eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Facultad de informática

Trabajo de Fin de Grado

Aplicación de técnicas de aprendizaje por refuerzo profundo en un entorno virtual simulado con Unity

Asier Aguayo Velasco

Junio 2018

Directores UPV/EHU

Iñigo Mendialdua Beitia

Basilio Sierra Araujo

Agradecimientos

En primer lugar quiero agradecer a mis directores del proyecto Iñigo y Basilio, por darme la oportunidad de hacer el trabajo sobre el tema que les propuse. Me han guiado y ayudado en todo momento y sobre todo, me han dado confianza durante todo el proyecto.

A mis padres por dejarme estudiar lo que he querido en todo momento. A mi hermano y Noemí por todos los momentos que hemos compartido jugando juntos.

También quiero agradecer a Ana todas las horas que ha pasado conmigo aconsejándome, corrigiendo, editando y moviendo imágenes en la memoria.

Y a mis amigos por aguantarme hablando del proyecto e interesarse por el transcurso del mismo entre partida y partida.

Resumen

En este proyecto, se estudia cómo dotar de inteligencia a objetos que componen un entorno virtual en Unity utilizando técnicas de Machine Learning, en concreto, mediante técnicas de aprendizaje por refuerzo profundo. Para ello, se utilizan herramientas como **Tensorflow**, el propio **Unity** y una librería especial creada para tal propósito llamada **ml-agents**.

En esta memoria, se explica el proceso de desarrollo, desde la preparación del entorno, hasta los entrenamientos y las pruebas posteriores para comprobar que los llamados agentes aprenden correctamente.

El proyecto está compuesto por 3 experimentos:

- El primero de ellos, sirve como aprendizaje por su simpleza. El objetivo del mismo es mantener una pelota en equilibrio sobre una plataforma, de forma que la plataforma aprenda a mantener la pelota estable y no se caiga.

- El segundo, consiste en entrenar a un vehículo en un circuito de forma que consiga llegar a meta en base a *checkpoints* (puntos intermedios) sin chocar con ninguna de las paredes que compone el circuito.

- El tercero, es una evolución del segundo. Consiste en que un vehículo consiga llegar a la línea de meta pero aprendiendo de forma diferente, basándose en la distancia que hay entre el coche y las paredes a su alrededor. Mediante esta forma de aprendizaje, el coche será capaz de recorrer cualquier circuito.

Índice

1. Introducción

En el año 2017 la industria del videojuego consiguió ser el sector cultural que más recaudó en el país por delante del cine y la música entre otros [1]. El crecimiento de las ganancias generadas a nivel mundial en un solo año ha subido el 8,5%, recaudando en 2015 91.800 millones de dólares y en 2016 99.600 millones de dólares [2]. La AEVI (Asociación Española de Videojuegos) destaca que "por cada euro invertido en la industria de los videojuegos en nuestro país se tiene un impacto de 3 euros en el conjunto de la economía". En España, un total de 8.790 personas trabajan en la industria del videojuego divididas en 330 empresas [3]. En Euskadi se concentra el 6,8% de las empresas del sector a nivel estatal. Además, el 41% de las personas juega a algún videojuego asiduamente [4]. Estos datos, recogidos de algunas de las fuentes más significativas, no hacen más que demostrar que el sector de los videojuegos está en auge desde hace unos años. La aparición de nuevas tecnologías, así como la accesibilidad a través de la red, no ha hecho más que aumentar el número de ventas y las ganancias en la industria.

Si se habla de nuevas tecnologías, hay que mencionar una de las que está llamada a ser la tecnología del futuro, el "Machine Learning" o aprendizaje automático. Se prevé que para el año 2020 la inversión en este tipo de tecnología por parte de las empresas aumente a los 44.000 millones de euros, un dato bastante significativo y más si se tiene en cuenta que en el año 2016 la inversión era de 6.000 millones de euros [5] [6]. Cada vez más, las empresas deciden invertir más dinero en el análisis de "datos" utilizando técnicas para predecir cómo va a afectar a su productividad, ventas e ingresos.

Por ello, la idea de este proyecto surgió de la mezcla de dos de los sectores que más están creciendo hoy en día, Machine Learning aplicado a videojuegos. En este proyecto no se desarrollar un videojuego completo, sino que se estudia en qué casos es aplicable esta técnica. A día de hoy muy pocos videojuegos utilizan Machine Learning para dotar a los componentes del juego de inteligencia.

Investigando sobre los posibles usos, a raíz de un blog de internet [7] surgió la idea de implementar un coche que aprenda a dar una vuelta en un circuito de forma autodidacta mediante este tipo de técnicas. Después de evaluar diferentes opciones, se decidió utilizar **Unity** como motor de desarrollo y su recién lanzada librería llamada **ml-agents** (Machine Learning - agents) como elemento para su aprendizaje.

Ml-agents es la librería para dotar de inteligencia a los objetos en un videojuego. Utiliza algoritmos proporcionados por Tensorflow para este propósito. La técnica que se estudia en el proyecto es el aprendizaje por refuerzo, técnica que se implementa mediante el uso de los algoritmos de Tensorflow.

Unity es un motor de desarrollo de videojuegos que permite a los desarrolladores a crear videojuegos de cualquier estilo y temática desde cero, tanto en 3 dimensiones como en 2

dimensiones.

En este proyecto se explican los pasos llevados a cabo para el uso de esta librería y se estudia el funcionamiento de la misma, explicando las técnicas que se utilizan y la manera de aplicarlas en los experimentos para lograr los objetivos que se mencionan a continuación.

1.1. Objetivos del proyecto

En esta sección se introducen los objetivos de los experimentos que se llevan a cabo para lograr el objetivo final, conseguir que un coche de una vuelta completa a un circuito mediante técnicas de aprendizaje por refuerzo.

Para conseguirlo, se ponen en práctica técnicas de Machine Learning. El capítulo 3 (Conceptos básicos) está dedicado a entender mejor qué son estas técnicas, qué se puede conseguir con ellas y cómo se pueden implementar. Se explican los diferentes tipos de aprendizaje de Machine Learning, aprendizaje supervisado y aprendizaje no supervisado, así como la que más influencia tiene en este proyecto, el aprendizaje por refuerzo. También se profundiza en los conceptos del d*eep learning* y Redes Neuronales.

A continuación (capítulo 4, Unity), se estudia el funcionamiento del entorno de desarrollo sobre el que se desarrolla el proyecto, Unity. Se explican las funciones más importantes, la interfaz y las herramientas asociadas utilizadas para implementar las técnicas de Machine Learning, Tensorflow y la librería ml-agents.

Finalmente, una vez explicados los pilares sobre los que se sustenta el proyecto, se explican los tres experimentos que se llevan a cabo para conseguir el objetivo final.

1.1.1. Pelota en equilibrio

Se parte de un experimento a modo introductorio. Consiste en mantener en equilibrio una pelota sobre una tabla de forma que no caiga. Para ello, se entrenará la plataforma en base a las técnicas de aprendizaje por refuerzo que se explica en los capítulos posteriores.

1.1.2. Circuitos

Una vez se haya conseguido el primero de los experimentos, se inicia el proceso al objetivo final. Para empezar, se realizan una serie de circuitos para poder probar el funcionamiento del coche en las distintas fases del experimento. Este coche, basándose en técnicas de aprendizaje por refuerzo, consigue llegar a meta en los diferentes circuitos creados sin chocar con las

paredes. Primero se entrenará en circuitos más sencillos y al final, será capaz de terminar circuitos de mayor complejidad.

- Simple - Checkpoints

La versión más sencilla del aprendizaje del coche se basa en seguir una serie de checkpoints, puntos intermedios. De este modo, el vehículo determina por donde debe continuar para llegar a la meta.

El problema de este tipo de aprendizaje es que sólo aprenderá a completar el circuito sobre el cual se entrena, es decir, el aprendizaje no se puede extrapolar a otros circuitos al no recibir información del entorno.

- Más complicado - Sensores

A diferencia que en el caso anterior, mediante sensores acoplados al coche, se puede entrenar al vehículo de forma que sea capaz de finalizar cualquier circuito, no sólo el circuito sobre el cual se entrena.

Al concluir este último experimento, se dará por cumplido el objetivo.

2. Documento de objetivos del proyecto

2.1. Descripción

A lo largo de este proyecto se estudia la librería proporcionada por Unity para dotar de inteligencia a los agentes que componen un entorno virtual. Con el fin de lograr el objetivo final del proyecto, conseguir que un coche simulado en Unity aprenda a dar una vuelta al circuito de forma autodidacta mediante un entrenamiento basado en una de las técnicas de Machine Learning, el **aprendizaje por refuerzo**.

Para ello, se estudia la base teórica en la que se sustenta tal librería para entender qué es lo que se trata de conseguir con ella. Machine Learning (redes neuronales y sobre todo, el aprendizaje por refuerzo) son las técnicas que se estudian durante el proyecto.

2.2. Objetivos y motivación

El objetivo principal es conseguir que los agentes una vez entrenados cumplan los objetivos para los que han sido diseñados. En el caso del primer experimento, mantener la pelota sobre la plataforma sin que se caiga. Y en los dos siguientes, conseguir que el coche de una vuelta completa al circuito sin chocar contra las paredes.

Para ello, lo primero es preparar el entorno de desarrollo, en este caso, instalar el software que se va a utilizar. A continuación, realizar uno de los tutoriales para ir comprendiendo el funcionamiento de la librería y aprender a manejarse con los controles de Unity. El tutorial escogido para tal tarea es el de mantener la pelota encima de una plataforma. Finalmente, aplicar a un nuevo caso los conocimientos adquiridos con el fin de lograr el objetivo. En este caso, conseguir que el coche de una vuelta a un circuito sin chocarse.

El alumno, mediante este proyecto, pone en práctica los conocimientos adquiridos durante la carrera y los aumenta estudiando nuevas técnicas de Machine Learning como es el aprendizaje por refuerzo. También aprende a utilizar uno de los motores de desarrollo de videojuegos más potentes, Unity. Por otro lado, al ser ml-agents una nueva herramienta, en una etapa de desarrollo inicial, se aprenden los conceptos necesarios para llevar a cabo los experimentos. Con todo esto encontrará otras aplicaciones prácticas que podría tener esta técnica.

2.3. Planificación

El proyecto se divide en varias fases compuestas por tareas.

- Primera fase:

  - **Informarse sobre posibles aplicaciones de Machine Learning en Unity**. A modo informativo, buscar formas de implementar técnicas de Machine Learning en Unity para conocer el potencial de la técnica.

  - **Preparar el entorno de desarrollo**. Siguiendo los pasos que aparecen en la documentación de ml-agents, instalar el software necesario para su funcionamiento.

  - **Estudiar el funcionamiento del software instalado.** Una vez instalado el software que se va a utilizar, es de vital importancia conocer qué puede hacer cada programa y sus limitaciones. En esta tarea también se añade el tiempo que se dedica a mirar la documentación de la librería, así como los ejemplos que ésta dispone para el aprendizaje.

- Segunda fase:

  - **Seguir el tutorial para afianzar ideas y ver en funcionamiento una aplicación**. Unity pone como ejemplo la aplicación de la pelota en equilibrio sobre una plataforma y propone un tutorial para su implementación. Esta tarea consiste en poner en marcha la aplicación para ver cómo es el proceso de aprendizaje del agente, familiarizarse con la estructura de directorios y ficheros, los scripts a utilizar, etc.

- Tercera fase, aplicable a los dos experimentos:

  - **Diseñar el experimento**. En esta tarea lo importante es estudiar diferentes ideas y formas distintas de afrontar el proyecto y plantear la mejor forma posible de llevarla a cabo, pensando sobre todo en las funciones de recompensa, las observaciones del entorno que debe realizar el agente y las diferentes acciones que toma en consecuencia.

  - **Desarrollo del experimento.** Una vez diseñado el proyecto a implementar, se empieza la tarea de desarrollo en Unity. Se comprobará que el agente recibe correctamente las observaciones del entorno como entrada para evitar cualquier problema posterior a la hora del entrenamiento. Se programarán los scripts necesarios para que el agente pueda entrenarse correctamente.

  - **Probar el agente manualmente.** Comprobar que el agente funciona correctamente manejándolo manualmente. Planear un entrenamiento breve para ver que aunque no aprenda del todo, evoluciona favorablemente.

  - **Entrenar al agente.** Diseñar un buen entrenamiento para garantizar que el agente aprende como es debido. La tarea consistirá en modificar los parámetros que utiliza el

algoritmo de aprendizaje, si hiciese falta, para modificar el entrenamiento.

- Fase final:

  - **Análisis de los resultados.** Comprobar que el aprendizaje funciona correctamente, es decir, que el coche es capaz de dar una vuelta al circuito sin chocar. Verificar que el coche una vez aprendido, es capaz de dar una vuelta a otros circuitos que se diseñen.

  - **Conclusiones y futuras aplicaciones.** Para dar por finalizado el proyecto se hablará de las posibles aplicaciones que pueden tener las técnicas estudiadas.

2.3.1. Planificación temporal inicial

Se ilustra la planificación mediante un diagrama de Gantt (Tabla 1). En este diagrama se pueden encontrar las tareas explicadas en el punto anterior. Hay que resaltar que son plazos orientativos. Se cuentan las semanas de lunes a viernes. Las tareas se llevarán a cabo progresivamente, excepto las marcadas a realizarse durante todo el proyecto.

Diagrama de Gantt - Estimación

	Fecha inicio	Fecha Fin	febrero			marzo				abril				mayo					junio	
Información aplicaciones Unity-ml agents	12/02/18	16/02/18																		
Preparar entorno de desarrollo	14/02/18	25/02/18																		
Estudiar funcionamiento software	17/02/18	28/05/18																		
Gestión del proyecto	05/03/18	15/06/18																		
Tutorial de inicio (Experimento 1)	05/03/18	23/03/18																		
Experimento 2	26/03/18	27/04/18																		
Diseño del experimento	26/03/18	06/04/18																		
Desarrollo del experimento	02/04/18	13/04/18																		
Pruebas	16/04/18	20/04/18																		
Entrenamiento de los agentes	23/04/18	27/04/18																		
Experimento 3	30/04/18	01/06/18																		
Diseño del experimento	30/04/18	11/05/18																		
Desarrollo del experimento	07/05/18	18/05/18																		
Pruebas	21/05/18	25/05/18																		
Entrenamiento de los agentes	28/05/18	01/06/18																		
Análisis de los resultados	23/04/18	01/06/18																		

Tabla 1. Diagrama de Gantt con las horas estimadas del proyecto.

En la tabla 2 se pueden ver cuántas horas estimadas se van a dedicar a cada tarea.

Tarea principal	Subtarea	Horas estimadas
Información aplicaciones Unity ml-agents		10
Preparar entorno de desarrollo		50
Estudiar funcionamiento software		50
Gestión del proyecto		60
Experimento 1		25
Experimento 2		50
	Diseño del experimento	15
	Desarrollo del experimento	15
	Pruebas	10
	Entrenamiento de los agentes	10
Experimento 3		55
	Diseño del experimento	15
	Desarrollo del experimento	20
	Pruebas	10
	Entrenamiento de los agentes	10
Análisis de los resultados		25
	Experimento 2	10
	Experimento 3	15
Suma total:		325

Tabla 2. Estimación de horas de proyecto.

Se estima que la tarea que más trabajo va a requerir es la gestión del proyecto, que incluye la redacción de la memoria. La preparación del entorno también implicará una gran dedicación debido a que puede haber problemas de compatibilidad con el equipo por la versión de la librería que se va a implementar. Además, al tratarse de un proyecto donde se requerirá utilizar funciones propias de la librería, se estima que se dedicarán muchas horas a consultar la documentación. Respecto a los experimentos, el primero de ellos, se realizará siguiendo un tutorial por lo que no se espera mucha dedicación. Para los experimentos dos y tres, sin embargo, se espera una mayor dedicación. Se estima un total de 325 horas de las 300 que componen el trabajo de fin de grado.

2.3.2. Planificación temporal real

Una vez terminado el proyecto y recogidos los tiempos de desarrollo de cada tarea, se ha elaborado una tabla (Tabla 3) de forma análoga a la del anterior punto con las horas invertidas a cada tarea.

Tabla de horas reales

Tarea principal	Subtarea	Horas reales
Información aplicaciones Unity ml-agents		20
Preparar entorno de desarrollo		30
Estudiar funcionamiento software		50
Gestión del proyecto		70
Experimento 1		15
Experimento 2		70
	Diseño del experimento	15
	Desarrollo del experimento	20
	Pruebas	20
	Entrenamiento de los agentes	15
Experimento 3		75
	Diseño del experimento	15
	Desarrollo del experimento	20
	Pruebas	20
	Entrenamiento de los agentes	20
Análisis de los resultados		10
	Experimento 2	5
	Experimento 3	5
Suma total:		340

Tabla 3. Horas totales invertidas en el proyecto.

Como cabía esperar, la mayor diferencia del tiempo estimado y el tiempo real son las horas invertidas en los experimentos 2 y 3. Esto se debe en parte al desconocimiento previo a la implementación de los experimentos.

Con el segundo de los experimentos el problema fue que las pruebas y entrenamiento de los agentes no respondían de la forma que se había esperado.

Con el tercer experimento ocurrió que la implementación de los sensores costó más de lo esperado. No detectaban las colisiones y el tiempo que se esperaba invertir en pruebas y más desarrollo, se dedicó a buscar una solución al problema.

De forma contraria, se esperaba invertir más tiempo en la preparación del entorno de desarrollo. A pesar de dedicar una gran cantidad de horas, debido a la gran cantidad de problemas que da el proceso de instalación, internet sirvió de base para encontrar soluciones a los problemas que fueron surgiendo.

Por último, el primer experimento se realizó de manera más rápida de la esperada. Al haber tan poca información sobre esta nueva técnica, la mayoría de los tutoriales implementaban el primer experimento, por ello se encontraron varios tutoriales y guías que explicaban paso por paso cómo llevarlo a cabo.

A continuación, mediante un diagrama de Gantt (Tabla 4), se muestran las fechas en las que se han implementado las tareas.

Diagrama de Gantt - Real

	Fecha inicio	Fecha Fin	febrero	marzo	abril	mayo	junio
Información aplicaciones Unity-ml agents	12/02/18	16/02/18					
Preparar entorno de desarrollo	14/02/18	25/02/18					
Estudiar funcionamiento software	17/02/18	28/05/18					
Gestión del proyecto	05/03/18	15/06/18					
Tutorial de inicio (Experimento 1)	05/03/18	09/03/18					
Experimento 2	12/03/18	13/04/18					
Diseño del experimento	12/03/18	23/03/18					
Desarrollo del experimento	26/03/18	06/04/18					
Pruebas	02/04/18	13/04/18					
Entrenamiento de los agentes	09/04/18	20/04/18					
Experimento 3	23/04/18	01/06/18					
Diseño del experimento	23/04/18	04/05/18					
Desarrollo del experimento	07/05/18	18/05/18					
Pruebas	14/05/18	25/05/18					
Entrenamiento de los agentes	28/05/18	01/06/18					
Análisis de los resultados	23/04/18	01/06/18					

Tabla 4. Diagrama de Gantt con las horas totales invertidas en el proyecto.

En comparación con la planificación inicial no hay demasiados cambios. Tanto para el experimento 2 y 3, se alargó una semana el tiempo estimado por los problemas que se han comentado anteriormente. Algunas de las tareas de los experimentos se solaparon por los problemas que se han comentado, teniendo que reestructurar los experimentos desde la parte del diseño de los mismos.

2.4. Análisis de riesgos

Para garantizar que el proyecto se pueda implementar correctamente, es interesante analizar cuáles son los posibles problemas a los que se puede enfrentar el proyecto. Se realizará un plan de contingencia para solucionar o minimizar los posibles problemas que puedan surgir. Son los siguientes.

- **Problemas técnicos**. Al tratarse de un proyecto que se va a desarrollar con un ordenador, no se pueden obviar los posibles problemas que puedan surgir con el hardware o el software instalado. Para evitar que estos problemas puedan retrasar el desarrollo, se realizarán copias de seguridad en la nube tanto de los proyectos, como de la memoria. Además, se instalará el entorno de desarrollo en otro equipo para posibilitar la opción de trabajar simultáneamente en dos ordenadores.

- **Falta de documentación.** Como la librería que se utiliza para la implementación de las técnicas de Machine Learning es prácticamente nueva, puede ocurrir que alguno de los problemas a los que haya que hacer frente no tenga una solución directa. Por eso, aunque no exista una solución a este problema, se hará uso de los foros de Unity para preguntar dudas e intentar que no se retrase el desarrollo del proyecto.

- **Sobreexceso de trabajo.** Durante el desarrollo del proyecto puede ocurrir que otras tareas laborales, académicas o personales, dificulten la realización del mismo. Para este tipo de situaciones, se valorarán las diferentes tareas a realizar y se realizará la más prioritaria dejando las demás para más adelante. Teniendo en cuenta que el objetivo principal es aprobar las asignaturas restantes, el desarrollo del proyecto quedará en un segundo plano hasta que no se terminen las asignaturas.

- **Enfermedad o problemas de salud.** Ya sean propias del alumno o de los directores del proyecto, para hacer frente a este tipo de situaciones, las tareas perjudicadas se pospondrán adelantando otras, en caso de que sea posible.

2.4.1. Valoración de los riesgos

En la siguiente tabla se pueden observar los posibles riesgos mencionados anteriormente y el impacto que supondrían en el desarrollo del proyecto.

Valoración de riesgos

	Probabilidad de ocurrir	Nivel de impacto
Problemas técnicos	Media	Alto
Falta de documentación	Alta	Alto
Sobreexceso de trabajo	Media	Medio
Problemas de salud	Baja	Bajo

Tabla 5. Valoración de riesgos.

Se puede apreciar que el mayor problema que puede surgir es hacer frente a la falta de documentación accesible en la red. El plan de contingencia diseñado para el primero de los problemas, disminuye las probabilidades de que ocurran. En caso de que hubiera algún problema técnico, el impacto que sufriría el proyecto sería alto.

No se prevé que los últimos dos riesgos vayan a tener demasiado impacto durante el desarrollo del proyecto, porque que se ha iniciado con tiempo suficiente para la implementación. Una buena reestructuración de las tareas, sería suficiente para subsanar estos problemas.

3. Conceptos básicos

En este capítulo, se explican las bases teóricas que se utilizan para poner en práctica las técnicas de aprendizaje. Se hace especial hincapié en la técnica de aprendizaje por refuerzo, técnica utilizada en este proyecto para la dotación de inteligencia a los agentes.

3.1. Machine Learning

Machine Learning (Aprendizaje automático) es una rama de la inteligencia artificial que da a un sistema la capacidad de aprender de forma autónoma, tomar decisiones o hacer predicciones. Estas decisiones o predicciones se toman en base a datos. Un algoritmo de Machine Learning se compone por una base de datos, una función de coste/pérdida, una función de optimización y un modelo [8].

En los últimos años, son muchos los sectores en los que se ha decidido empezar a utilizar este tipo de técnicas para resolución de todo tipo de problemas. Se debe sobre todo, a la mejora de los diferentes algoritmos y su probada eficacia. Hoy en día, estamos rodeados por tecnologías que aplican este tipo de técnicas, como por ejemplo, el motor de búsqueda de un buscador, el reconocimiento facial de una cámara de fotos y un largo etcétera. Machine Learning está basado en teoría de la probabilidad, estadísticas y optimización. Es la base del *Big Data*, *Data Science*, *Minería de datos*, *Modelos predíctivos,* etc. Su uso es de vital importancia para la *visión por computador*, *procesado del lenguaje natural* o la *robótica*.

En este proyecto, se utiliza uno de los paradigmas de **Machine Learning** llamado **Aprendizaje por refuerzo**, pero antes de explicar su funcionamiento es interesante conocer las bases sobre las que se fundamentan este tipo de técnicas.

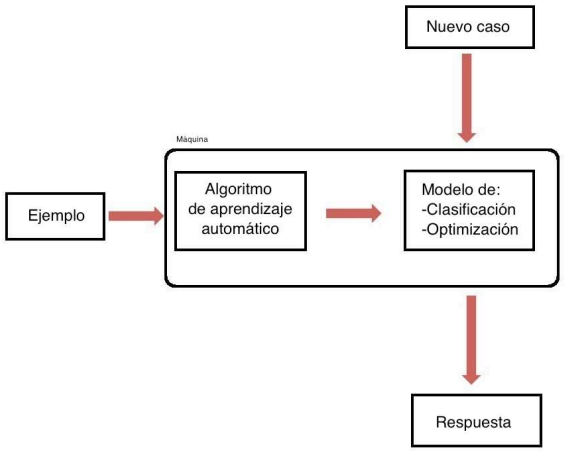

Imagen 3.1. Diagrama del funcionamiento de Machine Learning.

Normalmente se dividen las técnicas de Machine Learning en tres paradigmas diferentes, el aprendizaje supervisado, el aprendizaje no supervisado y el aprendizaje por refuerzo.

3.2. Aprendizaje supervisado

Es una técnica para predecir la clase de un dato basándose en un conjunto de entrenamiento. El conjunto de entrenamiento está compuesto por una serie de datos formado por un conjunto de características y una clase asociada. En base al conjunto de entrenamiento, se genera un modelo o regla general de forma que sirve para clasificar nuevos datos de los que se desconoce la clase. De esta forma, no se intenta simplemente agrupar los datos, si no

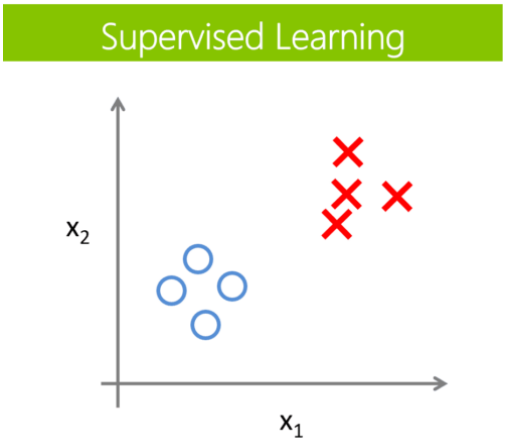

Imagen 3.2. Ejemplo sobre el aprendizaje supervisado.

encontrar una manera de clasificación en base a las características [9].

Esta técnica de aprendizaje se suele utilizar en temas de medicina para detectar enfermedades en base a datos de otros pacientes, detección de spam, reconocimiento de patrones, reconocimiento de objetos en visión por computador y otros.

3.3. Aprendizaje no supervisado

Esta técnica, también llamada *clustering*, no conoce la clase de los datos. A diferencia del aprendizaje supervisado, no utiliza un conjunto de entrenamiento para predecir la clase del dato. Los datos consisten en un vector de características. El objetivo de las técnicas no supervisadas consiste en agrupar los datos de manera que se puedan catalogar en base a las descripciones de los datos.

Este modelo se distingue del aprendizaje supervisado en que no existe un conocimiento previo de los datos. El conocimiento se va generando en base a observaciones [9].

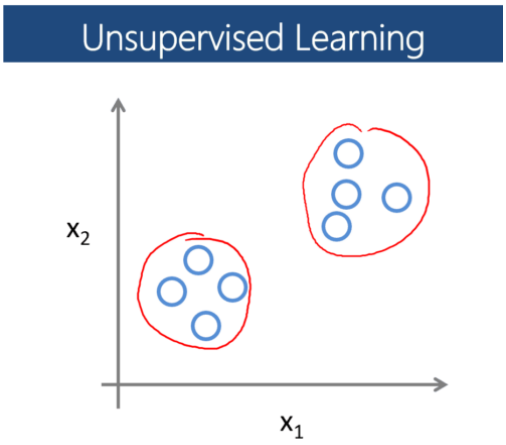

Imagen 3.3. Ejemplo sobre el aprendizaje no supervisado.

Esta técnica se suele utilizar para agrupación de páginas web en base a etiquetas, para la compresión de datos, monitorización de contenido en internet y otros.

3.4. Aprendizaje por refuerzo.

Para el proyecto se utiliza este modelo de aprendizaje que difiere de los dos paradigmas de aprendizaje anteriores en determinadas características.

El modelo de aprendizaje por refuerzo se inspira en la psicología conductista. Como definía el psicólogo estadounidense J. B. Watson (1878-1958), "el conductismo es el estudio experimental objetivo y natural de la conducta que puede ser observada"[1]. Basaba su estudio en que para que una conducta pueda ser modificada, se necesita un estímulo y una respuesta. Un estímulo puede llegar en forma de recompensa o premio cuando algo se hace de forma correcta y un castigo cuando algo se hace de forma errónea.

Un ejemplo representativo es aquel en el que se recompensa a un perro con una galleta (refuerzo positivo) cuando recoge una pelota al ser lanzada y se le golpea (refuerzo negativo) cuando no lo hace. De esta forma, se genera en el animal una conducta basada en el refuerzo [10].

De forma semejante ocurre en el aprendizaje por refuerzo que compone uno de los tres paradigmas de Machine Learning. Continuando con el paralelismo, un agente (perro), aprende de forma autónoma a tomar decisiones observando el entorno (coger la pelota) en base a recompensas (galleta o golpe).

[1] Del artículo sobre el conductismo en Wikipedia: https://es.wikipedia.org/wiki/Conductismo

Por lo tanto, se puede definir que el agente y el entorno que le rodea, modelan un estado *s,* el cual guarda toda la información observable del entorno.

Estos estados varían en función de un instante *t*. En cualquier estado, se puede realizar una acción *a* que llevará del estado *s* al estado *s'*, pudiendo ser *s'* la misma *s* en caso de tomar la acción de no actuar [11]. A esta transición se le llama **modelo de transición**. Hay dos tipos de transición.

- Aquella que ocurre en un **entorno determinístico**, donde el azar no tiene presencia y se genera una salida para una entrada sin contemplarse la incertidumbre del modo *T(s, a, s').*

- La que ocurre en un **entorno no determinístico o probabilístico**, donde el cambio de estado está influido por el azar y la probabilidad de manera que se genera una salida para una entrada de manera aleatoria. Esta es del modo *Pr(s' | s, a)*. Donde *Pr* hace referencia a la probabilidad de que ocurra el cambio de estado.

La toma de la decisión y la realización de una acción *a*, conlleva una recompensa *r*. Hay varios tipos de recompensa.

- Las basadas en el estado: *R(s)*.

- Las basadas en la acción que sucede en un estado: *R(s, a)*.

- Las basadas en la acción que sucede en un estado y como consecuencia se llega a otro nuevo estado: *R(s, a, s')*.

Este cambio de estado genera un cambio en lo que se llama **política**. La política, también llamada π, es una función que devuelve la acción a realizar dado un estado. El objetivo del aprendizaje por refuerzo será encontrar la función llamada **política óptima**, π*, es decir, encontrar la función que dado cualquier estado devuelva la acción que maximice la

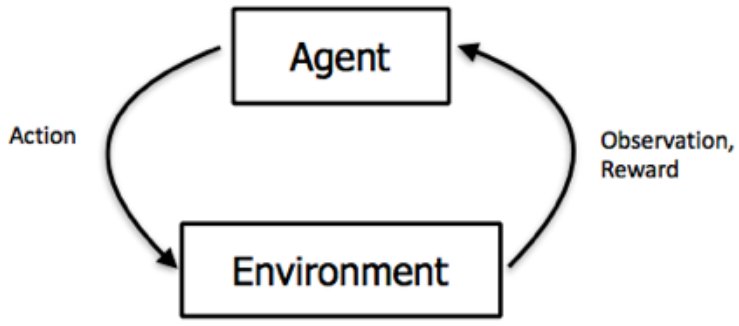

Imagen 3.4. Ciclo de funcionamiento del aprendizaje por refuerzo.

recompensa acumulada que se recibe.

Para encontrar la política óptima, el aprendizaje se divide en **episodios** o **iteraciones**. Estos episodios tienen un máximo **número de pasos** que indican cuando se terminan. El número de pasos varía en cada episodio dependiendo de lo que se intenta conseguir. Estas iteraciones forman un ciclo.

La forma de entrenar una política se basa en maximizar las recompensas obtenidas al realizar una determinada acción en un determinado estado. Teniendo en cuenta los instantes en los que se toman las acciones, se puede definir la fórmula de maximización de la siguiente manera:

$$R_t = r_{t+1} + r_{t+2} + \ldots + r_T$$

Donde Rt hace referencia a la recompensa total adquirida y T al estado final. Esto sucede cuando el agente alcanza uno de los llamados **estados terminales,** que indican que el agente ha llegado a un estado donde se da por terminado el episodio. Después de asignarle su correspondiente recompensa se reinicia, de modo que empieza un nuevo episodio en busca de optimizar más la política. A este tipo de tareas se les llama **tareas episódicas.**

Además de las tareas episódicas, existen las **tareas continuas.** Son aquellas en las que el aprendizaje se realiza de manera continuada sin reiniciar el entrenamiento a un estado de inicio o de salida. Para este tipo de tareas se introduce un nuevo concepto llamado **factor de descuento, *γ*.** Teniendo en cuenta que una tarea continuada no es finita, llegaría un punto en el que la suma de las recompensas tendería al infinito. Para ello, se introduce el factor de descuento que reducirá en cada instante *t* un valor comprendido entre 0 y 1, de forma que cuanto mayor sea el valor, mayor influencia tendrá en la recompensa adquirida por la política.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \Sigma_{k=0}^{N} \gamma^k r_{t+k+1}$$

La filosofía del aprendizaje por refuerzo no consiste en que el agente sea capaz de memorizar los pasos que tiene que dar hasta conseguir lo esperado. Se espera que sea capaz de decidir cuál es la acción más adecuada en cada estado. Por eso lo ideal es que un estado resuma la información de los estados anteriores y se tomen decisiones en base al estado actual sin importar de dónde ha llegado. Para representaciones de estados de este tipo existe el llamado **Proceso de Decisión de Markov** (MDP).

El MDP se basa en dos principios básicos.

- Sólo importa el presente. No se toman decisiones en base a estados anteriores, solo en base al actual.

- Inmóvil. Las observaciones que recibe el agente del entorno no varían.

De forma parecida a la mencionada anteriormente, un MDP viene definido por la cuádrupla *<S, A, P, R>*, donde *S* hace referencia a un espacio de estados, *A* es un espacio de acciones, *P* es una matriz que indica la probabilidad de pasar de un estado *s* a un estado *s'* dada una acción *a* y *R* es la función de refuerzo que indica la recompensa que recibe el agente al tomar la acción *a* en un estado *s.*

La mayoría de algoritmos de aprendizaje por refuerzo se basan en una estimación de cuan bueno es que un agente se encuentre en un determinado estado *s*. Esta estimación sirve para determinar si las recompensas venideras desde *s* van a mejorar lo obtenido hasta ahora o no. Como es lógico, las recompensas que el agente puede recibir en un futuro dependerán de las acciones que éste realice. Estas estimaciones se llaman **funciones de valor.**

Los modos de resolver problemas de aprendizaje por refuerzo se agrupan en tres tipos:

- **Programación dinámica**. Una técnica en desuso. Consiste en calcular las políticas óptimas suponiendo que el entorno del que se extraen las observaciones es perfecto, es decir, para cualquier estado, el entorno es siempre el mismo. Como no se suele cumplir, no se suele utilizar.

- **Métodos de Monte Carlo**. Útiles solo en casos pequeños y finitos que utilizan el modelo MDP y en problemas episódicos. Tratan de encontrar una política óptima en base a la recompensa media que se puede calcular de cada estado aplicando todas las diferentes acciones posibles. De este modo, al acabar cada episodio, se actualiza la política en caso de mejora.

- **Métodos de diferencias temporales**. Utiliza la experiencia para estimar la función de valor de una política. En base a esto, calcula la política óptima para el problema.

$$V(S_t) < V(S_t) + \alpha[G_t - V(S_t)]$$

  Donde *G* es la recompensa acumulada hasta el tiempo *t* y α es un parámetro constante llamado constant-α. De forma que al acabar una etapa de la iteración la función de valor $V(S_t)$ se actualiza.

Hoy en día, la mayoría de los algoritmos de aprendizaje por refuerzo utilizan el método de diferencias temporales como base. Uno de los primeros algoritmos de este tipo que se ideó es el **Q-Learning** (Watkins - 1989), algoritmo que sirve de base para la mayoría de los algoritmos que siguen creándose.

Para este proyecto se utilizarán las técnicas de aprendizaje por refuerzo junto con una red neuronal de forma que unidas, encuentren la acción que ha de realizar el agente de manera más precisa.

3.5. Deep learning

El *deep learning* forma parte de los métodos que componen Machine Learning. Métodos basados en el aprendizaje en base a características de datos. Es uno de los métodos más potentes por su utilidad en diferentes aplicaciones debido a que se han obtenido elevadas tasas de éxito con su uso a la hora de la predicción o clasificación de clases. Usan una red neuronal dividida en varias capas para tal tarea.

Una red neuronal es una estructura dividida en un conjunto de elementos llamados "neuronas" interconectadas entre sí que modela un sistema nervioso artificial. Esta estructura se inspira en las redes neuronales biológicas que componen un cerebro humano. Son unas de las herramientas computacionales más potentes utilizadas a día de hoy para resolver problemas complejos.

En 1943, Warren McCulloch y Walter Pitts [12], diseñaron un modelo informático basándose en los algoritmos llamados lógica de umbral. Este modelo derivó en la investigación de los procesos biológicos del cerebro y en la aplicación de redes neuronales en la inteligencia artificial. Se considera el inicio de las redes neuronales, hasta que no fue más adelante cuando se creó el primer algoritmo de reconocimiento de patrones.

Fue en 1958 y de la mano de Rosenblatt, un destacado psicólogo especializado en el campo de la inteligencia artificial. Rosenblatt diseñó los llamados "perceptrones", unos elementos creados para resolver problemas de clasificación que funcionan mediante la aproximación lineal de una función escalonada [13].

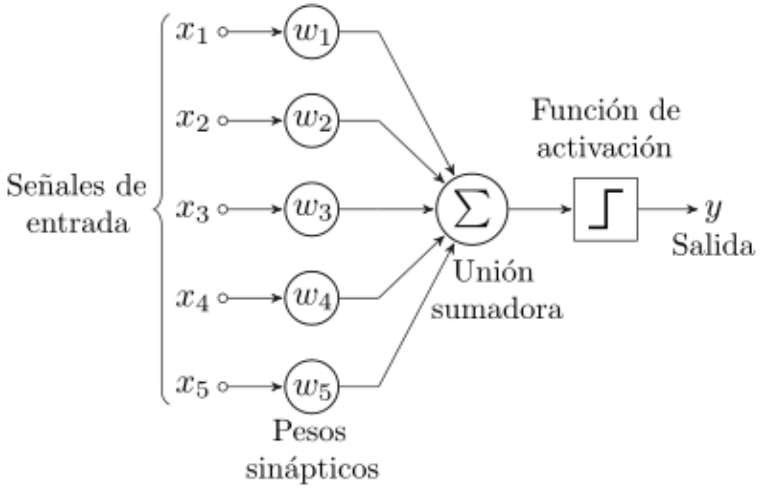

Imagen 3.5. Representación de un perceptrón.

El perceptrón, también llamado neurona, está formado por un vector de entrada $x_1, x_2, \ldots, x_j$, donde x_j denota el elemento en la posición j tal que $1 \leq j \leq n$. w_j es el peso asociado a la posición j del vector de entrada. Todos los elementos del vector y su peso asociado se suman junto a un valor constante llamado *bias (sesgo)*, dando como resultado un valor m, de la forma:

$$m = x_1 w_1 + x_2 w_2 + \ldots + x_n w_n + b$$

Al valor m se le aplica la función de activación (Función escalón de Heaviside) de forma que se genere una salida y. Como se ha comentado anteriormente, este tipo de técnica se utiliza para la clasificación, de modo que al encontrar una salida y se verifica si es la salida deseada y en base a ello se actualizan los pesos asociados al vector de entrada. Este proceso se repetirá hasta que los resultados converjan.

A pesar del gran avance que supuso su creación, durante los siguientes años se decidió no invertir tanto tiempo en esta técnica debido a que no era capaz de resolver problemas linealmente separables, como la operación de exclusividad XOR, entre otros.

Con el paso del tiempo, se logró combinar varios perceptrones simples y se consiguió romper la barrera de no poder resolver ciertos problemas no lineales. Pero no se conseguía automatizar el proceso de adaptar los pesos de las capas intermedias. En 1986, Rumelhart y otros autores, consiguen diseñar un algoritmo capaz de adaptar los pesos propagando los errores hacia las capas anteriores. Este algoritmo llamado **Backpropagation** conseguía resolver problemas no lineales. A raíz de este descubrimiento, se empezaron a utilizar los **perceptrones multicapa**, estructura de lo que hoy se cataloga como **redes neuronales** [14].

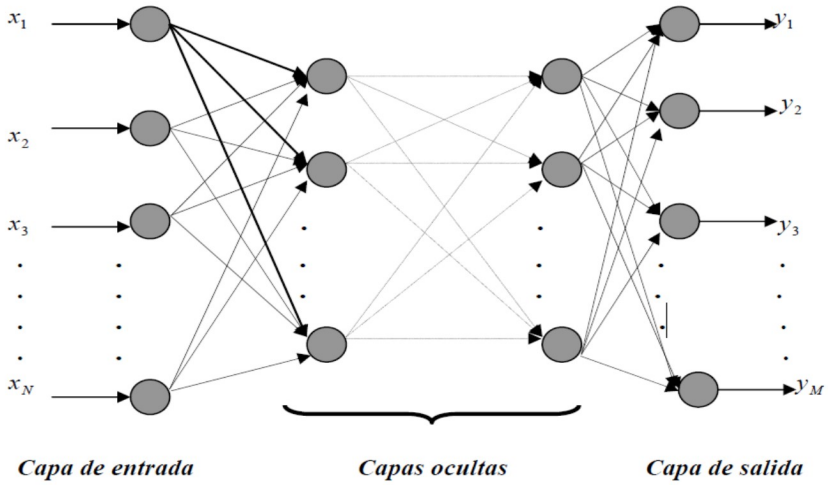

Imagen 3.6. Representación de un perceptrón multicapa.

Este tipo de estructura, como se puede observar en la imagen 3.6, está compuesto por:

- **Capa de entrada:** con sus respectivas neuronas y sus pesos asociados. En esta capa no se produce procesamiento alguno.

- C**apas ocultas:** transforman las entradas a salidas a través de una función de activación.

- C**apa de salida:** donde una vez generada una salida, se verifica y se procesan los pesos de las conexiones mediante el algoritmo **backpropagation.**

Una vez asentada esta estructura, empieza a utilizarse para resolver gran cantidad de problemas como pueden ser el reconocimiento de voz, reconocimiento de imágenes y traducción de lenguajes de programación.

Durante los años 90 hasta principios de los 2000 continúa la investigación de las redes neuronales, aunque será en el año 2012 cuando se alcanza el máximo auge de la tecnología. Esto es debido a un concurso de reconocimiento de imágenes[2], donde una **red neuronal convolucional**[3] ejecutada en una GPU, consiguió reducir el porcentaje de error de los competidores de un 26% a un 16%, un 10% de mejora al respecto.

Este hito se considera el inicio del ***deep learning***.

Estos algoritmos intentan modelar abstracciones de alto nivel para resolver problemas complejos. Utilizan funciones como el **descenso de gradiente con mini-batch,** función que reduce la varianza de actualización de los parámetros de la red neuronal, lo que conlleva encontrar una función de error más estable durante el entrenamiento. También utilizan otro tipo de funciones como la rectificación de las unidades lineales, descarte de capas no útiles, redes convolucionales, inicialización de pesos y aprendizaje residual, entre otras.

---

[2] "ImageNet Classification with Deep Convolutional Neural Networks", en el enlace se encuentra el proyecto que realizaron para el concurso. https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[3] Una red neuronal convolucional es un tipo de red neuronal compuesto por múltiples capas de filtros convolucinales. La convolución es un operador matemático que genera una función de la transformación de dos funciones mediante el producto punto a punto.

4. Unity

Un motor donde se pueden desarrollar los experimentos mencionados anteriormente es Unity. Es un motor de creación y desarrollo de videojuegos multiplataforma desarrollado por la empresa Danesa Unity Technologies en 2005, aunque hoy en día está afincada en San Francisco, Califormia [15].

Actualmente, es la mayor empresa de desarrollo de videojuegos a nivel mundial debido en parte a su política de licencias, gratuitas y de pago, y a que muchas empresas independientes desarrolladoras con poco poder económico, al no poder permitirse crear su propio motor de desarrollo, optan por su utilización [16].

Es compatible con otros conocidos softwares de diseño, permitiendo importar los recursos directamente desde las plataformas sin tener que realizar ninguna transformación. Algunos de los programas compatibles son: Blender, Maya, Adobe Photoshop, ZBrush, etc.

Unity, desarrollado en C++, permite el uso de los lenguajes C#, Javascript y Boo para el desarrollo de videojuegos, aunque los dos últimos, desde julio del año 2017, están obsoletos.

Una de las grandes ventajas para su utilización, es el fácil y sencillo uso de su interfaz gráfica (Imagen 4.1), con multitud de aplicaciones “arrastrar y soltar”, una pantalla que enseña en todo momento lo que el jugador puede ver en el juego, una sección con los parámetros de cada objeto dentro de la escena y muchas más funcionalidades “user friendly” para que cualquier tipo de usuario pueda lanzarse al desarrollo de videojuegos.

Permite la creación de videojuegos tanto en 2D como en 3D. En ambas opciones, posee una amplia librería programada llamada *físicas* para que el usuario no tenga que dedicar tiempo a su implementación. Resalta la facilidad de gestión de las cámaras, pudiendo crear en una escena varias de ellas y poder ir cambiando entre ellas de forma sencilla. También dispone de diferentes elementos lumínicos, para proporcionar a la escena la luz necesaria en cada momento, pudiendo crear luces direccionales, focales, de área, etc. y con la posibilidad de ajustar el color, el brillo y el ángulo de cada una de ellas.

La escena es el entorno donde se genera el mundo virtual para el videojuego. Ésta, estará compuesta por los llamados *Game Objects*, que pueden ser figuras 3D (cubos, esferas, cápsulas, etc.), figuras 2D, efectos, luces, audio, cámaras y los objetos llamados UI (interfaz de usuario), para la creación de menús, texto de referencia, etc.

Para la programación en C#, se hace uso de los llamados “scripts”, archivos independientes entre sí que se pueden asignar a los *Game Objects.* Los scripts sirven para dotar de la funcionalidad que se requiera a los objetos, como puede ser, controlar la colisión entre dos de ellos o el movimiento de un objeto por la escena.

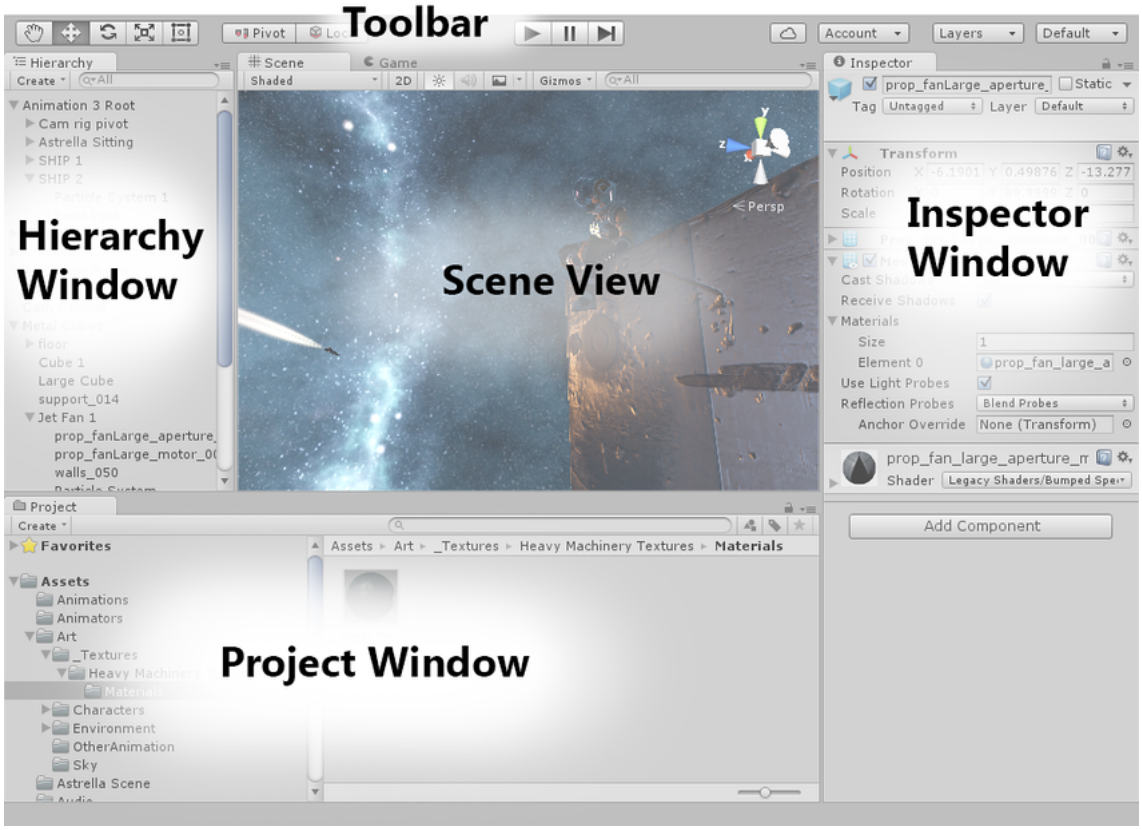

Imagen 4.1. Interfaz gráfica de Unity.

4.1. Tensorflow

Se utiliza Tensorflow como herramienta para dotar de inteligencia a los agentes de Unity. Los algoritmos que se utilizan con Tensorflow utilizan el *deep learning* como técnica de aprendizaje.

Tensorflow, desarrollado por Google y lanzado en Noviembre de 2015, es una librería de código abierto, escrita en Python y C++. Proporciona funcionalidades como entrenar redes neuronales, descifrar patrones… basándose en algoritmos para el aprendizaje automático.

Una de las mejoras que ha tenido con respecto a su predecesor *DistBelief* [4], es la de ofrecer la posibilidad del uso de varias CPUs y GPUs para el mismo objetivo, de forma que se agilice el proceso de entrenamiento.

Cuenta con una API escrita en Python. Con ella se pueden entrenar a los agentes mediante diferentes algoritmos. Esta API genera una la política de aprendizaje en el formato .bytes. Este archivo se añadirá al agente en Unity de modo que pueda poner en práctica los conocimientos

[4] DistBelief empezó su desarrollo en el año 2012 y se dejó de desarrollar en el año 2015.

adquiridos.

Una de las tareas para entrenar modelos es especificar los diferentes parámetros que requiere el algoritmo que se utilice. Estos parámetros, llamados *hiperparámetros,* son de mucha importancia para conseguir un entrenamiento adecuado y para ello se requiere de una serie de iteraciones hasta dar con los correctos. Por ello, Tensorflow cuenta con una herramienta para visualizar el proceso de aprendizaje para intentar ver dónde se puede mejorar. Esta herramienta se llama **Tensorboard.** Para este proyecto en cuestión, muestra unas gráficas sobre los *rewards* o recompensas que alcanza el agente durante las iteraciones que servirán para determinar si el entrenamiento está sirviendo para que el agente aprenda.

4.2. Aplicación de Machine Learning en Unity

Una vez explicado cada apartado por separado, es momento de juntar todas las piezas funcionando a la vez. Para ello, Unity cuenta con una librería propia especifica para el entrenamiento de agentes mediante Machine Learning llamada ***ml-agents.***

Esta librería desarrollada por Unity se encuentra todavía en una versión beta 0.3. No es estable. Todavía tiene muchas funcionalidades que no se han desarrollado y que se espera que en las próximas versiones vayan teniendo presencia. Hasta el momento, sólo es compatible con Windows y macOS [17].

La librería, para poder utilizarse, ha de importarse como un *asset* de Unity. Los *asset* son diferentes tipos de archivos importados que sirven para incorporarse a los objetos propios de Unity. Por ejemplo, una textura creada en Blender[5], se importaría a Unity como un *asset*, haciendo posible su uso. En el caso de este proyecto la librería ml-agents se importa como un *asset y* está formado por una estructura de directorios y ficheros.

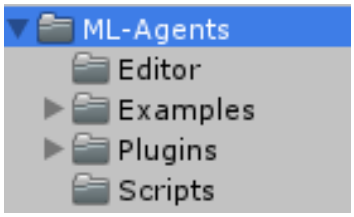

Imagen 4.2. Estructura de directorios y ficheros de ML-Agents.

Como se puede apreciar en la imagen 4.2, la librería cuenta con varias carpetas. En *Plugins* se encuentra la librería de Tensorflow. Con el plugin se podrá unir Tensorflow con Unity para entrenar los agentes (objeto con la capacidad de aprender). En la carpeta *Editor*, se pueden

[5] Wikipedia: Blender es un programa informático multi plataforma, dedicado especialmente al modelado, iluminación, renderizado, animación y creación de gráficos tridimensionales.

modificar algunos parámetros del comportamiento del cerebro (objeto que proporciona la inteligencia al agente) a la hora de aprender. *Examples* contiene una serie de ejemplos de aplicaciones prácticas para comprender su uso y utilizar parte del código si hiciese falta.

Las carpetas realmente importantes, donde se define el tipo de entrenamiento, las funciones de recompensa y demás cuestiones, son las siguientes.

- **Scripts**: Es el directorio principal de la librería. En él se encuentran todos los *scripts* que hacen que un objeto pueda aprender. Estos scripts, de código abierto, son modificables aunque no se recomienda hacerlo para no alterar el funcionamiento. Cada script representa una clase, con sus respectivas funciones y métodos que se pueden implementar para su uso. Para poder utilizar la librería, hay tres clases primordiales.

    - **Agent**: Para poder implementar las funciones propias de la clase, se extiende la clase a un script generado para un objeto que se quiere convertir en agente.

      El agente va adherido a un Game Object de Unity. El agente es el encargado de recoger la información del entorno, realizar una acción y recibir una recompensa, positiva o negativa, dependiendo de la situación.

      Este agente, pasa la información que recibe al *brain (*cerebro), el cual la procesa y envía de vuelta al agente la acción que debe realizar. Un agente sólo puede tener un cerebro.

      Por ejemplo, en el caso del proyecto, en el primer experimento, será la plataforma el agente que se aprenderá a mantener la pelota en equilibrio. En el segundo de los experimentos, el agente será el coche que aprenderá a recorrer el circuito.

      Las funciones más importantes de la clase, que deben implementarse para poder modificar el funcionamiento del agente son:

      A. *Start()*: Función donde se define el valor inicial de los parámetros del agente.

      B. *AgentAction(float[] vectorAction)*: Encargada de definir las funciones de recompensa y castigo. *vectorAction* es un vector del tipo float que recibe la acción que va a realizar el agente. Esta acción se procesa y dependiendo de lo que ocurra, se decide recompensar o castigar al agente. La función para añadir una recompensa se llama *AddReward(float reward)*.

      C. *CollectObservations()*: Se recogen los datos del entorno de modo que el cerebro tenga un contexto para poder realizar los cálculos y decidir la acción que ha de tomar. Dentro de la función, se añaden los datos del entorno mediante la función *AddVectorObs()*. Esta función recibe como parámetro cualquier tipo numérico, por ejemplo, puede recibir la posición en un eje en

concreto o incluso puede recibir un vector de tres posiciones con los tres ejes de una posición.

D. *AgentReset()*: Con esta función se reinician los valores que se han ido modificando durante la iteración una vez terminado el entrenamiento, para poder, nuevamente, llevar a cabo otra iteración del entrenamiento desde el inicio.

- **Brain**: El cerebro, como en el caso de los seres humanos, es el lugar donde se procesan todos los datos. Se hacen los cálculos necesarios y en consecuencia se toma la decisión que ha de realizar el agente. Es el objeto que encapsula la lógica de toma de decisiones en el agente.

  Hay varios tipos de cerebro. Cada uno de ellos se utiliza para determinados momentos del desarrollo.

  A. *External*: Mediante este tipo de cerebro, las decisiones se toman teniendo en cuenta el algoritmo PPO[6] que proporciona la API de Tensorflow. La información y la recompensa recogida se envía a la API mediante un comunicador externo. A continuación, la API devuelve la acción que ha de realizar el agente. Este proceso se repite hasta que termine el entrenamiento.

  B. *Internal*: Con este tipo de cerebro, las decisiones se toman desde la *política* generada por la API de Tensorflow. Este archivo del tipo .bytes, generado al terminar el entrenamiento, se añade al cerebro del agente que, basándose en los datos del archivo, tendrá la capacidad de decidir tomar una acción u otra.

  C. *Player*: Este tipo de cerebro se utiliza para comprobar que las recompensas y funciones definidas funcionan correctamente. Para ello, el desarrollador mediante algún tipo de controlador, como por ejemplo el teclado, toma las decisiones del agente. En este modo, el cerebro no aprende.

  D. *Heuristic*: Donde las decisiones se toman mediante código escrito. Es decir, se programan las decisiones que el agente va a adoptar y se entrena en base a ello. Se suele utilizar para comprobar que el entrenamiento llevado por parte de la API mejora al de código escrito.

- **Academy**: El script Academy sirve para que Unity reconozca el uso de agentes. Cualquier escena que contenga un agente ha de tener un Academy. Para el uso de la clase, hay que crear un script que extienda de la misma. De este modo aparecen obligatoriamente dos funciones para el funcionamiento de los agentes.

---

[6] Se explica en el punto 4.3 de la memoria.

A. *AcademyReset()*: Esta función se utiliza para modificar el entorno después de cada fase de entrenamiento. Por ejemplo, si se quisiese modificar la posición de un objeto que el agente tiene que alcanzar, se modificaría aquí.

B. *AcademyStep()*: Ocurre justo antes de la función *AgentStep()*, si se quisiese modificar el entorno durante el transcurso del entrenamiento, se debería hacer en este apartado.

4.3. Entrenamiento en Unity con Tensorflow - PPO

Como se ha mencionado durante el documento, se utiliza el aprendizaje por refuerzo como técnica de entrenamiento para los agentes. Para ello, se hace uso de la API de Tensorflow que se conecta con Unity mediante un comunicador externo, un socket [18].

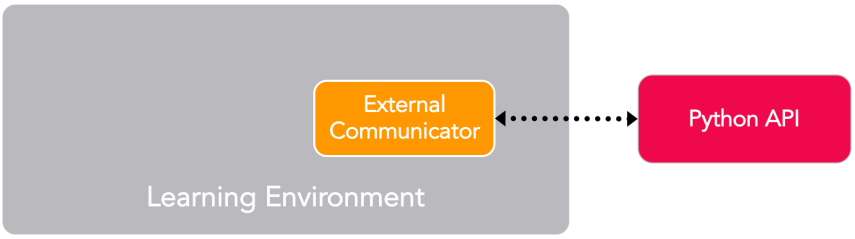

Imagen 4.3. Diagrama de comunicación entre Unity y Tensorflow.

Para el entrenamiento de los agentes, se usa el objeto *brain.* Una de las opciones que ofrece Unity es la capacidad de asociar a varios agentes un sólo cerebro, de manera que se paralelice el entrenamiento y en consecuencia, se agilice [18].

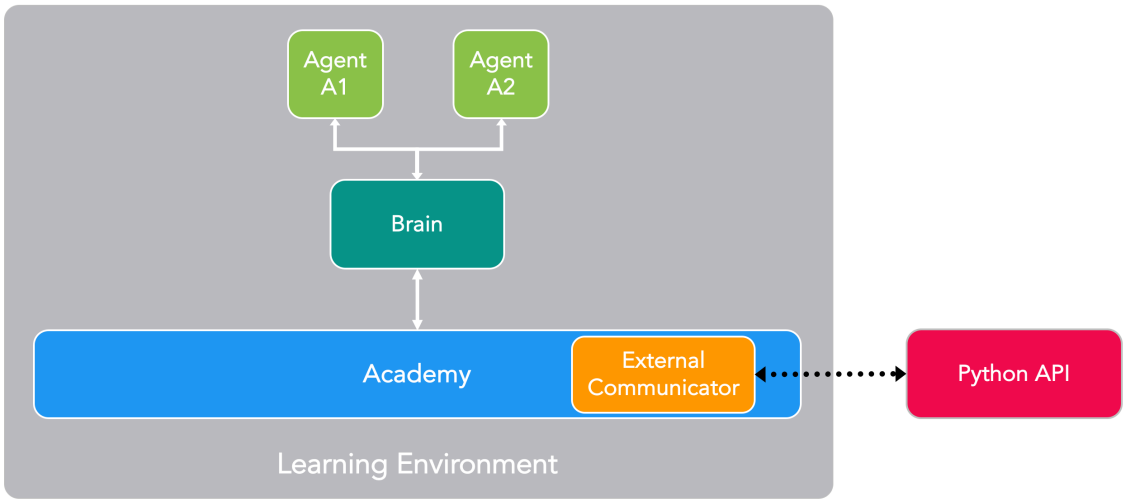

Imagen 4.4:. Diagrama del entrenamiento con varios agentes.

El algoritmo proporcionado por la API de Tensorflow para el aprendizaje por refuerzo es el algoritmo **PPO**, Proximal Policy Optimization.

PPO ha sido desarrollado por **OpenAI** en Julio de 2017. OpenAI es una compañía de investigación sin ánimo de lucro cuyo lema es "descubriendo y promulgando el camino a una inteligencia artificial segura"[7].

Se dedica a buscar nuevas técnicas de mejorar la inteligencia artificial. Cuenta con 60 investigadores e ingenieros a tiempo completo. Generan software gratuito para que cualquier usuario pueda entrenar y experimentar con la inteligencia artificial. Se subvenciona gracias a grandes empresas de la informática como Microsoft, Amazon e individuos como Elon Musk.

El algoritmo **PPO**, sustituye al algoritmo **Q-Learning** de la versión 0.2 de la librería ml-agents. Utiliza las técnicas del *deep learning* para tratar de generar la política óptima. Uno de los objetivos que busca el algoritmo es crear una **región de confianza** compatible con el **descenso de gradiente**, de modo que se simplifique el algoritmo y se garanticen actualizaciones adaptadas. En diferentes pruebas comparativas de algoritmos de aprendizaje por refuerzo profundo, se ha podido observar una mayor eficacia del algoritmo PPO con respecto a otros al tratar problemas que tratan tareas continuas [19].

Muchas veces no es suficiente con entrenar al agente para lograr los mejores resultados. El algoritmo cuenta con una serie de parámetros, llamados **hiperparámetros**, para tratar de mejorar los entrenamientos [20].

Algunos de los hiperparámetros más importantes son los siguientes:

- **Gamma**: Corresponde al grado de importancia que debe darle el agente al futuro en base a posibles recompensas. Es decir, este valor será mayor en caso de que el agente tenga más en cuenta los valores que ha obtenido en el presente en vez de los que pueda tener en el futuro.

- **Max Steps**: Cuántas iteraciones del entrenamiento ha de realizar el agente hasta terminar. Cuanto más complejo sea el problema, más iteraciones tendrá que utilizar.

- **Beta**: También llamada *entropía*, corresponde a la fuerza que tiene la política a la hora de tomar decisiones aleatorias, de modo que los agentes exploren una mayor cantidad de acciones durante el entrenamiento.

Durante el entrenamiento, **Tensorflow** ofrece la posibilidad de ver cómo va evolucionando el aprendizaje del agente mediante una herramienta llamada **Tensorboard**.

---

[7] *"discovering and enacting the path to safe artificial general intelligence"* recogido en la sección *about* de la página web de openAI.

Esta herramienta utiliza diferentes gráficas que muestran distintos datos relevantes para ver si el entrenamiento evoluciona favorablemente o por el contrario es mejor parar. Las gráficas (Imagen 4.5) que más datos van a aportar para el tipo de aprendizaje que se va a utilizar en el proyecto son las siguientes: [21]

- **Cumulative Reward**: Esta gráfica muestra cómo evoluciona la media de la recompensa que han ido acumulando los agentes durante las iteraciones del entrenamiento. Debe ir creciendo para un correcto entrenamiento.

- **Entropy**: Relacionado con el punto comentado en el apartado anterior, muestra la aleatoriedad en base a las decisiones que está tomando el agente. Para un uso correcto, debería tomar decisiones aleatorias al principio y posteriormente, debe decrecer de forma que se demuestre que las decisiones se toman en base al aprendizaje adquirido.

- **Learning rate**: Cuánto tarda en calcular la política óptima. Debe ir decreciendo con el paso de las iteraciones. Al llegar a 0, se puede considerar que se ha generado la política óptima.

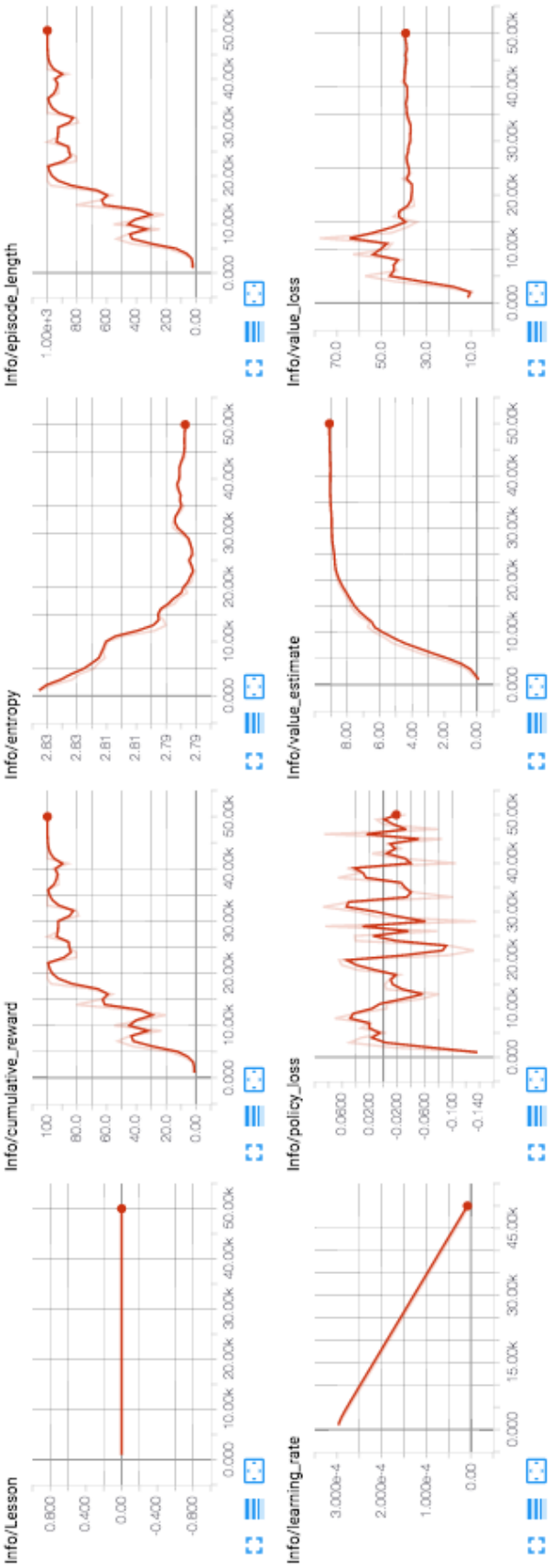

Imagen 4.5. Ejemplo de gráficas con la herramienta Tensorboard.

5. Experimentos

En esta sección se van a detallar todos los experimentos llevados a cabo con las tecnologías que se han ido mencionado en la memoria. Cada experimento está dividido en cuatro apartados.

5.1. Experimento 1 - Pelota en equilibrio

El objetivo de este primer experimento es el de conocer y aprender las posibilidades que permite el software. Por eso, tal y como recomienda **Unity**, se ha decidido implementar un pequeño tutorial básico basado en una pelota que mantiene el equilibrio sobre una tabla.

La pelota cae sobre la plataforma desde cierta altura y es la plataforma, que mediante la rotación sobre el eje X y el eje Z, ha de conseguir que se mantenga sobre la misma sin caerse. Para este propósito se utilizan las técnicas de **Aprendizaje por Refuerzo** comentadas anteriormente.

5.1.1. Preparación del entorno

Para la preparación del entorno, la escena está definida en 3 dimensiones. Está compuesta por sólo dos elementos, una esfera, que representará a la pelota y un plano, que será la plataforma que aprenderá a mantener la pelota en equilibrio.

La plataforma se coloca en el punto (0, 0, 0) en el sistema de referencia global de la escena. La pelota, en cambio, está colocada sobre la plataforma en el punto (0, 2, 0).

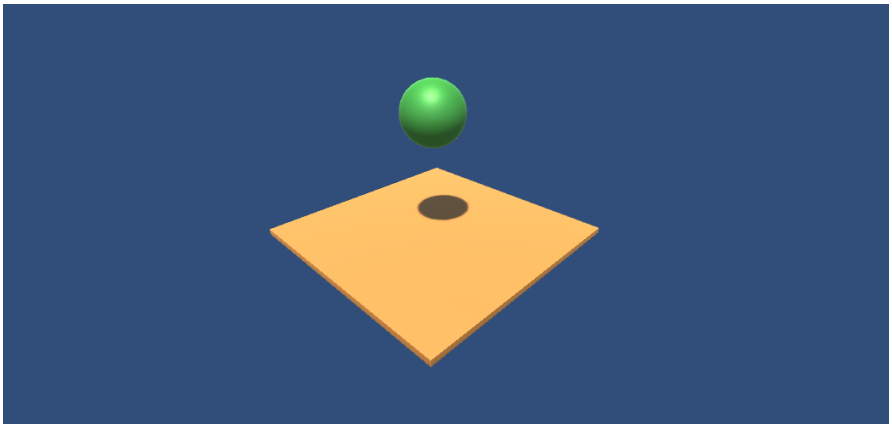

Imagen 5.1. Escena del experimento 1.

Estas serán las posiciones iniciales de los objetos durante las iteraciones que componen el entrenamiento.

5.1.2. Función de recompensa

Como se ha explicado anteriormente, las técnicas de **Aprendizaje por refuerzo** se basan en tomar decisiones en base a las observaciones que se toman del entorno. Por esto, el diseño de la función de recompensa es la parte más compleja a la hora de desarrollar un experimento que se basa en este tipo de técnicas. Esta función es la que utiliza el algoritmo para saber qué acción ha de realizar.

El algoritmo PPO, encargado del entrenamiento, recibe como parámetros de entrada las observaciones que se tienen del entorno de modo que se utilicen como inputs para una red neuronal.

Para este primer experimento, las observaciones que se hacen del entorno son las siguientes:

- **La rotación de la plataforma**, tanto del **eje X** como del **eje Z**. De esta forma, la red neuronal es consciente del ángulo de rotación en el que se encuentra la plataforma en todo momento. La rotación sobre el eje Y no se tiene en cuenta para la realización del experimento, ya que las acciones que tomará el agente, sólo se harán sobre los ejes X y Z.

- **La posición de la pelota** sobre la plataforma, lo que será un vector de 3 unidades, los **ejes X, Y y Z**.

- **La velocidad que tiene la pelota** al rodar sobre la plataforma, representado por un vector de 3 parámetros.

Estas 8 observaciones serán los parámetros de entrada de la red neuronal, es decir, compondrán la llamada input layer.

A continuación, dados los datos del entorno como entrada, el agente decidirá tomar una de las siguientes acciones:

- **Rotar** la plataforma en el **eje X**.

- **Rotar** la plataforma en el **eje Z**.

Estas dos salidas, corresponden a la llamada output layer.

Una vez decidida la acción que va a realizar el agente, es el momento de valorar si la acción que se ha realizado corresponde a lo que se esperaba del agente o no. Para ello, se hace uso de las recompensas.

- Si la pelota se mantiene sobre la plataforma en el siguiente estado, se recompensa positivamente al agente. En este caso, se ha recompensado con un valor de 0.1.

- Si la pelota se ha caído de la plataforma, se castiga al agente con un valor alto. En este caso, se recompensa con un valor negativo de -1.

De esta forma, la red neuronal, modificará la política en base a las recompensas que se han otorgado al agente, intentando que a lo largo de las iteraciones que componen el entrenamiento, el tiempo que la pelota aguante sobre la plataforma sea mayor.

5.1.3. Entrenamiento

Una vez se ha preparado el entorno y se han diseñado las funciones de recompensa, se debe diseñar un entrenamiento adecuado para el entorno sobre el que se está trabajando.

Para ello, lo primero es definir los hiperparámetros que componen el entrenamiento. Para este caso, debido a que es un proyecto bastante sencillo, no se han modificado los valores por defecto (gamma 0.99, max_steps 500.000 y beta $4 \bullet 10^{-3}$) y se han conseguido los resultados que se pueden ver en la imagen 5.2 después de realizar 500.000 iteraciones.

Como se puede observar en la gráfica de la recompensa acumulada (cumulative_reward), la línea tiende a subir de forma constante durante la fase de entrenamiento. Al principio, la media de la recompensa es muy pequeña, valores que rondan el 0, mientras que a partir de las 40.000 iteraciones, se empieza a ver un cambio y la media comienza a subir a valores cercanos a los 5 puntos. A partir de las 200.000 iteraciones ya se empiezan a ver unos cambios bastante significativos y al final de las 500.000 iteraciones se ve como los valores se estabilizan alcanzando el máximo posible.

En este caso, el máximo posible es 100 debido a que se trata de una tarea continua, no tiene por qué terminar la ejecución del entrenamiento si la pelota se mantiene en equilibrio. Por ello, se establece el número de pasos máximos[8] de cada iteración del entrenamiento a 5.000. En caso de llegar a estos 5.000 pasos, comenzará la siguiente iteración del entrenamiento.

Es significativa la gráfica de la entropía (entropy), que demuestra cómo el algoritmo va dejando de generar acciones aleatorias a medida que va adquiriendo conocimiento. De forma

[8] No confundir con el hiperparámetro “Max Steps”, explicado en la sección 4.3 de la memoria.

que al principio, el valor es alto y a partir de las 40.000 iteraciones, empieza a decrecer paulatinamente.

La gráfica de aprendizaje (learning_rate), mantiene un descenso lógico, ya que es al principio del entrenamiento donde se aprende más. De esta forma, a medida que continúa el entrenamiento, la línea desciende hasta un valor muy pequeño demostrando que se ha generado la política óptima para la resolución del problema.

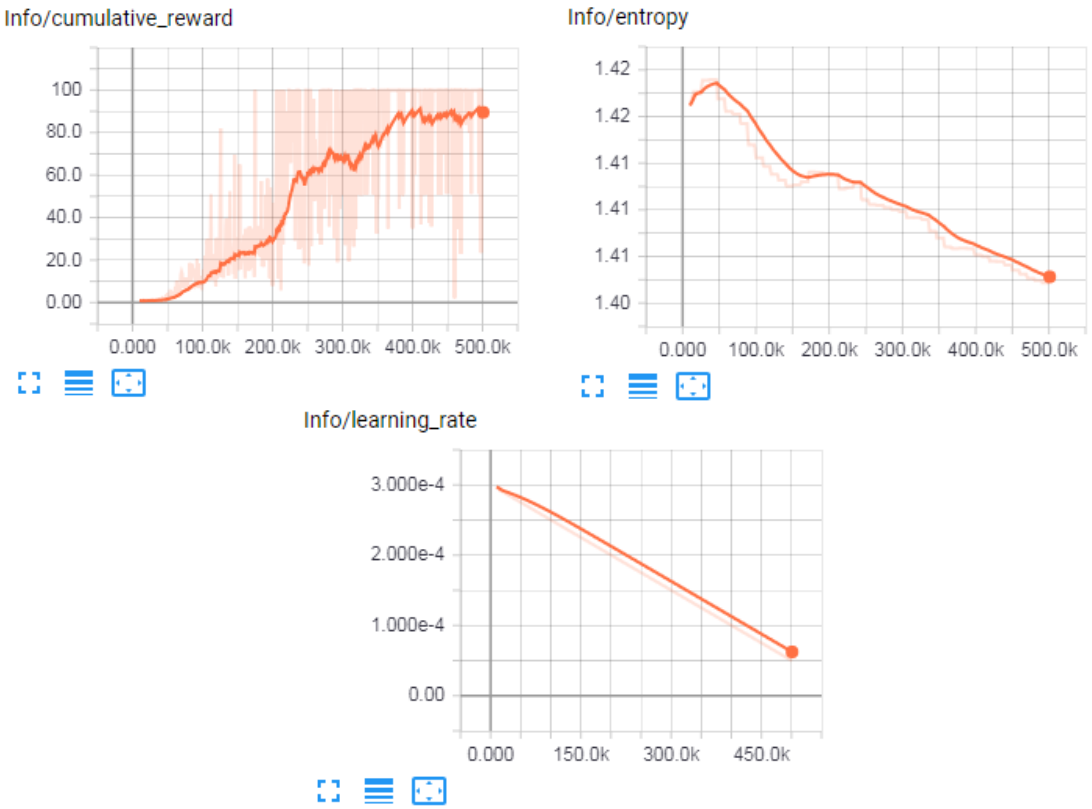

Imagen 5.2. Gráfica de entrenamiento del experimento 1.

Mediante estos datos se puede considerar que el algoritmo ha generado una política aproximada a la óptima ya que se ha detectado que la mejora a partir del punto en el que se ha parado el entrenamiento es mínima y se alcanza el máximo número de pasos que se llevan a cabo en cada iteración del entrenamiento. De este modo, la plataforma es capaz de mantener la pelota en equilibrio con un bajo porcentaje de fallo.

5.1.4. Conclusiones del experimento 1

En este último apartado del experimento, se comprueba si la política de aprendizaje generada funciona correctamente. Para ello, se añade la política al cerebro del agente y se comprueba si es capaz de mantener en equilibrio la pelota. Se ha creado un vídeo para poder

ver la evolución del aprendizaje del experimento accesible desde el apartado de las referencias [22].

En este caso, una vez añadida la política se puede ver que el agente consigue su función, mantener la pelota en equilibrio. Si bien es cierto que durante la evaluación final, no consigue mantener la pelota el 100% de las veces sobre la plataforma, se concluye que se ha logrado el objetivo. Con un número mayor de iteraciones sobre el modelo entrenado, se podrá conseguir una eficiencia del 100%.

5.2. Experimento 2 - Recorrer el circuito sin recibir información del entorno.

Una vez terminado el primero de los experimentos y adquiridos los conocimientos necesarios para desarrollar una aplicación que utilice las técnicas de Aprendizaje por refuerzo, se continúa con el desarrollo del proyecto principal.

El objetivo de este segundo experimento es conseguir que el coche llegue a la línea de meta en un circuito. El coche utiliza para ello checkpoints, puntos intermedios, para saber que está avanzando correctamente por el circuito. Estos checkpoints son los elementos que proporcionan la recompensa al agente. Se decidió realizar está implementación para verificar que el coche puede aprender de manera autónoma. A pesar de no ser la manera más eficiente de entrenar, se siguió adelante con la implementación debido a las pocas entradas que utiliza la red neuronal para entrenar.

5.2.1. Preparación del entorno

El entorno está definido en 3 dimensiones. Sobre un plano que hace de suelo, se colocan unos cubos que una vez escalados hacen de paredes. Se coloca una meta al final del circuito y finalmente se añade un cubo (coche) como agente que aprenderá a completar el circuito.

Además, se colocan los checkpoints en el suelo para ayudar al coche a seguir la trazada del circuito. El entorno final queda de la manera ilustrada en la imagen 5.3.

La idea principal es utilizar checkpoints colocados de forma simétrica por todo el circuito para verificar que son suficientes para que aprenda a recorrer el circuito. Por motivos que se estudian en la sección *entrenamiento* se añadieron una cantidad mayor de checkpoints en la segunda curva (Imagen 5.4).

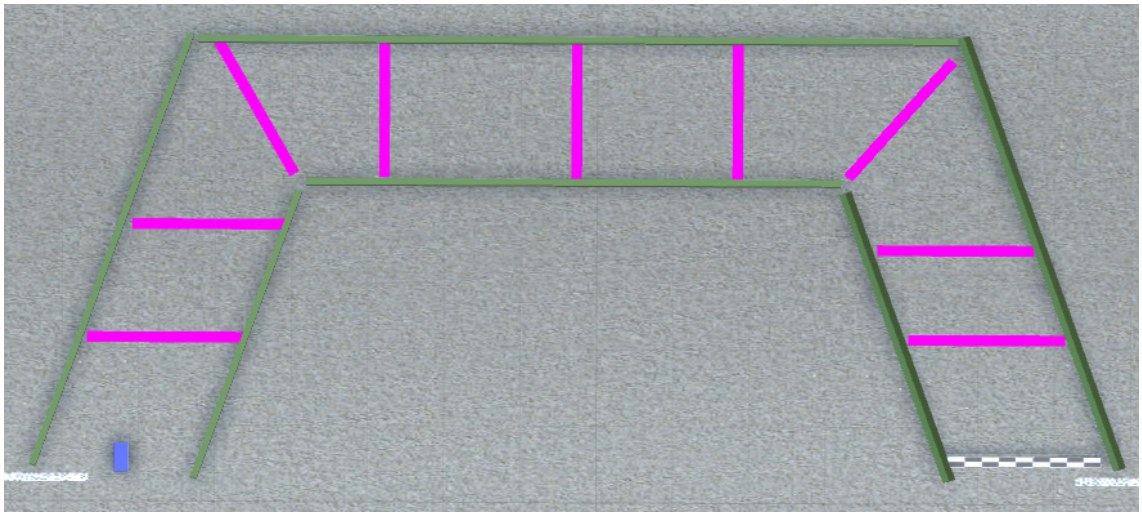

Imagen 5.3. Primera escena del experimento 2.

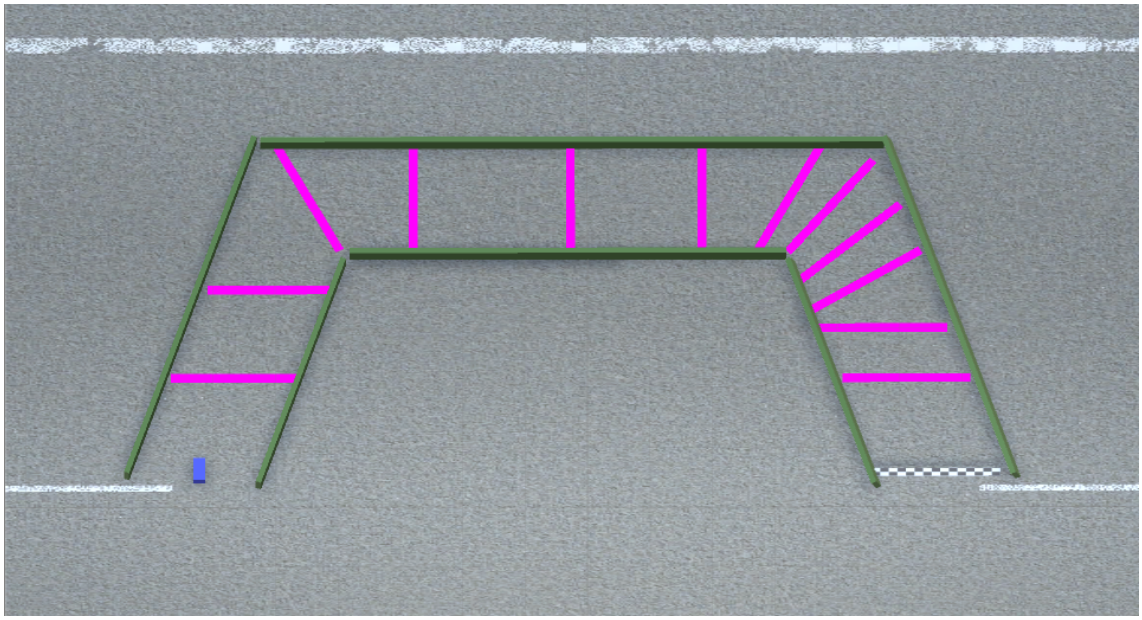

Imagen 5.4. Segunda escena del experimento 2.

5.2.2. Función de recompensa

Para este experimento, las decisiones que adopta el agente se basan mayoritariamente en el paso por los checkpoints y las recompensas que se generan al pasar por ellos y no se basan tanto en las observaciones que recibe el agente del entorno.

Las observaciones del entorno que recibe el agente y que son los 3 parámetros de entrada a de la red neuronal, son las siguientes:

- La **posición** del vehículo, **ejes X y Z**.

- La **rotación** del vehículo en el **eje Y**.

Una vez definidas las entradas, y teniendo en cuenta que la velocidad que tiene el coche es constante durante todo el circuito, la acción que ha de tomar el agente es girar el coche. Por ello, las salidas se definen de la siguiente manera:

- **Rotar** el vehículo sobre su **eje Y**. De esta forma, el coche girará hacia la derecha e izquierda dependiendo del valor que reciba. Teniendo en cuenta que el algoritmo PPO devuelve un parámetro float positivo o negativo, se decidió reducir ese valor a dos posibles valores, -1 o 1. Al recibir un -1 el agente girará hacia la izquierda y al recibir un 1 el agente girará a la derecha. El ángulo de giro es de 1 grado.

- **Continuar recto**. Otra posibilidad es que la mejor opción para el agente sea no girar. Si no se hubiese implementado esta acción, el coche estaría constantemente girando y no continuaría recto en el circuito.

Finalmente, una vez el agente ha tomado una acción, se han definido las recompensas que recibe de la siguiente manera.

- Al chocar contra la pared, recibirá la recompensa negativa más severa, -1. De esta forma, el agente debe aprender a evitar las paredes.

- Al llegar a meta, sin embargo, recibirá la mayor recompensa, 1. Una vez se alcanza la meta, se da por satisfactoria la iteración del entrenamiento y el agente vuelve a empezar de nuevo desde la salida.

- Cuando pase por alguno de los checkpoints, recibirá una recompensa positiva de 0.2. De esta forma, el coche sabe que va por el camino adecuado a medida que va acumulando una mayor recompensa.

Una vez definida la función de recompensa, se entrena al agente.

5.2.3. Entrenamiento

Mediante el entrenamiento se puede comprobar si la forma en la que se ha afrontado el experimento con el sistema de recompensas mencionado en el punto anterior es la más adecuada para el objetivo que se intenta conseguir.

Se han mantenido los hiperparámetros que vienen por defecto. Tampoco se ha definido el número máximo de iteraciones que tiene el entrenamiento por el mismo motivo.

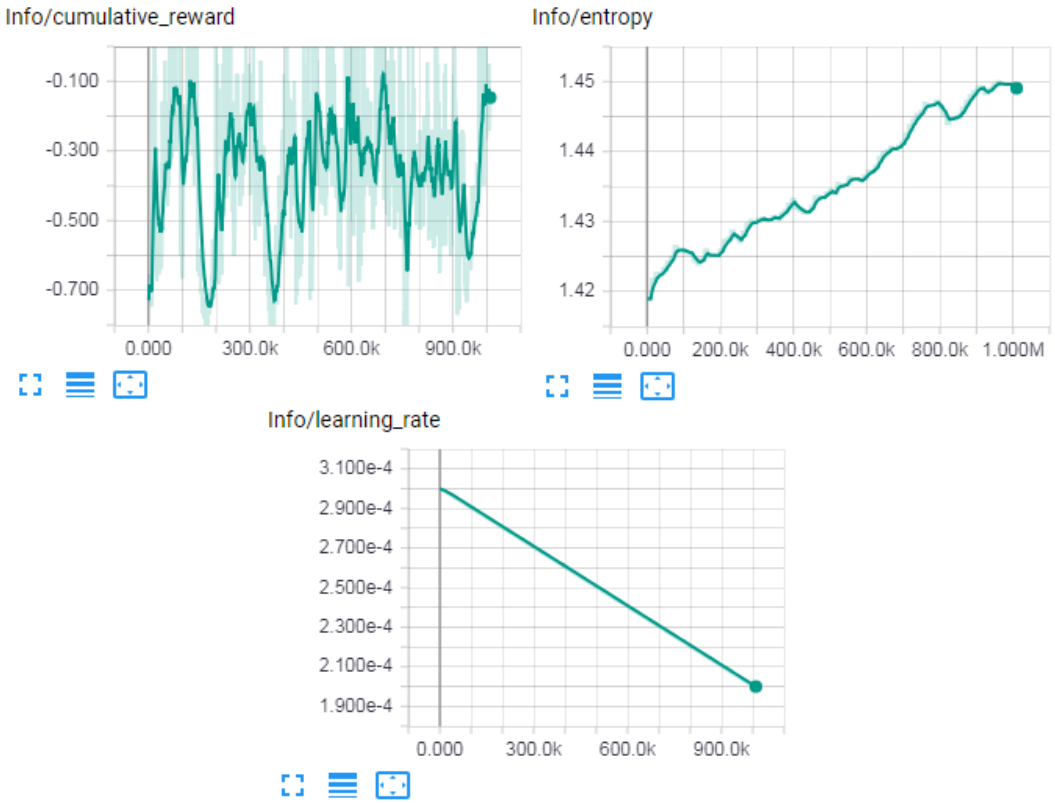

Imagen 5.5. Primera gráfica de entrenamiento del experimento 2.

Después de 1.000.000 de iteraciones entrenando, se observa que no se consiguen los resultados esperados. La máxima recompensa alcanzada corresponde a llegar a la segunda curva, a partir de ahí no evoluciona positivamente, es decir, no avanza hacia la meta. Como muestra la gráfica de la entropía, los valores que se toman son en su mayoría aleatorios y a medida que avanza el entrenamiento cada vez lo son más, comprobando así que el agente no aprende adecuadamente (Imagen 5.5).

Aún así, se observa que el agente mejora mediante este sistema de recompensas, por lo que se continúa con el experimento para comprobar si es posible que el agente aprenda basándose casi exclusivamente en las recompensas al pasar por los puntos de referencia. Por ello, se decide modificar la parte del entorno donde más fallaba el coche, la segunda curva, y añadir más checkpoints para tener más puntos de referencia en el aprendizaje (Imagen 5.4). Los resultados del entrenamiento se muestran en la gráfica de la imagen 5.6.

Después de más de 6.000.000 de iteraciones, los últimos resultados que se observan en la gráfica de la recompensa acumulada, se estabilizan e indican que se ha alcanzado la máxima recompensa posible en el circuito. La gráfica de la entropía a partir de 5.000.000 de iteraciones empieza a tomar valores descendentes. La gráfica del aprendizaje muestra que si se sigue entrenando, se alcanzará el valor 0, valor que indica que se ha generado (o está cerca de generar) una política óptima.

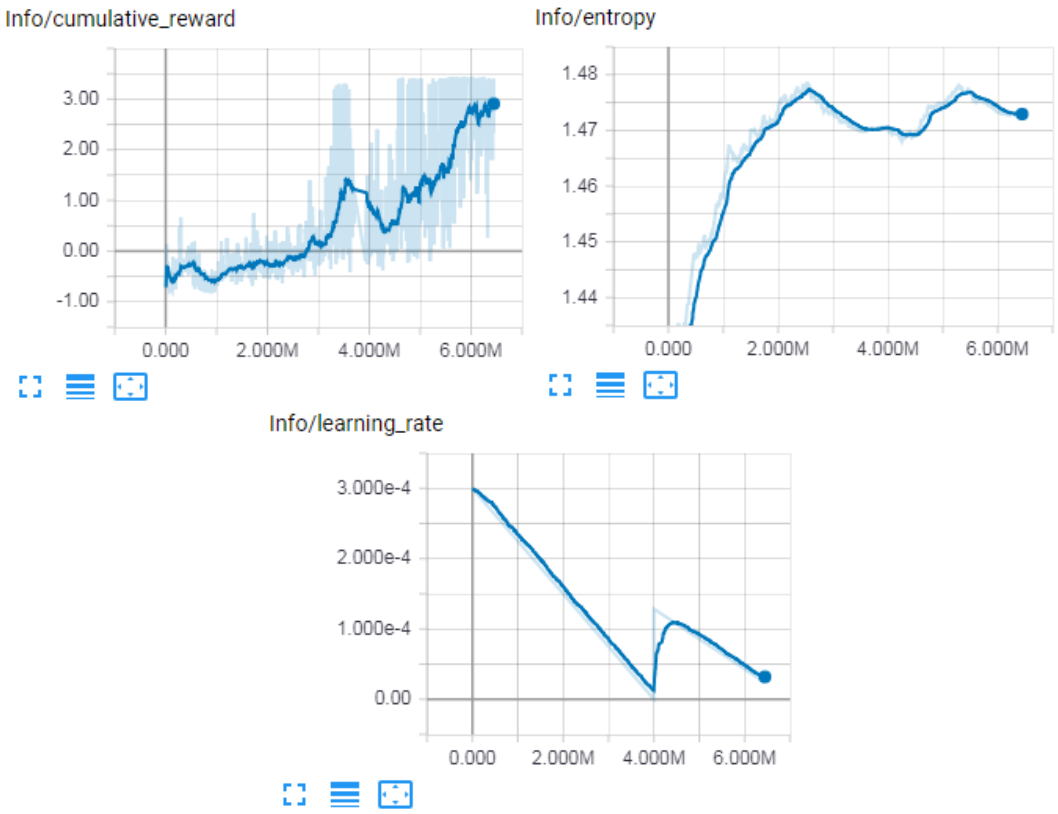

Imagen 5.6. Segunda gráfica de entrenamiento del experimento 2.

5.2.4. Conclusiones del experimento 2

Una vez implementada la política de aprendizaje generada por el algoritmo en el agente, se consiguió que este alcanzara la meta del circuito, aunque después de muchas iteraciones de entrenamiento y varios intentos fallidos. Como se hizo en el experimento anterior, se ha creado un vídeo para ver los diferentes estados de aprendizaje del agente accesible desde la bibliografía [23].

Se demuestra así que cuantas menos entradas obtenga el agente observando el entorno, más tiempo necesita para recorrer el circuito y menos conocimiento adquiere.

A pesar de haber conseguido el objetivo, este experimento no se considera como satisfactorio porque el agente solo es capaz de recorrer este circuito en concreto y no otro diferente. No ha adquirido los conocimientos suficientes para ello.

5.3. Experimento 3 - Recorrer el circuito en base a las distancias con las paredes

Una vez comprobado que el experimento anterior no cumplía con las expectativas para las que se había diseñado en un principio, se decidió diseñar una nueva propuesta. Este experimento se basa en recoger más datos del entorno, retirando los checkpoints y basándose en las distancias que hay con las paredes para tomar las decisiones. En este diseño se da más importancia a las observaciones del entorno utilizando unos sensores que comprueban en todo momento si el agente está colisionando con alguna de las paredes del circuito. Además, se quiere comparar este aprendizaje con el del experimento 2 y comprobar que de esta forma el agente aprende más rápido

5.3.1. Preparación del entorno

Para garantizar el aprendizaje progresivo del agente, se han diseñado varios circuitos que irán incrementando en dificultad. La política generada en un circuito y que ha resultado satisfactoria, se utiliza en el siguiente circuito (más difícil) para optimizar el aprendizaje y no partir de cero. Así, se necesita un menor número de iteraciones en el entrenamiento y se aumenta la eficiencia de aprendizaje. De esta manera se va añadiendo la experiencia adquirida en los entrenamientos previos.

Como se explicó en el experimento anterior, el entorno se compone por cubos escalados para formar las paredes, otro cubo escalado como agente y un plano como línea de meta.

El primero de estos circuitos se compone de una única curva a la derecha (imagen 5.7).

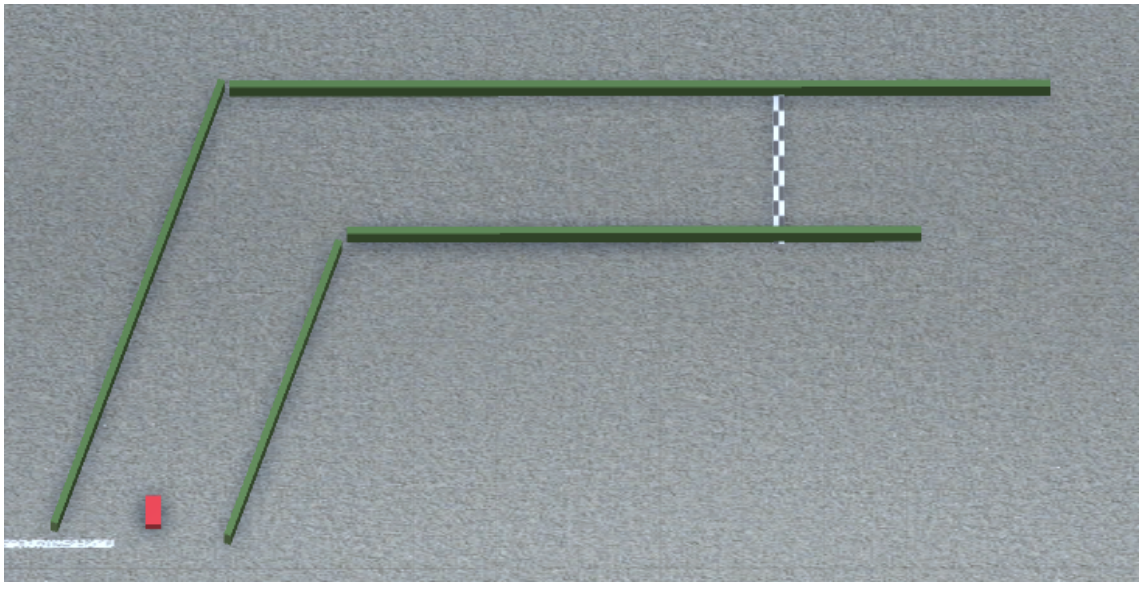

Imagen 5.7. Primer circuito del experimento 3.

El segundo entorno de entrenamiento es el del experimento anterior que consiste en dos curvas a la derecha (imagen 5.8). De esta forma, se comprueba que efectivamente, este modo de aprendizaje funciona mejor que el anterior.

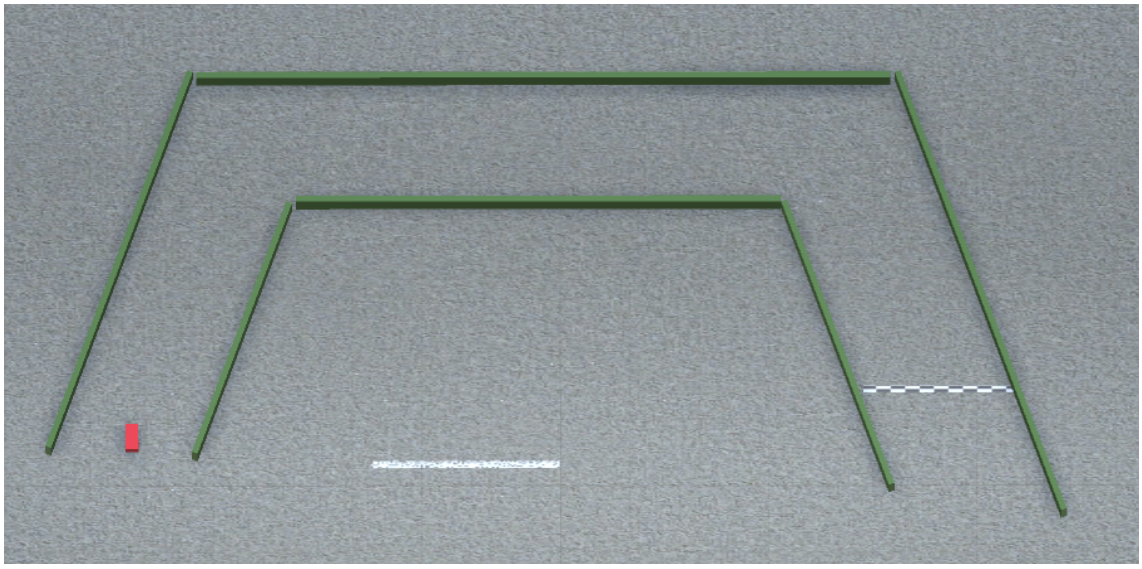

Imagen 5.8. Segundo circuito del experimento 3.

Una vez comprobado que el agente ha aprendido a recorrer el circuito del segundo experimento, se introducen las curvas a la izquierda, para continuar con el aprendizaje progresivo comentado anteriormente (imagen 5.9).

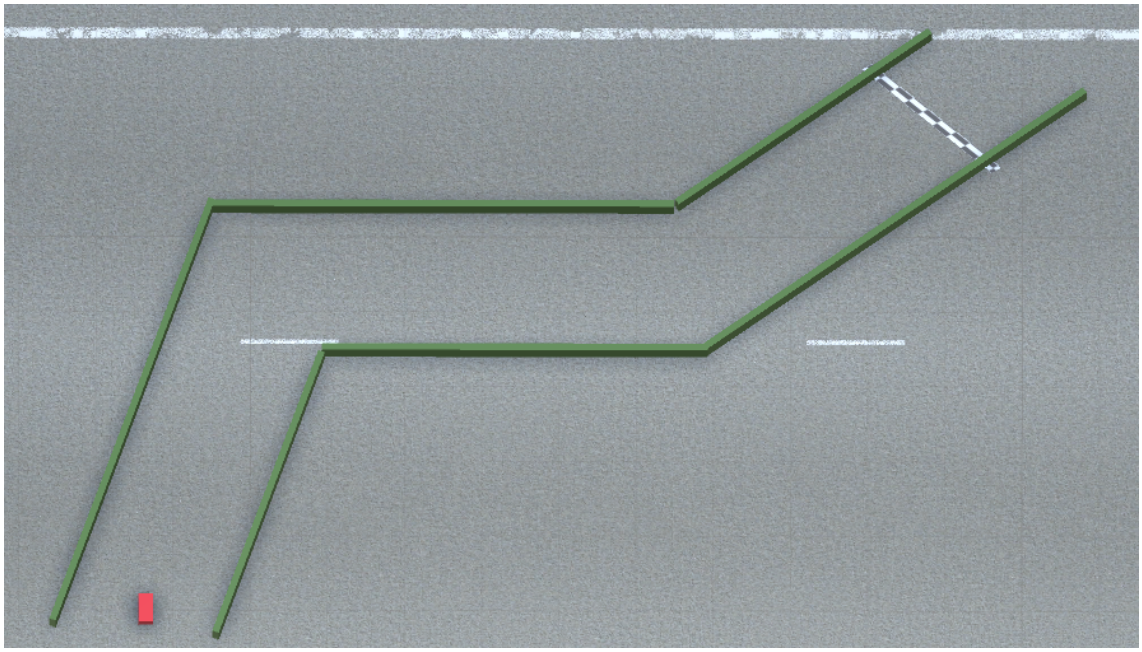

Imagen 5.9. Tercer circuito del experimento 3.

En este punto, el agente debe ser capaz de recorrer un circuito compuesto por varias curvas a la izquierda y a la derecha (imagen 5.10).

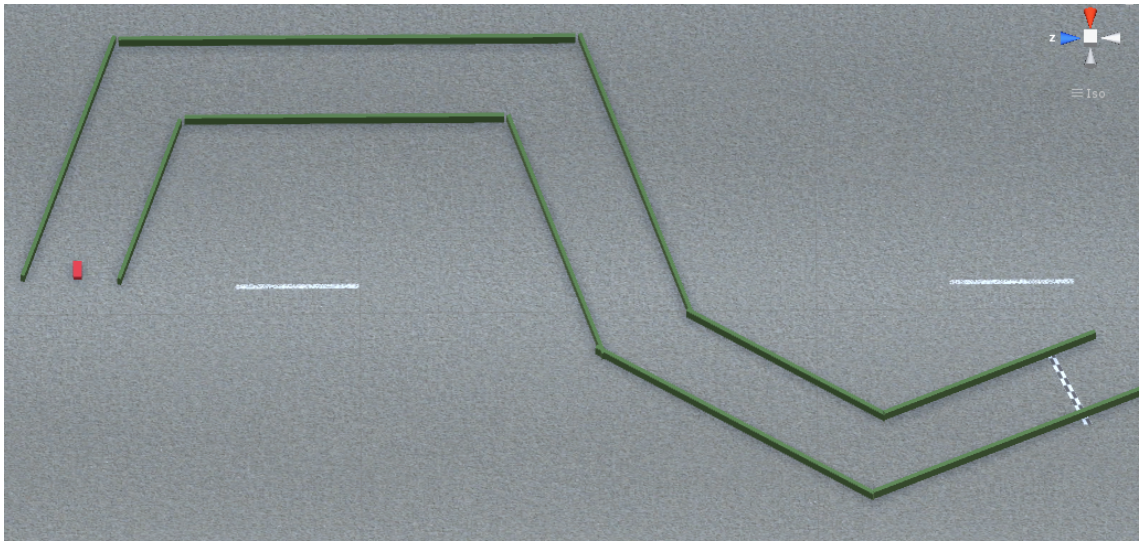

Imagen 5.10. Cuarto circuito del experimento 3.

5.3.2. Función de recompensa

A diferencia del experimento anterior, el agente percibe y recoge más observaciones del entorno. Para ello se utilizan unos sensores que dicen en todo momento si está colisionando con las paredes o no. A estos sensores se les llama ***ray casts.*** Los sensores, representados mediante líneas negras rectas (imagen 5.11), parten del centro del agente y su posición es relativa al objeto que los utiliza, es decir, al rotar el coche, los sensores rotan a la vez la misma cantidad de grados. La longitud de los sensores no es excesivamente larga para poder observar la capacidad de reacción que tiene el agente a la hora de tomar decisiones. Se ha decidido utilizar 5 sensores posicionados de la siguiente manera.

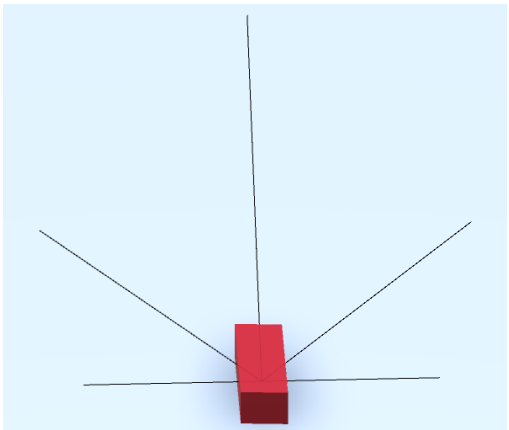

Imagen 5.11. Observaciones del entorno que recibe el agente en el experimento 3.

- El **sensor frontal**, con **0°** de ángulo respecto a la trayectoria del agente y cuya longitud es mayor que los otros sensores, sirve para detectar paredes frontales. La idea de utilizar un sensor más largo que los demás es dotar al agente de la capacidad de anticiparse y poder tomar la curva de la forma más adecuada.

- Dos **sensores laterales** cuya longitud sumada es igual a la anchura del circuito. Estos sensores están rotados **90° y 270°** respecto al sensor frontal (y por lo tanto, respecto a la trayectoria del agente). Se utilizan estos sensores con la idea de colocar el coche en el centro de la carretera.

- Dos **sensores diagonales**, rotados **45° y 315°** respecto al sensor frontal, para que el agente al tomar una curva sepa en todo momento si se aproxima a una pared o no.

- Además, aparte de los sensores, se recoge también la **rotación** del agente en el **eje *y*** para que sepa en todo momento el ángulo de rotación.

En este experimento cada *raycast* recoge 4 datos del entorno, 2 son los objetos detectables por el agente (paredes y meta) y los otros 2 son datos auxiliares generados por la función definida en la librería ml-agents, basados en la distancia entre el agente y los objetos. De esta manera, la capacidad de entrada de la red neuronal es de 21 variables (5 *raycast* x 4 datos del entorno + el eje *y* de rotación), 7 veces más que en el experimento anterior. Así, se le da más importancia a las observaciones del entorno.

Estos datos de entrada se van a traducir en una salida que es la acción que el agente tomará. Al igual que en el experimento anterior hay dos posibles salidas:

- **Rotar** el coche sobre su **eje Y**.

- **Continuar recto**.

Basándose en los datos recogido del entorno, el proceso de recompensas en el que se basa el aprendizaje por refuerzo varía con respecto al experimento anterior.

- Se mantiene la recompensa que se le asigna al agente cuando choca contra una pared, la más severa, -1.

- Se mantiene la recompensa que se asigna cuando cruza la línea de meta, 1.

En teoría, con estas recompensas, el coche debía ser capaz de encontrar la línea de meta, ya que se esperaba que el agente fuese capaz de evitar chocar con las paredes al cabo de un tiempo de entrenamiento. Sin embargo, después de una cantidad pequeña de iteraciones, se observó que el coche tomaba las mismas acciones una y otra vez. Las acciones hacían que el agente chocase contra la misma pared. Por ello, para garantizar que el coche alcanzase la meta,

se implementó una recompensa muy pequeña que se asignaba cada vez que el agente tomaba una acción y no derivaba en chocar contra alguna pared.

- La recompensa que se asigna es de valor 5 * 10^{-4}. Mediante esta recompensa, se consigue aumentar bastante el ritmo de aprendizaje. Aunque bien es cierto que no es la mejor opción, ya que se está premiando aguantar lo máximo posible en el circuito. Lo ideal sería que se recorriese el circuito de la manera más rápida posible.

5.3.3. Entrenamiento

Como se ha comentado anteriormente, se ha pretendido implementar un entrenamiento progresivo, con el fin de ir mejorando de lo que el agente ha aprendido en los entornos anteriores. Además, para agilizar el proceso, se ha realizado un entrenamiento con múltiples agentes asociados a un único cerebro (imagen 5.12).

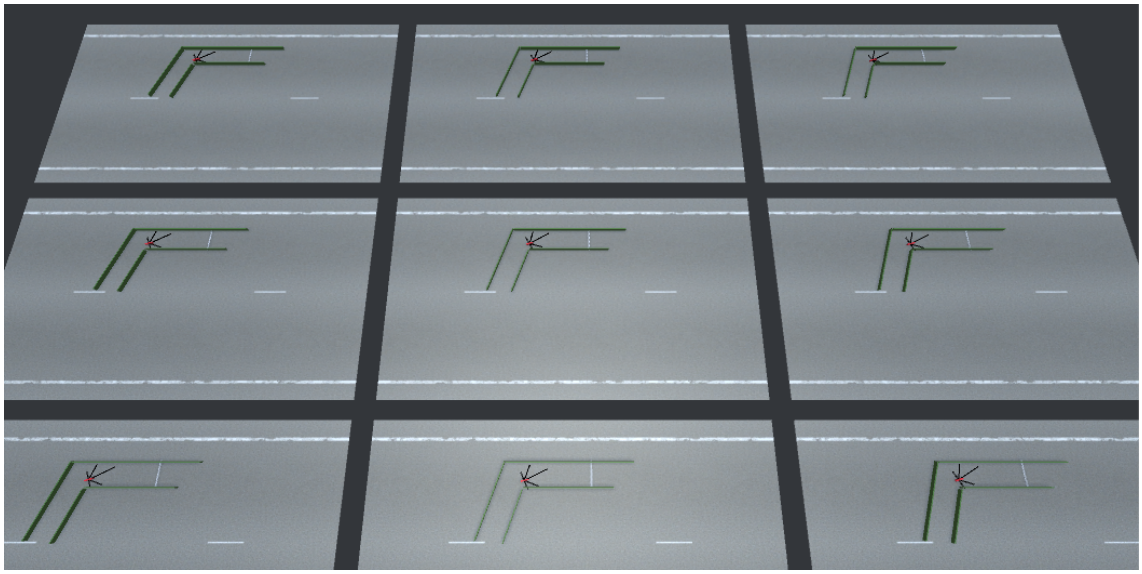

Imagen 5.12. Entrenamiento simultáneo de los agentes.

El primer entrenamiento se realizó sobre el entorno mostrado en la imagen 5.7, con el objetivo de que el agente adoptase los conocimientos suficientes para tomar curvas a la derecha. Los resultados son los que se pueden observar en la imagen 5.13.

La gráfica de la *cumulative reward* muestra como a partir de las 40.000 iteraciones del entrenamiento se van alcanzando los valores máximos. Aún así, según muestra la gráfica *learning rate*, el aprendizaje no se estabiliza hasta pasadas las 60.000 iteraciones, ahí es donde la gráfica empieza a descender indicando que se ha generado la política óptima. Este dato concuerda con el que muestra la gráfica de la *entropía*, en la cual a partir de las 60.000 iteraciones desciende el número de acciones que se toman de manera aleatoria.

Imagen 5.13. Primera gráfica de entrenamiento del experimento 3.

Una vez entrenado, también se ha utilizado la política generada en el coche que recorre el circuito de la imagen 5.8. Y se demuestra que es válido para ambos circuitos.

A continuación, se utiliza el entorno de la imagen 5.9. El objetivo es que el agente tenga consciencia de que existen curvas a la izquierda. Los resultados obtenidos una vez entrenado son los que se pueden ver en la imagen 5.14:

Los valores de las gráficas parten de las iteraciones del entrenamiento anterior. Sorprende la poca cantidad de iteraciones que ha necesitado el agente para estabilizarse en el máximo valor posible de recompensa (*cumulative reward*), solo 50.000. Hay que tener en cuenta que la cantidad de agentes entrenando simultáneamente (en este caso 9) agiliza el proceso de aprendizaje, multiplicando la velocidad de entrenamiento aproximadamente por la cantidad de agentes.

La gráfica del *learning rate* es bastante significativa en este caso, ya que indica que apenas ha aprendido algo durante este entrenamiento. De igual manera, la gráfica de la *entropía* al tener una forma descendente, indica que las acciones tomadas han sido en base a la política generada.

Imagen 5.14. Segunda gráfica de entrenamiento del experimento 3.

Finalmente, después de cargar en el agente la política generada en el entrenamiento hasta el momento, a pesar de progresar bastante en el circuito más complejo (imagen 5.10), no conseguía llegar a meta, por lo tanto se decidió entrenar al agente en el entorno de este circuito. El entrenamiento generó los datos observables en la imagen 5.15.

Después de 500.000 iteraciones se empiezan a alcanzar los máximos valores de recompensa. Aun así, la gráfica de la *entropía* no es descendente, lo que indica que las acciones que se están tomando son en gran parte aleatorias y que no se está generando una política óptima. Lo corrobora la gráfica del *learning rate*, donde se puede apreciar que no se alcanza en ningún momento el valor 0 en el eje de las ordenadas que es indicador de que ya se ha generado la política óptima.

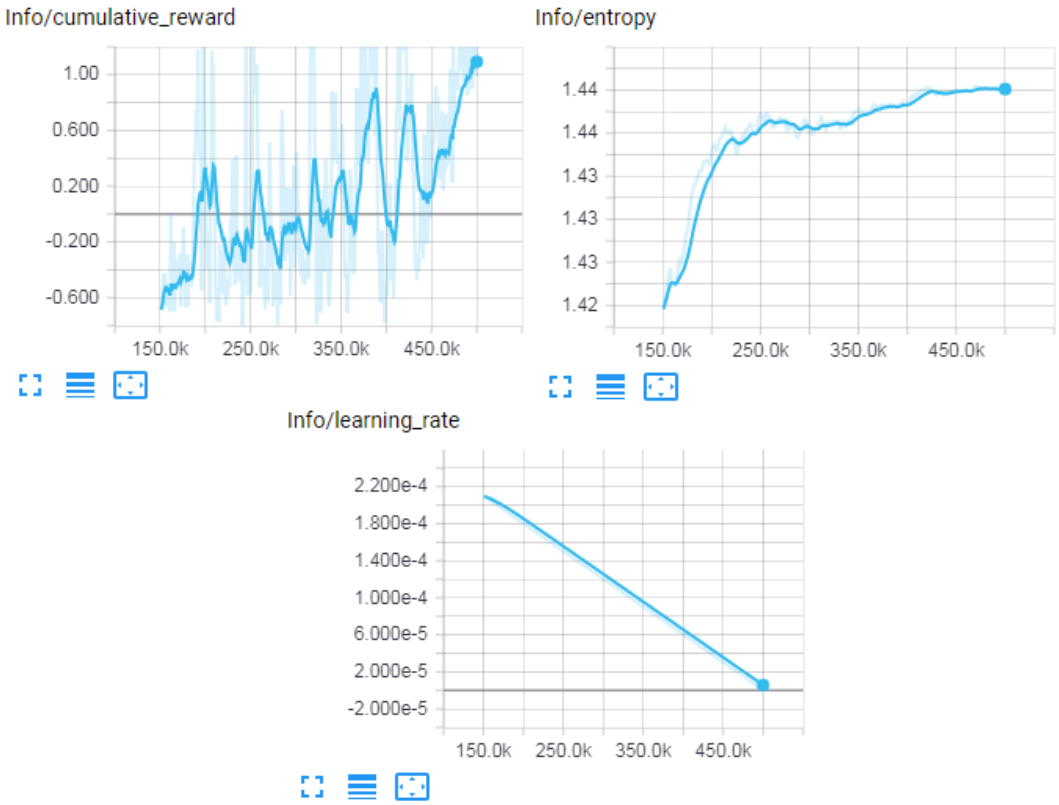

Imagen 5.15. Tercera gráfica de entrenamiento del experimento 3.

5.3.4. Conclusiones del experimento 3

Se considera que se ha conseguido el objetivo, el agente ha aprendido a recorrer el circuito y llegar a línea de meta. Aun así, seguramente, con más iteraciones en el entrenamiento, se garantizaría generar la política óptima para un mejor rendimiento del agente. También se ha creado un vídeo con el proceso de aprendizaje del agente durante los diferentes circuitos que componen este experimento accesible desde el apartado de las referencias [24].

Se ha ejecutado la política generada después del primer entrenamiento en el circuito de la imagen 5.7 en el circuito de la imagen 5.8. Después de compararlos, se ha podido comprobar que de este segundo modo, el aprendizaje es más rápido y además, de este modo no solo es aplicable a un único circuito, como ha quedado demostrado al aplicarse en el siguiente circuito (imagen 5.9).

Aun así, hay diferentes aspectos en este último experimento mejorables. Antes se ha mencionado que se recompensa al agente por continuar en el circuito. Lo ideal sería introducir otro tipo de recompensa que apremie al agente a terminar cuanto antes el circuito. Para ello, una vez llegado a meta y recompensado con el máximo valor posible, se le penalizará dependiendo del tiempo que ha estado recorriendo el circuito, consiguiendo así, que cuánto más rápido recorra el circuito, menor penalización recibe. Además, en caso de no alcanzar la

meta, se recompensará de manera positiva haber avanzado más por el circuito, incluso si esto incluye chocar con una pared. De este modo, se conseguiría una mejora en el aprendizaje y se mejoraría el tiempo por vuelta del coche.

También se ha podido observar que el problema por el que en el último circuito el coche no alcanzaba la meta se debía a que en la última curva, no reaccionaba a tiempo y chocaba con la pared. Una posible solución al problema sería alargar los sensores para que detecte antes las paredes y así tenga más tiempo para reaccionar. Por otro lado se ha observado que la información obtenida por los sensores laterales no tienen tanta trascendencia a la hora del aprendizaje del agente, ya que a la hora de ejecutar la política generada, se observa que las decisiones se toman en mayor medida en base a los sensores diagonales y al sensor frontal.

6. Conclusiones y líneas futuras

Una vez terminados los experimentos se considera que el objetivo propuesto se ha cumplido. Si bien es cierto que se podrían haber implementado algunas mejoras, como se ha comentado en los análisis de cada experimento realizado.

Durante la realización de este proyecto, por una parte, se han estudiado e implementado algunas de las técnicas actuales más potentes, como son las técnicas de Machine Learning y más en profundidad en este proyecto, el aprendizaje por refuerzo. Por otra parte, para implementar lo estudiado, se ha aprendido a utilizar la herramienta Unity junto con la librería ml-agents que se encarga de aplicar esas técnicas de Machine Learning. A pesar de no haber utilizado las técnicas en una aplicación real, queda constancia de la potencia que pueden tener.

En este caso, el proyecto se ha enfocado al campo de los videojuegos, un sector, que como se comentó en la introducción, está en auge. A pesar de que aun no se han utilizado demasiado estas técnicas en los videojuegos, poco a poco se empiezan a implementar. Sin ir más lejos, en el año 2012 desarrollaron un juego en el que las estadísticas de las unidades enemigas se balanceaban en base a las acciones que tomaba el jugador [27]. Los llamados NPC (personaje no jugador) podrán desarrollar una inteligencia artificial mucho más profunda que las actuales de manera que podrán dotar a los videojuegos de una experiencia mucho más parecida a enfrentarse a otras personas que a un entorno simulado.

De hecho, en la versión utilizada para el proyecto (versión 0.3 aún en fase beta), se añadieron dos formas de aprendizaje distintas más, **aprendizaje por imitación** [28] y **aprendizaje por curriculum** [29]. Para la siguiente versión se ha anunciado que habrá diferentes métodos de aprendizaje, como por ejemplo, el uso de memoria en los agentes [30].

La verdadera fuerza de este tipo de algoritmos reside en la amplia aplicabilidad que tiene en todos los sectores de hoy en día. Precisamente por eso, una de las razones por las que se decidió llevar a cabo este proyecto era aprender a utilizar la que llaman, “la herramienta del futuro” [31]. Técnicas aplicables a todo lo posible imaginable, como por ejemplo la seguridad de datos, la seguridad personal, cuidado de la salud, detección de fraudes y un largo etcétera [32].

Además, como posible mejora a lo ya implementado en los experimentos, se podría añadir la capacidad de aumentar o disminuir la velocidad del coche según convenga. Se añadiría una variable extra a las posibles acciones que ha de tomar el agente y se podría observar de manera más visual la mejora del aprendizaje. Otro de los posibles trabajos futuros, es el de hacer carreras en diferentes circuitos con diferentes tipos de aprendizaje, de modo que se pueda verificar cuál de los métodos de aprendizaje es más rápido en aprender. De esta forma se podría valorar cuál de los métodos es el más adecuado para estos experimentos.

De forma semejante a la implementada en este proyecto, este tipo de técnicas se podrían utilizar para dotar a un robot de inteligencia para que pudiese ir de un sitio a otro en un entorno semiestructurado. Algo parecido a lo que empiezan a implementar algunas de las mayores marcas de coches del mundo, como por ejemplo el *Volkswagen SEDRIC* [25], un vehículo que no precisa de un conductor para su conducción. Sin ir más lejos en Donostia se puso en marcha un proyecto pionero que utilizaba este tipo de técnicas y hacía un recorrido de varias paradas en el parque científico y tecnológico de Miramón [26].

Para finalizar, una vez terminado el proyecto, no han hecho más que aumentar las ganas de descubrir las nuevas técnicas que irán surgiendo con el paso del tiempo. No hay que olvidar que, como quien dice, esta tecnología acaba de nacer.

7. Bibliografía

[1] El Plural. (2017). *El sector del videojuego, pilar de la industria cultural en España.* Recuperado de https://www.elplural.com/tech/2017/11/20/el-sector-del-videojuego-pilar-de-la-industria-cultural-en-espana

[2] Rodrigo, Cristina. (2017). *El poder de la industria de los videojuegos.* Recuperado de http://www.expansion.com/economia-digital/companias/2017/08/26/5996fc9722601d6d3b8b4598.html

[3] Ortega, Jose L. (2018). *La industria de los videojuegos emplea a 8.790 personas en España.* Recuperado de https://www.hobbyconsolas.com/noticias/industria-videojuegos-emplea-8790-personas-espana-184010

[4] El Plural. (2017). *El 41% de los españoles juega a videojuegos.* Recuperado de https://www.elplural.com/tech/2017/02/17/el-41-de-los-espanoles-juega-videojuegos

[5] Vidal, Marc. (2017). *La inversión en 'machine learning' aumenta y marca el futuro empresarial inmediato.* Recuperado de https://www.marcvidal.net/blog/2017/11/3/la-inversin-en-machine-learning-aumenta-y-marca-el-futuro-empresarial-inmediato

[6] Martín, Enrique. (2017). *Por qué 'machine learning' sera la tecnología más importante en 2018.* Recuperado de https://elpais.com/tecnologia/2017/11/28/actualidad/1511866764_933798.html

[7] Niggreti, Alessia. (2017). *Using Machine Learning Agents in a real game: a beginner's guide.* Recuperado de https://blogs.unity3d.com/es/2017/12/11/using-machine-learning-agents-in-a-real-game-a-beginners-guide/

[8] Shalev-Shwartz, Shai y Ben-David, Shai. (2014). *Understanding Machine Learning: From Theory to Algorithms.* Recuperado de http://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/understanding-machine-learning-theory-algorithms.pdf

[9] Sierra, Basilio. (2006). *Aprendizaje Automático: conceptos básicos y avanzados,* Pearson Educación. Capítulo 2: Paradigmas de aprendizaje automático.

[10] Nueva Acrópolis. *El perro de Pavlov y el reflejo condicionado.* Recuperado de http://gandia.nueva-acropolis.es/gandia-articulos/26024-el-perro-de-pavlov-y-el-reflejo-condicionado

[11] Sierra, Basilio. (2006). *Aprendizaje Automático: conceptos básicos y avanzados,* Pearson Educación. Capítulo 11: Aprendizaje por refuerzo.

[12] McCulloch, Warren S. y Pitts, Walter H. (1943). *A logical calculus of the ideas immanent in nervous activity*. Recuperado de http://www.cse.chalmers.se/~coquand/AUTOMATA/mcp.pdf

[13] Web Archive. (2012). *Perceptron Simple y Adaline*. Recuperado de http://web.archive.org/web/20121221114040/http://www.lab.inf.uc3m.es/~a0080630/redes-de-neuronas/perceptron-simple.html

[14] Sathyanarayana, Shashi. (2014). *A Gentle Introduction to Backpropagation.* Recogido de http://numericinsight.com/uploads/A_Gentle_Introduction_to_Backpropagation.pdf

[15] Unity. (2018). *El software líder a nivel mundial en la industria de los juegos.* Recuperado de https://unity3d.com/es/public-relations

[16] Keene, Jamie. (2012). *Unity Technologies marks one million developers for its game development tools.* Recuperado de https://www.theverge.com/2012/4/10/2938040/unity-game-tools-one-million-registered-developers

[17] Unity ml-agents. (2018). *Unity ML-Agents (Beta).* Recuperado de https://github.com/Unity-Technologies/ml-agents#unity-ml-agents-beta

[18] Unity ml-agents. (2018). *ML-Agents Overview.* Recuperado de https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md

[19] Schulman, John; Klimov Oleg; Wolski, Flip; Dhariwal, Prafulla y Radford, Alec. (2017). *Proximal Policy Optimization.* Recogido de https://blog.openai.com/openai-baselines-ppo/#content

[20] Unity ml-agents. (2018). *Training with Proximal Policy Optimization.* Recuperado de https://github.com/Unity-Technologies/ml-agents/blob/20569f942300dc9279587a17ea3d3a4981f4429b/docs/Training-PPO.md

[21] Unity ml-agents. (2018). *Training Statistics.* Recuperado de https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md#training-statistics

[22] Aguayo, Asier. (2018). *Experimento 1. Pelota en equilibrio.* Recuperado de https://www.youtube.com/watch?v=8QFOKv4hI1c

[23] Aguayo, Asier. (2018). *Experimento 2. Aprendizaje en base a puntos de referencia.* Recuperado de https://www.youtube.com/watch?v=SC8DJpyLJIw

[24] Aguayo, Asier. (2018). *Experimento 3. Aprendizaje en base a sensores.* Recuperado de https://www.youtube.com/watch?v=xfNSO_DjxPA

[25] Muela, Cesar. (2018). *Probamos el Volkswagen SEDRIC, el coche autónomo de nivel 5 que promete ser tu taxi del futuro.* Recuperado de https://www.xataka.com/automovil/probamos-volkswagen-sedric-coche-autonomo-nivel-5-que-promete-ser-tu-taxi-futuro

[26] Muñoz, Ainhoa. (2016). *El primer autobús sin conductor ya circula por Miramón*. Recuperado de http://www.diariovasco.com/san-sebastian/201604/06/primer-autobus-conductor-circula-20160406124937.html

[27] Champandard, Alex J. (2012). *Making Designers Obsolete? Evolution in Game Design.* Recuperado de http://aigamedev.com/open/interview/evolution-in-cityconquest/

[28] Unity ml-agents. (2018). *Imitation Learning.* Recuperado de https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md#imitation-learning

[29] Unity ml-agents. (2018). *Curriculum Learning.* Recuperado de https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md#curriculum-learning

[30] Unity ml-agents. (2018). *Memory-enhanced Agents using Recurrent Neural Networks.* Recuperado de https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Feature-Memory.md

[31] El País Retina. (2018). *Think with Google. 'Machine Learning' como herramienta de futuro (y presente) para la redefinición del marketing digital*. Recuperado de https://retina.elpais.com/retina/2018/03/06/innovacion/1520342184_406366.html

[32] Gónzalez Dono, María. (2017). *Los 10 usos más comunes en machine learning e inteligencia artificial.* Recuperado de https://blogthinkbig.com/los-10-usos-mas-comunes-en-machine-learning-e-inteligencia-artificial

8. Anexo 1: Instalación del software

8.1. Preparación del entorno

Para la puesta en marcha del proyecto se necesita una serie de programas compatibles tanto para Windows como para macOS. En el caso de este proyecto, se ha implementado sobre un ordenador con sistema operativo Windows 7 de 64 bits.

Todo el software mencionado en esta sección se puede descargar gratuitamente siempre y cuando sea para uso didáctico y no uso comercial.

8.1.1. Unity

Se puede descargar desde la propia página web de Unity. La versión utilizada para este proyecto es la 2017.3.1f1 de 64-bits. Las versiones inferiores a la 2017.1 no son compatibles con la librería ml-agents.

8.1.2. Python

Es un lenguaje de programación multiparadigma, es decir, soporta la programación orientada a objetos, la programación imperativa, así como la programación funcional. Con una sintaxis que favorece la lectura del código, ha ascendido hasta convertirse en uno de los lenguajes más usados a día de hoy.

La librería ml-agents utiliza una API escrita en Python, por lo que es necesaria su instalación y funcionamiento a la hora de desarrollar este proyecto.

8.1.2.1. Anaconda

En el caso de windows, para la instalación de Python, se recomienda utilizar Anaconda. Es un sistema de gestión de paquetes y distribuciones de software escrito en Python utilizado, entre otras cosas, para la instalación de Python en el sistema operativo. La versión que se utiliza en este proyecto es la 5.1 para windows de 64 bits. Esta versión, instala la versión 3.6.3 de Python.

8.1.2.2. Tensorflow

Para generar los archivos de datos interpretables por el cerebro del agente se necesita instalar Tensorflow, así como para hacer uso de Tensorboard, para poder analizar el entrenamiento en base a datos y estadísticas. En este caso, se ha instalado la versión 1.4.0 de Tensorflow ya que, hasta la fecha, es la única versión compatible con ml-agents debido a que la librería no es estable aún.

8.2. Ml-agents

Por último, la librería ml-agents, se puede descargar desde el repositorio en GitHub. Para este proyecto se utiliza la versión ml-agents0.3, última versión funcional con todo el software especificado anteriormente.

9. Anexo 2: Código fuente utilizado en el proyecto

9.1. Experimento 1

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MyBallAgent : Agent
{
    public GameObject pelota;

    //Observaciones del entorno que recoge el agente
    public override void CollectObservations()
    {
        AddVectorObs(gameObject.transform.rotation.z);
        AddVectorObs(gameObject.transform.rotation.x);
        AddVectorObs((pelota.transform.position - gameObject.transform.position));
        AddVectorObs(pelota.transform.GetComponent<Rigidbody>().velocity);
    }

     //Al iniciar una nueva iteración del entrenamiento, se reiniciarán a estos
       valores.
    public override void AgentReset()
    {
        gameObject.transform.rotation = new Quaternion(0f, 0f, 0f, 0f);
        gameObject.transform.Rotate(new Vector3(1, 0, 0), Random.Range(-10f, 10f));
        gameObject.transform.Rotate(new Vector3(0, 0, 1), Random.Range(-10f, 10f));
        pelota.GetComponent<Rigidbody>().velocity = new Vector3(0f, 0f, 0f);
        pelota.transform.position = new Vector3(Random.Range(-1.5f, 1.5f), 4f,
               Random.Range(-1.5f, 1.5f)) + gameObject.transform.position;

     }
```

Anexo 2

```
//Función de recompensa del agente.
    public override void AgentAction(float[] vectorAction, string textAction)
    {
        if (brain.brainParameters.vectorActionSpaceType == SpaceType.continuous)
        {
            float action_z = 2f * Mathf.Clamp(vectorAction[0], -1f, 1f);
            if ((gameObject.transform.rotation.z < 0.25f && action_z > 0f) ||
                (gameObject.transform.rotation.z > -0.25f && action_z < 0f))
            {
                gameObject.transform.Rotate(new Vector3(0, 0, 1), action_z);
            }
            float action_x = 2f * Mathf.Clamp(vectorAction[1], -1f, 1f);
            if ((gameObject.transform.rotation.x < 0.25f && action_x > 0f) ||
                (gameObject.transform.rotation.x > -0.25f && action_x < 0f))
            {
                gameObject.transform.Rotate(new Vector3(1, 0, 0), action_x);
            }

            SetReward(0.1f);

        }

              //Si la pelota está por debajo de la plataforma, se dará por
              concluido la iteración del entrenamiento.
        if ((pelota.transform.position.y - gameObject.transform.position.y) < -2f
              || Mathf.Abs(pelota.transform.position.x -
                 gameObject.transform.position.x) > 3f
              || Mathf.Abs(pelota.transform.position.z -
                 gameObject.transform.position.z) > 3f)
        {
            Done();
            SetReward(-1f);
        }

}
```

9.2. Experimento 2

```
using System.Collections.Generic;
using UnityEngine;

public class CocheAgent : Agent
{
    public Vector3 carStartPosition;
    private int colision = 0; //1 ha chocado con una pared. 2 ha llegado a meta.

    private void Start()
    {
        carStartPosition = gameObject.transform.position;
    }

    public override void CollectObservations()
    {

        //Posicion 'x' y 'z' del gameObject, en este caso el coche.
        AddVectorObs(this.transform.position.x);
        AddVectorObs(this.transform.position.z);

        //La rotación del coche, en este caso, el eje 'y' será sobre el que rotará
el coche al girar en las curvas.
        AddVectorObs(this.transform.rotation.y);

    }

    public override void AgentReset()
    {
        gameObject.transform.rotation = new Quaternion(0f, 0f, 0f, 0f);
        gameObject.transform.position = carStartPosition;
        colision = 0;
        //Debug.Log(GetCumulativeReward());
    }
```

```
public override void AgentAction(float[] vectorAction, string textAction)
{
    switch (colision)
    {
        case 1: //Ha chocado con una pared
            Done();
            AddReward(-1f);
            break;
        case 2: //Ha llegado a meta
            Done();
            AddReward(1f);
            break;
        case 3: //Ha pasado por el checkpoint
            AddReward(0.2f);
            break;
    }
    colision = 0;
    gameObject.transform.Translate(2f * Time.deltaTime, 0f, 0f);
    gameObject.transform.Rotate(new Vector3(0, vectorAction[0], 0), 1);
}

private void OnCollisionEnter(Collision collision)
{
    //Debug.Log(collision.gameObject.name);
    if (collision.gameObject.name == "Paredes")
    {
        //Debug.Log("Ha chocado contra una pared.");
        colision = 1;
    }
    else if (collision.gameObject.name == "Meta")
    {
        //Debug.Log("Ha llegado a meta.");
        colision = 2;
    }
     else if (collision.gameObject.tag == "CheckPoints")
    {
        colision = 3;
    }
}
```

9.3. Experimento 3

```
using System.Collections.Generic;
using UnityEngine;

public class CocheDAgent : Agent
{
    public Vector3 carDStartPosition;
    private int Dcolision = 0; //1 ha chocado con una pared. 2 ha llegado a meta.
    RayPerception rayPer;
    public GameObject Meta;

    public override void InitializeAgent()
    {
        rayPer = GetComponent<RayPerception>();
    }

    private void Start()
    {
        carDStartPosition = gameObject.transform.position;
    }

    public override void AgentReset()
    {
        gameObject.transform.rotation = new Quaternion(0f, 0f, 0f, 0f);
        gameObject.transform.position = carDStartPosition;
        Dcolision = 0;
        //Debug.Log(GetCumulativeReward());
    }
```

```
public override void CollectObservations()
{

    //0f = (1,0) en circle unit
    //90f = izquierda
    //270f = derecha

    string [] detectableObjects = new string[] { "Paredes", "Meta" };

    float[] rayosLaterales = { 90f, 270f };
    float rayDistanceLaterales = 0.7f;
    AddVectorObs(rayPer.Perceive(rayDistanceLaterales, rayosLaterales,
   detectableObjects, 0f, 0f));

    float[] rayoFrontal = { 0f };
    float distanciaFrontal = 2f;
    AddVectorObs(rayPer.Perceive(distanciaFrontal, rayoFrontal,
   detectableObjects, 0f, 0f));

    float[] rayosDiagonales = { 45f, 315f };
    float distanciaDiagonal = 1.2f;
    AddVectorObs(rayPer.Perceive(distanciaDiagonal, rayosDiagonales,
   detectableObjects, 0f, 0f));

    //La rotación del coche, en este caso, el eje 'y' será sobre el que rotará
   el coche al girar en las curvas.
    AddVectorObs(this.transform.localRotation.y);

}
```

```
public override void AgentAction(float[] vectorAction, string textAction)
{

    gameObject.transform.Translate(2f * Time.deltaTime, 0f, 0f);

    //Al llegar un float en el vector de acción, de esta forma, lo redondeamos
   a -1, 0 o 1.
    int ejeY = Mathf.RoundToInt(Mathf.Clamp(vectorAction[0], -1, 1));

    gameObject.transform.Rotate(new Vector3(0, ejeY, 0), 1);

    switch (Dcolision)
    {
        case 1: //Ha chocado con una pared
            AddReward(-1f);
            Done();
            break;
        case 2: //Ha llegado a meta
            AddReward(1f);
            Done();
            break;
    }

    //Le recompensamos que aguante en el circuito sin chocar.
    AddReward(0.0005f);

    Dcolision = 0;

}
```

```
    private void OnCollisionEnter(Collision collision)
    {
        //Debug.Log(collision.gameObject.name);
        if (collision.gameObject.name == "Paredes")
        {
            //Debug.Log("Ha chocado contra una pared.");
            Dcolision = 1;
        }
        else if (collision.gameObject.name == "Meta")
        {
            //Debug.Log("Ha llegado a meta.");
           Dcolision = 2;
        }
    }
}
```