sistema nos disponemos a ejecutarlo. Ejecutaremos nuestro sistema con diferentes configuraciones para experimentar con los parámetros que consideramos que influyen en el aprendizaje de nuestra red neuronal y observar qué configuración nos proporciona un mejor aprendizaje.

5.1 VERSIÓN 0

En primer lugar hemos lanzado a entrenar la versión cero para realizar una primera aproximación con el caso más sencillo y experimentar con cuantos episodios sería capaz nuestro agente de aprender a acelerar, es decir, cuánto tardará en modificar los pesos de las neuronas de su red neuronal para que la elección de nuestra red sea la de avanzar.

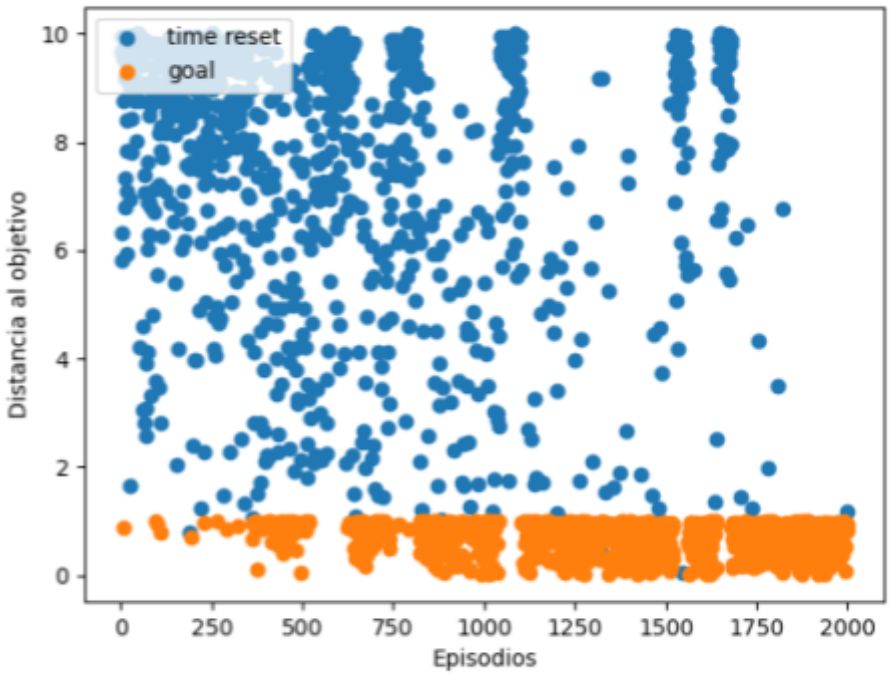

Figura 5.1: Distancias del entrenamiento de la versión 0

Para esta prueba hemos utilizado una memoria de recuerdo de 5.000 transiciones, tamaño de batch de 32 transiciones y tras 2.000 episodios con un tiempo límite de diez segundos por episodio podemos observar en la figura 5.1 como nuestro agente consigue acortar la distancia hasta la meta, con reinicios debidos al límite de tiempo pintados de azul y en los últimos 250 episodios logra el objetivo prácticamente en cada uno de ellos, representado por los puntos pintados en naranja.

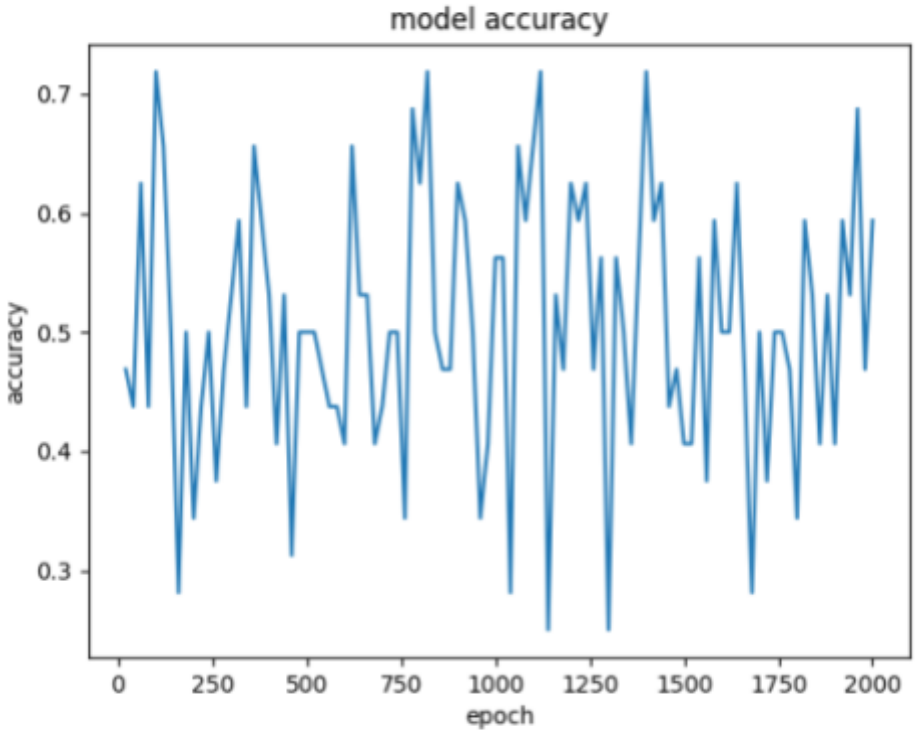

Figura 5.2: Precisión del entrenamiento de la versión 0

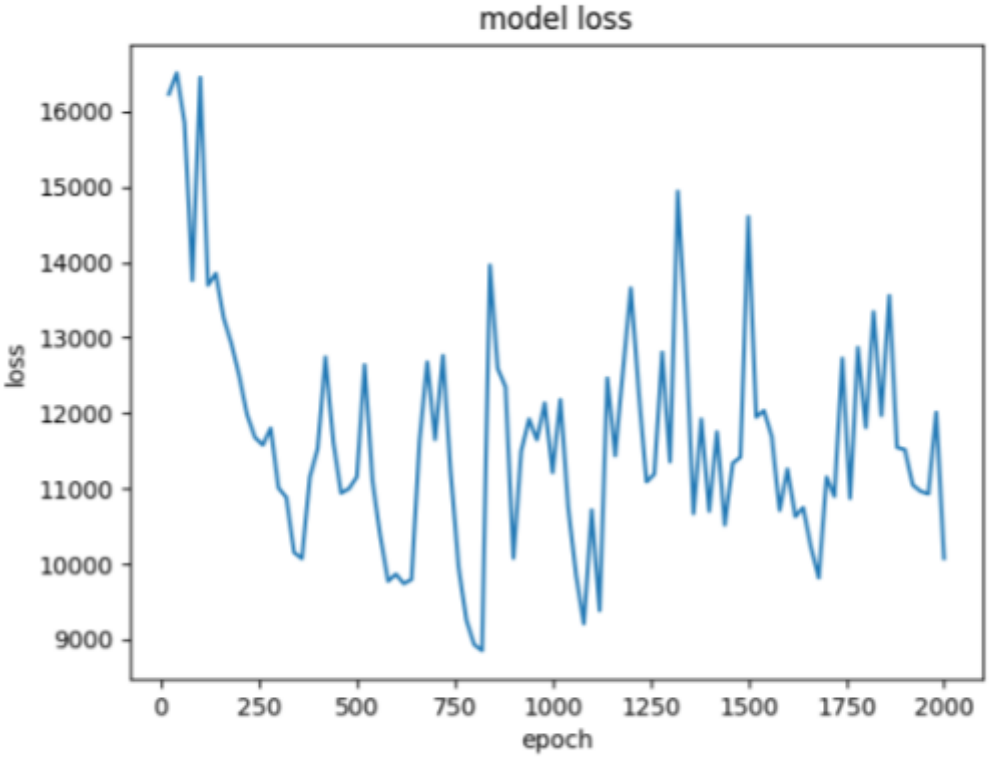

Figura 5.3: Pérdida del entrenamiento de la versión 0

Aparte de las distancias que alcanza nuestro agente a lo largo de su entrenamiento, también hemos obtenido la gráfica que muestra la evolución de la precisión de nuestro modelo al realizar el entrenamiento respecto al modelo objetivo en la figura 5.2. Se puede observar como la precisión comienza a adoptar una tendencia más ascendente a partir del episodio 1.750, coincidiendo con la gráfica de las distancias en la figura 5.1 donde se aprecia que a partir de ese episodio nuestro agente logra el objetivo con mayor frecuencia. Así mismo, en la figura 5.3 se puede discernir una ligera estabilización ya que los dientes de sierra formados por la gráfica van reduciendo su tamaño, es decir, la diferencia entre un entrenamiento y otro se reduce y tiende a converger.

5.2 VERSIÓN 1

Después hemos lanzado varias ejecuciones de la versión uno, en esta ocasión probando distintas combinaciones de tamaño de memoria de recuerdo y batch. Las pruebas han combinado tamaños de memoria de recuerdo de 5.000, 10.000 y 15.000 transiciones y tamaños de batch de 32 y 64 transiciones, lo que supone un total de seis configuraciones distintas todas con una duración de 1.000 episodios y diez segundos como máximo cada episodio.

Tras realizar los experimentos con esta versión hemos obtenido las gráficas de distancias y precisión.

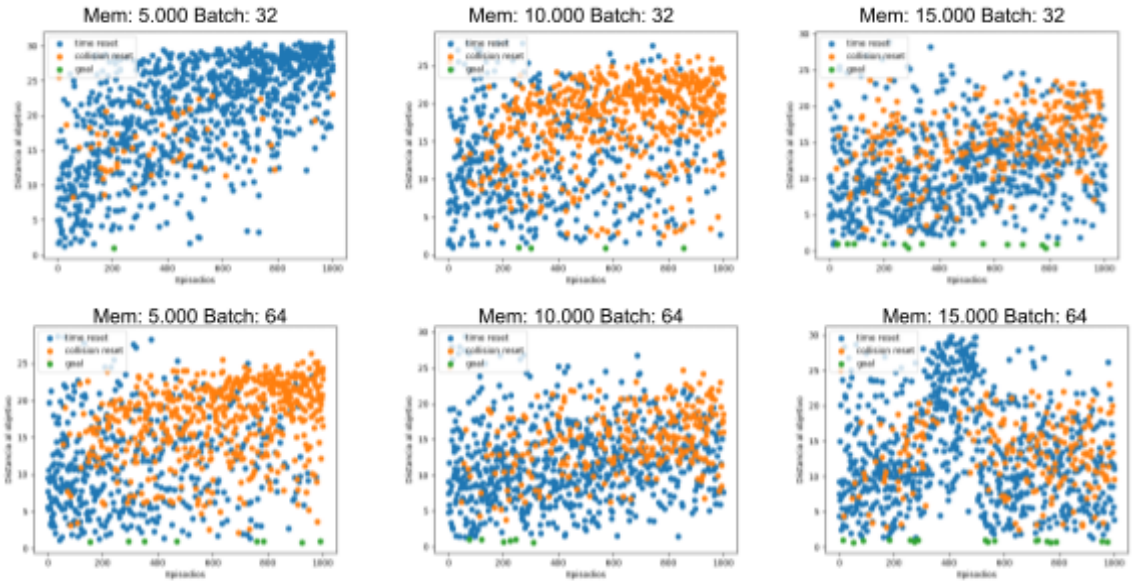

Figura 5.4: Gráficas de distancias del entrenamiento de la versión 1

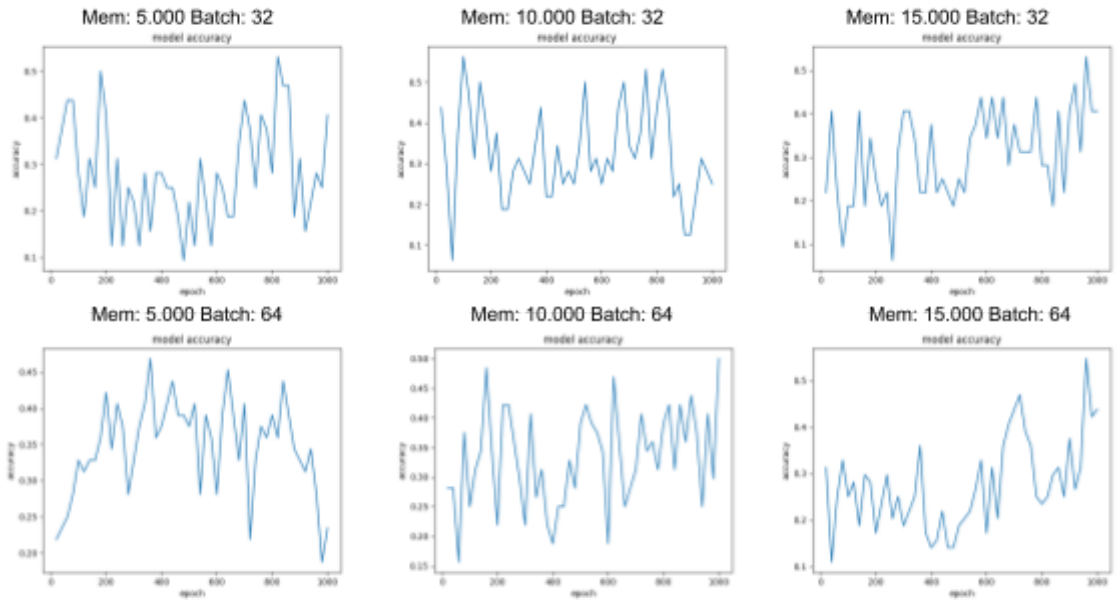

Figura 5.5: Gráficas de precisión del entrenamiento de la versión 1

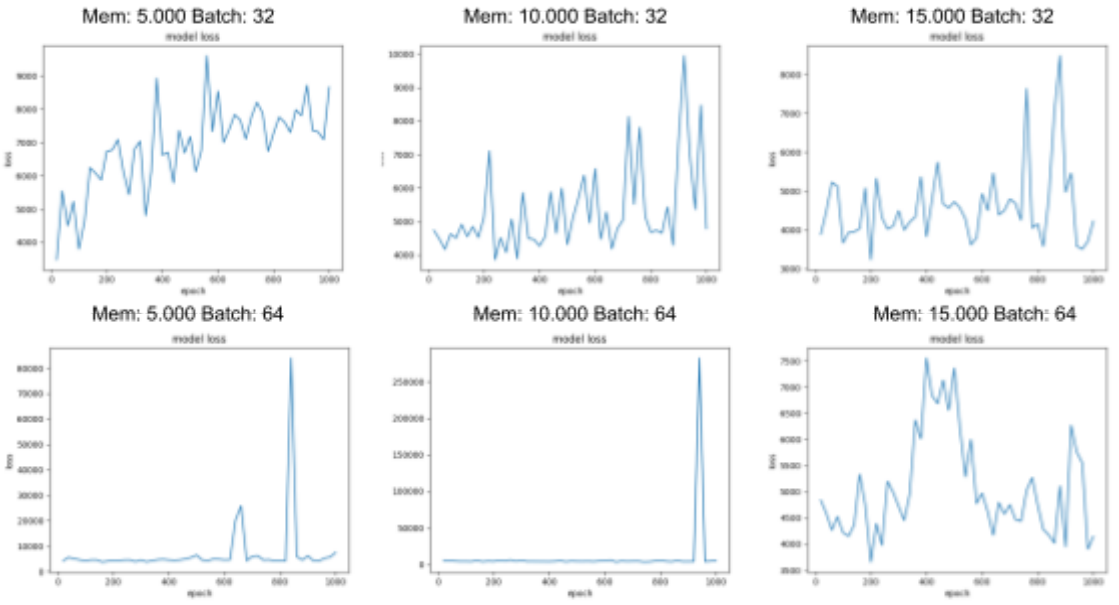

Figura 5.6: Gráficas de pérdida del entrenamiento de la versión 1

De entre todas las ejecuciones que podemos observar en las figuras 5.4, 5.5 y 5.6, la que mejor rendimiento parece tener es la de 15.000 de memoria de recuerdo y 64 de batch, tanto en distancia como en precisión. En las gráficas de distancias se aprecia que esta última presenta mayor aproximación al objetivo que las demás, que tienden a alejarse más de la meta.

Esto por sí solo no indica que un entrenamiento más duradero con esta configuración vaya a durar menos, pero es considerable la ayuda que aporta al aprendizaje por refuerzo tener guardadas en la memoria de recuerdo esas transiciones que se acercan más al objetivo y de las cuales nuestro agente aprenderá.

Respecto a la precisión, los mismos parámetros que generan la gráfica de mejor distancias también presentan una precisión de tendencia ascendente con mayor pendiente que las demás y con picos más bajos, lo que denota menor desviación entre los entrenamientos.

5.3 VERSIÓN 2

Por último realizamos los experimentos con la versión dos. Estos experimentos son los mismos que las pruebas de la versión uno en cuanto a las combinaciones de configuraciones utilizadas.

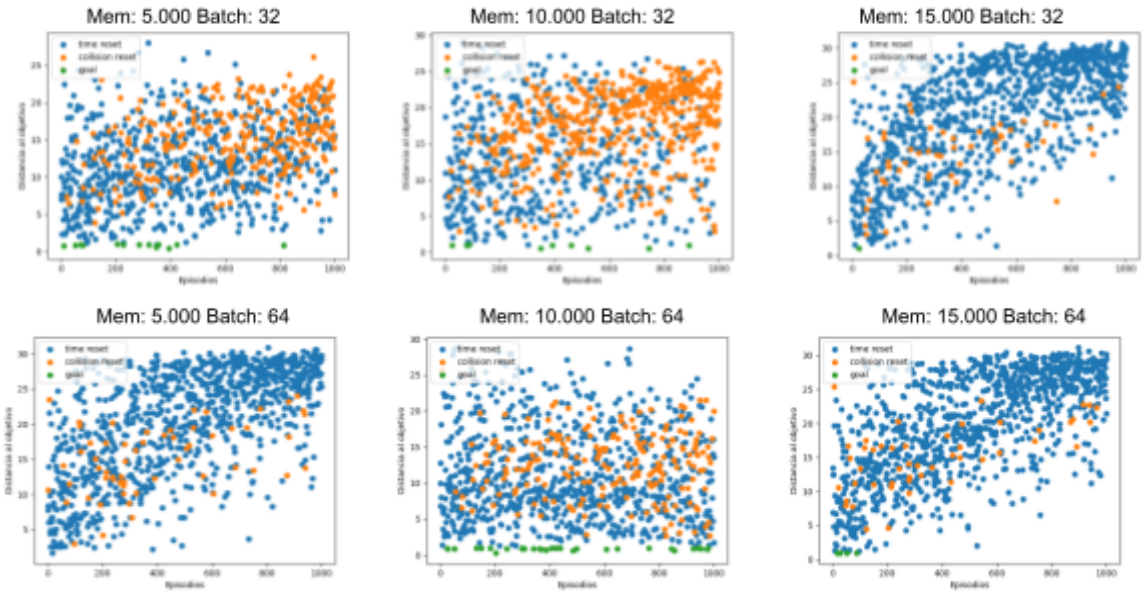

Figura 5.7: Gráficas de distancias del entrenamiento de la versión 2

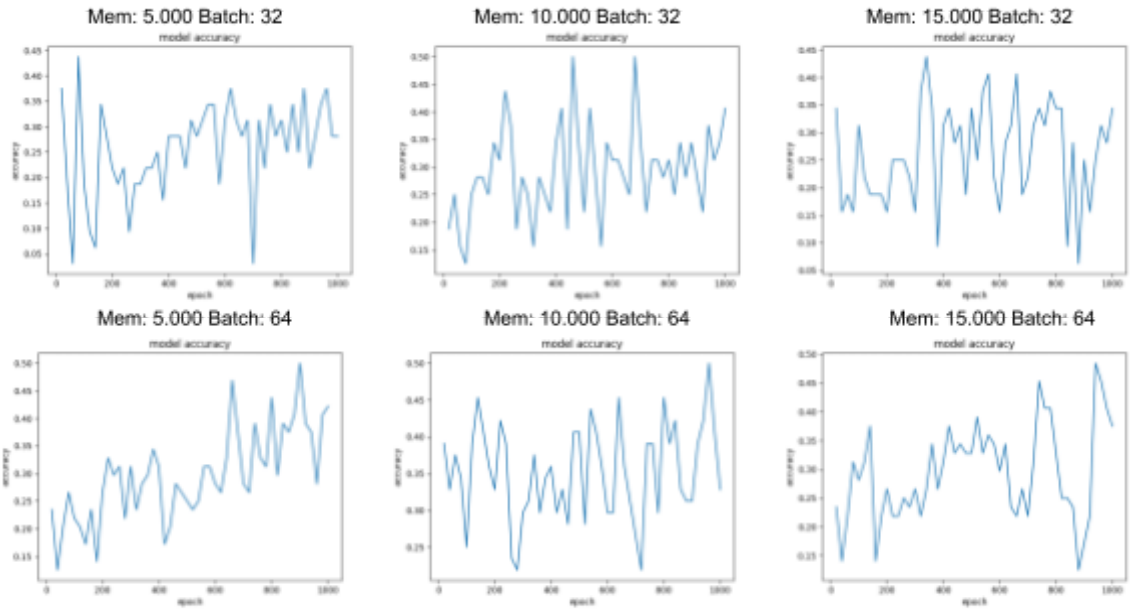

Figura 5.8: Gráficas de precisión del entrenamiento de la versión 2

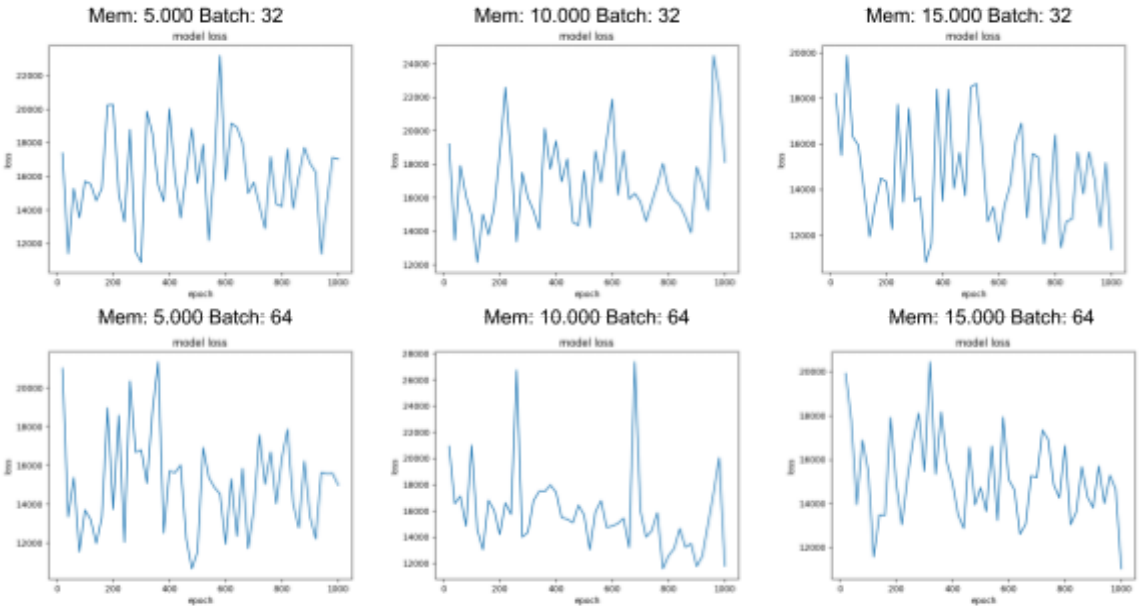

Figura 5.9: Gráficas de pérdida del entrenamiento de la versión 2

Una vez obtenidos los resultados de los entrenamientos mostrados en las figuras 5.7, 5.6 y 5.9 nos enfrentamos a la duda de qué configuración es la mejor. Por una parte una memoria de recuerdo de 10.000 transiciones y un tamaño de batch de 64 transiciones parece producir mejores resultados respecto a la distancia. Pero por otro lado la precisión aparece con mayor tendencia ascendente al entrenar con una memoria de recuerdo de 5.000 transiciones y un tamaño de batch de 64 transiciones.

Ante esta duda cabe recordar que la gráfica de distancias representa el estado de la memoria de recuerdo, es decir, cuanto más próximas sean las distancias al objetivo o más bajos estén los puntos, habrá un mayor número de transiciones con recompensas altas resultado de haberse acercado más a la meta y de las cuales el agente aprenderá.

En el caso contrario, si las distancias resultantes del entrenamiento están más alejadas de la meta las transiciones que se almacenen en la memoria de recuerdo contendrán peores recompensas, pero esto no es algo negativo. Al igual que el agente puede aprender a partir de transiciones “buenas” también puede aprender de las “malas”. Es como partir desde dos sitios diferentes, los caminos a recorrer son distintos pero la meta es la misma, solo cambia el tiempo que tome llegar. El agente también podrá aprender de esas experiencias que lo penalizaron y modificar los pesos de las neuronas para obtener un resultado distinto.

Dicho esto, ya que las gráficas de las distancias por si solas no proporcionan suficiente información sobre el rendimiento de nuestro sistema durante estos experimentos, usaremos las gráficas de precisión para decidir la configuración óptima. Por lo tanto, los parámetros que mejores datos han arrojado en estas pruebas son una memoria de recuerdo de 5.000 transiciones y un tamaño de batch de 64 transiciones. Esta será la configuración que utilizaremos para entrenar nuestro modelo de forma mucho más exhaustiva.

5.4 VERSIÓN FINAL

En esta última versión utilizaremos la mejor configuración obtenida en el apartado anterior y pondremos al agente a entrenar durante un largo periodo de tiempo para posteriormente, analizar los resultados de dicho entrenamiento.

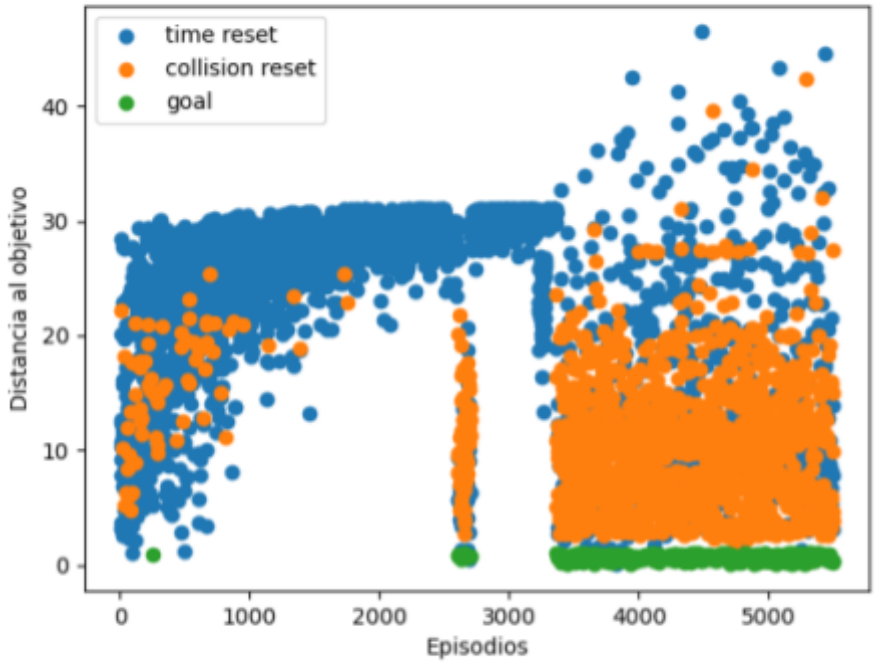

Figura 5.10: Evolución de las distancias en el entrenamiento de la versión final

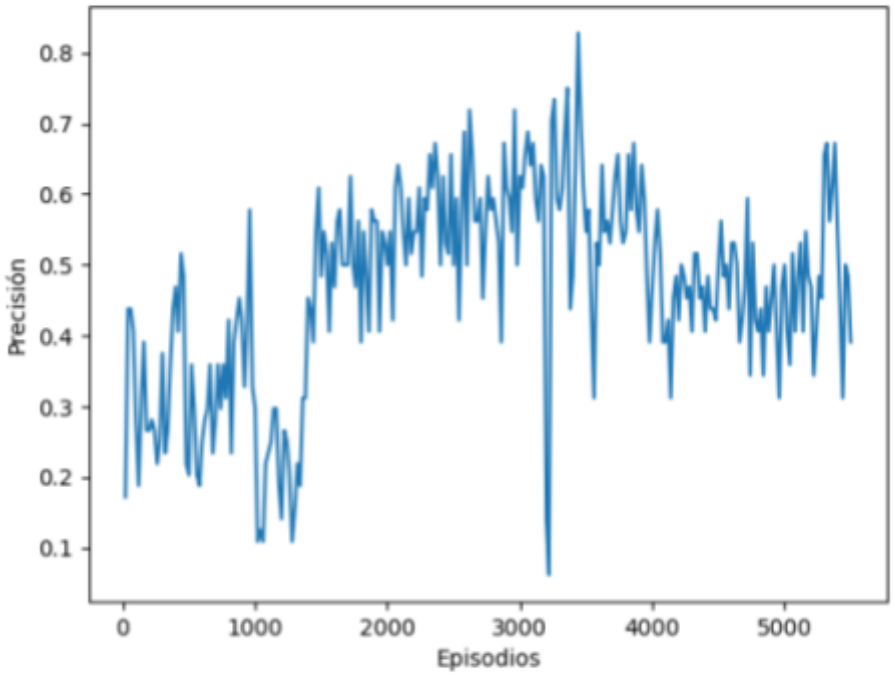

Figura 5.11: Evolución de la precisión en el entrenamiento de la versión final

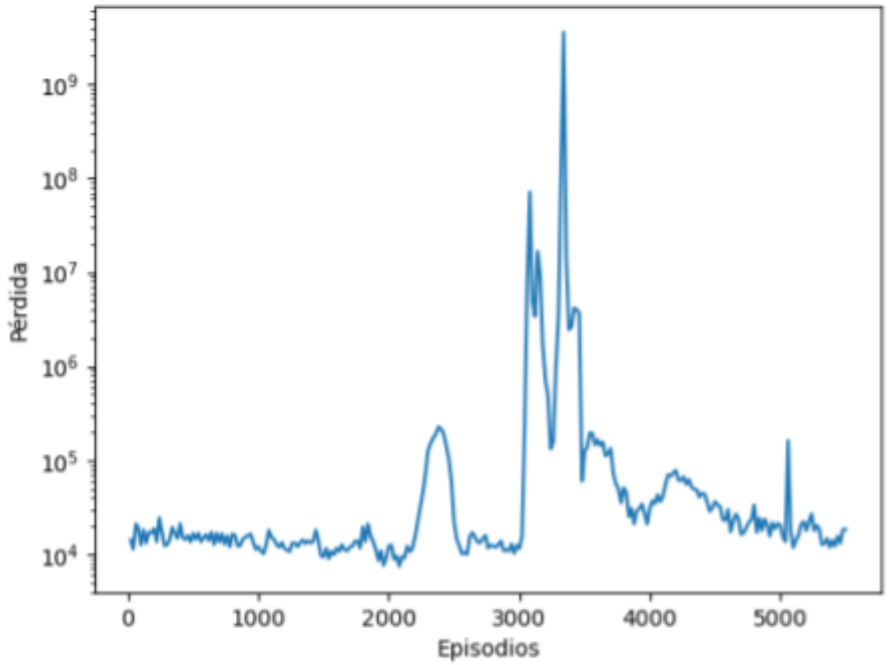

Figura 5.12: Evolución de la pérdida en el entrenamiento de la versión final

Tras un entrenamiento de 5.500 episodios y una duración de aproximadamente 22 horas hemos obtenido un modelo que ha aprendido a avanzar. En la figura 5.10 se puede observar como el agente ha logrado acercarse cada vez más a la meta, en muchas ocasiones alcanzándola exitosamente. Las colisiones se van concentrando en la zona inferior de la gráfica con lo que parece una tendencia a converger en la meta.

También se van reduciendo los reinicios por límite de tiempo y la mayoría se aproximan a la meta a medida que avanzan los episodios. Algunos de estos reinicio por tiempo sobrepasan la distancia inicial entre el punto de salida del agente y la meta de 30 metros, esto se debe a que el vehículo ha sobrepasado la meta por algún lateral y ha seguido avanzando sin chocarse.

Puede que no haya logrado el objetivo por completo al no acercarse lo suficiente a la zona a alcanzar pero el hecho que haya continuado circulando por la carretera hasta un máximo de 40 metros pasada la meta sin chocar también puede considerarse un éxito parcial en este entrenamiento. Recordemos que nuestro modelo tiene un factor de aleatoriedad que va disminuyendo para proporcionar a nuestra memoria de recuerdo una variedad de transiciones con las que pueda aprender de la mayor cantidad de posibles situaciones.

En el episodio 3.500 el agente ya se desplaza con mayor frecuencia que en los episodios anteriores, en los cuales desde el episodio 2.000 no se movía más de cinco metros. En el episodio 3.500 el factor de aleatoriedad de nuestro modelo es del 41,68%, por lo que el avance del agente es gracias a las predicciones de nuestra red neuronal en más del 50% de las acciones que toma.

Respecto a la precisión de este modelo, en la figura 5.11 no parece haber una tendencia claramente ascendente hacia el final del entrenamiento, pero si aparenta quedar estabilizada en torno a una precisión del 50% a partir del episodio 3.500. Esta estabilidad puede correlacionarse con las distancias ya que la mayoría de los puntos a partir del episodio 3.500 se concentran por debajo los 15 metros que es el 50% de la distancia inicial que debe recorrer el agente para lograr su objetivo.

En cuanto a la pérdida de nuestro modelo se aprecia una ligera tendencia descendente en la figura 5.12 excepto por el repentino incremento entre los episodios 3.000 y 3.500. Este aumento de la pérdida coincide con el descenso de la precisión y el aumento de la distancia. La precisión solo disminuye en un punto pero la pérdida se mantiene alta en esos 500 episodios al igual que se mantiene la distancia a la meta en la parte superior de la gráfica durante un periodo similar pero con un inicio anterior.

Esto se debe a que la pérdida del modelo se calcula en base a lo sucedido con anterioridad. Antes de ese periodo con las distancias cercanas a los 30 metros el agente había logrado acercarse e incluso alcanzar la meta poco después del episodio 2.500 durante unos 100 o 200 episodios. Tras este éxito en su progreso vuelve a una situación peor en la que no avanza más de cinco metros, por este motivo la pérdida se dispara, el modelo parecía haber empezado a mostrar signos de aprendizaje y poco después volvió a un estado anterior como si no hubiese aprendido.

6. Conclusiones y trabajos futuros

6.1 CONCLUSIONES

Una vez concluido el proyecto y la memoria, es posible obtener unas conclusiones finales respecto al proyecto desarrollado. El proyecto consta de una serie de objetivos y subobjetivos que se han realizado correctamente y permiten obtener una conclusión final.

En primer lugar, se ha abordado exitosamente el objetivo que consta en estudiar las posibles opciones para interconectar un sistema multiagente con un simulador de vehículos. A su vez, este objetivo consta de subobjetivos, tal y como recopilar información del sistema multiagente y el simulador de vehículos para su selección frente a otras posibilidades.

Respecto a la información que se ha recopilado sobre el sistema multiagente se encuentra el sistema SPADE del cual se detalla suficiente información que justifica el porqué ha sido seleccionado para su incorporación en el proyecto desarrollado.

En cuanto a la recolección de información sobre simuladores de vehículos se encuentran AirSim, un software de simulación de drones y coches, SUMMIT, un simulador de conducción urbana con tráfico mixto masivo, y Flow, un framework de aprendizaje profundo por refuerzo para el tráfico de autonomía mixta. Pero ante estos simuladores hay que destacar el simulador de vehículos CARLA, que permite la simulación de todo tipo de vehículos y peatones y muchas más ventajas que se detallan en la memoria.

Por ello fue el simulador seleccionado para desarrollar, junto con la plataforma de agentes SPADE, el entorno que se ha realizado en este proyecto.

Una vez seleccionados los sistemas que se requerían para la implementación del proyecto, se abordó exitosamente el objetivo de diseñar e implementar un entorno de simulación de vehículos que permitiese la implementación y comparación de diferentes modelos de aprendizaje por refuerzo con DQN, así como desarrollar pruebas para experimentar con estos modelos. De este modo queda completado el segundo objetivo.

Respecto al tercer punto de los objetivos que hace foco en realizar pruebas con diferentes modelos y analizar sus resultados, se ha conseguido probar con éxito hasta seis modelos distintos para dos de las tres versiones de nuestro sistema con un total de doce pruebas más la prueba preliminar de la primera versión. Tras realizar las pruebas se han analizado los resultados de sus correspondientes entrenamientos y se ha seleccionado el mejor modelo de los doce.

Finalmente, el último objetivo trataba de utilizar el mejor modelo seleccionado para realizar un entrenamiento de máxima duración en el que el agente pueda llegar a aprender a conducir lo más posible. Si bien no se ha demostrado un perfecto rendimiento por parte del agente, si que se ha podido probar el progreso del aprendizaje y por tanto el cumplimiento de este objetivo al obtener un agente que va “aprendiendo” dentro del entorno simulado.

Tras esta enumeración de los cumplimientos de todos los puntos y subpuntos de los objetivos. Se puede decir que se han completado correctamente todos los objetivos establecidos para este proyecto, dicho de otro modo el proyecto ha finalizado satisfactoriamente.

6.2 TRABAJOS FUTUROS

En cuanto a trabajos futuros, el proyecto ofrece grandes posibilidades de ampliación por distintas partes. A continuación, se detallan algunas de ellas.

En primer lugar, y una de las más interesantes. Sería realizar una investigación más detallada sobre algoritmos de aprendizaje por refuerzo para poder optimizar por completo el algoritmo DQN. De este modo, proporcionar un modelo que permita la conducción autónoma lo más perfecta posible. Además, ya que lo permite la plataforma desarrollada, migrar el modelo a un coche real para realizar pruebas en el mundo real.

En segundo lugar, este proyecto ha sufrido limitaciones debido a que se ha tenido que realizar por completo en un ordenador personal que no estaba acondicionado en su totalidad para este tipo de experimentos. Por ello resultaría conveniente poder realizar este proyecto en una máquina de mayor potencia que permitiese modelos de mayor capacidad computacional.

Por último, dejamos como trabajo futuro el entrenamiento completo del modelo seleccionado por este proyecto para lograr un aprendizaje con el que el agente logre conducir con una mayor precisión. Debido al corto periodo de ejecución que hemos podido conseguir no se ha podido observar los resultados de un aprendizaje en su totalidad o con un nivel deseado, por lo tanto resultaría interesante poder entrenar el modelo mucho más tiempo del experimentado en este trabajo.

7. Bibliografía

[1] Shah, S., Dey, D., Lovett, C., & Kapoor, A. (2018). Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In Field and service robotics (pp. 621-635). Springer, Cham.

[2] Cai, P., Lee, Y., Luo, Y., & Hsu, D. (2020, May). Summit: A simulator for urban driving in massive mixed traffic. In 2020 IEEE International Conference on Robotics and Automation (ICRA) (pp. 4023-4029). IEEE.

[3] Wu, C., Kreidieh, A., Parvate, K., Vinitsky, E., & Bayen, A. M. (2017). Flow: Architecture and benchmarking for reinforcement learning in traffic control. arXiv preprint arXiv:1710.05465, 10.

[4] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., & Koltun, V. (2017, October). CARLA: An open urban driving simulator. In *Conference on robot learning* (pp. 1-16). PMLR.

[5] Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of artificial intelligence research*, *4*, 237-285.

[6] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

[7] Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3), 279-292.

[8] Watkins, C. J. C. H. (1989). Learning from delayed rewards.

[9] Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 30, No. 1).

[10] Lin, L. J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, *8*(3), 293-321.

[11] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver. (2015). Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*.

[12] Roderick, M., MacGlashan, J., & Tellex, S. (2017). Implementing the deep q-network. *arXiv preprint arXiv:1711.07478*.

[13] Fan, J., Wang, Z., Xie, Y., & Yang, Z. (2020, July). A theoretical analysis of deep Q-learning. In *Learning for Dynamics and Control* (pp. 486-489). PMLR.

[14] Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning Proceedings 1995* (pp. 30-37). Morgan Kaufmann.

[15] Ong, H. Y., Chavez, K., & Hong, A. (2015). Distributed deep Q-learning. *arXiv preprint arXiv:1508.04186*.

[16] Palanca, J., Terrasa, A., Julian, V., & Carrascosa, C. (2020). Spade 3: Supporting the new generation of multi-agent systems. *IEEE Access*, *8*, 182537-182549.

[17] Riquelme, A. J., Abellán, A., Tomás, R., & Jaboyedoff, M. (2014). A new approach for semi-automatic rock mass joints recognition from 3D point clouds. *Computers & Geosciences*, *68*, 38-52.

8. Anexo

8.1 OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**	X			
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.	X			
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**			X	
ODS 14. **Vida submarina.**				X

ODS 15. **Vida de ecosistemas terrestres.**				**X**
ODS 16. **Paz, justicia e instituciones sólidas.**				**X**
ODS 17. **Alianzas para lograr objetivos.**				**X**

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

En este trabajo se ha desarrollado un sistema con la finalidad de entrenar a un agente mediante una red neuronal para que aprenda a conducir igual o mejor que un ser humano. Con este enfoque podemos relacionar en primer lugar este proyecto con el ODS 3, salud y bienestar, en un alto grado. Gracias a la incorporación de un software a los mandos de un vehículo se pueden evitar los accidentes de tráfico y otro tipo de siniestros causados por distracciones al volante. Tampoco habría problemas causados por conductores bajo los efectos del alcohol o drogas y también se evitarían los accidentes consecuencia del cansancio. Todas estas ventajas de tener un sistema automático de conducción reducirían la mortalidad en carretera considerablemente así como permitir a los usuarios más tiempo de descanso al no tener que conducir lo que afectaría positivamente a su estado mental.

En segundo lugar, este trabajo también se relaciona en un alto grado con el ODS 9, industria, innovación e infraestructuras. Respecto a este punto se centra más en la parte de la innovación por el desarrollo del aprendizaje por refuerzo. Este proyecto se ha centrado en el uso de este campo de la inteligencia artificial en auge y con amplias posibilidades en el futuro gracias a su potencial método de entrenamiento sin supervisión. El tema de las redes neuronales es un campo de investigación muy amplio y las configuraciones utilizadas en este proyecto pueden ayudar a diseñar modelos para usos similares en el futuro.

En tercer lugar también podemos relacionar nuestro proyecto con el ODS 11, ciudades y comunidades sostenibles, en un alto grado. La eficiencia que proporciona un coche capaz de conducir de forma autónoma permitiría reducir el consumo de combustible o cualquier tipo de energía que utilice el vehículo. Además, se podría prescindir de la logística e infraestructura dedicada a vigilar las infracciones de los conductores con lo que disminuiría el gasto público dedicado a esas áreas y podría repartirse entre todas las demás.

En cuarto lugar, el proyecto puede relacionarse con el ODS 13, acción por el clima, en bajo grado por lo comentado anteriormente. Un menor consumo de energía en la conducción favorece la disminución de emisiones de CO_2 a la atmósfera y ayuda, aunque sea de forma reducida, contra el cambio climático. Respecto al resto de los ODS, este trabajo no tiene relación alguna dada su naturaleza intangible al ser un software, su uso e implementación en los vehículos para uso personal en un principio tampoco deja un margen de actuación para que este trabajo pueda influir en otros aspectos del entorno o la sociedad.

En definitiva, hemos podido relacionar este proyecto con varios de los ODS en mayor o menor grado, con lo que podemos concluir que este trabajo puede ser potencialmente beneficioso para el desarrollo sostenible de nuestra sociedad y el planeta.