

**Universidad
Rey Juan Carlos**

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería Informática

Curso 2024-2025

Trabajo Fin de Grado

**APRENDIZAJE POR REFUERZO CON
ENTRENAMIENTO PARALELO DE UN COCHE DE
CARRERAS**

**Autor: Fernando López Berrocal
Tutor: Alberto Sánchez Campos**

Índice de contenidos

Índice de tablas	V
Índice de figuras	VIII
Índice de algoritmos	X
1. Introducción	1
1.1. Contexto general	1
1.2. Antecedentes	2
1.3. Objetivo del proyecto	2
1.4. Estructura del documento	3
2. Estado del arte	6
2.1. Machine Learning	6
2.2. Redes neuronales	7
2.3. Aprendizaje por refuerzo	9
2.3.1. Aprendizaje por refuerzo profundo	11
2.4. Proximal Policy Optimization	12
3. Herramientas	14
3.1. Unity	14
3.2. ML-Agents	16
4. Desarrollo	20
4.1. Planteamiento del problema	20
4.2. Componentes del escenario	21
4.2.1. Carretera	21
4.2.2. Checkpoints	22
4.2.3. Muros	23
4.2.4. Coche	23
4.3. Agente y entorno	25
4.3.1. Observaciones	26
4.3.2. Acciones	27
4.3.3. Recompensas	28

4.4.	Escenarios	28
4.4.1.	Forward	29
4.4.2.	Reorientation	30
4.4.3.	Turn	30
4.4.4.	Loop	31
4.4.5.	Curves	31
4.4.6.	Circuit	32
5.	Entrenamiento	34
5.1.	Configuración del entrenamiento	35
5.2.	Evolución del entrenamiento	40
5.2.1.	Comparación de entrenamiento con durabilidad	44
5.3.	Problemas encontrados durante el entrenamiento	45
5.3.1.	Velocidad del coche	45
5.3.2.	Baja utilización de GPU	46
6.	Conclusión	48
6.1.	Resultado final	48
6.2.	Trabajo futuro	49
	Bibliografía	52

Índice de tablas

5.1. Tabla con tiempos de entrenamiento según número de instancias . 37

Índice de figuras

2.1. Jerarquía en Machine Learning	7
2.2. Perceptrón de una ANN	8
2.3. Representación de una DNN	8
2.4. Ciclo de interacción en RL	10
3.1. Logo de Unity	14
3.2. Editor de Unity. Muestra una escena con <i>GameObjects</i> y los componentes de un <i>GameObject</i> seleccionado	15
3.3. Componentes principales de ML-Agents	16
3.4. Learning Environment	17
4.1. Jerarquía de un circuito	21
4.2. Tipos de tramos de carretera	22
4.3. Checkpoint en un tramo de recta	22
4.4. Muros colocados en un tramo de recta	23
4.5. Modelo del coche en un tramo de recta	24
4.6. Raycasts del agente	27
4.7. Escenario Forward	29
4.8. Escenario Reorientation	30
4.9. Escenario Turn	30
4.10. Escenario Loop	31
4.11. Escenario Curves	32
4.12. Escenario Circuit	32
5.1. Gráfica que representa la mejora de tiempos según número de agentes en escena	38
5.2. Tabla y gráfica de los entrenamientos de prueba de número de agentes por instancia	39
5.3. Gráfica con entrenamientos de prueba de tamaño de <i>batch</i>	39
5.4. Video resumen del entrenamiento en Forward	40
5.5. Gráfica del entrenamiento en Forward	40
5.6. Video resumen del entrenamiento en Reorientation	41
5.7. Gráfica del entrenamiento en Reorientation	41

5.8.	Video resumen del entrenamiento en Turn	41
5.9.	Gráfica del entrenamiento en Turn	42
5.10.	Video resumen del entrenamiento en Loop	42
5.11.	Gráfica del entrenamiento en Loop	42
5.12.	Video resumen del entrenamiento en Curves	43
5.13.	Gráfica del entrenamiento en Curves	43
5.14.	Video resumen del entrenamiento en Circuit	43
5.15.	Gráfica del entrenamiento en Circuit	44
5.16.	Video comparativa de los 2 agentes con durabilidad al 100 %	44
5.17.	Video comparativa de los 2 agentes con durabilidad al 30 %	45

Índice de algoritmos

1.	PPO Simplificado	36
----	----------------------------	----

1

Introducción

1.1. Contexto general

Entre los principales objetivos de la Inteligencia Artificial [1] está el desarrollo de agentes autónomos que sean capaces de interactuar con su entorno aprendiendo de él y tomar decisiones que le lleven a sus objetivos. El aprendizaje por refuerzo (Reinforcement Learning, RL) es un área centrada en ese objetivo. En RL, los agentes aprenden mediante prueba y error mejorando su comportamiento con recompensas recibidas de un entorno que no conocen. El RL se aplica en muchas áreas como: controlar robots físicos, planificación y toma de decisiones, optimización de procesos, videojuegos... [2] Sin embargo, hace años el RL estaba limitado a problemas muy sencillos y no se podía usar en la práctica. El problema era la demanda de cómputo, memoria y la generación de muestras para el aprendizaje. El desarrollo del aprendizaje profundo (Deep Learning) trajo muchos avances en el RL (Deep Reinforcement Learning). Utilizando redes neuronales profundas (Deep Neural Network) ahora se pueden resolver problemas que antes eran considerados intratables en RL por espacios estados y acciones demasiado grandes. Uno de los primeros grandes hitos en el uso de DRL fue con el algoritmo de Deep Q-Network (DQN) desarrollado por DeepMind [3] con el que lograron entrenar un agente a jugar videojuegos de la Atari 2600 como el Pong, Breakout y Space Invaders entre otros. Fueron entrenados tomando directamente los píxeles como observación y en algunos de ellos lograron mejores rendimientos a los humanos. Otros importantes hitos fueron el AlphaGo, que venció al campeón de Go con una combinación de DRL, aprendizaje supervisado y búsqueda heurística [4]. Más recientemente, OpenAI entreno un agente usando el algoritmo PPO para jugar al Dota 2 [5] y ponerlo a prueba con jugadores de eSports. Con estos avances,

los videojuegos y los entornos simulados son un entorno de pruebas importante para la investigación en DRL ya que proporcionan entornos controlados y sin riesgos físicos. Pero el objetivo final del RL es adaptarse al mundo real y crear agentes capaces de realizar tareas físicas o cognitivas fuera del entretenimiento como: mover objetos, conducción autónoma o gestionar recursos

1.2. Antecedentes

El DRL ha comenzado a tener un impacto considerable en algunas áreas como la conducción autónoma [6]. Tradicionalmente los sistemas de conducción autónoma se dividen en varios módulos que se encargaban de diferentes tareas como percepción, mapeado, planificación de trayectorias y control del vehículo. Los módulos se implementaban principalmente usando algoritmos de aprendizaje supervisado o con reglas de comportamiento programadas explícitamente. En los últimos años se ha incluido el uso de DRL en algunos de los anteriores módulos o incluso realizando un aprendizaje end-to-end donde no se dividen módulos si no que se usa una red neuronal que mapea directamente las entradas con el control del vehículo. Hay empresas como Wayve que ha desarrollado modelos que combinan DRL con Imitation Learning entrenados en simuladores y en entornos reales controlados. También existe el proyecto CARLA (Car Learning to Act) [7], que es un simulador de conducción autónoma utilizado en investigación. En el área de la robótica también se ha aplicado el DRL con éxito en tareas complejas como locomoción y manipulación de objetos. La empresa Boston Dynamics han incorporado DRL en el sistema de locomoción de algunos de los robots cuadrúpedos. En todos los casos, los entrenamientos primero se realizan en entornos simulados antes de probarlos en el mundo real, evitando de esta manera riesgos materiales y humanos. Es la estrategia “Sim-to-Real” [8] que permite entrenar los agentes en simulaciones como CARLA o Unity ML-Agents y después transferir el conocimiento a robots reales. En los videojuegos se han realizado algunos intentos de implementar DRL para el comportamiento de los NPC's [9]. Pero su uso no es muy popular debido al alto coste computacional y la complejidad de implementación innecesaria resulta con técnicas más simples como Máquinas de Estados Finitos, Waypoints y Heurísticas basadas en reglas.

1.3. Objetivo del proyecto

El objetivo final del proyecto es desarrollar y entrenar un agente inteligente capaz de controlar un coche por un circuito (en un entorno simulado) lo más rápido posible. Este objetivo se divide en los siguientes sub-objetivos:

1. Diseñar las características y movimiento del coche en el entorno simulado.

2. Diseñar el entorno de aprendizaje por refuerzo dividiendo el aprendizaje en escenarios.
3. Ajuste de las configuraciones del entrenamiento y los hiperparámetros del algoritmo utilizado para disminuir los tiempos de entrenamiento.
4. Entrenar al agente logrando que cumpla sus propios objetivos.

Por otro lado, el agente tiene sus propios objetivos que son superar eficazmente los escenarios de aprendizaje. Una vez haya superado todos, el agente deberá ser capaz de:

- Controlar el coche sin salirse del recorrido.
- Seguir los checkpoints del recorrido.
- Diferenciar los tipos de checkpoints para anticiparse al siguiente tramo y mejorar su velocidad total.
- Adaptar su forma de conducir al estado de las ruedas.
- Conducir rápido.

El propósito del proyecto no es crear una solución generalizable ni un producto comercial ya que el agente estará limitado al entorno diseñado. El fin es adquirir conocimientos prácticos sobre la creación de entornos de DRL, ajuste de hiperparámetros y aprovechamiento del hardware para optimizar el proceso de entrenamiento.

1.4. Estructura del documento

El documento se estructura en las siguientes secciones:

- **Sección 2. Estado del arte:** Se explican los conceptos fundamentales necesarios para comprender el desarrollo del proyecto. Se introducen las bases del Machine Learning, las redes neuronales, el aprendizaje por refuerzo y por último el algoritmo Proximal Policy Optimization que será el utilizado en el entrenamiento.
- **Sección 3. Herramientas:** Se explica brevemente las herramientas más relevantes utilizadas en el proyecto: Unity y ML-Agents

- **Sección 4. Desarrollo:** Esta sección describe detalladamente el planteamiento del problema. Se explica el diseño del entorno de aprendizaje, describiendo los componentes usados en los escenarios creados y el diseño del agente y su interacción con el entorno. Además, se presentan los diversos escenarios donde entrenará el agente para un aprendizaje progresivo.
- **Sección 5. Entrenamiento:** Se analiza el proceso de entrenamiento del agente y las pruebas realizadas con diferentes conjuntos de hiperparámetros para encontrar la configuración óptima para el entrenamiento. Se muestra la evolución del agente a través de su entrenamiento por los escenarios. También se explican algunos de los problemas encontrados durante el proceso de entrenamiento y las soluciones aplicadas.
- **Sección 6. Conclusión:** Se resume el resultado final del agente tras sus entrenamientos. También se proponen ampliaciones del proyecto para mejorar el comportamiento del agente.

2

Estado del arte

2.1. Machine Learning

El Machine Learning (ML) [10] es una rama de la Inteligencia Artificial centrada en el desarrollo de algoritmos que pueden aprender a realizar tareas concretas a partir de unos datos de entrada que sirven como ejemplos. Al contrario que con algoritmos tradicionales en los que se programan directamente las reglas para resolver la tarea, el Machine Learning permite que el sistema aprenda a partir de los datos de entrada. Para ello habitualmente hay que proporcionar grandes volúmenes de datos al modelo y un objetivo. De esta forma puede relacionar atributos de los datos identificando patrones y con ello mejorar su rendimiento. Según los tipos de datos disponibles y las tareas a realizar se pueden clasificar varios tipos de algoritmos como se muestra en Fig. 2.1.

- **Aprendizaje supervisado:** Es el más común. Se posee un conjunto de datos etiquetados y el objetivo es aprender a predecir la etiqueta. Cuando se haya completado el entrenamiento, el modelo debe poder predecir la etiqueta de datos que no se hayan utilizado durante el entrenamiento. Algoritmos de aprendizaje supervisado son los de regresión y los de clasificación. Los de clasificación generan algoritmos que separan los datos en clases y los de regresión predicen un valor continuo según la entrada como una función matemática.
- **Aprendizaje no supervisado:** Se entrena el modelo usando datos sin etiqueta. Los problemas son de agrupar los datos según similitud y patrones.

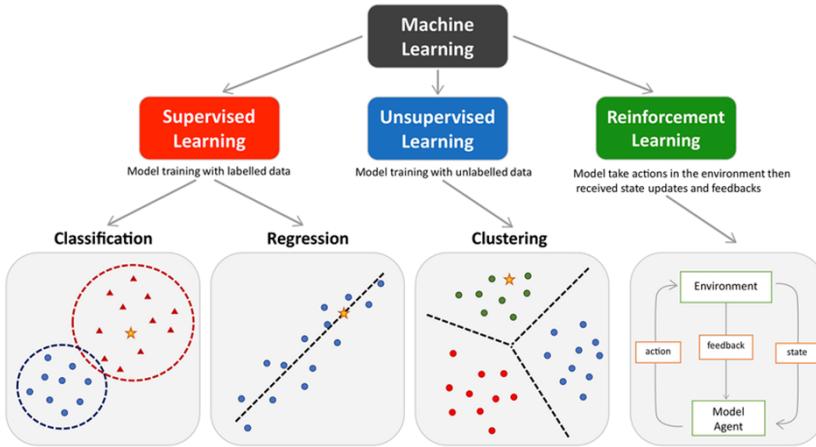


Figura 2.1: Jerarquía en Machine Learning

El tipo de algoritmo más representativo es el clustering que agrupa datos similares según sus atributos en conjuntos.

- **Aprendizaje por refuerzo:** En este tipo de aprendizaje hay un agente que aprende a tomar decisiones interactuando con su entorno. Estas interacciones le devolverán unas recompensas positivas o negativas. El objetivo final será maximizar la suma total de las recompensas. Este tipo de aprendizaje es diferente a los anteriores por la obtención de los datos que se realiza durante el proceso de entrenamiento y no se preparan antes como los anteriores tipos.

En 1986 surgió una rama del Machine Learning, el Deep Learning [11]. El Deep Learning ha sido fundamental en los últimos avances de la Inteligencia Artificial gracias a su capacidad para detectar patrones de mayor complejidad. Esta subárea se basa en el uso de las redes neuronales profundas que están formadas por múltiples capas intermedias que permiten modelar patrones más complejos.

2.2. Redes neuronales

Las redes neuronales artificiales (ANN) son un sistema basado en las redes neuronales del cerebro humano. Están compuestas por una unidad básica llamada perceptrón que tiene entradas, una función de activación y una salida. Los perceptrones se organizan en capas conectadas entre sí. El objetivo es que, al dar unas entradas a la red neuronal, las procese para obtener una salida. Las redes neuronales son capaces de aprender patrones complejos y se pueden usar en regresión, clasificación, visión artificial y aprendizaje por refuerzo entre otras. Los perceptrones (Ver Fig. 2.2) tienen una serie de entradas que vienen por unas aristas, estas aristas tienen cada una un peso. Los perceptrones también reci-

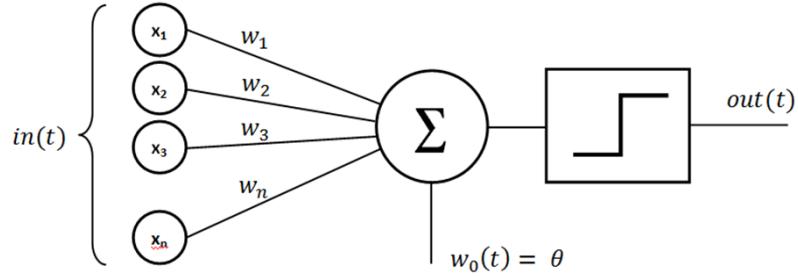


Figura 2.2: Perceptrón de una ANN

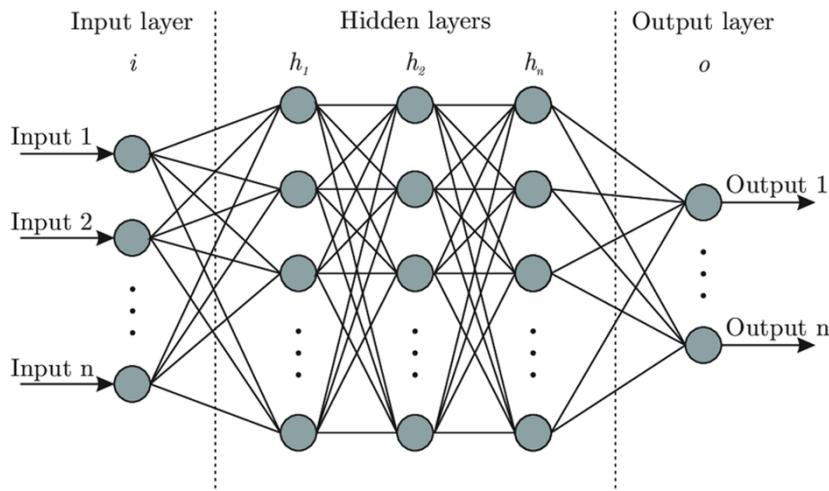


Figura 2.3: Representación de una DNN

ben un valor fijo llamado sesgo. A continuación, se suman todas las entradas (ponderadas por el peso de su arista) y el sesgo. Luego se aplica una función de activación a la suma anterior y su resultado es la salida del perceptrón. Hay diferentes funciones de activación como: sigmoide, tangente hiperbólica o ReLU que se usan dependiendo de la situación [12]. Las redes neuronales están formadas por una capa de entrada y una de salida. La capa de entrada no está formada por perceptrones, sino que son los datos de entrada directamente sin aplicarle pesos ni funciones de activación. Las entradas estarán comunicadas con todos los perceptrones de la siguiente capa. Una red neuronal puede contener capas intermedias, llamadas capas ocultas, si tiene más de una capa oculta se le considera una red neuronal profunda (DNN) como en la Fig. 2.3. Para que una red neuronal realice correctamente la tarea para la que se quiere usar, hay que “entrenarla”. Entrenarla consiste en ajustar los valores de los pesos y los sesgos para que las salidas resulten correctas según las entradas proporcionadas. Una red neuronal inicia sus pesos “aleatoriamente” (no arbitrariamente y sin control, hay métodos de inicialización) por lo que el comportamiento inicial es incorrecto y errático. Proporcionando una función objetivo y muchos datos, se modifican progresivamente los pesos para llegar al comportamiento deseado. Para las actualizaciones

se usa el algoritmo de backpropagation que es un método que usa el error de las predicciones para ajustar los pesos. Las GPU son especialmente útiles para tratar con redes neuronales. Tienen capacidad de realizar muchas operaciones matemáticas en paralelo lo que acelera los entrenamientos e inferencias de redes neuronales. Esto es posible porque se requieren gran cantidad de operaciones matemáticas que pueden paralelizarse al procesarse por capas.

2.3. Aprendizaje por refuerzo

El aprendizaje por refuerzo (RL)[\[13\]](#)[\[14\]](#) [\[15\]](#) es un tipo de aprendizaje en Machine Learning orientado a resolver problemas con tomas de decisiones. Para resolverlos se usa un agente que interactúa en un entorno con el objetivo de lograr una meta como llegar a un destino o conseguir la máxima puntuación. Al contrario que otros tipos de Machine Learning, en el aprendizaje por refuerzo no existe un conjunto de datos preparados sobre los que aprender. Si no que el agente aprende mediante prueba y error en un entorno del cual no conoce las reglas de su funcionamiento ni que acciones tomar en un momento determinado. Por ello, aprendizaje por refuerzo es muy útil para resolver problemas reales donde las acciones óptimas no se conocen o no son fáciles de encontrar. Entornos de juegos y robótica son problemas típicos de aprendizaje por refuerzo. Un problema de aprendizaje se representa como un sistema entre un entorno y al menos un agente. El entorno es donde el agente interactúa y este tiene un estado el cual puede cambiar con las acciones del agente. El agente observa el entorno para conocer su estado, las observaciones del agente deben tener toda la información necesaria para que el agente tome una decisión. Según las observaciones que reciba el agente debe realizar una acción en el entorno. La acción puede modificar el estado del entorno y además proporcionará una recompensa al agente que puede ser positiva, negativa o 0. Cuando ocurre un ciclo estado-acción-recompensa pasa un *time – step*. En Fig. 2.4 se muestra el ciclo que realiza el agente. La política representa el comportamiento del agente. La política es una función que mapea estados a acciones. El objetivo final del aprendizaje por refuerzo es ajustar esta política para que tome las mejores acciones según las observaciones que obtiene. Para ello la suma de las recompensas obtenidas a lo largo del recorrido debe ser la máxima. Una tupla estado-acción-recompensa obtenida en un *time – step*, se llama experiencia. El agente recopila experiencias en bucle hasta llegar a un estado final (llegar al destino, por ejemplo). Los *time – step* que ocurren desde el inicio hasta el estado final forma un episodio. Y una trayectoria es la secuencia de experiencias recopiladas en un episodio. Para entrenar un agente y ajustar su política, se tienen que recopilar miles o millones de trayectorias dependiendo de la complejidad del problema.

Para representar un problema de aprendizaje por refuerzo se usa un Proceso de Decisión de Markov (MDP). Un MDP es un marco matemático para modelar

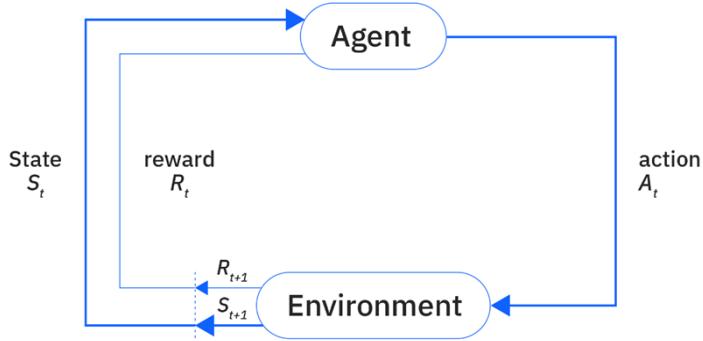


Figura 2.4: Ciclo de interacción en RL

problemas de toma de decisiones con incertidumbre. Se representa el problema usando estados, acciones, una función de transición y una función de recompensas. Un MDP es una tupla (S, A, P, R) donde:

- S es el conjunto de todos los estados posibles del entorno.
- A es el conjunto de todas las acciones posibles que el agente puede tomar.
- P es la función de transición. Define la probabilidad de transicionar a un nuevo estado teniendo en cuenta el estado actual y una acción.
- R es la función de recompensa. Define la recompensa obtenida tras realizar la acción a en un estado s . También se puede definir como la recompensa obtenida tras llegar al estado s' realizando la acción a en el estado s . Es una definición más apropiada para sistemas estocásticos.

Los agentes no tienen acceso a la función de transición ni de recompensa. Para obtener información de ellas tienen que obtener experiencias (s, a, r) interactuando en el entorno.

También hay que definir una función objetivo (Ver Ecuación 2.1) que es la expectativa de la suma descontada de las recompensas

$$J(\tau) = \mathbb{E}_{\tau \sim \pi} [R(\tau)] = \mathbb{E}_{\tau} [\sum_{t=0}^T \gamma^t r_t] \quad (2.1)$$

Hay 3 principales funciones que puede aprender un agente de aprendizaje por refuerzo: una política, la función valor y el modelo del entorno.

- La política π mapea estados a acciones y es esencial en el funcionamiento

del agente porque es lo que genera las acciones que toma. La política debe generar acciones que maximicen el objetivo.

- La función valor estima la expectativa de recompensa. Ayuda al agente a entender como de buenos los estados y acciones disponibles. La función valor $V(s)$ evalúa como de bueno es un estado midiendo la expectativa de recompensa si en el estado s se sigue usando la política π hasta el final del episodio. La función valor $Q(s, a)$ evalúa pares estado-acción. Mide la expectativa de recompensa tras realizar la acción a en el estado s y luego seguir usando la política π hasta el final del episodio.
- El modelo del entorno es la función de transición que este usa. Si el agente aprende esta función puede predecir los estados siguientes. De esta forma puede predecir resultados antes de realizar acciones y con ello cambiar el entorno.

2.3.1. Aprendizaje por refuerzo profundo

El Deep Reinforcement Learning (DRL)[16] [17] [18] usa redes neuronales profundas como método para aproximar funciones. Los algoritmos de DRL se dividen en 3 grupos según cuál de las 3 anteriores funciones aprende el agente.

- Algoritmos basados en la política. Estos algoritmos aprenden una política que genere trayectorias que maximicen el objetivo. La función de la política toma como entrada y produce una acción. Las ventajas de estos algoritmos son: optimizan directamente el objetivo, son muy generales y sirven para cualquier tipo de acciones y garantizan converger a un óptimo local. La desventaja es que tienen alta varianza y son inefficientes en cuanto a muestras.
- Algoritmos basados en la función valor. El agente aprende la función $V(s)$ o $Q(s, a)$ y la usan para generar una política, por ejemplo, seleccionar la acción en el estado que tenga la mayor recompensa estimada. Son más eficientes en cuanto a muestras que el anterior grupo por tener menor varianza. El problema es que no garantizan converger a un óptimo.
- Algoritmos basados en el modelo. Estos algoritmos aprenden un modelo de la función de transición del entorno. Con esa función pueden predecir la trayectoria en el futuro sin producir ninguna acción en el entorno que lo modifique. Es muy útil en situaciones donde realizar acciones es muy costoso. Entornos con muchos estados y acciones son difíciles de modelar y pueden considerarse intratables si es complejo. Solo es útil cuando se pueden predecir precisamente las transiciones del entorno varios pasos en el futuro.

También hay algoritmos que aprenden más de una de estas funciones. Combinan las funciones para sacar lo mejor de cada una. Son muy comunes los algoritmos que aprenden función de política y de valor. Son llamados Actor-Critic [19] porque la política actúa mientras la función valor critica la acción.

2.4. Proximal Policy Optimization

Proximal Policy Optimization (PPO) es un algoritmo Actor-Critic desarrollado en 2017 por Schulman [20] en OpenAI. Los algoritmos Actor-Critic están formados por 2 componentes: un actor, que aprende la política usando optimización por gradiente; y un crítico que aprende una función valor. Durante el entrenamiento la política del actor será reforzada con recompensas generadas por la función valor del crítico. Las recompensas que recibe la política son generadas por la función valor que se aprende, no directamente por el entorno como ocurriría en un algoritmo basado en política únicamente. Esto se hace así para que las nuevas recompensas sean más “informativas” para la política. El algoritmo PPO introduce como cambio principal una nueva función objetivo. La idea de PPO es mejorar la estabilidad del entrenamiento limitando el tamaño de las actualizaciones de la red neuronal. Limitar el tamaño de las actualizaciones resulta en un mayor número de actualizaciones para cambiar la red neuronal y por ello en más tiempo de entrenamiento. Esto se hace para evitar los “performance-collapse” que son situaciones en las que, al realizar una actualización muy grande, se salta a un espacio de políticas “malas” que generan trayectorias pobres. Cuando esto ocurre, es difícil que el algoritmo se recupere y suele converger a un óptimo local con bajos resultados.

3

Herramientas

3.1. Unity

Unity [21] es un motor de videojuegos lanzado en 2005 por la empresa danesa Unity Technologies, originalmente ideado como una herramienta para facilitar y acelerar el desarrollo de videojuegos para Mac OS X. La propuesta era crear un entorno de desarrollo potente y con herramientas robustas pero accesible orientado a desarrolladores profesionales y principiantes. El objetivo de Unity cambio convirtiéndose en multiplataforma siendo compatible en más de 20 plataformas diferentes: móviles, escritorio, web, consola, realidad virtual y aumentada. Unity se ha convertido en uno de los motores de videojuegos más populares gracias a su accesibilidad. Incorpora un editor visual que permite crear escenas y objetos gráficamente reduciendo la curva de aprendizaje. Además, tiene una licencia de uso gratuita para uso personal (y empresas que generen menos de \$100.000 anuales) lo que lo hace muy popular entre principiantes que quieren entrar en el desarrollo de videojuegos y en empresas indies.



Figura 3.1: Logo de Unity



Figura 3.2: Editor de Unity. Muestra una escena con *GameObjects* y los componentes de un *GameObject* seleccionado

El motor de Unity permite crear entornos interactivos en 2D y 3D, lo que lo convierte en una herramienta versátil más allá del desarrollo de videojuegos: ha sido utilizado para crear simulaciones y experiencias interactivas, películas animadas, prototipos industriales, aplicaciones de formación y entrenamiento, experiencias inmersivas en realidad aumentada y realidad virtual entre otros. Por ello se ha usado en otras industrias como: cine, automoción, arquitectura, ingeniería, construcción, aeroespacial, educación e investigación. Compañías como DeepMind y Alphabet lo han usado para entrenar modelos de inteligencia artificial. En el área del aprendizaje por refuerzo profundo, se ha usado Unity para crear entornos sobre los que agentes pueden interactuar. Por ejemplo, para desarrollar robots. El motor de Unity incluye todas las funcionalidades principales de un motor de videojuegos. Para el motor gráfico, Unity usa un sistema de renderizado propio. En cuanto al manejo de físicas, Unity integra el motor NVIDIA PhysX para simular colisiones, fuerzas, gravedad, fricción... en tiempo real. También incluye herramientas para la creación de animaciones, gestión del audio, control de cámara y gestión de escenas entre otras.

Una instancia de Unity (Ver Fig. 3.2) se compone de una o múltiples escenas las cuales contienen objetos y pueden representar niveles de un juego, menús o entornos de simulación. Cada escena tiene una jerarquía formada por **GameObjects**. Cualquier objeto de una escena es un **GameObject** que puede representar un personaje, luz, objeto, audio... Un **GameObject** por sí solo no es nada, solo un contenedor de más **GameObjects**. Pero se le pueden dar distintos comportamientos y propiedades añadiéndole **Components**. Hay muchos **Components** como: **Transform**, que determina su posición, rotación y escalado; **Mesh Filter** y **Mesh Renderer** para dibujar su forma; **Collider** que representa el volumen físico para gestionar colisiones; **Script** para añadir un script en **C#** que añada un comportamiento personalizado.

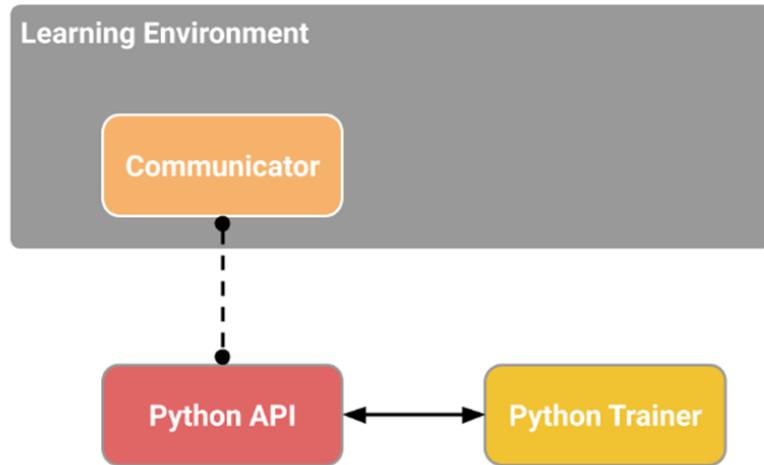


Figura 3.3: Componentes principales de ML-Agents

3.2. ML-Agents

Unity Machine Learning Agents Toolkit (ML-Agents Toolkit) [22] es un proyecto open-source que permite usar juegos y simulaciones de Unity como entornos para entrenar agentes de Inteligencia Artificial [23]. Se pueden entrenar usando Reinforcement learning, Imitation learning, neuroevolution y otros métodos de machine learning. ML-Agents tiene implementaciones de algoritmos de estado del arte basados en PyTorch para entrenar agentes inteligentes en entornos diseñados en Unity. Puede ser usado desde desarrolladores de videojuegos que quieran implementar Inteligencia artificial en el comportamiento de NPC's de sus juegos hasta investigadores de Inteligencia artificial que quieran usar el motor de físicas de Unity para crear un entorno personalizado para entrenar sus agentes y probar diferentes algoritmos. ML-Agents está formado de 4 componentes principales (Ver Fig. 3.3): Learning Environment, Python Low-Level API, External Communicator y Python Trainers.

- **Learning Environment** - Contiene la escena de Unity que proporciona el entorno en que los agentes van a interactuar para recoger experiencias. El ML-Agents Unity SDK te permite convertir una escena de Unity en un entorno de aprendizaje definiendo clases para los agentes e incluyendo herramientas para conectarse con la API de Python
- **Python Low-Level API** - Es una interfaz que sirve para interactuar y controlar el Learning Environment. No es parte de Unity, se comunica con él a través del External Communicator.
- **External Communicator** - Conecta Learning Environment y Python Low-Level API.

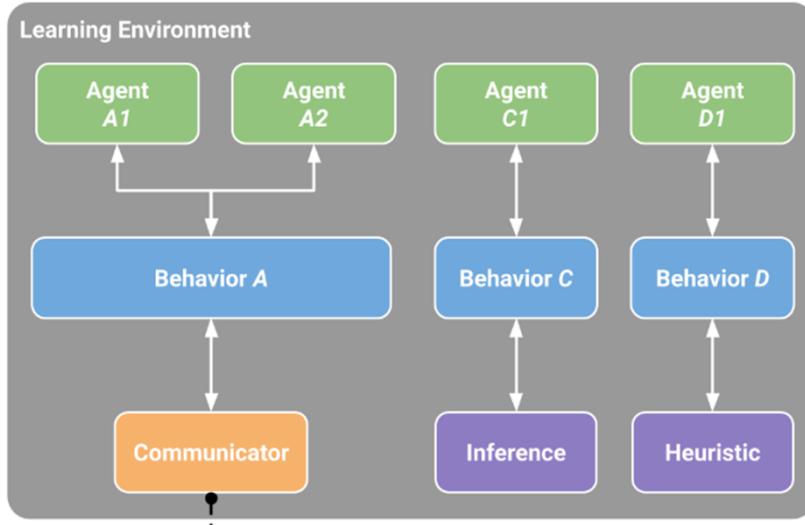


Figura 3.4: Learning Environment

- **Python Trainers** - Es un paquete de Python (mlagents) que contiene la implementación de algoritmos para entrenar agentes.

Un Learning Environment (Ver Fig. 3.4) contiene agentes, que son GameObjects con un componente de agente que controla la generación de observaciones, realiza las acciones que le llegan y se le asigna recompensas. Los agentes están ligados a un Behavior. El Behavior por su parte define los atributos del agente, recibe las observaciones y recompensas de este y le devuelve acciones que ejecuta el agente. Los Behavior pueden ser de aprendizaje (todavía deben aprender), heurísticos (tienen las acciones implementadas directamente en código) y de inferencia (que usan una red neuronal ya entrenada para generar las acciones)

Para el entrenamiento se usa la utilidad mlagents-learn del paquete de Python mlagents junto a un archivo YAML con los hiperparámetros del entrenamiento. En este archivo hay parámetros para Behaviors, entorno, motor, control de checkpoints y opciones de PyTorch. Durante el entrenamiento los agentes envían sus observaciones a su Behavior, y el Behavior a la API de Python por el External Communicator. La API de Python la usará el algoritmo de IA utilizado. El proceso de entrenamiento genera 3 artefactos:

- **Summaries:** Métricas del entrenamiento durante el proceso útiles para monitorizar el aprendizaje. Se puede usar Tensorboard para observar gráficas de la recompensa acumulada, longitud de episodios, Policy Loss, Value Loss...
- **Models:** Son las redes neuronales entrenadas usando un formato ONNX.
- **Timers:** Para observar métricas relacionadas con el consumo de hardware y el tiempo usado en determinadas zonas de código.

ML-Agents permite entrenar en varios tipos de entornos:

- **Agente único:** Situación normal en la que se entrena un solo tipo de comportamiento. Pero también puedes multiplicar estos agentes para que sean independientes entre ellos, pero usen un mismo Behavior, de esta manera se puede paralelizar el entrenamiento reduciendo tiempos.
- **Self-Play:** Para entrenar a 2 agentes que sean adversarios, las recompensas son inversas. De esta manera los agentes compiten contra un adversario de su mismo nivel.
- **Multiagente:** Situaciones con múltiples agentes que pueden tener el mismo o distinto Behavior. Pueden ser escenarios cooperativos o competitivos.

ML-Agents implementa varios algoritmos de IA del estado del arte. En cuanto a Deep Reinforcement Learning, ofrece Proximal Policy Optimization (PPO) y Soft Actor-Critic (SAC). PPO se usa por defecto y es más estable que otros algoritmos de RL así que es más propósito general. SAC es un algoritmo off-policy lo que lo hace más sample-efficient que un algoritmo on-policy como PPO, pero suele requerir más actualizaciones. ML-Agents también implementa algoritmos de Imitation Learning: Behavioral Cloning (BC) y Generative Adversarial Imitation Learning (GAIL). Estos se pueden usar solos o en conjunto con otros de DRL.

4

Desarrollo

4.1. Planteamiento del problema

En esta sección se va a explicar el entorno de aprendizaje por refuerzo que se ha diseñado. Se ha usado el motor de videojuegos Unity como plataforma para la creación de escenarios y manejo de las físicas con la librería ML-Agents que se encarga de los agentes y sus recompensas. Este entorno dará como resultado diversas builds ejecutables de Unity que se usarán durante el propio entrenamiento de ML-Agents. Esto permite separar el entorno de Unity del entrenamiento.

El agente tiene el objetivo de controlar un coche y debe recorrer una carretera hasta el final sin salirse de ella lo más rápido posible. El agente puede controlar el coche haciéndolo acelerar, frenar y girar.

No se proporciona al agente un mapa de la carretera, debe aprender con información parcial desde su perspectiva. Para observar el entorno, el agente usa raycasts líneas proyectadas desde un punto (centro del modelo del coche) en varias direcciones, principalmente hacia delante. Simulan un sistema de visión simple sin necesidad de usar visión artificial ni redes convolucionales para tratar las imágenes. Los raycasts podrán detectar diferentes tipos de objetos (carretera, checkpoints y muros) y el agente usará esa información para orientarse.

Además, el coche tiene una durabilidad en las ruedas, cuanto menor sea la durabilidad, menos se van a agarrar las ruedas y por ello el agente tendrá más dificultades para tomar curvas sin derrapar o perder el control. El agente deberá adaptarse a la durabilidad de sus ruedas para seguir cumpliendo los objetivos de

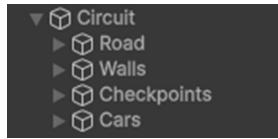


Figura 4.1: Jerarquía de un circuito

no salirse de la carretera y llegar al final lo más rápido que pueda. El entorno está formado por varios escenarios con diferentes carreteras. Esto se hace para enseñar al agente de una forma progresiva diversas situaciones que puede encontrar en escenarios más avanzados. El agente no aprendería bien, se quedaría atascado si se le entrena desde un principio en un circuito avanzado. Primero debe aprender a conducir recto, orientarse hacia los checkpoints y tomar curvas. Después situaciones más avanzadas como curvas complejas. Mediante esta introducción gradual de la complejidad, el agente aprende patrones simples preparándose para situaciones complejas.

4.2. Componentes del escenario

El entorno de entrenamiento está compuesto por varios elementos fundamentales: carreteras, checkpoints, muros y coches. Estos componentes se agrupan en **Circuitos**. Pueden coexistir varios circuitos en una misma escena de Unity. Con esta organización jerárquica permite reutilizar código en todas las escenas porque siguen una misma estructura. La Fig. 4.1 muestra un ejemplo de la jerarquía en el editor de Unity.

4.2.1. Carretera

La carretera de un circuito es formada por tramos modulares (ver Fig. 4.2) que permiten crear circuitos personalizados adaptados para diferentes entrenamientos. Los tramos pueden ser rectas, curva izquierda o curva derecha. Cada tramo incluye un componente **MeshCollider** que permite detectar colisiones con los coches. Los componentes **WheelColliders** de los coches interactúan con los **MeshCollider** para simular la conducción.

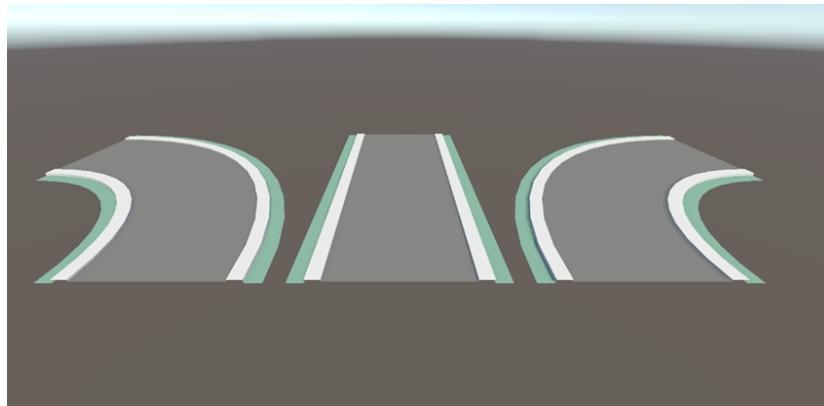


Figura 4.2: Tipos de tramos de carretera

4.2.2. Checkpoints

Los checkpoints (Ver Fig. 4.3) están colocados a lo largo del circuito para guiar al agente durante su conducción. Cada checkpoint se sitúa delante de cada tramo de carretera y existen 3 tipos distintos según al tramo que preceden: recta, curva derecha y curva izquierda. Cada tipo de checkpoint representa el tipo de tramo que hay a continuación. El motivo de diferenciar los checkpoints es porque los agentes no pueden ver más allá del propio checkpoint, son opacos para ellos porque usan raycasts como visión (explicado en profundidad en el apartado 4.3.1). Por lo tanto, hacer diferentes checkpoints da al agente información del tramo siguiente y poder anticiparse al mismo. Por ejemplo, frenar y colocarse cuando el siguiente tramo es una curva. Para diferenciar los tipos de checkpoints, los objetos están etiquetados y el agente puede diferenciar ver las etiquetas.

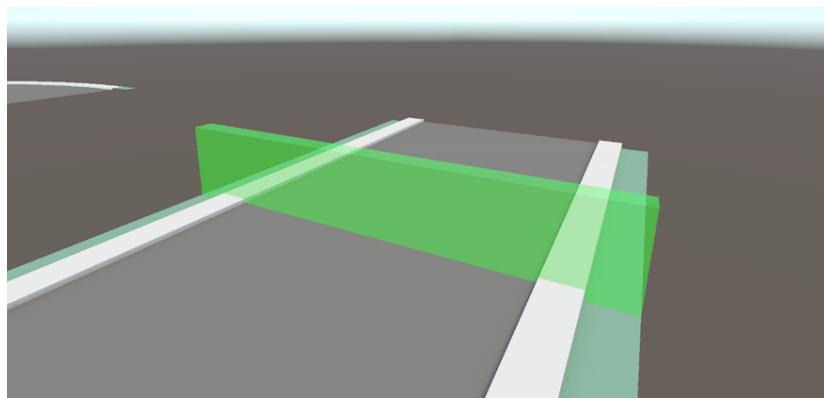


Figura 4.3: Checkpoint en un tramo de recta

Los objetos de checkpoint no tienen un cuerpo físico, son atravesados por los coches. Pero si incorporan un `BoxCollider` configurado como *trigger*. Cuando un

coche atravesese un checkpoint, se detectará una colisión mediante los **Colliders** del checkpoint y del coche lo que realizará una acción, en este caso una recompensa al agente del coche que colisiona.

Los checkpoints en realidad no se comunican directamente con el agente del coche. Si no con un componente **CheckpointController** intermedio. Es un objeto instanciado mediante un componente **Script** que tiene cada coche y se ocupa de la gestión de los checkpoints. Se encarga de llevar la cuenta de los checkpoints cruzados por el coche y dar sus recompensas. Si cruza un checkpoint ya cruzado anteriormente (el coche vuelve hacia atrás), la recompensa será negativa y terminará el episodio. También lleva un temporizador de tiempo pasado desde el último checkpoint que terminará el episodio si llega a cierto umbral, para agentes que se quedan atascados.

4.2.3. Muros

Los muros (Ver Fig. 4.4) definen los límites de la carretera y son zonas de penalización para el agente. Como ocurre con los checkpoints, tienen **BoxColliders** que funcionan como *triggers* que detectan una colisión de un coche. Cuando esto sucede el agente del coche recibe una recompensa negativa y se termina el episodio del agente. Los muros sirven para que el agente aprenda a mantenerse dentro de la carretera.

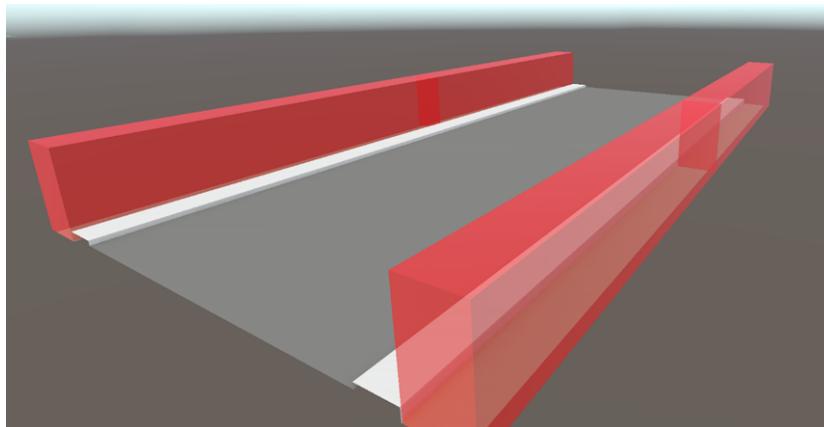


Figura 4.4: Muros colocados en un tramo de recta

4.2.4. Coche

Los coches (Ver Fig. 4.5) son los objetos controlados directamente por los agentes. El agente realmente es un componente **Script** que tiene cada **GameObject** de coche, se explica en mayor profundidad en la sección 4.3. Aparte del agente, el coche está formado por otros componentes que definen sus físicas y movimientos.

- Un **Rigidbody**, que le da una masa, gravedad y comportamiento físico al coche.
- Un **BoxCollider** para que el coche sea detectado por los **BoxCollider** de los checkpoints y muros.
- Cuatro **WheelColliders**, uno en cada rueda, que simulan la fricción, aceleración, frenado, suspensión y giro de las ruedas de forma realista.
- Un script **CheckpointController** encargado de la gestión de los checkpoints cruzados por el coche.
- Un script **CarController** que funciona como una interfaz entre el agente y el funcionamiento físico del coche. Este script, por un lado, contiene medidas que el agente usara como observaciones. Por el otro lado, traduce las acciones del agente a funciones del coche como acelerar, frenar y girar. Realmente el **CarController** ejecuta un bucle de acciones durante toda la ejecución de la escena. En el bucle siempre gira y, acelera o frena, pero con la intensidad que el agente a definido.



Figura 4.5: Modelo del coche en un tramo de recta

También es importante destacar que en una misma escena puede haber múltiples coches. Pero estos múltiples coches se usan únicamente para acelerar el proceso de entrenamiento. Aunque haya varios coches en el mismo circuito, no pueden interactuar ni interferir entre ellos. Los coches no pueden chocarse ni bloquear su visión. Tampoco se pueden cruzar los checkpoints de otros ya que cada uno controla los checkpoints que llevan, no es el circuito quien los gestiona. En resumen, los coches son independientes unos de otros, no compiten entre ellos como si fuera una carrera.

4.3. Agente y entorno

En la anterior sección se ha descrito el funcionamiento del entorno desde una perspectiva física, explicando como interactúan los objetos, como se estructuran los escenarios y que componentes tienen. En esta sección se profundizará en la configuración del entorno como un entorno de aprendizaje por refuerzo utilizando el framework ML-Agents.

ML-Agents necesita un Learning Environment que usará para entrenar. Para ello hay que convertir la escena simple de Unity en un Learning Environment. ML-Agents incluye una clase base **Agent** que puede ser usada para crear agentes añadiéndole un script como componente a un **GameObject**. En este entorno los coches son los objetos que contienen a los agentes. Todo agente debe estar ligado a un **Behavior**, en este caso todos los agentes están ligados al Behavior **CarAgent**. Un **Behavior** sirve para configurar un agente y cada **Behavior** está asociada a una política que es lo que finalmente se está entrenando.

Un **Behavior** define los atributos que deben tener sus agentes. Principalmente especifica el número de observaciones, y el número y tipo de acciones disponibles. Sin embargo, es en el script del agente donde se implementa cuáles son las observaciones y como se gestionan las acciones recibidas.

Los agentes son entidades que recopilan información del entorno y realizar acciones en él. Un agente se gestiona con la clase **Agent**. El agente pasará sus observaciones a su política, esta se ocupará de tomar decisiones y pasarele las acciones que el agente debe tomar. Para crear un agente se crea una clase que herede de **Agent** para implementar los siguientes métodos:

- **OnEpisodeBegin()** - Llamado en el inicio de un episodio, se usa para reiniciar el estado del agente tras terminar un episodio. Como la posición o temporizadores.
- **CollectObservations(VectorSensor sensor)** - El parámetro es un array de un tamaño especificado en el **Behavior** y se debe llenar con las observaciones del entorno que el agente necesitaría para cumplir la tarea.
- **OnActionReceived (ActionBuffers actions)** - El parámetro es un array de tamaño especificado en el **Behavior** y contiene las acciones decididas por la política. Se deben usar esos valores para realizar acciones en el entorno.

El ciclo observaciones-decisión-acción-recompensa se realiza cada vez que el agente pide una decisión, cuando se llama al método **RequestDecision()** del agente. Llamar **RequestDecision()** puede ser útil en entornos donde haya momentos bien marcados de realizar acciones como en un juego por turnos. En cambio, en este entorno de conducción, se realizan decisiones constantemente,

para que el agente mantenga al coche recto, acelerando, girando... Para ello se incluye un componente `Decision Requester` al `GameObject` del coche que se ocupara de realizar llamadas a `RequestDecision()` en intervalos periódicamente. Este agente esta ajustado para realizar una llamada cada 0.1 segundos por lo que realiza 10 ciclos observación-decisión-acción-recompensa en un segundo. Se podría haber seleccionado un intervalo mayor, pero para que el coche fuera reactivo a altas velocidades era necesario uno más bajo a costa de mayor coste de cómputo.

4.3.1. Observaciones

Las observaciones representan la información que el agente obtiene del entorno. El agente debe tener como observaciones, suficiente información para cumplir la tarea. Si no tiene información relevante no aprenderá mucho. Para generar observaciones se puede usar el método `CollectObservations()` para incluir datos numéricos y sensores raycasts.

En el método `CollectObservations()` se rellena un vector con datos principalmente numéricos y no visuales. El agente creado obtiene observaciones relacionados con el estado del coche: velocidad, ángulo de giro, ángulo de derrape, durabilidad de las ruedas y vector dirección. Todas estas observaciones están normalizadas con rangos $[-1, 1]$ o $[0, 1]$ para mejores resultados durante el entrenamiento. Además, se usa la técnica de Stacking que consiste en repetir observaciones de anteriores pasos. De esta forma se consigue una pequeña memoria a corto plazo sin necesidad de añadir una red neuronal recurrente (RNN). El agente tiene un tamaño de stack de 11, esto quiere decir que en cada decisión observa el entorno actual y el entorno en las 10 decisiones anteriores. Como el agente realiza una decisión cada 0.1 segundos, tiene una memoria de 1 segundo. Esto le puede ayudar en curvas para mantener girando el coche mientras siga dentro de la curva.

Estas observaciones solo daban información acerca del coche. Para que el agente tenga una visión del entorno exterior al coche se usan raycasts. Los raycasts son una técnica más ligera que usar observaciones visuales mediante una cámara. Estas observaciones son imágenes, lo que son mucho más pesadas y sería necesaria usar una red neuronal convolucional (CNN). Los raycasts son rayos que salen de un punto (el centro del coche en este caso) y tienen una dirección. Pueden colisionar con un objeto del que obtendrán su etiqueta (Muro y tipos de checkpoints) y distancia del rayo. De esta manera el agente puede ver objetos, conociendo su tipo y a la distancia que esta de ellos. El agente usa 2 grupos de raycasts (Ver Fig. 4.6) apuntando hacia delante: uno de ellos con ángulos en el intervalo $[-7, 7]$ para centrarse en los objetos de delante mientras que el otro grupo usa ángulos en el intervalo $[-80, 80]$ por lo que detecta objetos con más amplitud, útil para giros. Los raycasts también tienen un stack de 11.

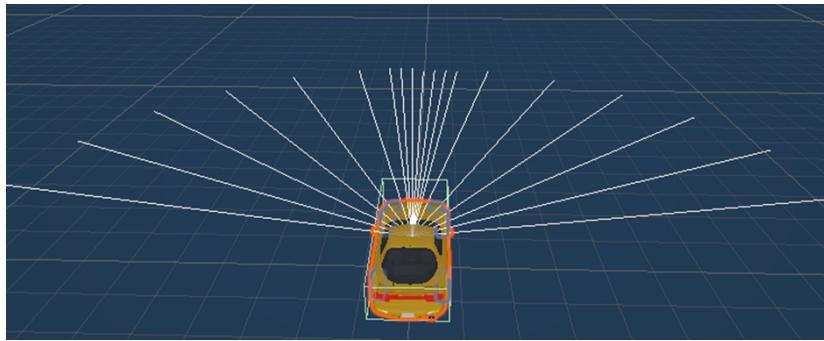


Figura 4.6: Raycasts del agente

4.3.2. Acciones

Una acción es una salida de la política y que el agente debe realizar. ML-Agents y el algoritmo de PPO son capaces de usar tanto acciones discretas como continuas. Para definir que hacen las acciones hay que usar el método `OnActionReceived()`. Si se usan acciones discretas, hay que definir el número de acciones y cuantas posibilidades hay de cada acción. El resultado de una acción discreta será un integer entre $[0, N - 1]$ siendo N el número de posibilidades. En cuanto a acciones continuas simplemente hay que seleccionar el número de acciones, los resultados son siempre un float entre $[-1, 1]$.

En este proyecto se usa un espacio de acciones complemento continuo, ya que se ajusta mejor al control analógico que se usa en un coche real. Simplemente se utilizan 2 acciones:

1. **Aceleración/Frenar** - Un valor $(0, 1]$ significaría aceleración hacia delante, cuanto más cercano a 1 más intensamente acelera. El coche no puede conducirse marcha atrás, por lo tanto, un valor negativo en la acción significa frenar, cuanto más cercano a 1, frena con más intensidad. 0 significa no acelerar ni frenar.
2. **Girar** - Un valor positivo girara hacia la derecha y uno negativo a la izquierda. Igualmente, cuanto más cercano a los extremos, girará con más intensidad. 0 significa no girar.

En el método `OnActionReceived()`, se comunican estos valores al script `CarController` encargado del movimiento del coche. Este script usara esos nuevos valores para realizar las acciones interactuando con las ruedas. según el sistema de físicas en Unity y el `DecisionRequester`, el script `CarController` realizara 5 ciclos de instrucciones por cada acción tomada por el agente. Esto parece contraproducente, pero hace que el coche tenga un movimiento más estable y menos errático.

4.3.3. Recompensas

Las recompensas son señales que guían al agente para obtener un buen resultado. Para implementar recompensas usando ML-Agents hay que usar el método `AddReward()` o `SetReward()` en el agente. Las recompensas deben estar en el intervalo $[-1, 1]$ para una mayor estabilidad durante el entrenamiento. La recompensa del agente se resetea cada nueva decisión. Para observar el entrenamiento se hace una suma total de la recompensa durante todo el episodio.

En este entorno el agente recibe las siguientes recompensas:

- **Checkpoint** - Cada vez que el coche pasa por un checkpoint, el agente recibe un $+0,2$. Los checkpoints guían al agente por la carretera. Si el coche pasa por un checkpoint ya cruzado, recibirá un -1 y también terminará el episodio. La recompensa negativa significa que lo está haciendo mal y se termina el episodio porque es difícil que se recupere de esa situación y es más eficiente que empiece de nuevo.
- **Muros** - Si el coche pasa por un muro recibirá un -1 y se termina el episodio.
- **Velocidad** - El agente recibe constantemente una recompensa negativa muy pequeña. Esto se hace para incentivar la velocidad. Si no existe esta recompensa, el agente podría aprender a pasar por todos los checkpoints, pero muy lentamente ya que la recompensa total será la máxima al pasar por todos. Pero con esta penalización constante, el agente intentará llegar cuanto antes a los checkpoints para que se le penalice las menos veces posibles. Recibe esta penalización en el método `CollectObservations()`, por lo tanto, en cada ciclo observaciones-decisión-acción. La penalización depende de la velocidad del coche.

4.4. Escenarios

En esta sección se mostrarán y explicarán todos los escenarios utilizados para el entrenamiento del agente. Como se ha comentado previamente, el entrenamiento se ha estructurado de forma progresiva en escenarios con cada vez una dificultad mayor. El objetivo de dividir el aprendizaje en varios escenarios es que el agente aprenda situaciones más sencillas como moverse en línea recta y no salirse de la carretera. Y después aprenda a tratar situaciones más complejas como diferenciar tipos de checkpoints, gestionar curvas encadenadas y adaptar su conducción a la durabilidad de las ruedas.

Entrenar al agente directamente en el escenario más complejo no da buenos resultados, se queda atascado y no consigue aprender.

Para poder completar el último escenario el agente debe haber aprendido a:

- Mantener el coche recto y no salirse de la carretera.
- Girar hacia los checkpoints y evitar los muros.
- Reconocer los tipos de checkpoints y anticipar el siguiente tipo de tramo.
- Adaptar su conducción a la durabilidad de las ruedas para evitar salirse de la pista o derrapes fallidos.

A continuación, se detallan los escenarios creados:

4.4.1. Forward

En este primer escenario (Ver Fig. 4.7) el circuito es una pequeña recta. El objetivo es que el agente aprenda a moverse y no desviarse y salirse de la carretera. El escenario está formado por 3 circuitos y cada uno usa 1 de los 3 tipos de checkpoints. Se usan checkpoints de curvas, aunque no las haya. El motivo es que el agente aprenderá que esos 3 tipos de etiquetas son buenos, le dan recompensas y los muros le penalizan. Todavía no se le enseña a diferenciar los tipos de checkpoints, eso será en un escenario más adelante. También esta desactivada la durabilidad de las ruedas está al 100 % para que no tenga problemas con ello, se mantendrá desactivada hasta el penúltimo escenario. Los agentes aparecen aleatoriamente en cualquier punto dentro del primer tramo, pero siempre orientados hacia delante.

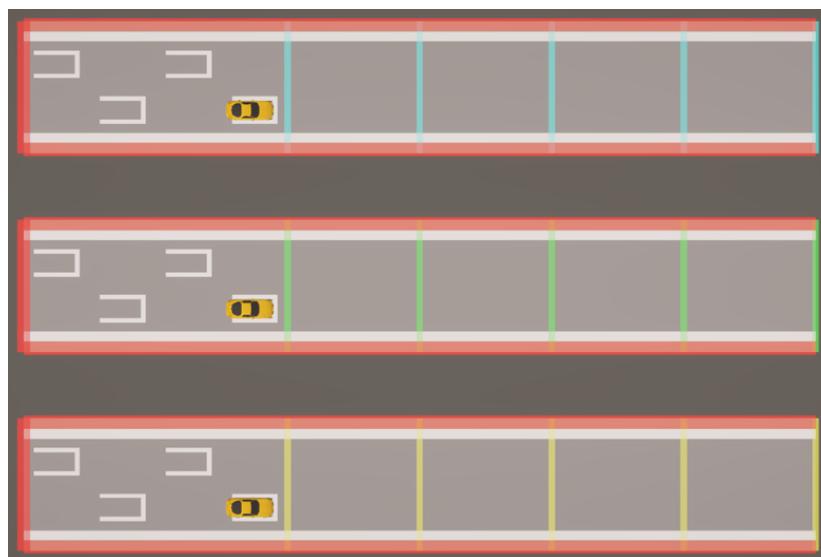


Figura 4.7: Escenario Forward

4.4.2. Reorientation

Este simple escenario (Ver Fig. 4.8) sirve para que el agente aprenda a orientarse hacia el checkpoint para moverse hacia él. Los agentes aparecen en el centro, pero con ángulos aleatorios entre $[-90, -30] \cup [30, 90]$ con centro en el checkpoint. Es una preparación antes de tomar curvas y el coche este desalineado con el siguiente checkpoint.

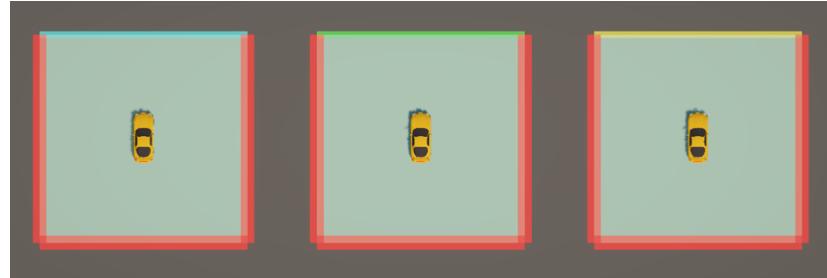


Figura 4.8: Escenario Reorientation

4.4.3. Turn

En el tercer escenario (Ver Fig. 4.9) el agente aprende a tomar curvas y diferenciar los tipos de checkpoint que les informa sobre que viene en el siguiente tramo. Los agentes aparecen aleatoriamente como en Forward, lo que le permite tomar curvas desde diferentes puntos. La durabilidad sigue desactivada para que el agente pueda perfeccionar el giro sin más condiciones.



Figura 4.9: Escenario Turn

4.4.4. Loop

El escenario (Ver Fig. 4.10) se compone de 2 circuitos donde el agente aprende a dar vueltas. Ya aprendió a girar en el anterior escenario, pero este combina curvas con rectas para que aprenda a alternar entre diferentes tramos.

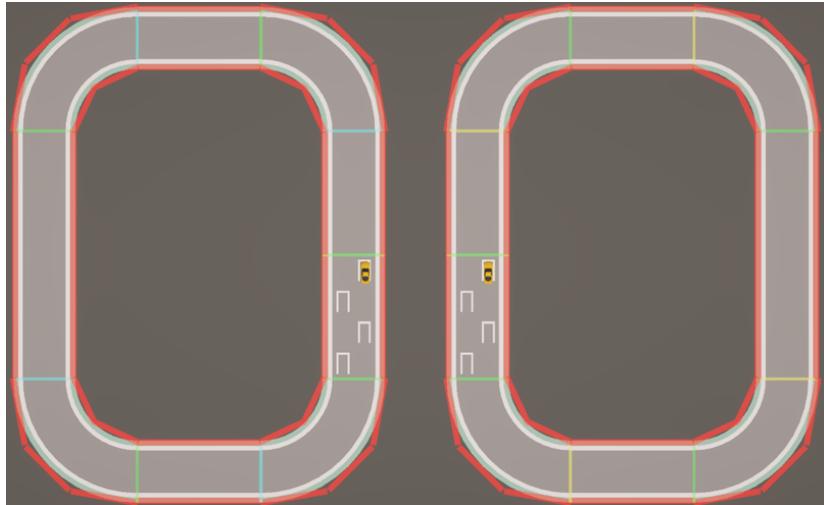


Figura 4.10: Escenario Loop

4.4.5. Curves

El agente aprenderá técnicas más complejas en este escenario (Ver Fig. 4.11). El escenario está formado por 3 circuitos duplicados en modo espejo. En los circuitos hay composiciones de curvas con otro tramo detrás complicado y que el agente puede salirse con facilidad si no ha ajustado correctamente la velocidad y ángulo de giro. El agente aparece también aleatoriamente dentro del primer tramo como en escenarios anteriores, pero además esta vez iniciará con cierta velocidad aleatoria (o quieto). El agente aprenderá a resolver estas situaciones más complejas.

Este escenario tiene 2 versiones. La primera con la durabilidad desactivada. La segunda esta activada y los agentes aparecerán con una durabilidad aleatoria en las ruedas del coche. Por lo que, tras aprender a realizar las curvas con el máximo agarre en las ruedas, ahora deben aprender a entender su observación de durabilidad y adaptar su forma de conducir según ello.

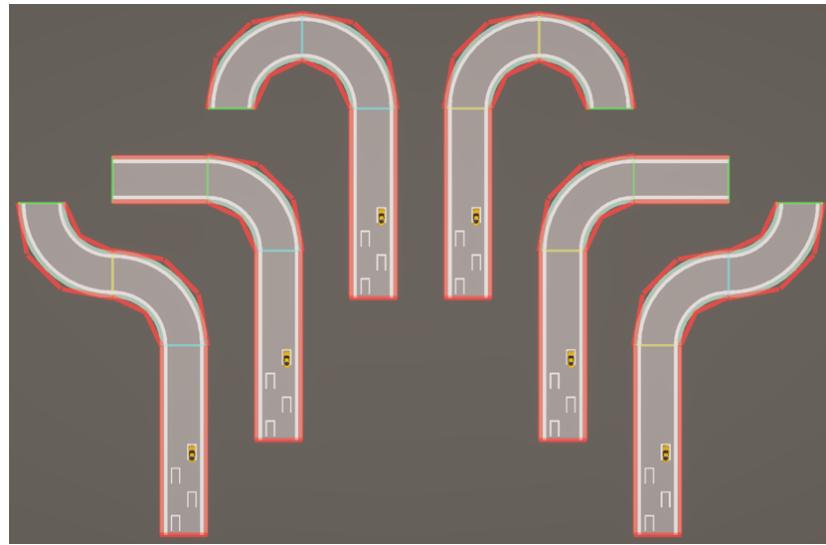


Figura 4.11: Escenario Curves

4.4.6. Circuit

En el último escenario (Ver Fig. 4.12) el agente demostrará todo lo que ha aprendido en los escenarios anteriores. El circuito está compuesto por situaciones encontradas en los anteriores escenarios formando un circuito: tramos rectos, curvas encadenadas y desgaste de ruedas. La durabilidad de las ruedas esta activada desde el principio por lo que también se adaptará a ello.

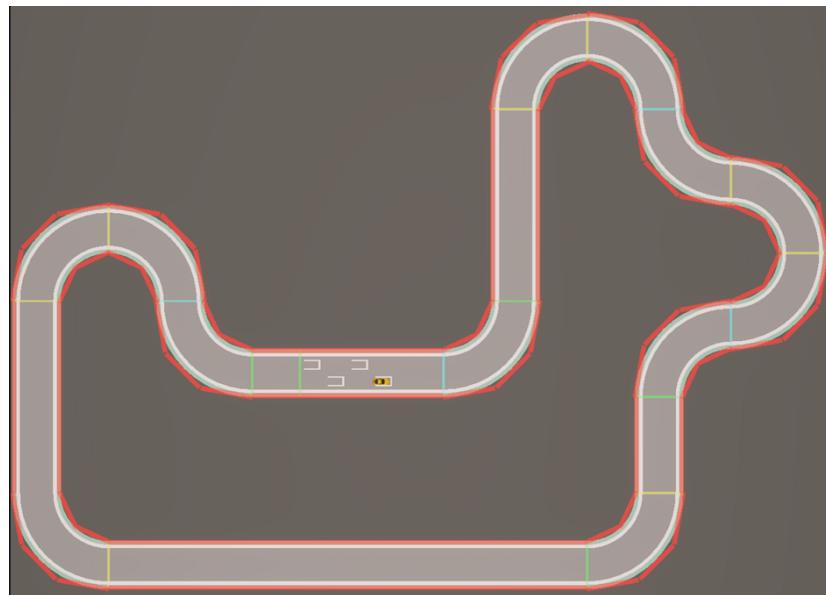


Figura 4.12: Escenario Circuit

5

Entrenamiento

En esta sección se describe el proceso de entrenamiento del agente en el entorno de aprendizaje desarrollado en la anterior sección. En la sección de Desarrollo tenemos como resultado final builds ejecutables de Unity (una por cada escena) que servirán como Learning Environments. El framework de ML-Agents contiene un paquete de Python (`mlagents`) que tiene algoritmos de DRL ya implementados. El framework usará las builds de Unity como entornos de aprendizaje para el algoritmo que se use para entrenar. El proceso de entrenamiento consiste en:

1. Preparar el archivo de configuración que contiene todas las configuraciones e hiperparámetros [24] [25] del algoritmo que se usarán durante el entrenamiento .
2. Ejecutar el entrenamiento mediante el comando `mlagents-learn` usando el archivo de configuración creado y la build de Unity.
3. Analizar los resultados utilizando herramientas de visualización como Tensorboard.

El proceso no es sencillo ni inmediato porque muchos de los hiperparámetros requieren prueba y error para ajustarse correctamente. Los resultados obtenidos tras los entrenamientos se analizan para evaluar el rendimiento del agente observando gráficas sobre la recompensa acumulada en Tensorboard o probando la red neuronal en el entorno directamente. Si no se obtiene un buen rendimiento puede ser necesario modificar los hiperparámetros o incluso modificar el entorno

(observaciones del agente, posición de checkpoints, recompensas...). Por lo tanto, el ciclo de entrenamiento se repite en múltiples ocasiones hasta obtener un comportamiento bueno.

A continuación, se explicará:

- La configuración final utilizada para el entrenamiento.
- La evolución del agente durante su entrenamiento en los diversos escenarios.
- Algunos de los problemas encontrados durante el entrenamiento y como se han solucionado.

5.1. Configuración del entrenamiento

El archivo de configuración es la única entrada del comando `mlagents-learn` que es el que realiza el entrenamiento. El archivo contiene los datos necesarios para comenzar el entrenamiento. Es un archivo YAML con opciones de configuración sobre diferentes temas:

- Opciones de entorno: principalmente para especificar la build de Unity utilizada y el número de instancias a lanzar simultáneamente.
- Opciones del motor: relacionadas con la resolución durante el entrenamiento (no se usan gráficos en este caso) y la escala de tiempo (ajustada en x20 para aumentar la velocidad, valores mayores pueden producir errores en la simulación de las físicas).
- Opciones de puntos de control: Especificar ID's de entrenamiento y posibles inicializaciones a partir de un entrenamiento anterior.
- Opciones de dispositivo: Seleccionado CUDA para ejecutar el algoritmo en GPU (las instancias de Unity siguen usando CPU).
- Hiperparámetros de agentes: Incluye los hiperparámetros y el algoritmo usado para entrenar los agentes. A continuación, se explicarán los valores seleccionados en los hiperparámetros.

Pero antes, para entender los hiperparámetros utilizados, se mostrará y explicará un pseudocódigo muy simplificado del algoritmo PPO [26] (Ver Algoritmo 1) que es el algoritmo utilizado en los entrenamientos.

Algoritmo 1 PPO Simplificado

```

1: Iniciar parámetros y redes neuronales
2: while  $steps < max\_steps$  do
3:   {Recopilación de experiencias}
4:   for  $1..num\_agents$  do
5:     Interactuar durante  $time\_horizon steps$ 
6:     Introducir experiencias en el  $batch$ 
7:   end for
8:   {Actualización de redes neuronales}
9:   for  $1..num\_epoch$  do
10:    for  $minibatch$  in  $batch$  do
11:      Actualizar redes neuronales
12:    end for
13:   end for
14: end while

```

El algoritmo consiste en iterar en un bucle hasta llegar al número máximo de $steps$ (max_steps) especificado. Este hiperparámetro se aumentará al principio hasta saber cuántos $steps$ necesita el agente para aprender en el entorno. Cada iteración se divide en 2 partes: recopilación de experiencias y actualización de las redes neuronales. En la primera parte, los agentes utilizados interactúan en el entorno. Recogen experiencias durante $time_horizon steps$ y las introducen en un buffer llamado $batch$. Los agentes se ejecutan en paralelo (también puede haber un solo agente). Después se realiza la actualización de las redes neuronales. Para ello se usan los datos recogidos en el $batch$ en la anterior etapa. Se divide el contenido del $batch$ en $minibatch$ y se realizan actualizaciones por cada $minibatch$. Además, todo ese proceso se repite un número de $epoch$ especificado.

El hiperparámetro $time_horizon$ especifica el número de $steps$ que recorre un agente antes de introducir todas las experiencias en el $batch$. El tamaño del $time_horizon$ debe ser lo suficientemente grande para capturar una secuencia de acciones importante de una vez. Por ejemplo, todos los $steps$ que ocurren desde que comienza una curva hasta que la acaba. En los entornos creados se ha llegado a la conclusión de que lo ideal sería que esta secuencia de acciones consiga abarcar 2 tramos de carretera ya que en el primero, el agente se puede anticipar gracias al checkpoint y en segundo tramo se realiza con la anticipación propuesta. Según múltiples pruebas realizadas, el tiempo medio en recorrer 2 tramos una vez ya ha sido entrenado es de $\sim 60 steps$. Así que el $time_horizon$ final es de $64 steps$.

En cuanto al número de agentes en paralelo. ML-Agents permite colocar múltiples agentes en una escena como se ha visto en el apartado anterior, lo que es útil para acelerar y estabilizar el proceso de entrenamiento. Pero ML-Agents también permite lanzar múltiples instancias de Unity durante el entrenamiento, y cada instancia puede contener múltiples agentes. Todas las pruebas las siguen-

Número de instancias	Tiempo de entrenamiento (min)
1	9.883
2	6.16
3	5.395
4	4.73
5	4.371
6	4.221
7	4.191
8	4.17

Tabla 5.1: Tabla con tiempos de entrenamiento según número de instancias

tes pruebas se realizaron en el primer escenario Forward manteniendo todos los parámetros que no están siendo valorados en la prueba en específico. El aprendizaje con una única instancia y un agente en ella duró más de 2 horas, y con las mejoras aplicadas el tiempo baja a ~ 1.5 minutos. Primero se ha ajustado el número de instancias que se van a ejecutar simultáneamente. Las instancias de Unity se ejecutan en CPU, principalmente mononúcleo, por lo tanto, hay que tener en cuenta el número de núcleos de la CPU utilizada para el entrenamiento. En este caso se utilizó una CPU con 6 núcleos (Ryzen 5 7600). Se realizaron pruebas con distintos números de instancias, pero manteniendo el resto de los parámetros y hubo mejoras significativas del tiempo entre 1 y 6 instancias, pero a partir de 6 las diferencias de tiempos eran mínimas. Así que al final se seleccionó 6 instancias simultaneas. En la Tabla 5.1 se pueden observar los tiempos de los entrenamientos. Todas las pruebas fueron realizadas en el escenario Forward durante $1M$ de *steps*.

Después de ajustar el número de instancias se especificó el número de agentes por instancia. La primera idea fue poner tantos agentes como el procesador sea capaz de soportar observando el uso de la CPU durante la ejecución y los tiempos en realizar los entrenamientos. Se realizaron diversas pruebas con diferente número de agentes por instancia y se puede observar lo siguiente: cuanto más agentes, menor es el tiempo en realizar $Xsteps$, pero la disminución de tiempo, como con el número de instancias, no escala de forma lineal. La gráfica que se muestra en Fig. 5.1 muestra la escalada de tiempo según número de agentes que ocurre en realidad.

Según la anterior gráfica se propuso poner tantos agentes en instancia como soporte la CPU o hasta que los tiempos dejen de crecer porque parece que hay una asíntota al tener forma logarítmica. Sin embargo, esta idea cambio analizando la gráfica de Fig. 5.2. En esta gráfica se demuestra que cuantos más agentes, más tarde se llega a la máxima recompensa acumulada del escenario donde se realizan las pruebas. Pero también cuanto más agentes, menos tiempo se tarda en llegar a $Xstep$. Por lo tanto, hay que encontrar un equilibrio entre en que *step* de ha

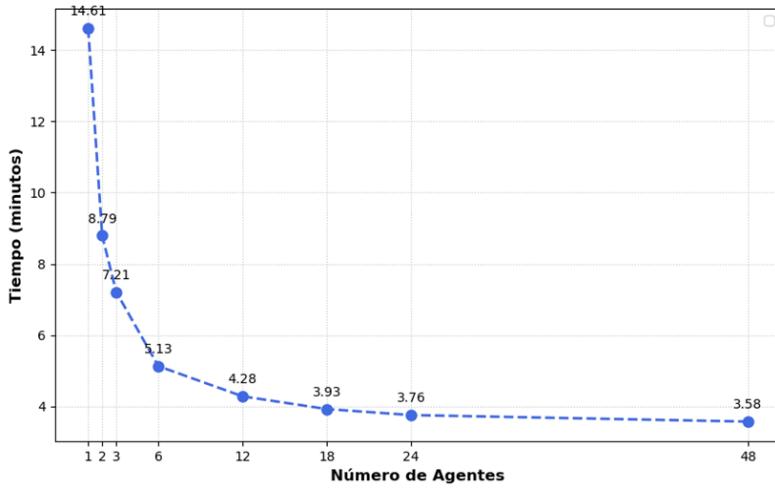


Figura 5.1: Gráfica que representa la mejora de tiempos según número de agentes en escena

llegado a la máxima recompensa y el tiempo que tarda en llegar a él. El número óptimo de agentes por instancia encontrado es de 12.

Una vez se ha ajustado el número total de agentes se selecciona el valor óptimo de tamaño de *batch*. El *batch* es un buffer que los agentes llenan mientras recopilan experiencias con las observaciones, acciones y recompensas. Un mayor *batch* conlleva un entrenamiento más estable. Sin embargo, en los entornos utilizados, la recompensa es demasiado dispersa y un aumento de *batch* resulta en peor rendimiento y no aprendizaje (representado en la Fig. 5.3). Se ha escogido un tamaño de *batch* de 1280 (*minibatch* de 128) por su pronto aprendizaje en menor tiempo.

El resto de hiperparámetros se han mantenido en los valores por defecto que incluye ML-Agents. Se probaron algunas configuraciones con pequeñas variaciones, pero no producían mejoras significativas.

- α (Learning rate): $3e^{-4}$ con disminución lineal hasta 0
- β (entropía): $5e^{-3}$
- ϵ (clip ratio): 0,2 con disminución lineal hasta 0
- λ (GAE): 0,95
- Num_epoch : 3
- Neural Network
 1. *Hidden_units*: 256
 2. *Num_layers*: 2

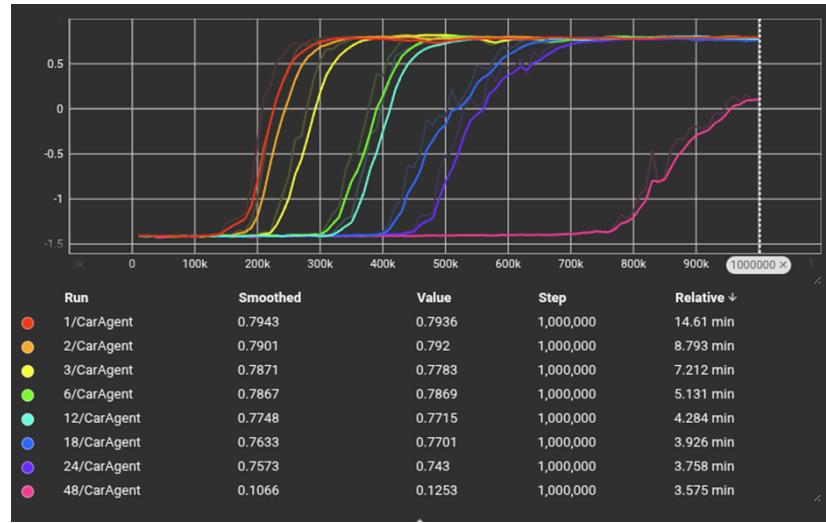


Figura 5.2: Tabla y gráfica de los entrenamientos de prueba de número de agentes por instancia

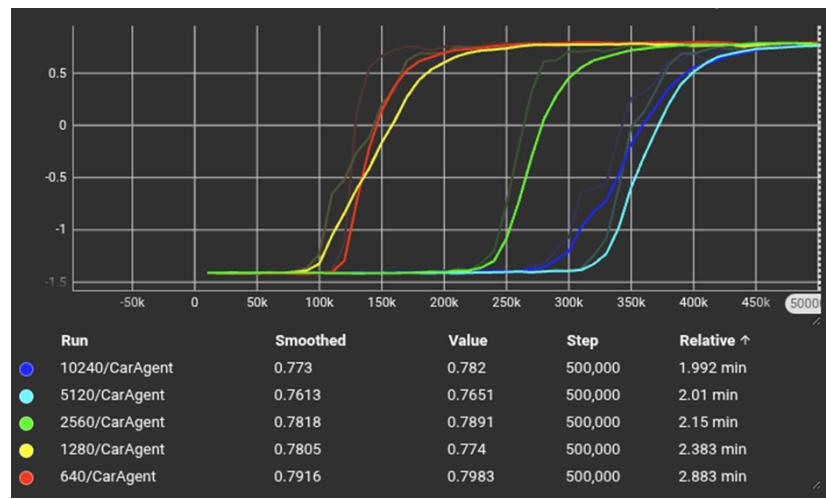


Figura 5.3: Gráfica con entrenamientos de prueba de tamaño de *batch*

El tiempo total para realizar todas las pruebas ha sido de más de 70 horas.

5.2. Evolución del entrenamiento

En esta sección se muestra el progreso del agente a lo largo del entrenamiento en los escenarios desarrollados. Se presentan videos acelerados de varias fases del proceso y gráficas de la recompensa acumulada. Las gráficas también incluyen el tiempo requerido para cada entrenamiento. El agente tarda un total de ~157 minutos en pasar por todos los escenarios.

Forward

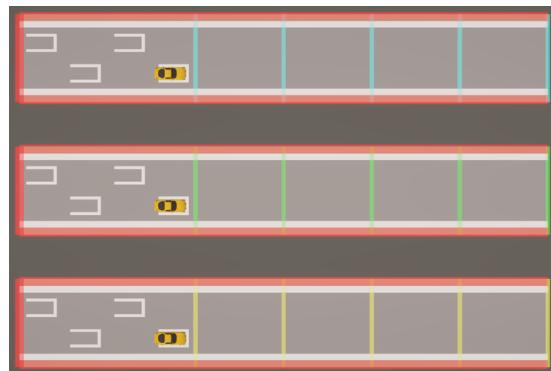


Figura 5.4: Video resumen del entrenamiento en Forward

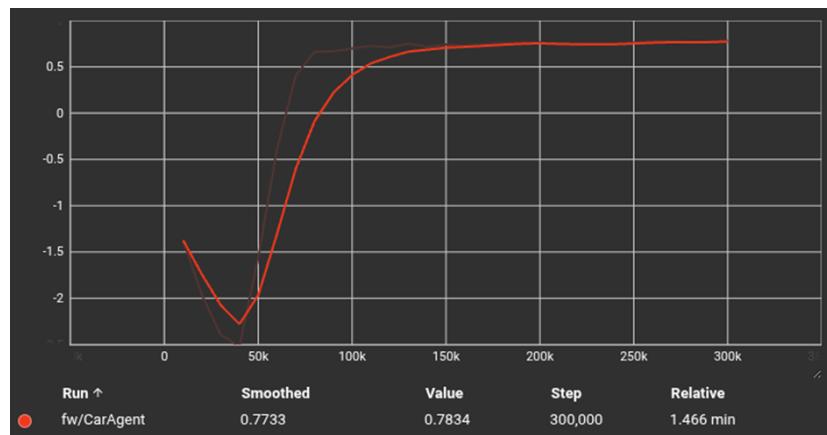


Figura 5.5: Gráfica del entrenamiento en Forward

Reorientation

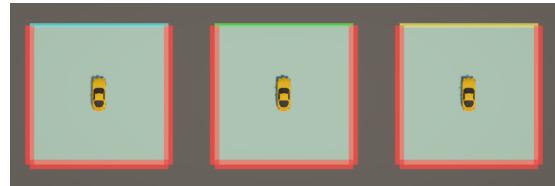


Figura 5.6: Video resumen del entrenamiento en Reorientation

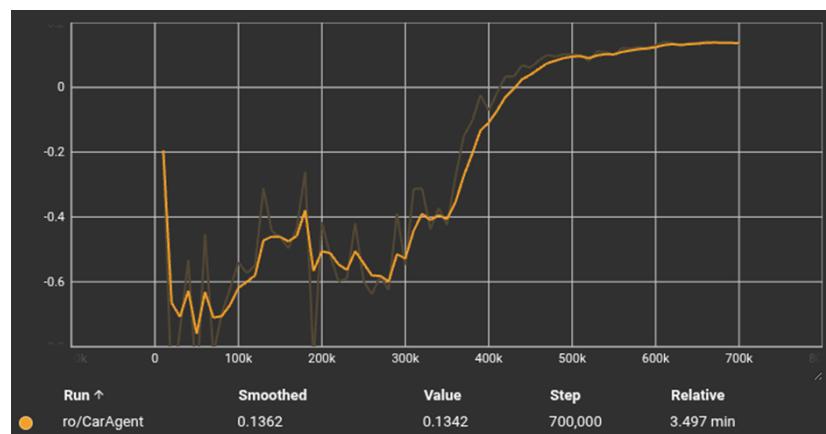


Figura 5.7: Gráfica del entrenamiento en Reorientation

Turn



Figura 5.8: Video resumen del entrenamiento en Turn

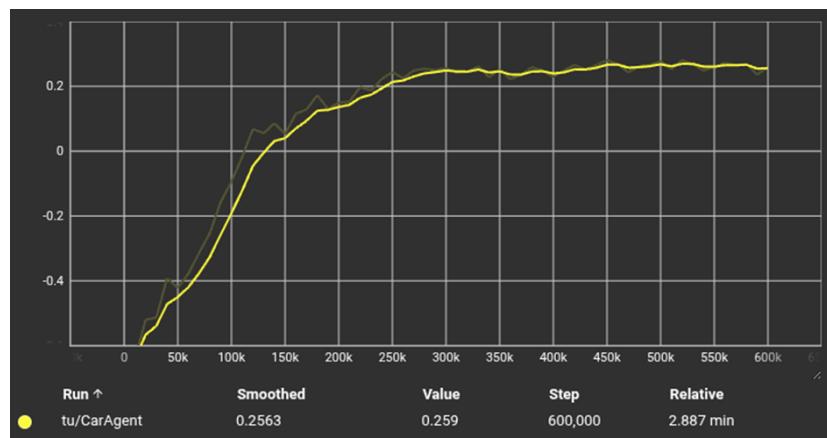


Figura 5.9: Gráfica del entrenamiento en Turn

Loop

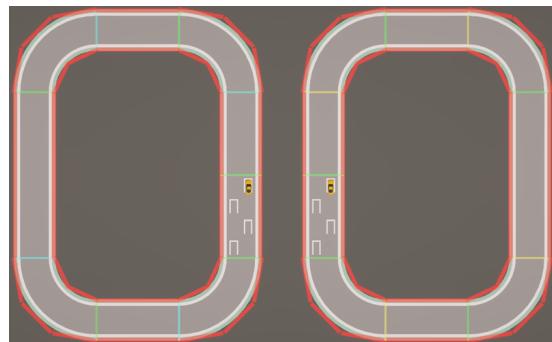


Figura 5.10: Video resumen del entrenamiento en Loop



Figura 5.11: Gráfica del entrenamiento en Loop

Curves

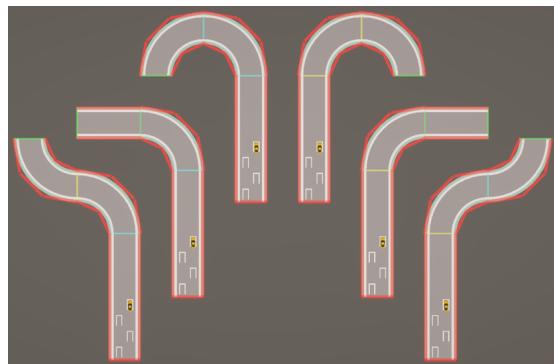


Figura 5.12: Video resumen del entrenamiento en Curves

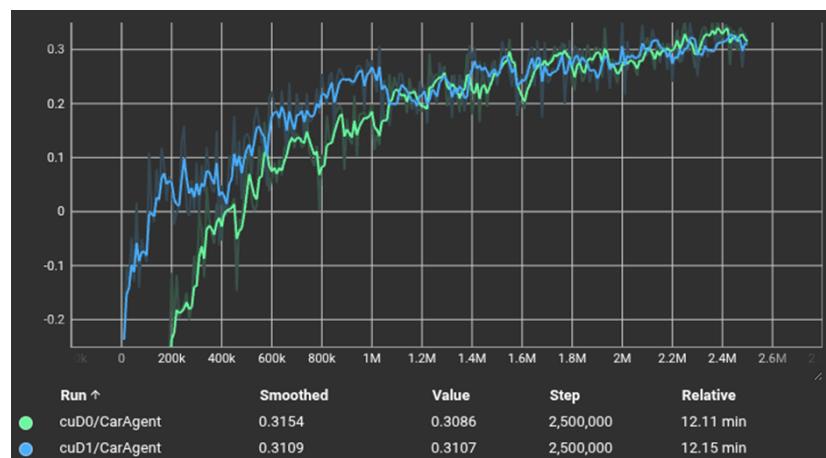


Figura 5.13: Gráfica del entrenamiento en Curves

Circuit

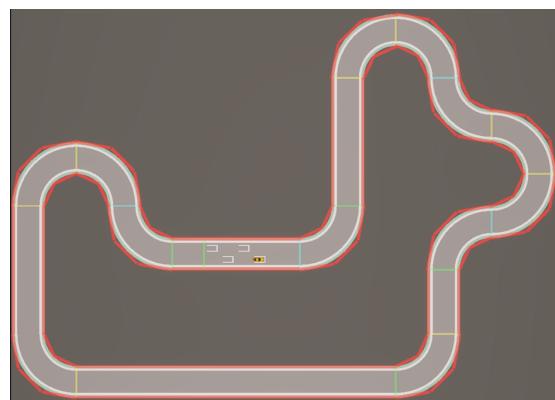


Figura 5.14: Video resumen del entrenamiento en Circuit

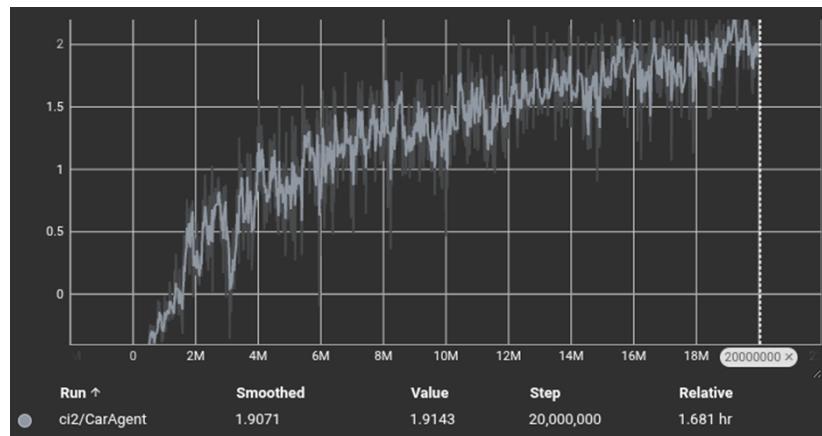


Figura 5.15: Gráfica del entrenamiento en Circuit

5.2.1. Comparación de entrenamiento con durabilidad

El siguiente video muestra una comparación entre 2 agentes entrenados en condiciones diferentes. Los 2 fueron entrenados igual hasta el escenario Curves, a partir de ahí:

- **Agente A:** Entrenó en Curves y paso a Circuit, los 2 escenarios con la durabilidad desactivada.
- **Agente B:** Tras entrenar en Curves con durabilidad desactivada, después entrenó en el mismo escenario y en Circuit con la durabilidad activada.

En el video los 2 agentes se enfrentarán en el escenario Circuit con una durabilidad de 30 % en las ruedas para comprobar si el entrenamiento en durabilidad ha mejorado realmente la adaptación del agente.



Figura 5.16: Video comparativa de los 2 agentes con durabilidad al 100 %



Figura 5.17: Video comparativa de los 2 agentes con durabilidad al 30%

5.3. Problemas encontrados durante el entrenamiento

Durante el proceso de entrenamiento se detectaron problemas que afectaban al rendimiento y comportamiento del agente. A continuación, se describen algunos de los problemas y las soluciones aplicadas.

5.3.1. Velocidad del coche

Una vez el agente fue capaz de superar correctamente todos los escenarios propuestos sin salirse de la carretera, se observó que el comportamiento todavía no era óptimo. Aunque el agente completaba los circuitos, lo hacía a una velocidad relativamente alta, pero con frenadas innecesarias justo después de cada checkpoint. Este comportamiento afectaba negativamente a la fluidez de la conducción, ya que el agente disminuía la velocidad incluso en tramos rectos donde no era necesario frenar. El problema estaba en el diseño original de los checkpoints. Inicialmente, solo existía un único tipo de checkpoint con la función era marcar el camino a seguir. Pero para un agente, un checkpoint es un objeto opaco que no permite ver lo que hay detrás hasta que lo atraviesa, ya que los raycasts no pueden traspasarlo. Por tanto, el agente no podía anticipar si tras cruzarle habría una curva o una recta. Para evitar penalizaciones por salirse de la carretera, reducía la velocidad como medida de precaución, incluso en tramos donde podía mantenerse a alta velocidad. El motivo realmente es que la penalización por salirse de la carretera es mayor que ir un poco más lento. La solución fue la implementación de 3 tipos de checkpoint con el objetivo de que cada uno “informara” sobre el siguiente tramo. La solución está inspirada en un copiloto de rally que avisa al piloto sobre los tramos que vienen a continuación. También se rediseñaron algunos escenarios para incluir un aprendizaje de diferenciación de cada checkpoint. De esta forma, el agente puede anticiparse al tramo siguiente sin necesidad de reducir la velocidad.

5.3.2. Baja utilización de GPU

Uno de los objetivos durante el entrenamiento fue aprovechar la GPU para reducir los tiempos de cómputo. Los entrenamientos se configuraron para utilizar GPU, por lo que se usarían el algoritmo de PPO versión CUDA implementado en ML-Agents. Sin embargo, la utilización de la GPU no fue la esperada y no se consiguieron mejoras significativas de los tiempos de entrenamiento. El bajo aprovechamiento se debe a dos factores principales:

- **Instancias de Unity:** En cada iteración del algoritmo PPO, los agentes interactúan con el entorno para recopilar experiencias que luego se utilizan para actualizar la red neuronal. Esta fase de recopilación de experiencias se ejecuta mediante múltiples instancias de Unity, que corren en paralelo utilizando principalmente la CPU. Esta etapa es la más costosa en tiempo dentro del ciclo de entrenamiento, pero no se beneficia del uso de GPU, ya que las instancias de Unity no están optimizadas para aprovechamiento de GPU.
- **Red neuronal pequeña:** La red neuronal utilizada para los entrenamientos es pequeña, con 2 capas ocultas de 256 unidades. Este tamaño es suficiente para resolver los entornos planteados. Se realizaron pruebas aumentando el tamaño de la red, lo cual permitió alcanzar una utilización de GPU de aproximadamente un 50 %, pero no se observó una mejora sustancial en el aprendizaje. Además, una red más grande implica un mayor tiempo de entrenamiento, ya que hay más parámetros que ajustar.

6

Conclusión

6.1. Resultado final

Se han cumplido los objetivos propuestos para el proyecto. En este proyecto se ha diseñado un agente y un entorno de aprendizaje por refuerzo. Después se ha entrenado exitosamente dicho agente para superar los escenarios y aprender lo propuesto. Además, se ha ajustado la configuración del entrenamiento para optimizar el proceso.

El agente consigue controlar el coche por el entorno con unas simples observaciones del estado del coche y unos raycasts que usa como visión del exterior. Las observaciones del coche le permiten ajustar las entradas que le da al coche para controlarlo con mayor eficacia. La visión exterior que le proporcionan los raycasts le permite diferenciar tipos de objetos y la distancia y dirección a la que están. El agente ha aprendido a diferenciar muros de checkpoints y qué hacer ante cada uno. Además, puede diferenciar los tipos de checkpoints para anticiparse al siguiente tramo posicionándose y frenando o seguir acelerando.

Tras los entrenamientos, el agente es capaz de recorrer tanto el circuito final propuesto, como cualquier circuito que se cree con las mismas reglas, posicionamiento de checkpoints y muros, y uso de mismo tipo de tramos que en los escenarios utilizados para el entrenamiento. Se ha logrado que el agente no se dedique simplemente a moverse en dirección hacia los checkpoints sino que también lo hace rápidamente. Gracias a la penalización constante, el agente intenta llegar al siguiente checkpoint lo más rápido que puede para recibir menos pena-

lizaciones, esto resulta en ir a mayor velocidad. En los últimos escenarios se ha cambiado la durabilidad de las ruedas haciéndolas más deslizantes. Sin embargo, el agente ha sido capaz de aprender a adaptarse a la durabilidad y cambiar su conducción para seguir cumpliendo los objetivos de no salirse de la carretera e ir lo más rápido posible.

El método de entrenamiento progresivo en escenarios con cada vez más complejidad ha funcionado correctamente. Se han realizado pruebas entrenando al agente de 0 en el último escenario y los resultados no son positivos ya que no llega a aprender nada. Teóricamente, aumentando mucho la entropía para mayor exploración, no sería imposible que pudiera ser entrenado de esta manera. Sin embargo, es un entrenamiento demasiado aleatorio para ser utilizado en la práctica. En el entrenamiento progresivo se han creado escenarios ideados para el aprendizaje de técnicas simples al principio e incorporar otras más complejas después. Este método progresivo guía al agente durante su aprendizaje y se acelera en gran medida.

En cuanto a la durabilidad de las ruedas, los entrenamientos realizados explícitamente para aprender a adaptarse han sido efectivos como se mostró en el video de la Fig. 5.17. Como se puede observar el agente que no ha sido entrenado para ello, no es capaz de completar el circuito cuando la durabilidad de los neumáticos no está al 100 %.

Con el ajuste de las configuraciones para el entrenamiento y los hiperparámetros del algoritmo se ha logrado una mejora muy significativa de los tiempos empleados para los entrenamientos. Antes de ajustar ninguna configuración, utilizando los parámetros por defecto, el agente necesitaba un entrenamiento de más de 2 horas en el primer escenario para completarlo. En cambio, al final, el agente tarda ~ 1.5 minutos.

6.2. Trabajo futuro

Este proyecto tiene algunas limitaciones y optimizaciones que pueden mejorarse. El agente está muy limitado al entorno y sus reglas. Se pueden probar diferentes tipos de tramos: diferentes tipos de curvas (más abiertas o más cerradas), diferentes anchos de carretera e incluso tramos que sean irregulares (curva que empieza más cerrada y luego se abre, o tramo que cambie su ancho progresivamente). Por otro lado, las carreteras utilizadas en el entorno son totalmente planas, también se podría implementar subidas y bajadas o terrenos irregulares simulando tierra, arena o grava. Actualmente la única condición del entorno es la durabilidad de los neumáticos, pero también podría cambiar el “clima” simulando lluvia o nieve para mayor deslizamiento o viento que empuje el coche. Otra mejora sería que el agente compita contra otros agentes. En los escenarios de entrenamiento había múltiples agentes en la misma carretera, pero los agentes eran

independientes, sus interacciones no afectaban a la de los otros. Se podrían implementar coches que puedan colisionar y agentes puedan observar el resto de los coches. El agente debería aprender a no chocar con el resto de los coches, seguir conduciendo rápido, adelantar o incluso bloquear adelantamientos. Estos cambios requerirían más observaciones exteriores por parte del agente, más tipos de checkpoints que den información del siguiente tramo o nuevos tipos de observaciones para diferenciar tramo y condición del terreno. Es posible que haya que aumentar el tamaño de la red neuronal para ser capaz de aprender un comportamiento más complejo.

En cuanto al entrenamiento, se ha usado el algoritmo PPO por su versatilidad siendo funcional en muchos tipos de entornos y ser muy usado en robótica que es un entorno similar a este. Sin embargo, hay otros algoritmos como SAC (Soft Actor-Critic) que también da buenos en este tipo de entornos y es más sample-efficient al ser off-policy. A parte de probar otros algoritmos se pueden añadir otras técnicas como el Imitation Learning o recompensas intrínsecas para acelerar los entrenamientos.

La configuración del entrenamiento y los hiperparámetros ha dado buenos resultados reduciendo en gran medida los tiempos de aprendizaje. Sin embargo, el tiempo empleado entrenando agentes de prueba para analizar y comparar resultados ha sido demasiado grande. Sería necesario experimentar con herramientas dedicadas a la optimización de hiperparámetros como Weights&Biases-Sweeps que se encargan de automatizar los entrenamientos de prueba y analizar los hiperparámetros más importantes y su ajuste óptimo.

Bibliografía

- [1] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach.* pearson, 2016.
- [2] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, “A survey of deep reinforcement learning in video games,” *arXiv preprint arXiv:1912.10944*, 2019.
- [3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, “Mastering the game of go with deep neural networks and tree search,” *nature*, vol. 529, no. 7587, pp. 484–489, 2016.
- [5] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse *et al.*, “Dota 2 with large scale deep reinforcement learning,” *arXiv preprint arXiv:1912.06680*, 2019.
- [6] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, “Deep reinforcement learning for autonomous driving: A survey,” *IEEE transactions on intelligent transportation systems*, vol. 23, no. 6, pp. 4909–4926, 2021.
- [7] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “Carla: An open urban driving simulator,” in *Conference on robot learning.* PMLR, 2017, pp. 1–16.
- [8] W. Zhao, J. P. Queralta, and T. Westerlund, “Sim-to-real transfer in deep reinforcement learning for robotics: a survey,” in *2020 IEEE symposium series on computational intelligence (SSCI).* IEEE, 2020, pp. 737–744.
- [9] M. Vasco, T. Seno, K. Kawamoto, K. Subramanian, P. R. Wurman, and P. Stone, “A superhuman vision-based reinforcement learning agent for autonomous racing in gran turismo,” *arXiv preprint arXiv:2406.12563*, 2024.
- [10] A. Ng, A. Bagul, G. Ladwig, and E. Shyu, “Supervised machine learning: Regression and classification,” [Online]. Disponible: <https://www.coursera.org/learn/machine-learning>, 2022, [Accedido: 25-junio-2025].
- [11] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [12] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* .O'Reilly Media, Inc., 2022.
- [13] L. Graesser and W. L. Keng, *Foundations of deep reinforcement learning: theory and practice in Python.* Addison-Wesley Professional, 2019.
- [14] M. Hu, *The Art of Reinforcement Learning.* Springer, 2023.
- [15] A. White and M. White, “Fundamentals of reinforcement learning,” [Online]. Disponible: <https://www.coursera.org/learn/fundamentals-of-reinforcement-learning>, 2019, [Accedido: 25-junio-2025].

BIBLIOGRAFÍA

- [16] Y. Li, “Deep reinforcement learning: An overview,” *arXiv preprint arXiv:1701.07274*, 2017.
- [17] T. Simonini and O. Sanseviero, “The hugging face deep reinforcement learning class,” [Online]. Disponible: <https://github.com/huggingface/deep-rl-class>, 2023, [Accedido: 25-junio-2025].
- [18] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.
- [19] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel *et al.*, “Soft actor-critic algorithms and applications,” *arXiv preprint arXiv:1812.05905*, 2018.
- [20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *arXiv preprint arXiv:1707.06347*, 2017.
- [21] Unity Technologies, “Unity user manual,” [Online]. Disponible: <https://docs.unity3d.com/6000.1/Documentation/Manual/UnityManual.html>, 2025, [Accedido: 25-junio-2025].
- [22] Unity Technologies, “Ml-agents documentation,” [Online]. Disponible: <https://github.com/Unity-Technologies/ml-agents>, 2024, [Accedido: 25-junio-2025].
- [23] A. Juliani, V.-P. Berges, E. Teng, A. Cohen, J. Harper, C. Elion, C. Goy, Y. Gao, H. Henry, M. Mattar, and D. Lange, “Unity: A general platform for intelligent agents,” *arXiv preprint arXiv:1809.02627*, 2020. [Online]. Available: <https://arxiv.org/pdf/1809.02627.pdf>
- [24] T. Eimer, M. Lindauer, and R. Raileanu, “Hyperparameters in reinforcement learning and how to tune them,” in *International conference on machine learning*. PMLR, 2023, pp. 9104–9149.
- [25] Baptista, Baptista, “Ppo hyperparameter optimization,” [Online]. Disponible: <https://joel-baptista.github.io/phd-weekly-report/posts/hyper-op/>, 2024, [Accedido: 25-junio-2025].
- [26] Huang, Shengyi; Dossa, Rousslan Fernand Julien; Raffin, Antonin; Kanervisto, Anssi; Wang, Weixun, “The 37 implementation details of proximal policy optimization,” [Online]. Disponible: <https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/>, 2022, [Accedido: 25-junio-2025].