Universidad de Córdoba
Escuela Politécnica Superior

Grado en Ingeniería Informática
Especialidad en Computación

Trabajo de Fin de Grado

Entrenamiento de una IA mediante aprendizaje por refuerzo para un juego hecho en Unreal Engine

- Manual Técnico -

Autor:

Francisco David Castejón Soto

Directores:

Dr. Manuel Jesús Marín Jiménez

Dr. Javier Sánchez Monedero

Córdoba, 9 de junio de 2024

Índice general

Índice de figuras

Índice de tablas

Parte I

La idea

Agradecimientos

A mi familia y a mi pareja, por seguir depositando su total confianza en mi independientemente de cuánto se alargara el proyecto.

Capítulo 1

Introducción

Un concepto tiene tantos inicios como acepciones en el diccionario, y la inteligencia artificial (IA) no es una excepción a esta regla. Sus definiciones han ido evolucionando a lo largo de la historia para dar forma a la idea general de la IA actual. Aun así, es posible afirmar que el siglo XX fue clave en su desarrollo. Por ejemplo, se considera que la IA como campo de investigación nació en un taller de verano de la universidad de Darthmouth, en 1956. Allí, matemáticos y científicos hablaron sobre la posibilidad de crear una máquina que pudiera pensar como un humano (Kaplan y Haenlein, 2019). Sin embargo, la IA como el concepto de «la máquina que aprende», llevaba en el colectivo de la mente humana desde principios de siglo, cuando los primeros cineastas mostraron al mundo uno de los géneros más famosos de su incipiente arte: la ciencia ficción. Durante esos cien años se realizaron múltiples estudios relativos a la IA (McCulloch y Pitts, 1943), los cuáles quizá se puedan resumir con una frase célebre. En 1950, Alan Turing sugirió lo siguiente: si los humanos utilizan la información de su alrededor junto a su capacidad de razón para tomar decisiones, ¿por qué no podrían las máquinas hacer lo mismo? (Turing, 1950).

De forma paralela a los avances informáticos que permitieron el progreso de la IA, los videojuegos han ido ganando terreno en la industria del en-

tretenimiento. Lo que empezó como una divertida patente sobre un tubo de rayos catódicos (Goldsmith y Ray, 1948) ahora se proclama como uno de los gigantes multimedia de nuestra sociedad. Para ser más exactos, se proyecta que la industria del videojuego superará los doscientos mil millones de dólares de beneficio anual en el año 2025 (McDonald, 2023) y, al igual que el campo de la IA, no parece que esas cifras vayan a empezar a bajar pronto. El auge de los juegos para móviles y las multimillonarias inversiones en videojuegos AAA[1] han hecho que el mercado se expanda rápidamente en los últimos años, creando nuevas e innovadoras formas de jugar. Estas han sido potenciadas además por el crecimiento de los servicios de vídeo en directo de Twitch o YouTube, que permiten a los espectadores ver e interactuar con sus ídolos mientras estos juegan. Es una combinación prácticamente simbiótica.

El rápido progreso de la IA ha dado a científicos e investigadores la capacidad de crear desafíos a la altura de expertos del género. Desde los años noventa se han ido utilizando varios juegos y videojuegos[2] como entornos donde probar diferentes algoritmos y técnicas de aprendizaje automático (de lo que se comentará más en el Capítulo 3). Este proyecto trata de explorar una posibilidad similar: la de crear una IA que pueda jugar a un pequeño videojuego por sí misma. Todo con la mera intención de crear un desafío para el jugador humano y que se vea incentivado a mejorar su puntuación para superar a la máquina. Como se detalla en el Capítulo 2, la intención es ser una prueba de concepto que demuestre hacer uso del aprendizaje automático en el contexto de los videojuegos, pero dejando un amplio margen de evolución para futuros trabajos.

[1] Un videojuego AAA o Triple A es una clasificación utilizada para los videojuegos producidos por una distribuidora importante, típicamente teniendo un marketing y desarrollo de alto presupuesto (Wikipedia, 2022).

[2] Aunque en muchas ocasiones se utilicen indistintamente, si aparecen juntos, "videojuego" se refiere a un juego electrónico como el Tetris, mientras que "juego" es una actividad lúdica que no requiere de electrónica, como el Parchís.

Capítulo 2

Objetivos

En este capítulo se muestra la información de aquello que se pretende obtener, dando una descripción de las funciones que debe cumplir el prototipo. Los objetivos se revisarán al final del manual como conclusión al proyecto.

2.1. Objetivo Principal

El objetivo de este proyecto es exploratorio e informativo. La finalidad es aprender sobre varias materias específicas que respondan a preguntas relacionadas entre sí. ¿Cuál es el estado del arte del aprendizaje automático en el mundo los videojuegos? ¿Puede un motor gráfico como Unreal Engine 5 manejar agentes pre-entrenados? ¿Qué herramientas serían necesarias para semejante tarea? El propósito es responder a esas preguntas y, al mismo tiempo, añadir contenido al portfolio personal. Por tanto, este trabajo consta de un doble objetivo: engrosar el conocimiento sobre la materia y desarrollar un prototipo que sirva como carta de presentación a la industria laboral del videojuego.

Como ya se ha mencionado en el Capítulo 1, esto se conseguirá a través de la creación de una IA que sea capaz de jugar a un pequeño videojuego de disparos. Dicha tarea abarca el diseño y desarrollo de un entorno de pruebas,

una red neuronal y un programa que gestione el entrenamiento y testeo de la red.

Al desarrollar un videojuego, es de suma importancia que la experiencia de jugarlo no sea ni tan fácil como para que el jugador se aburra, ni tan difícil como para que se frustre. Ambas situaciones terminan con alguien molesto, por eso es preciso un balanceo correcto. Gran parte de la responsabilidad de encontrar ese fino balance recae en los diseñadores del videojuego y en los equipos de control de calidad. Los primeros trazan las mecánicas[3] del juego, mientras que los segundos aportan comentarios y críticas para ajustarlas debidamente. Aplicando el aprendizaje automático a este apartado de los videojuegos no se pretende sustituir a ninguno de ellos, sino más bien simplificar un trabajo que a menudo requiere de cientos de horas de pruebas y ajustes dependiendo del alcance del juego y tamaño del equipo de desarrollo. Algo que muchas veces es inviable de hacer correctamente por falta de recursos.

2.2. Objetivos Específicos

Para cumplir con estas metas, es necesario alcanzar ciertos objetivos específicos, los cuales se especifican a continuación:

- Diseño, desarrollo e implementación de un pequeño juego de disparos en 3D como entorno de pruebas.

- Diseño, desarrollo e implementación de una red neuronal capaz de jugar al juego.

[3] Gran parte de lo que define un juego es sus mecánicas, sus reglas, lo que los jugadores deben llevar a cabo para ganar. La labor del creador de videojuegos es conseguir un conjunto de reglas que estén equilibradas y permitan disfrutar a los jugadores (Wikipedia, 2023a).

- Diseño, desarrollo e implementación de un programa adyacente al juego capaz de gestionar el entrenamiento y testeo de la red neuronal.
- Análisis de datos y establecimiento de parámetros determinantes para el funcionamiento óptimo de la IA.
- Validación y corrección del sistema. Se deberá entregar un sistema que actúe con la mayor eficiencia que el estudiante pueda conseguir.

2.3. Objetivos didácticos

Además, el desarrollo de este proyecto pretende ampliar el conocimiento de las siguiente materias y herramientas:

- Unreal Engine, como herramienta de desarrollo de videojuegos.
- C++ y Python, como lenguajes polivalentes de programación.
- Stable Baselines, como biblioteca de aprendizaje por refuerzo.
- Gymnasium, como biblioteca de entornos para aprendizaje por refuerzo.
- Pytorch, como biblioteca de aceleración por GPU para redes neuronales.
- LaTeX , como lenguaje de maquetación de documentos.
- Múltiples conceptos de aprendizaje automático y redes neuronales.
- Redacción de documentos técnicos en el ámbito ingenieril con un acercamiento científico.
- La aceleración de software mediante hardware gráfico.
- El desarrollo de proyectos de ingeniería informática.

Capítulo 3

Antecedentes

Tras la introducción y el desglose de los objetivos, es común mencionar otros proyectos similares a modo de comparativa y como aporte contextual del tema a tratar. Sin embargo, primero se van a nombrar y definir ciertos programas, bibliotecas y algoritmos a modo de glosario. De esta forma siempre habrá una sección del texto donde volver si el lector no recuerda del todo algún concepto. Está dividido en tres secciones: 3.1.1 ¿Qué es la IA?, que trata de definir las ideas más importantes de la inteligencia artificial relacionadas con este trabajo; 3.1.2 ¿Qué es el aprendizaje por refuerzo?, que define su vocabulario clave, y 3.1.3 ¿Qué es un motor gráfico?, que hace lo propio con el apartado de los videojuegos. Tras esto, se procederá a hablar de los antecedentes en sí.

3.1. Glosario

3.1.1. ¿Qué es la IA?

Existen muchas definiciones de IA, pero todas ellas se podrían catalogar como combinaciones de «humanidad frente a razón y pensamiento frente a comportamiento», siendo el enfoque del comportamiento-racional uno de los más aceptados: "IA es el estudio y construcción de sistemas que hacen «lo

correcto», en función de su objetivo" (Russell y Norvig, 2020). Sin embargo, Alan Turing se inclinaba más por el comportamiento-humano, llegando incluso a diseñar una prueba para definir operacional y satisfactoriamente si un sistema es inteligente. Dicho test utiliza como evaluador a un humano, y se basa en realizar preguntas y obtener respuestas. Si el evaluador no es capaz de discernir si la respuesta viene de una persona o no, entonces el sistema al otro lado se cataloga de inteligente (Turing, 1950). Siguiendo esta filosofía, se podría decir que una "IA es una entidad informática con capacidad de emular ciertas características mentales humanas". Según el enfoque que se use, "IA" puede ser el estudio de una materia o el producto de dicho estudio, independientemente de que use aprendizaje automático para lograrlo.

Hay, por ejemplo, ciertas diferencias en torno a las diversas acepciones con las que se acuñará el término "IA" durante este escrito. Cuando se habla de "IA" en el contexto de los videojuegos, generalmente se está haciendo referencia a un sistema que controla las acciones de un personaje no jugable (PNJ)[4] de forma autónoma. Sería irrelevante la manera en la que ese PNJ toma sus decisiones, ya sea por medio de un árbol de comportamiento (Sekhavat, 2017), una máquina de estados finitos (Jagdale, 2021), una red neuronal o cualquier otro modo. Es decir, la IA en este caso es el propio personaje del juego. Pero si, por ejemplo, un PNJ usara una red neuronal como controlador, esta sería descrita por un fichero llamado el "modelo" de la red, que también es IA en si mismo pero no es equivalente al PNJ.

Otras acepciones que podrían originar confusión son la diferencia entre IA y aprendizaje automático. La mayor parte de la IA contemporánea se realiza mediante aprendizaje automático, por lo que ambos términos suelen utilizarse como sinónimos, pero en realidad "IA" se refiere al concepto general de crear una cognición similar a la humana mediante programas informáticos,

[4]Un ejemplo de personaje no jugable sería el ladrón en el juego de "Los Sims" o los compañeros extra que se asignan a cada equipo cuando una partida de "Call of Duty" no está llena.

mientras que el aprendizaje automático es sólo uno de los métodos para conseguirlo (Coursera, 2024).

Por lo tanto, si se dijera que "la IA de nuestro ejemplo juega a un videojuego", realmente se estaría diciendo que el modelo de la red toma las decisiones del PNJ, el cual a su vez interpreta un papel en el videojuego de mayor o menor importancia.

A modo de representación visual de esta relación de conceptos, se muestra la Figura 3.1. En ella se puede apreciar como, por ejemplo, un robot inteligente es una forma de IA, el cual puede estar controlado por modelos entrenados con diferentes técnicas de aprendizaje automático, como el aprendizaje por refuerzo o los árboles de comportamiento. Esos modelos podrían a su vez hacer uso de redes neuronales, basadas en arquitecturas como la de un perceptrón multicapa, el cual, podría aumentarse hasta generar arquitecturas más complejas como las redes neuronales convolucionales.

Definida la idea principal, estos son otros conceptos clave que se usarán a lo largo del manual técnico:

- Aprendizaje automático: es la rama de la inteligencia artificial que trata de conseguir que la máquina aprenda, acercándose así a su cometido principal. Si catalogamos a la IA como "agente", entonces el aprendizaje automático intenta desarrollar técnicas que le permitan aprender a partir de "experiencias o datos" (generalmente recogidos en una base de datos). Además, el aprendizaje automático se centra en aquellos problemas que, por su complejidad, no pueden ser resueltos por un humano de forma eficiente. Por lo tanto es la propia máquina (con ayuda de programadores humanos) la que debe descubrir sus propios algoritmos. Un ejemplo sería suministrarle muchas fotos de gatos a un modelo, para ver si más tarde puede diferenciarlos en una foto con perros y gatos (Jordan y Mitchell, 2015). Como pequeño apunte, quizá al lector le suene más por su homónimo en inglés: «Machine Learning».

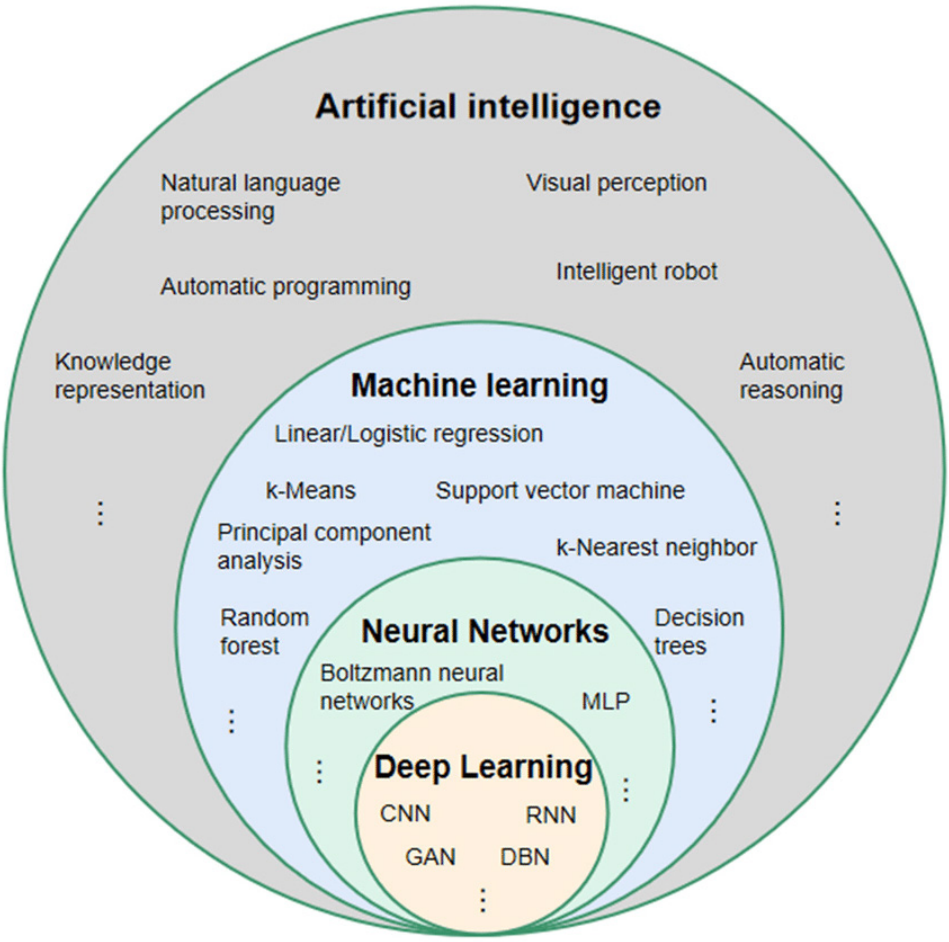

Figura 3.1: Relación entre inteligencia artificial, aprendizaje automático, red neuronal y aprendizaje profundo

Fuente: (Song, 2021)

- Modelo: en el ámbito de la IA es un programa, archivo o agente que ha sido entrenado en un conjunto de datos para reconocer patrones y hacer predicciones o tomar decisiones. Es decir, es una IA ya entrenada o parcialmente entrenada. El modelo tiene otra acepción en el aprendizaje por refuerzo, del cual se habla más en detalle en la siguiente sección. En ese caso, el modelo sería una representación del entorno, existiendo tanto algoritmos que generan modelos exactos del entorno,

como otros que no los necesitan.

- Red neuronal: se basa en un conjunto de nodos llamados "neuronas artificiales", que modelan vagamente las neuronas de un cerebro. Cada conexión, como las sinapsis de un cerebro biológico, puede transmitir una señal a otras neuronas. Los nodos de una red neuronal reciben señales, las procesan y las envían a otros nodos conectados ellos. Todas las redes tienen al menos una entrada y una salida, siendo la última la que presenta la utilidad para quien las usa, pues conlleva un "entendimiento" sobre la entrada. Las redes neuronales se entrenan cambiando los parámetros que hay dentro de cada uno de los nodos, así como la propia arquitectura de la red, lo que a menudo carece de sentido lógico para la mente humana. De ahí que las denoten como "modelos de caja negra" (IBM, 2024).

- Red neuronal profunda: se refiere al aprendizaje automático que utiliza redes neuronales de múltiples capas de elementos computacionales sencillos y ajustables (neuronas). El aprendizaje profundo depende en gran medida de un hardware potente. Mientras que una CPU de un PC estándar puede realizar 10^9 o 10^{10} operaciones por segundo, un algoritmo de aprendizaje profundo que se ejecute en un hardware especializado (por ejemplo, una GPU) puede consumir entre 10^{14} y 10^{17} operaciones por segundo, principalmente en forma de operaciones matriciales y vectoriales altamente paralelizadas (Russell y Norvig, 2020).

3.1.2. ¿Qué es el aprendizaje por refuerzo?

Es un área del aprendizaje automático en la que un agente intenta averiguar una buena política para interactuar con un entorno. La clave es que el tiempo es discreto; en cada paso temporal, el agente percibe el estado actual del entorno y selecciona una única acción. Tras realizar la acción, el

agente obtiene una recompensa o penalización, el tiempo avanza y el entorno pasa a un nuevo estado. A base de prueba y error el agente intenta aprender una política de elección de acciones que conduzca a la mejor suma posible de recompensas a largo plazo (Elkan, 2012). En la Figura 3.2 se muestra un diagrama de este bucle de pasos en el tiempo que genera este tipo de aprendizaje automático.

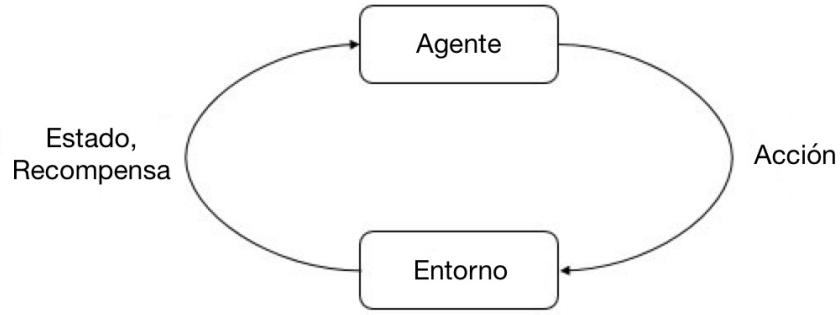

Figura 3.2: Bucle de interacción agente-entorno

Fuente: OpenAI

Cabe destacar que el aprendizaje por refuerzo difiere de otros métodos de aprendizaje automático en varios aspectos. Los datos utilizados para entrenar al agente se recogen a través de interacciones con el entorno por el propio agente (en comparación con el aprendizaje supervisado, en el que se dispone de un conjunto de datos fijo, por ejemplo). Esta dependencia puede dar lugar a un círculo vicioso: si el agente recoge datos de mala calidad (por ejemplo, trayectorias sin recompensas), no mejorará y seguirá acumulando malas trayectorias (E. d. StableBaselines3, 2021).

Estos son algunos de los conceptos y partes más importantes del aprendizaje por refuerzo (Nicholson, 2023):

- Agente: quien genera acciones. Controlado por el modelo descrito anteriormente. Por ejemplo, sería aquel que controla si el personaje debe disparar al pasar la mirilla por el objetivo en la galería de tiro.

- Acción (A): es el conjunto de todas las posibles acciones que el agente puede realizar. Los agentes suelen elegir de una lista de acciones discretas posibles o un conjunto de valores continuos.

- Recompensa (R): es la respuesta que recibe el agente tras mandar su acción para un cierto estado. Por ejemplo, en un videojuego, cuando Sonic toca un anillo, gana puntos. Las recompensas pueden ser inmediatas o retrasadas en el tiempo. Evalúan la acción del agente.

- Entorno: es el mundo a través del cual se mueve el agente y que responde a este. El entorno fabrica un nuevo estado tomando el estado actual y la acción del agente. Después, devuelve al agente una recompensa y el nuevo estado.

- Estado (S): es una situación concreta e inmediata en la que se encuentra el agente dentro del entorno.

- Factor de descuento (γ): es el factor por el que se multiplican las recompensas para amortiguar el efecto de estas en la elección de la acción del agente. Se usa para que las recompensas futuras valgan menos que las recompensas inmediatas, al fin y al cabo el futuro es más incierto que el ahora.

- Política (π): es la estrategia que el agente emplea para determinar la siguiente acción basándose en el estado actual del entorno. Mapea estados a acciones, las acciones que prometen la mayor recompensa. Si, por ejemplo, un agente puede moverse hacia arriba, abajo, izquierda o derecha en una cuadrícula, cada una de las cuatro acciones deberá tener asignada una recompensa esperada en función del estado actual. La política se encarga de asignar esos valores y decidir cual es la acción que más le conviene al agente.

- Valor: es la recompensa esperada a largo de una serie de estados tras

aplicarle el descuento. $V_\pi(s)$ se define como el retorno esperado a largo plazo del estado actual bajo la política π. La recompensa es una señal inmediata que se recibe en un estado dado, mientras que el valor es la suma de todas las recompensas que se pueden anticipar de ese estado.

- Valor-Acción (Q): es similar al valor, excepto que toma un parámetro extra, la acción actual A. $Q_\pi(s, a)$ se refiere al retorno a largo plazo tras tomar la acción A bajo la política π desde el estado actual S. Q mapea pares de estado-acción a recompensas.

- Curriculum learning: El aprendizaje por refuerzo exige una gran interacción con el entorno, lo que puede resultar prohibitivamente caro en escenarios realistas. Para resolver este problema, se puede aplicar el aprendizaje por transferencia al aprendizaje por refuerzo, de modo que la experiencia adquirida en una tarea puede aprovecharse al empezar a aprender la siguiente, más difícil. Las tareas, pueden secuenciarse en un currículum con el fin de aprender un problema que, de otro modo, podría ser demasiado difícil de aprender desde cero (Narvekar et al., 2020).

- Stable Baselines 3: conjunto de implementaciones de algoritmos de aprendizaje por refuerzo en PyTorch basados en el trabajo previo de OpenAI Baselines (Hill, 2023).

- Gymnasium: biblioteca de Python de código abierto para desarrollar y comparar algoritmos de aprendizaje por refuerzo que proporciona una API estándar para comunicarse entre algoritmos de aprendizaje y entornos. (C. OpenAI, 2023) Con esta API se pueden crear entornos virtuales, donde los agentes pueden entrenar.

Gran parte de estos conceptos tomarán un significado más holístico en el Capítulo 6, donde se explicará cómo el algoritmo de aprendizaje hace uso de cada uno de ellos.

3.1.3. ¿Qué es un motor gráfico?

Según la RAE (ASALE y RAE, 2024), un videojuego es "Juego electrónico que se visualiza en una pantalla", pero ¿qué es "Unreal Engine"?

- Motor gráfico o de videojuegos: programa para desarrollar videojuegos. Abarca desde la creación de entornos gráficos en 2D o 3D hasta la programación de toda la funcionalidad necesaria. Para este Trabajo de fin de grado se eligió usar el motor gráfico "Unreal Engine" (UE). Aunque en un principio UE, se concibió únicamente para la creación de videojuegos, su nivel de realismo es tal que actualmente su uso se ha extendido al mundo del cine en incluso a la arquitectura (Plante, 2015).

- MindMaker: extensión de Unreal Engine que hace uso de la librería "SocketIO" para conectar el motor gráfico con una instancia de Python. De esta forma, UE sería el entorno, mientras que en Python se ejecuta el agente (o modelo) que toma las decisiones. MindMaker permite, por tanto, una comunicación bidireccional a tiempo real entre ambos.

Si hubiera otros pequeños conceptos que necesitasen de una explicación, estos se definen en el momento de su aparición mediante una nota al pie.

3.2. La IA en los videojuegos

Los primeros videojuegos de la historia se basaban en algoritmos sencillos para proporcionar comportamientos y movimientos básicos de los enemigos, como moverse en el espacio y perseguir al jugador. A medida que la tecnología y la potencia de cálculo ha ido avanzando, también lo ha hecho el uso de la IA en los videojuegos (Skinner y Walmsley, 2019). A continuación, se describen algunos de sus usos más comunes en este ámbito:

3.2.1. Búsqueda de caminos

Uno de los avances más tempranos fue la introducción de algoritmos de búsqueda de caminos, que permite a los PNJs navegar por su entorno y encontrar el camino óptimo para llegar al lugar deseado. Así se conseguía un comportamiento más realista y dinámico de los PNJs, ya que ahora podrían esquivar obstáculos y subir o bajar elevaciones (Duong, 2021).

3.2.2. Toma de decisiones

Otro de los primeros avances fue el desarrollo de algoritmos de toma de decisiones, que permiten a los PNJs tomar decisiones basadas en la información disponible y sus objetivos, dando lugar a un comportamiento más realista. Por ejemplo, un PNJ puede decidir atacar al jugador si lo percibe como una amenaza o huir si le queda poca salud (Anderson, 2003).

3.2.3. Generación de contenido procedural

En tiempos más recientes, la IA se ha extendido más allá del comportamiento de los PNJs, empleándose para generar contenido como niveles, misiones y objetos de forma procedural, aumentando así la rejugabilidad (Yannakakis, 2012). Esto haría que en cada partida cierto contenido del juego fuera diferente. Incluso se puede utilizar para crear elementos creativos, como música, arte conceptual, animaciones y voces (Statt, 2019). Cabe destacar que actualmente su utilización en estos ámbitos tiene una connotación mayoritariamente negativa, ya que se considera que estas herramientas están reemplazando el trabajo de muchos artistas, diseñadores y actores.

3.2.4. Adaptación de la dificultad

En un primer momento se diseñó el sencillo algoritmo MiniMax, que es capaz de ganar las partidas de juegos deterministas al planificar cada uno de los posibles movimientos después de cada jugada. Lamentablemente, su

sencillez hace que también sea poco eficiente. En juegos con pocos movimientos, como el tres en raya, se pueden manejar los árboles de movimientos necesarios para su uso, pero si se intenta utilizar MiniMax sin ningún tipo de heurística en un juego como el ajedrez, incluso los supercomputadores más potentes del mundo en 1997 (año en el que Kasparov fue derrotado por una IA) tendrían problemas para ejecutarlo (Helsinki, 2024) (Campbell et al., 2002). Es decir, la escalabilidad de este tipo de algoritmos deterministas es nula.

3.3. Aprendizaje automático en los videojuegos

3.3.1. Frameworks

En las últimas décadas, se ha observado un aumento de las iniciativas relacionadas con el aprendizaje profundo[5] aplicadas a videojuegos. Esto es en parte debido a la creación en 2013 del Arcade Learning Environment (ALE), que proporciona un framework común para el desarrollo de agentes de IA en juegos de Atari 2600 y permite una forma de cuantificar y valorar el funcionamiento de los diferentes algoritmos (Emu, 1995). A raíz de este proyecto, han surgido muchas otras plataformas para probar técnicas de aprendizaje profundo, incluyendo juegos más modernos y con mayor complejidad que los juegos tradicionales de arcade, principalmente Starcraft II Learning Environment y VizDoom (Justesen et al., 2019).

[5]El aprendizaje profundo permite a los modelos computacionales compuestos por múltiples capas de procesamiento aprender representaciones de datos de varios niveles de abstracción. Estos métodos han mejorado drásticamente el estado del arte en el reconocimiento del habla, el reconocimiento visual de objetos y muchos otros dominios como el descubrimiento de fármacos y la genómica. (LeCun et al., 2015).

3.3.2. AlphaGo y AlphaStar

Uno de los ejemplos más populares en la aplicación del aprendizaje profundo es AlphaGo, del equipo DeepMind de Google, que surge de la combinación de técnicas de aprendizaje por refuerzo con enfoques de aprendizaje profundo más tradicionales. Fue el primer programa de ordenador capaz de derrotar a un jugador de Go profesional y a al campeón mundial, convirtiéndose en el mejor jugador de Go hasta entonces (Silver et al., 2016).

Dos años después, este mismo equipo desarrolló AlphaStar, un agente de aprendizaje profundo capaz de derrotar a equipos profesionales en Starcraft II, un juego de estrategia a tiempo real desarrollado por Blizzard Entertainment.

3.3.3. OpenAI Five

De forma paralela al desarrollo de AlphaStar, se creó otro gran proyecto relacionado con los juegos competitivos: OpenAI Five, una IA desarrollada con técnicas de aprendizaje por refuerzo profundo para jugar a Dota 2, un juego multijugador de estrategia por equipos a tiempo real con una gran complejidad. Al igual que AlphaStar, esta IA demostró ser capaz de vencer a la gran mayoría de equipos profesionales de su momento (OpenAI et al., 2019).

Capítulo 4

Recursos

A lo largo del desarrollo del trabajo se hará uso de diversos recursos humanos, materiales e inmateriales. Los siguientes tres puntos intentarán hacer un breve resumen de estos, estando suscritos a la variabilidad del proyecto.

4.1. Recursos Software

- **Sistema operativo**: Windows 11.
- **Entorno de programación y desarrollo**: Microsoft Visual Studio Code y JetBrains Rider.
- **Entorno de creación del videojuego**: Unreal Engine 5.
- **Editor de documentación**: LaTeX .
- **Lenguajes de programación**: C++ y Python.
- **Librerías de aprendizaje por refuerzo**: Stable Baselines 3, Gymnasium y Optuna.

4.2. Recursos Hardware

Ordenador personal con las siguientes características:

- **Procesador**: AMD Ryzen 7 5800X.
- **Tarjeta gráfica**: NVidia RTX 3070 8GB.
- **Memoria RAM**: 32GB DDR4.

4.3. Recursos Humanos

- **Autor**: Francisco David Castejón Soto.
- **Codirector**: Manuel Jesús Marín Jiménez.
- **Codirector**: Javier Sánchez Monedero.

Capítulo 5

Restricciones

Durante la etapa de diseño y planificación del proyecto, se realizarán una serie de elecciones que condicionarán el desarrollo del mismo. Estas elecciones se basarán en una serie de restricciones que se han identificado y que se expondrán a continuación.

5.1. La elección del motor

En la actualidad hay docenas de motores gráficos disponibles para desarrollar videojuegos. En 2019, un estudio de la revista "Game Developer" mostró que los dos motores gráficos más utilizados eran Unreal Engine y Unity, con un 26 % y un 13 % de uso respectivamente (Toftedahl, 2019).

Unity tiene una curva de aprendizaje menos pronunciada, está pensado para tener unos gráficos más estilizados y se suele utilizar en proyectos de menor envergadura debido a su gran facilidad de prototipado rápido. Por otro lado, Unreal Engine tiene una curva de aprendizaje más inclinada, genera unos gráficos más realistas por defecto, posee un potente sistema de "scripting" (Blueprints) y se suele utilizar en proyectos de gran calibre, como videojuegos AAA.

Pero la mayor diferencia entre ambos motores gráficos en cuanto al apren-

dizaje con refuerzo radica en que Unity cuenta con una librería diseñada expresamente para ello, "ML-Agents", la cual permite entrenar una IA directamente en el motor gráfico. Esta biblioteca, creada por el mismo equipo encargado del desarrollo del motor, hace que Unity sea la elección más obvia para cualquier proyecto que requiera de este tipo de técnicas. Por ejemplo, en el trabajo de fin de master "Estudio de agentes inteligentes aplicando aprendizaje por refuerzo profundo en Unity" realizado por Jorge Zafra Palma en 2022, se utilizó ML-Agents para entrenar una IA en un videojuego de Unity por medio de redes convolucionales profundas[6]. El uso de este tipo de redes neuronales y otras características de ML-Agents, como la vectorización de entornos para entrenar a varios agentes a la vez, hicieron posible el TFM anteriormente mencionado así como otros muchos estudios.

Dadas las claras ventajas de Unity en cuanto a la facilidad de uso y la disponibilidad de herramientas para el aprendizaje por refuerzo, el lector podría preguntarse por qué no se ha optado por utilizar este motor gráfico en el presente trabajo. La elección de Unreal Engine es una decisión consciente que se ha tomado por varias razones.

5.1.1. Mayor aplicabilidad en la industria

La razón de peso vuelve a los porcentajes de uso de los motores gráficos. Ya en 2019, Unreal Engine era el motor gráfico más utilizado en la plataforma Steam[7] y, con la salida de su quinta versión en 2022, muchos estudios de videojuegos decidieron utilizar UE5 para sus futuros proyectos (Obedkov, 2022). Hay varias razones para ello, como la increíble versatilidad y calidad gráfica del motor, o la relativamente baja tasa de regalías que cobra Epic

[6] Una red neuronal convolucional es una clase de red neuronal artificial que se aplica sobre todo al análisis de imágenes. Su característica distintiva es el uso de una operación matemática llamada "convolución". (Wikipedia, 2023b).

[7] Steam es la mayor plataforma de distribución digital de juegos para PC, con una cuota de mercado estimada del 75 % en 2013 según IHS Screen Digest. (Wikipedia, 2024).

Games por su uso. Pero una de las razones más importantes por las que tantos estudios de videojuegos eligen Unreal Engine en lugar de un motor propio es la facilidad de encontrar profesionales cualificados en el mercado laboral. Aprender a utilizar un motor de videojuegos a alto nivel es una tarea que podría llevar meses incluso para los equipos más experimentados, por lo que al hacer uso de Unreal Engine, estos estudios se aseguran tener acceso a la mayor bolsa de talento posible del mercado.

Es por esto que, teniendo en cuenta que parte del propósito de este Trabajo de Fin de Grado se trata de ser una carta de presentación al mercado laboral, la elección de Unreal Engine es la más acertada. Aunque las alternativas a ML-Agents no estén tan desarrolladas, la experiencia de haber usado UE en un proyecto real es un activo de increíble valor, que impulsa positivamente este objetivo en concreto del proyecto.

5.1.2. Proyecto base estable y extendible

Si bien Unity tiene una biblioteca que facilita el aprendizaje por refuerzo, Unreal Engine cuenta con un proyecto que facilita la creación del entorno de aprendizaje. Lyra es un videojuego de código abierto creado por Epic Games para demostrar las capacidades de Unreal Engine en el entorno de los juegos de disparos multijugador. Aunque para este proyecto no es necesario ese componente multijugador (más detalles en el Capítulo 7), desde un principio se pretendía que el entorno de aprendizaje fuera un juego de disparos (debido principalmente a la inspiración de Resident Evil 4). Por lo tanto, tener disponible una plantilla que modificar y adaptar a las necesidades del proyecto confiere una ventaja que no se puede despreciar.

Figura 5.1: Lyra, juego de disparos multijugador

Fuente: Epic Games

5.2. Ausencia de una biblioteca integrada de entrenamiento

Como se ha comentado anteriormente, en este trabajo se usarán diferentes herramientas para entrenar la IA y mostrar su comportamiento. Herramientas como Stable Baselines, Gymnasium o MindMaker son fundamentales para el desarrollo del proyecto. Sin embargo, la falta de una biblioteca de aprendizaje por refuerzo en Unreal Engine constituye una restricción importante. Por ejemplo, ML-Agents cuenta con sus propias implementaciones de los algoritmos más comunes en aprendizaje por refuerzo; en este proyecto, esta parte es suplida por Stable Baselines 3. Por otro lado, la integración de UE con Python está mucho menos desarrollada que la de Unity, lo que complica el proceso de entrenamiento y visualización de la IA. Es ahí donde entra MindMaker, un plugin que proporciona un ejemplo de cómo conectar Unreal Engine con Python. Es importante la distinción de "ejemplo", ya que quien realmente establece dicha conexión es una librería incluida en MindMaker

llamada SocketIO. Esta subbiblioteca es muy potente, pero no está pensada para la paralelización necesaria en el entrenamiento de este tipo de IAs, por lo que se tendrá que hacer uso de otras herramientas como Gymnasium para poder entrenar la IA de forma eficiente fuera de UE.

La elección de Unreal Engine supone un reto, pero también proporciona una oportunidad de aprender a sortear las limitaciones de un motor gráfico que, aunque no está pensado para el aprendizaje por refuerzo, puede ser capaz de albergar un proyecto de este tipo.

En cualquier caso, y a modo de mirada al futuro, existe un plugin experimental que Epic Games está desarrollando para solventar este problema. Se llama "Learning Agents" y se podría decir que es la respuesta de Epic Games a ML-Agents. Aunque aún no esté listo para su uso en un proyecto como este, es una muestra de que la empresa está interesada en el aprendizaje por refuerzo y que, en un futuro, Unreal Engine podría ser una herramienta tan válida como Unity para este tipo de proyectos.

Parte II

Preparación de los experimentos

Capítulo 6

Modelos de aprendizaje

En este primer capítulo de la segunda parte del manual técnico se presentará el camino tomado para la elección de los algoritmos de aprendizaje por refuerzo seleccionados. Además, también se explicará la abstracción necesaria para adaptar el entorno de Python al de Unreal Engine.

6.1. Un algoritmo adecuado

Tras la explicación de los puntos más importantes del aprendizaje por refuerzo en la Sección 3.1, esta sección se centrará en los tipos de algoritmos existentes con el objetivo de elegir los más adecuados para este proyecto.

El objetivo del aprendizaje por refuerzo es generar una política que maximice la recompensa esperada cuando el agente actúe de acuerdo con ella (OpenAI, 2018a). Para ello, hay fundamentalmente dos tipos de algoritmos: los que tienen acceso a un modelo del entorno[8] ("Model-Based") y los que no lo necesitan ("Model-Free"). Es decir, hay algoritmos que necesitan saber lo más fielmente posible cómo se comporta el entorno y algoritmos que

[8]Con "modelo del entorno" no se está haciendo referencia al archivo que representa a la IA o agente, sino a una función que predice los estados a los que transiciona el entorno y sus recompensas.

sólo necesitan la interacción con el entorno.

Tener acceso a dicho modelo genera una gran ventaja, pues ofrece la posibilidad de calcular los posibles futuros estados de este y consecuentemente planificar sus acciones de forma más eficiente. De esta forma consiguen generar políticas que maximizan la recompensa esperada con un menor número de muestras necesarias en su entrenamiento. Sin embargo, la mayoría de los entornos del mundo real son demasiado complejos para ser modelados con precisión. Hay demasiadas variables a tener en cuenta, lo que acaba degenerando en un sesgo en la política generada por el modelo. Esa es la razón por la que los algoritmos Model-Based son menos populares actualmente (OpenAI, 2018b).

Por su parte, los algoritmos Model-Free son más simples de implementar y ajustar, pero requieren de muchas muestras (a veces millones de interacciones) para aprender algo útil. Por eso, la mayoría de los éxitos en el aprendizaje por refuerzo se han logrado en juegos o en simulación virtual de entornos (E. d. StableBaselines3, 2021), donde se pueden simular un número ilimitado de episodios.

Otra manera de clasificar a los algoritmos es en función de lo que aprenden o de lo que ajustan. En la Figura 6.1 [9] se muestra una taxonomía de diferentes algoritmos de aprendizaje por refuerzo. Como se aprecia en la imagen, hay tres tipos adicionales dentro de los Model-Free:

- Policy-Based: optimizan los parámetros de la política de forma directa aplicando un ascenso de gradiente o de forma indirecta mediante la maximización de aproximaciones locales de la recompensa esperada. La mayoría de estos algoritmos son "on-policy", lo que quiere decir que

[9] Se recomienda encarecidamente al lector indagar más en la fuente de esta imagen si este quisiera saber más acerca del tema. La página web de documentación "Spinning Up" de OpenAI es un fuente detallada, explicativa y realmente útil de todo lo relativo al aprendizaje por refuerzo y sin ella hubiera sido mucho más complicado explicar esta sección adecuadamente (OpenAI, 2018b).

sólo usan datos recogidos por la política más reciente, lo que los hace unos métodos muy estables.

- Q-Learning: estiman una función acción-valor basada en las ecuaciones de Bellman[10] En el mayor número de los casos, esta optimización se realiza de manera "off-policy". De esta forma se pueden usar datos recogidos en cualquier punto del entrenamiento, lo que les permite tener una alta eficiencia de muestras al entrenar.

- Interpolación de ambas: intentan combinar lo mejor de ambos mundos, implementando los puntos fuertes de cada uno y minimizando sus debilidades, pero suelen ser más complejos de entender. Estos serían los algoritmos DDPG, TD3 y SAC de la Figura.

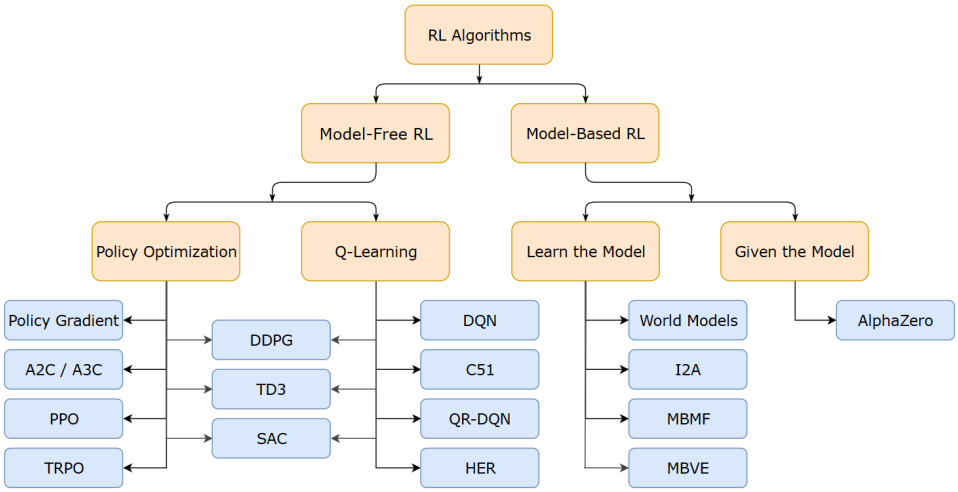

Figura 6.1: Una taxonomía de los algoritmos de aprendizaje por refuerzo
Fuente: OpenAI (OpenAI, 2018b)

De entre todos los algoritmos que se muestran en la figura, Stable Baselines 3 sólo implementa un subconjunto de ellos de forma oficial, que son:

[10]La idea básica de las ecuaciones de Bellman es la siguiente: El valor de tu punto de partida es la recompensa que esperas obtener por estar allí, más el valor de dondequiera que aterrices después (GeeksforGeeks, 2021).

A2C, DDPG, DQN, HER, PPO, SAC y TD3. A la hora de elegir el algoritmo o algoritmos adecuados para este proyecto, se deben tener principalmente en cuenta dos factores: el espacio de acción del entorno y la eficiencia de muestras requerida por el algoritmo.

En la Figura 6.2 se muestra un diagrama de flujo que ayuda a elegir el algoritmo adecuado en función de estos dos factores. En este caso, el espacio de acción es discreto, ya que el agente puede moverse en 4 direcciones y disparar. Al utilizar un entorno virtual como campo de pruebas, no es necesario que el algoritmo sea muy eficiente en este aspecto, ya que siempre se pueden simular más episodios. Por lo tanto, los algoritmos PPO y A2C parecen los más apropiados, ya que son relativamente sencillos de implementar y permiten ser intercambiados entre sí mediante pequeñas modificaciones de código. Es necesario mencionar que, aunque en el diagrama se muestran como algoritmos que están pensados para diferentes tipos de espacios de acción, Stable Baselines 3 permite que ambos usen tanto espacios discretos como continuos (StableBaselines3, 2021a) (StableBaselines3, 2021b).

Tanto PPO como A2C pertenecen a la clasificación de tipo Policy-Based, lo que significa que siguen un método "on-policy", brindándole una mayor estabilidad durante su entrenamiento. Sin embargo, A2C realiza un ascenso de gradiente para maximizar directamente el rendimiento, mientras que PPO actualiza indirectamente la política para maximizar su resultado. Además, PPO emplea una región de confianza para evitar actualizaciones demasiado grandes en su política (OpenAI, 2018b).

6.2. La separación entre el entorno y el agente

En el Capítulo 5 se menciona que el agente se entrenará únicamente en un entorno de Gymnasium y luego se usará Unreal Engine para mostrar los resultados del entrenamiento de forma visual. Esta es una decisión que, impulsada por las limitaciones de las herramientas con las que se cuenta,

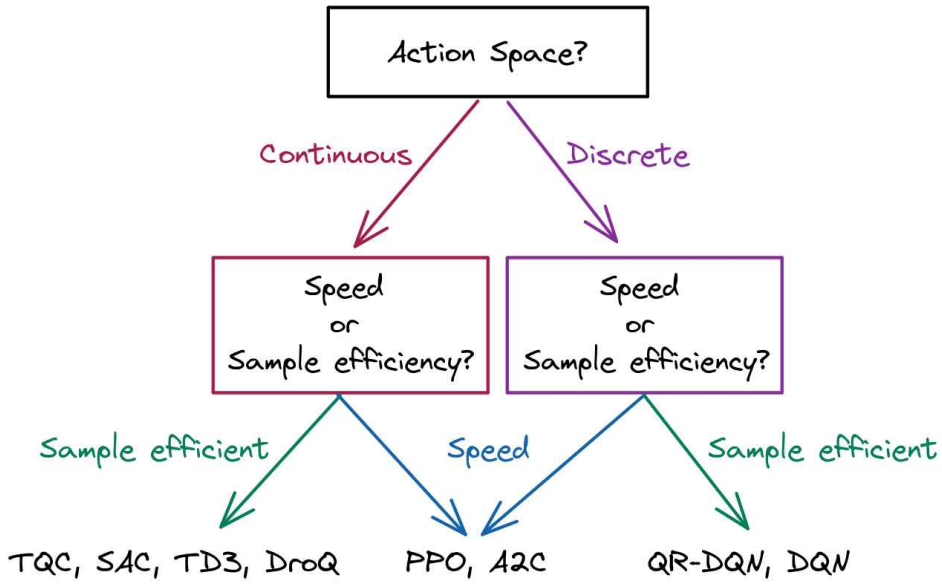

Figura 6.2: Cómo elegir el algoritmo de aprendizaje por refuerzo adecuado
Fuente: Antonin Raffin (E. d. StableBaselines3, 2021)

mejora la eficiencia del entrenamiento de la IA. Sin embargo, esto genera un problema adicional: la necesidad de recrear un mismo entorno usando dos herramientas completamente distintas.

Por un lado está Gymnasium, que permite crear diferentes espacios de observación y de acción a partir de diferentes espacios fundamentales. Estos son algunos de ellos:

- Discreto: un conjunto de valores discretos, como -1, 0, 1.
- Box: un espacio n-dimensional de valores continuos, como [0, 1]
- Multibinario: un espacio de valores binarios, como 101 o 111.
- Diccionario: un conjunto de varios tipos de espacios de observación, ordenados como un diccionaro de Python.

Por otro lado está Unreal Engine, que permite crear entornos 2D o 3D y mostrarlos a través de una pantalla como si de una grabación de vídeo se

tratase. En sentido estricto, UE tendría que ser capaz de emular el espacio de observación en el que se entrenará el agente en Gymnasium (para que el agente entienda los estímulos), a la vez que muestra el resultado de sus acciones por pantalla (para que el usuario humano entienda lo que está ocurriendo). Esto se puede conseguir mediante una abstracción del espacio de observación del entorno de Gymnasium en el entorno de Unreal Engine.

6.2.1. Proyección en perspectiva

La abstracción haría uso de una proyección en perspectiva para generar un sistema cartesiano, como si de una cuadrícula de objetivos se tratase. El modelo se entrenaría con esta cuadrícula y, una vez completado el entrenamiento, se proyectaría en el motor gráfico. En la Figura 6.3 se muestra un ejemplo de cómo se vería la proyección en perspectiva.

Como se puede apreciar en la figura, el objetivo (un maniquí de tiro en 3D como el de Resident Evil 4) se va abstrayendo poco a poco hasta transformarse en un punto en un espacio 2D. Empezando desde arriba, a través de una cámara virtual en el mundo de UE se consigue que dicho mundo en 3D se convierta en una imagen en 2D. Esta imagen es lo que el usuario humano vería en la pantalla. Así, una coordenada en el mundo 3D, como el centro del maniquí, se convertiría en un punto en nuestra pantalla. Eliminando la tercera dimensión, es decir la profundidad, se consigue generar un espacio de observación que el agente pueda entender. Por ello, se creará un espacio Box de dos dimensiones, que es el espacio de observación que será usado en el entrenamiento de la IA.

Es cierto que si al agente no le vamos a pasar una imagen directa del juego como input, no es necesario utilizar la cámara; simplemente se le podrían pasar las coordenadas de los objetivos usando otra abstracción distinta. Sin embargo, hay una mecánica importante dentro de los juegos de disparos actuales: apuntar con la cámara. En la Figura 6.4 se muestra un ejemplo de

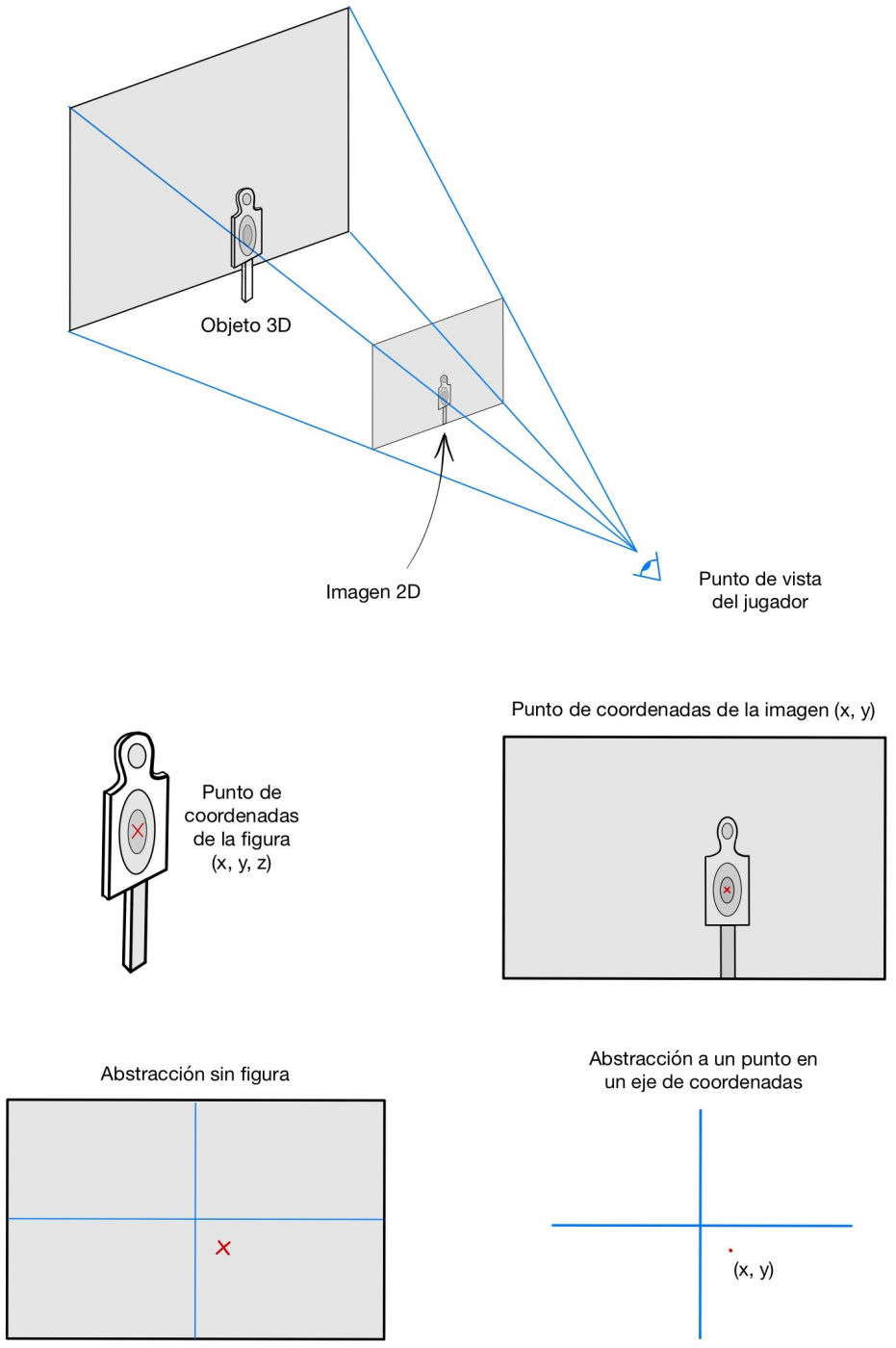

Figura 6.3: Diseño de la proyección en perspectiva

cómo se apunta en Lyra. Aunque sea lógico pensar que la bala sale del arma y llega hasta el objetivo, la realidad es que esto es sólo un efecto visual. Lo que ocurre realmente es que el jugador (o la IA en este caso) apunta con la cámara al maniquí, haciendo que este se encuentre en el centro de la pantalla (el punto (0, 0) en la abstracción). Esa es la verdadera forma de apuntar a un objetivo: una línea imaginaria que va desde la cámara en el mundo hasta el infinito, siguiendo la dirección a la cual esta apunta. Por lo tanto, la cámara es un elemento fundamental en el juego y, por ende, necesaria para la abstracción.

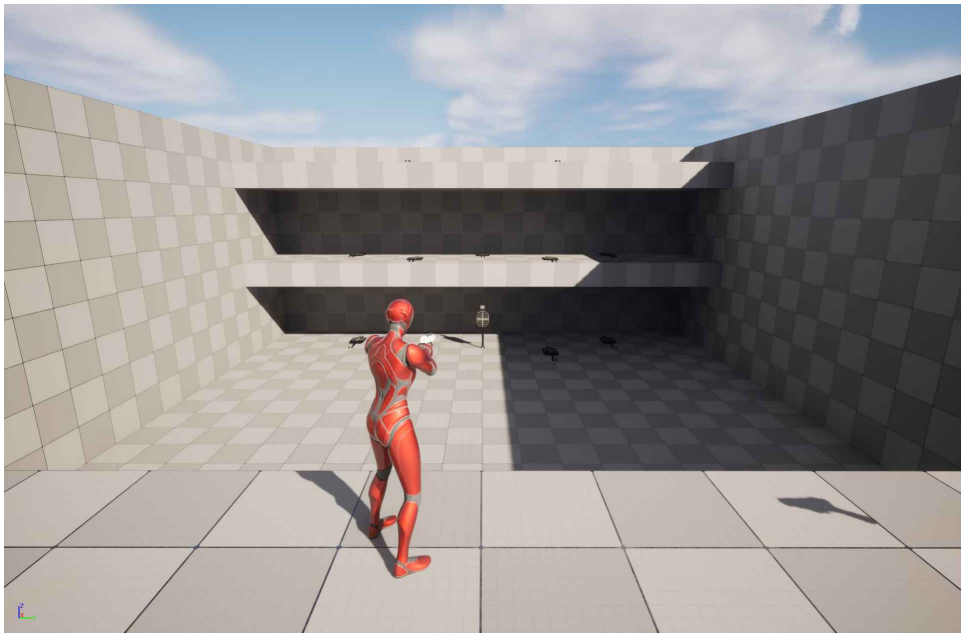

Figura 6.4: Ejemplo del campo de tiro

De esta forma, la toma de decisiones de la IA se realizaría en base a la proyección de la figura. El agente simplemente observaría unos puntos en un espacio 2D y "movería" la base de los ejes para que coincidieran con el punto al que quiere apuntar. De esta forma, el modelo no tendría que preocuparse de la rotación de la cámara, ya que el sistema de proyección de la cámara se encargaría de ello. Sí que tendría que encargarse de saber cuándo disparar,

que sería cuando el punto estuviera en el centro de la pantalla. Es decir, cuando el maniquí estuviera en el punto (0, 0) de la abstracción.

Si imaginamos que un juego se actualiza 30 veces por segundo, viendo por tanto 30 imágenes o fotogramas (*frames*) por segundo en pantalla, este sería el aspecto que tendría la toma de decisiones de la IA imagen a imagen.

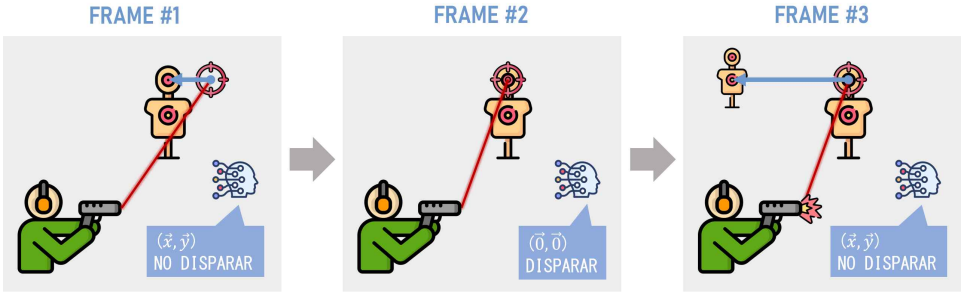

Figura 6.5: Toma de acciones de la IA

- Frame #1: la IA "ve" que la pistola no está apuntando a ningún objetivo, por lo que su acción es mover la cámara hacia la izquierda y no disparar por el momento.

- Frame #2: una vez se ha ejecutado el movimiento, se vuelve a actualizar a la IA con la situación actual y decide que ahora sí es un buen momento para disparar, pero sin mover la cámara para no alterar la posición del arma.

- Frame #3: tras haber disparado, ese objetivo ya no proporcionará más puntos a la IA, por lo que se le muestra el siguiente objetivo y esta decide mover la cámara hacia él.

Capítulo 7

Software desarrollado

En este capítulo, se describe el método de trabajo general que se ha tomado durante el desarrollo del proyecto. Este método comienza con un pequeño prototipo de cada una de las partes el cual más tarde se enriquece con el contenido final.

Además, se mencionan los desafíos de trabajar con Lyra y la creación de un entorno de tiro para la IA y los jugadores humanos, incluyendo la modificación de la interfaz, la creación de un maniquí interactivo y la implementación de la comunicación entre UE y Python.

7.1. El agente y el entorno textual

7.1.1. Prototipado

Aunque el diseño de la arquitectura del agente y el entorno ya estaban decididos cuando se empezó a desarrollar el software de aprendizaje por refuerzo, se decidió realizar un prototipo más sencillo del problema para entender más a fondo las herramientas que se iban a emplear en el agente final. Durante la lectura de la documentación de Gymnasium (Farama, 2022), se encontró una página que explicaba cómo crear un entorno personalizado. En ella, se explicaba que un entorno debía tener al menos 2 métodos en

su clase: `reset()` y `step()`. El primero se encarga de reiniciar el entorno y devolver el estado inicial, mientras que el segundo recibe una acción y devuelve el estado siguiente, la recompensa, si el episodio ha terminado, además de alguna otra información extra que se quiera devolver.

Tras realizar múltiples iteraciones mejorando el código y arreglando errores, el entorno estaba preparado para usarse en el entrenamiento de un agente. Se decidió utilizar Proximal Policy Optimization (PPO) por ser un algoritmo muy versátil. En un primer intento, por mucho tiempo que se dejara a la IA entrenando, nunca llegaba a completar un episodio: no conseguía llegar a la meta. Tras investigar más detenidamente la documentación de Stable Baselines 3, se encontró una biblioteca auxiliar llamada "RL Baselines3 Zoo" creada por Antonin Raffin en 2019 (Raffin, 2019). Esta biblioteca contiene múltiples entornos y algoritmos pre-entrenados, así como *scripts* para entrenar agentes en entornos personalizados. Pero lo que realmente se necesitaba de ella eran sus *scripts* para el ajuste de hiperparámetros mediante Optuna[11].

Una vez el ajuste de hiperparámetros estaba completado, se entrenó un nuevo modelo usando PPO. En este caso, el agente fue capaz de completar múltiples episodios sin problemas. Fue entonces cuando se decidió que el prototipo hacía sido completado y se podía empezar a trabajar en el agente final.

7.1.2. Entorno final en Python

7.1.2.1. El espacio de observación

En el capítulo 6 se explicó la razón por la que el espacio de observación sería de tipo Box. Debido a la proyección en perspectiva, el agente sólo necesita dos coordenadas para saber dónde está el objetivo. Y dado que siempre tiene que llevarlo al mismo sitio (0, 0), no necesita saber hacia dónde está

[11] Optuna es una biblioteca de ajuste de hiperparámetros diseñada específicamente para no depender de ningún marco de trabajo, lo cual la hace muy versátil.

apuntando en todo momento. Esto sin embargo conlleva una limitación: sólo es posible presentarle un objetivo a la vez. Sería teóricamente posible entrenar un agente que aceptase múltiples objetivos, pero se le tendría que añadir más "espacio" al espacio de observación. Utilizando, por ejemplo, un espacio de tipo diccionario, se podrían añadir tantos objetivos como se quisiera. No obstante, este cambio tampoco es perfecto pues el número máximo de objetivos tendría que ser fijo. Es decir, para que el agente aprendiera a manejar un número diferente de objetivos, se tendría que volver a entrenar. No es, por tanto, una solución escalable.

Durante la realización de este TFG no ha sido posible resolver esta limitación, por lo que se ha optado por usar el número de objetivos más extrapolable: uno. Si bien no es lo más adecuado, es suficiente para demostrar que el agente es capaz de aprender a apuntar a un objetivo y saber cuándo tiene que disparar. De todas maneras, si el campo de tiro tuviera más de un objetivo (como es el caso en la Figura 6.4), simplemente se podría resetear el entorno en Python después de acertar a cada objetivo y asignarle un nuevo objetivo desde UE.

7.1.2.2. El espacio de acción

En el prototipo del entorno se utilizó un espacio de acción de tipo Box. Si esto se aplicara al entorno final podría generar muy buenos resultados, ya que permitiría al agente moverse en cualquier dirección e incluso modular la cantidad de movimiento en cada paso. Además, si en su lugar se usara un espacio de tipo diccionario, se podrían añadir otros tipos de espacio, como por ejemplo, un binario que represente la acción de disparo.

Esto en la práctica no es posible, ya que ningún algoritmo de Stable Baselines 3 soporta espacios de acción de tipo diccionario. Es por esa razón que se ha decidido usar un espacio de acción de tipo Discreto, el cual es capaz de manejar múltiples acciones predefinidas. Y esa es precisamente la desven-

taja de este enfoque: las acciones deben estar predefinidas. Si queremos que el agente se pueda mover hacia arriba, abajo, izquierda y derecha, hay que contar con cuatro acciones discretas. Pero si, por ejemplo, quisiéramos que el agente pudiera moverse en cualquier ángulo en un rango de 360 grados, tendríamos que tener 360 acciones discretas. Esto, de nuevo, no es escalable, pero es suficiente para demostrar que el agente es capaz de aprender a moverse y a disparar. Como ya se mencionó en la Sección 2.1, el objetivo de este trabajo es crear un prototipo funcional y aprender acerca del aprendizaje por refuerzo. Por tanto, para este propósito, el espacio de acción de tipo Discreto es suficiente.

Sabiendo entonces las limitaciones del diseño, las acciones que puede tomar el agente en cada paso son las siguientes:

- 1: Moverse hacia derecha.
- 2: Moverse hacia arriba.
- 3: Moverse hacia la izquierda.
- 4: Moverse hacia abajo.
- 5: Disparar.

7.1.2.3. La función de recompensa

La ingeniería de recompensas es una parte muy importante del aprendizaje por refuerzo: una recompensa mal diseñada puede hacer que el agente no aprenda nada o que aprenda algo que no queremos. Pero también es una de las partes más complejas debido a su subjetividad: para cada problema, se podría tener infinitas maneras de decirle al agente que lo que está haciendo está bien o mal.

Por ejemplo, en cierto momento del desarrollo de la función de recompensa, se incluyó una pequeña recompensa positiva simplemente por llevar

el objetivo al centro de la pantalla y mantenerlo ahí. Esto se hizo pensando que motivaría al agente a no mover el objetivo una vez estuviera en el punto de mira. Por supuesto, no acabó funcionando en absoluto así. Como no había ninguna comprobación para ver cuánto tiempo estaba el agente en esa precisa situación, este aprendió que la forma de maximizar la recompensa era mover el objetivo alrededor del centro de la pantalla y nunca disparar. Así, en lugar de recibir una recompensa total de 1,6 tras disparar al objetivo correctamente, podía llegar a conseguir una recompensa de 4999,6. Existe un tope a la cantidad de pasos que puede tener un episodio, ese tope está puesto en 50000 pasos, y la recompensa por quedarse en el centro de la pantalla era de 0,1 por paso. Por tanto, el agente podía llegar a conseguir una recompensa de 4999,6 si se quedaba en el centro de la pantalla durante 49996 pasos. Este es un ejemplo de cómo una recompensa mal diseñada puede hacer que el agente aprenda algo que no queremos.

Dado que el problema a resolver en este proyecto es relativamente sencillo, la función de recompensa también debe serla. Aun así, se han intentado seguir conceptos generales de la ingeniería de recompensas. Por ejemplo, se ha optado por normalizar las recompensas en un rango [-1, 1], no dar valores numéricos demasiado dispares entre sí y no penalizar al agente por tomar acciones que fomentan la exploración. Pero al fin y al cabo, es el ingeniero quien tiene que decidir la función de recompensa (Eschmann, 2021), por lo que así se ha diseñado:

- Si el agente ha disparado y le ha dado al objetivo [Recompensa = 1]: dar una recompensa de 1 al agente usualmente significa que ha terminado el episodio satisfactoriamente.

- Si el agente ha disparado y no le ha dado al objetivo [Recompensa = 0]: se le podría dar una recompensa negativa por no acertar, pero entonces se estaría penalizando al agente por explorar. Teniendo en cuenta que el

agente debe obligatoriamente disparar al objetivo en algún momento, se ha decidido no penalizar al agente por intentarlo.

- Si el agente se ha acercado al objetivo en este paso [Recompensa = 0.1]: se recompensa positivamente al agente por hacer progresos.

- Si el agente no se ha acercado al objetivo en este paso [Recompensa = -0.1]: se recompensa negativamente al agente por no hacer progresos. Esta regla además fomenta que el episodio acabe lo antes posible.

- Si el agente ha llegado al objetivo por primera vez [Recompensa = 0.5]: se le da una gran recompensa al agente por apuntar correctamente al objetivo.

- Si el agente ha llegado al objetivo y su siguiente acción no es disparar [Recompensa = -0.5]: el agente es penalizado con una gran recompensa negativa por no acabar el episodio cuando estaba a un paso de hacerlo.

7.2. El entorno y el agente virtual

7.2.1. Prototipado

La fase de prototipado en Unreal Engine fue una de las primeras fases del TFG. En ella, se revisó y analizó el proyecto de ejemplo que incluye MindMaker y se realizó asimismo un pequeño experimento con los conocimientos adquiridos hasta entonces.

El proyecto que incluye MindMaker con su documentación fue de gran ayuda, ya que representa un ejemplo funcional de cómo es el ciclo completo de entrenamiento de un agente en UE y Python. El código que incluye dentro de UE está escrito mediante Blueprints, lo cual lo hace más sencillo de entender para alguien sin experiencia previa con el motor. Por otro lado, el *script* de Python, aunque algo anticuado y desorganizado, proporciona ejemplos de

muchas funciones útiles que se han acabado utilizando en el *script* final de este trabajo.

Tras completar el estudio de MindMaker, se desarrolló un entorno sencillo en el que el objetivo era que un robot anduviera hasta un punto en el mapa. Lamentablemente el agente no fue capaz de aprender a llegar a su destino incluso desactivando la visualización de la cámara y las animaciones del robot para que el entrenamiento fuera más rápido. Este fue el momento en el que quedó claro que ese flujo de trabajo no iba a dar resultados y se tendría que separar el entrenamiento de UE.

En concreto, el número de iteraciones por segundo durante este entrenamiento estuvo alrededor de las 15 it/s. Es decir, que cada segundo se realizaban alrededor de 15 pasos en el entorno. Este número queda muy por debajo de las 4000 it/s que se consiguieron al entrenar un agente únicamente en Python usando 16 entornos al mismo tiempo. El simple hecho de dejar de conectar UE y Python mediante un servidor local ya supuso una mejora sustancial en la velocidad de entrenamiento, ya que así el algoritmo no tenía que esperar a que UE procesara cada paso y enviara sus resultados antes de poder seguir adelante. Pero sin duda alguna fue la paralelización lo que más influyó en la mejora de la velocidad de entrenamiento.

7.2.2. Lyra como plantilla

Trabajar con Lyra indudablemente ha sido un reto: en ocasiones enriquecedor y en otras frustrante. Cuando se empezó con esta fase del proyecto, ya se había completado un curso en UE y se había utilizado el motor durante toda la fase del prototipo, por lo que empezar a usar Lyra parecía ser el siguiente paso lógico. Sin embargo, Lyra estaba (y hasta cierto punto sigue estando) muy por encima del nivel de conocimiento que se tenía en ese momento. Esta extrema dificultad para entender el funcionamiento de Lyra se debe a varias razones:

- Lyra no sigue el método usual de desarrollo en UE, o al menos el método que se suele enseñar a principiantes. Dado que es un proyecto diseñado por veteranos de la industria, utiliza abstracciones y módulos no del todo frecuentes.

- Lyra es un proyecto multijugador, lo que complica todo su planteamiento. Aunque en este TFG no se haya utilizado la funcionalidad multijugador, la estructura del proyecto sigue siendo la misma. Por otro lado, separar sólo la parte útil para este proyecto habría sido una tarea demasiado complicada.

- La documentación oficial de UE no es muy detallada. A menudo, buscar una función de código en la documentación de Unreal Engine sólo aporta una brevísima descripción de la función y en qué fichero de cabecera está definida. Por lo que o bien es necesario buscar ejemplos de su uso en el código fuente de UE o bien buscar en foros y comunidades de desarrolladores y tener la suerte de que alguien haya explicado su funcionamiento.

En resumen, la idea de utilizar Lyra como plataforma de aprendizaje cuando se tenía tan poca experiencia con Unreal Engine ha sido como intentar comprender el funcionamiento del motor de combustión de un coche deportivo cuando mi experiencia como mecánico se limitaba a haber cambiado una rueda. Afortunadamente, existe un desarrollador llamado Xist, cuya documentación[12] sobre Lyra ha sido de gran ayuda.

7.2.3. Entorno final en UE

El entorno final representa un campo de tiro para que un jugador humano o una IA puedan practicar su puntería. Como se ha mencionado en la sección anterior, gran parte del trabajo se ha centrado en modificar Lyra para que

[12]Su página web de documentación es: x157

se ajuste a las necesidades de este proyecto. Pero esa no ha sido la única tarea realizada. Además de preparar el entorno jugable, era necesario crear la conexión entre UE y Python usando la biblioteca de SocketIO.

En la creación de un videojuego, uno de los primeros pasos es el diseño de niveles. Aunque el juego vaya a ser simple y no tenga ninguna historia, es muy importante que el diseño sea robusto y esté bien definido. Es por esa misma razón que se ha escogido un nivel probado y con gran éxito. En el aclamado juego de Capcom, Resident Evil 4, existe una galería de tiro en la que Leon, el protagonista, dispara a muñecos de madera que representan zombies. En las Figuras 7.1 y 7.2 se muestra un ejemplo de estas salas, las cuales van cambiando a lo largo del juego con diferentes escenarios y enemigos.

Figura 7.1: Diseño inspirador de la galería de tiro
Fuente: Resident Evil 4: HD Remaster

La primera de las diferencias en la forma de trabajar con UE por defecto frente a Lyra vino con la creación de un nuevo modo de juego. Para esto, se siguió la guía de Epic Games[13] en conjunto con la de Xist para crear una nueva instancia donde crear el modo en sí.

[13]Esta guía en concreto: Inicio en Lyra

Figura 7.2: Cámara de disparo y objetivos

Fuente: Resident Evil 4: HD Remaster

Tras esto, se realizaron diversas modificaciones a diferentes secciones del código de Lyra. Por ejemplo, se ajustó la precisión de las armas para que fuera perfecta. Hay una mecánica muy usual en los juegos de disparos que hace que cuanto más se dispare un arma, más complicado sea que los disparos de esta vayan justo a donde se apunta. Esta mecánica se llama "spread" y se utiliza para equilibrar las armas en los juegos. En este caso, se ha decidido que podría confundir al agente al no estar presente en el entorno de Python, por lo que se ha decidido eliminarla.

Asimismo, se ajustaron el número de puntos para terminar una partida, al igual que el tiempo máximo que tiene el jugador o la IA para completar la ronda. Igualmente, el contador de puntos se rediseñó para que sólo mostrase a un equipo, ya que en este caso no había dos equipos enfrentándose. En la Figura 7.3 se puede ver la comparación entre la interfaz original y la interfaz modificada.

La siguiente tarea fue la de crear un maniquí que pudiera recibir daño y que mostrara los estados de vida y muerte. Para ello, se descargó y arregló

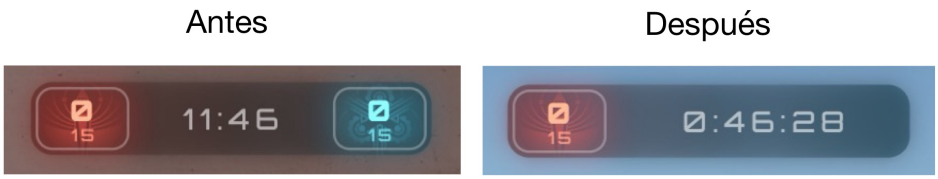

Figura 7.3: Comparación entre las dos interfaces

un diseño 3D de un maniquí y luego se le añadió funcionalidad. Aunque en los anteriores casos se usó únicamente código de *blueprint*, en este caso se utilizó una mezcla de ambas. Se creó una clase base en C++ que heredaba de la clase base de los personajes de Lyra y se le añadió la funcionalidad de recibir daño. Tras esto, se creó un *blueprint* que heredaba de esta clase y se le añadió la funcionalidad de tumbarse al recibir daño y de levantarse al ser revivido. En la Figura 7.4 se puede ver el maniquí en cuestión, mientras que en la 7.5 se muestra el código *blueprint* que hace que este gire.

Figura 7.4: Maniquí

Tras esto, se creó la funcionalidad que levantaba un solo maniquí al mismo

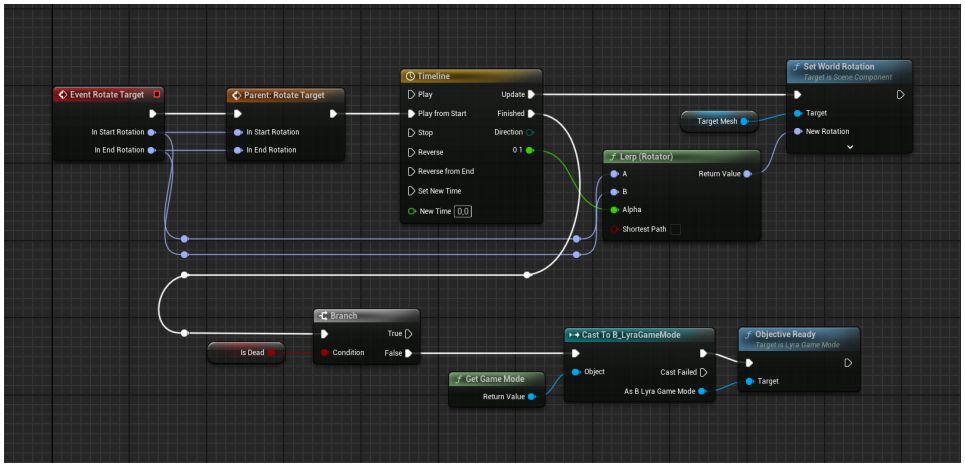

Figura 7.5: Blueprint que hace girar al maniquí

tiempo de entre todos los disponibles y le mandaba a Python la información de su posición en pantalla. Esta parte, escrita únicamente en C++, requirió revisar el código fuente de SocketIO para entender cómo enviar mensajes al servidor de Python. Esto es así ya que en el ejemplo de Lyra sólo se usaban *blueprints* para enviar o recibir mensajes, así que en este caso se optó por hacerlo en C++ para entender más a fondo el funcionamiento de la conexión entre UE y Python.

Por último, se hizo el propio mapa de la galería de tiro. Este cuenta con varios niveles con el objetivo de comprobar las capacidades del agente de apuntar no sólo hacia la izquierda o derecha, sino también hacia arriba y abajo. Para el jugador humano se añadió de forma adicional un lugar donde cambiar las armas y un botón para reiniciar la ronda. En la Figura 7.6 se puede ver el mapa final.

7.3. Resumen de tecnologías

A modo de recapitulación de todas las bibliotecas, programas y tecnologías que se han utilizando durante este proyecto, se ha creado la Figura 7.7. En ella se muestran dichas herramientas, así como la forma en la que inter-

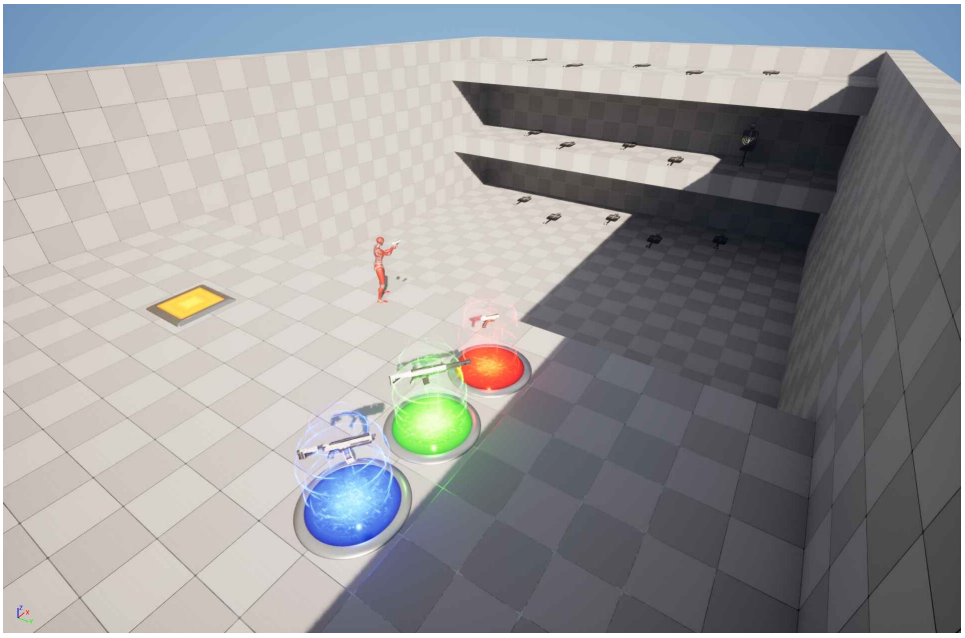

Figura 7.6: Mapa final del juego

actúan entre ellas. Las más importantes son Unreal Engine y Python, que se encargan de mostrar el entorno por pantalla y dirigir al agente, respectivamente. Entre ellas está SocketIO, que está dentro del *plugin* MindMaker y se encarga de la comunicación entre UE y Python. Este Trabajo de Fin de Grado utiliza Lyra, desarrollado en UE, como base para crear un entorno de juego. A su vez, se ha usado Gymnasium para diseñar el entorno en Python, donde se entrenan los agentes. En la parte inferior del diagrama se muestran Stable Baselines 3 y Optuna, los cuales se usan en conjunto. Los algoritmos de SB3 entrenan al agente en el entorno diseñado en Gymnasium y Optuna utiliza esta capacidad de entrenamiento para buscar los hiperparámetros que mejoren en mayor medida la recompensa que consigue dicho agente.

Figura 7.7: Diagrama de las tecnologías que se han utilizado en el proyecto

Capítulo 8

Métricas de evaluación

Stable Baselines 3 (SB3) proporciona una serie de métricas de evaluación que se registran durante el entrenamiento y se pueden utilizar para evaluar el rendimiento del agente hasta el momento (StableBaselines3, 2021c). Además de las métricas esperadas, como un contador de iteraciones, pasos y tiempo transcurrido, SB3 también genera otro tipo de datos que pueden ser de utilidad. Algunos de los más importantes son:

- Longitud media del episodio (`ep_len_mean`): la longitud media de los episodios durante el entrenamiento. Indica la rapidez del agente en resolver el entorno.

- Recompensa media del episodio (`ep_rew_mean`): la recompensa media de los episodios durante el entrenamiento. Su valor depende completamente del entorno, pero teniendo en cuenta la función de recompensa, se puede estimar la eficacia del agente.

- Aproximación de KL (`approx_kl`): la divergencia de Kullback-Leibler aproximada entre las políticas antiguas y nuevas. Un valor alto puede indicar que la política está cambiando demasiado rápido. Este es un indicador clave y exclusivo del algoritmo PPO.

- Varianza explicada (`explained_variance`): representa la precisión con la que la función de valor predice la recompensa, lo que es crucial para una toma de decisiones eficaz. Un valor cercano a 1 indica una buena predicción, mientras que un valor cercano a 0 representa que el modelo no es mejor que simplemente usar la media de la predicciones (enfoque "ingenuo"). Por último, un valor negativo indica que el modelo es peor que utilizar dicho enfoque y por lo tanto tiene un mal rendimiento.

Estas métricas se utilizarán para analizar el rendimiento de los agentes y compararlos entre sí. Esto se verá en el Capítulo 10.

Capítulo 9

Parámetros de los algoritmos

En el aprendizaje por refuerzo profundo tratamos con políticas parametrizadas: políticas cuyos resultados son funciones computables que dependen de un conjunto de parámetros (por ejemplo, los pesos y sesgos de una red neuronal) que podemos ajustar para cambiar el comportamiento a través de algún algoritmo de optimización (OpenAI, 2018a).

A continuación, se describen los parámetros comunes de Proximal Policy Optimization (PPO) y Advantage Actor Critic (A2C) (StableBaselines3, 2021d), así como los exclusivos de cada uno de ellos. En el Capítulo 10, se detallan los valores específicos utilizados en los experimentos.

9.1. Parámetros comunes

- Número de entornos (`n_envs`): cantidad de entornos paralelos que se utilizan para recolectar experiencias de manera simultánea. Aumentar este valor puede mejorar la eficiencia de la recolección de datos, pero depende en gran medida de las capacidades de la máquina donde se ejecute el entrenamiento.

- Número total de pasos en el tiempo (`n_timesteps`): cantidad total de pasos de tiempo que se utilizarán para entrenar el modelo.

- Política (`policy`): tipo de política que se usará. Por ejemplo, 'MlpPolicy' indica que se utilizará una red neuronal perceptrón multicapa (MLP).
- Número de pasos de actualización (`n_steps`): cantidad de pasos entre actualizaciones del modelo. Un valor más alto puede mejorar la estabilidad de la política, pero también aumenta el uso de memoria.
- Factor de descuento para GAE (`gae_lambda`): descuento para el cálculo de la ventaja generalizada (GAE). Un valor más cercano a 1.0 hace que el GAE sea más similar a la ventaja Monte Carlo.
- Factor de descuento (`gamma` γ): factor de descuento para las recompensas futuras. Un valor más cercano a 1.0 considera más las recompensas futuras.
- Coeficiente de entropía (`ent_coef`): se utiliza para regular la exploración. Un valor más alto la fomenta en mayor medida.
- Tasa de aprendizaje (`learning_rate`): tasa de aprendizaje para el optimizador. Un valor más bajo puede mejorar la estabilidad del entrenamiento, pero puede requerir más tiempo para converger.
- Norma máxima del gradiente (`max_grad_norm`): máximo valor de la norma de los gradientes. Se utiliza para la normalización de gradientes, evitando grandes cambios en estos.
- Coeficiente de la función de valor (`vf_coef`): coeficiente para la pérdida de la función de valor. Controla la importancia de la función de valor en la actualización de la política.
- Parámetros adicionales de la política (`policy_kwargs`):
    - Función de activación (`activation_fn`): especifica la función de activación utilizada en las capas de la red neuronal.

- Arquitectura de la red (`net_arch`): define la estructura de la red neuronal, tanto la cantidad de capas ocultas como la cantidad de neuronas por capa.

9.2. Parámetros exclusivos de PPO

- Tamaño del lote (`batch_size`): tamaño del lote de datos utilizado para actualizar el modelo. Un tamaño de lote mayor puede proporcionar estimaciones más estables de los gradientes, pero también requiere más memoria.
- Número de épocas (`n_epochs`): número de épocas para las que se actualiza la política en cada iteración. Más épocas pueden mejorar la convergencia, pero también pueden llevar a sobreajuste.
- Rango de recorte (`clip_range`): rango de recorte para la probabilidad de la política. Ayuda a prevenir cambios drásticos en esta, mejorando la estabilidad del entrenamiento.

9.3. Parámetros exclusivos de A2C

- Uso de RMSProp (`use_rms_prop`): indica si se debe usar el optimizador RMSProp en lugar de Adam.
- Normalización de la ventaja (`normalize_advantage`): si se debe normalizar la ventaja antes de actualizar la política.
- Parámetros adicionales de la política (`policy_kwargs`):
  - Inicialización de desviación estándar (`log_std_init`): valor inicial para la desviación estándar logarítmica de la política.
  - Inicialización ortogonal (`ortho_init`): si se debe usar inicialización ortogonal para los pesos de la red.

Parte III

Experimentación y Resultados

Capítulo 10

Entrenamiento de los agentes

Tras la descripción de los espacios de acción y observación, así como de la función de recompensa del entorno en la Sección 7.1.2, en este capítulo se detallarán los experimentos realizados durante el entrenamiento de los agentes.

Se empezará por mostrar los hiperparámetros finales aplicados a los algoritmos de entrenamiento, así como los efectos que algunos de ellos pueden tener. En secciones posteriores se mostrarán comparaciones entre los propios agentes, pudiéndose ver los puntos fuertes y débiles de cada uno. Por último, se describirán las consecuencias de ampliar las posibilidades de movimiento del agente.

10.1. Acciones por defecto

10.1.1. Entrenamiento del agente con PPO

Aunque el entorno ha ido evolucionando en cada iteración a lo largo del proyecto, siempre se ha utilizado el mismo algoritmo de entrenamiento, PPO por defecto, para evaluar los cambios realizados. Por ello, sus valores óptimos de hiperparámetros han ido cambiando conforme avanzaba el TFG, requiriendo de múltiples búsquedas para encontrar los valores que mejor se

adaptaban a las necesidades del entorno. Los hiperparámetros que se utilizaron para el entrenamiento final del agente con PPO se pueden ver en la siguiente tabla:

Hiperparámetro	**Valor**
Número de entornos	16
Número total de pasos en el tiempo	1×10^6
Política	"MlpPolicy"
Número de pasos de actualización	128
Tamaño del lote	256
Factor de descuento para GAE	0.9
Factor de descuento	0.98
Número de épocas	5
Coeficiente de entropía	0.00017
Tasa de aprendizaje	0.0067
Rango de recorte	0.4
Norma máxima del gradiente	5
Coeficiente de función de valor	0.54
Función de activación	ReLU
Arquitectura de la red	pi=[64], vf=[64]

Tabla 10.1: Hiperparámetros finales usados durante el entrenamiento del agente mediante Proximal Policy Optimization.

En cada una de las rondas de optimización de hiperparámetros, se probaron 15 conjuntos de valores diferentes y se seleccionó el conjunto que ofrecía mejor rendimiento, lo que tardaba 50 minutos. Tras encontrar los hiperparámetros óptimos, se procedió a entrenar el agente con ellos durante un millón de pasos a una velocidad de 4500 interacciones por segundo, lo que tomó aproximadamente 10 minutos. Por lo tanto, el proceso completo de optimi-

zación y entrenamiento del agente PPO por defecto duró aproximadamente una hora en la máquina utilizada.

Como parámetros a destacar se encuentra la función de activación de la red, que en este caso es ReLU, una función de activación muy popular en las redes neuronales profundas debido a su simplicidad computacional y a su capacidad para mitigar el problema del desvanecimiento del gradiente[14] (Krishnamurthy, 2024). Otro aspecto importante es que la arquitectura está formada por una capa oculta de 64 neuronas en ambas redes neuronales. PPO es un algoritmo tipo Actor-Critic, lo que quiere decir que optimiza dos redes neuronales al mismo tiempo. Una de las redes es la que optimiza la política, que es la que decide qué acción tomar, y la otra es la que ajusta la función de valor, que estima la recompensa esperada de un estado.

Las métricas que SB3 proporcionó durante el entrenamiento del agente concuerdan con un progreso estable en el entrenamiento. Como este se realizó en 16 entornos al mismo tiempo y con un número de pasos de actualización de 128, eso quiere decir que cada 2048 pasos se realizaba una actualización de la red neuronal. En la primera actualización con métricas, la segunda, se suelen encontrar valores de longitud media de episodios de varios cientos de pasos, y recompensas medias negativas. Además, en estas fases tan tempranas del entrenamiento se solían ver varianzas explicadas por debajo de 0, lo cual indicaba una elección de acciones prácticamente peor que si fuera aleatoria.

Por último, la aproximación de KL podía tener un valor de 0.1 o más, lo que indica que la política cambia rápidamente. Este valor se va reduciendo a medida que el entrenamiento avanza, llegando a valores de 0.01 o menos en las últimas actualizaciones.

Además de la aproximación de KL, el resto de métricas también se van estabilizando a lo largo del entrenamiento. La longitud media se va redu-

[14]El problema del desvanecimiento del gradiente ocurre cuando los valores de un gradiente son demasiado pequeños y el modelo deja de aprender o tarda demasiado en hacerlo (Donges, 2024).

ciendo hasta alcanzar valores de unas pocas decenas de pasos por episodio. Por el contrario, la recompensa media va aumentando, llegando a los valores positivos típicos de un episodio exitoso (el valor en concreto depende de la función de recompensa). La varianza explicada también se va acercando a 1. Por ejemplo, en la última actualización, la varianza explicada fue de 0.936. Es decir, la función de valor estaba explicando mucho mejor las recompensas que al principio del entrenamiento.

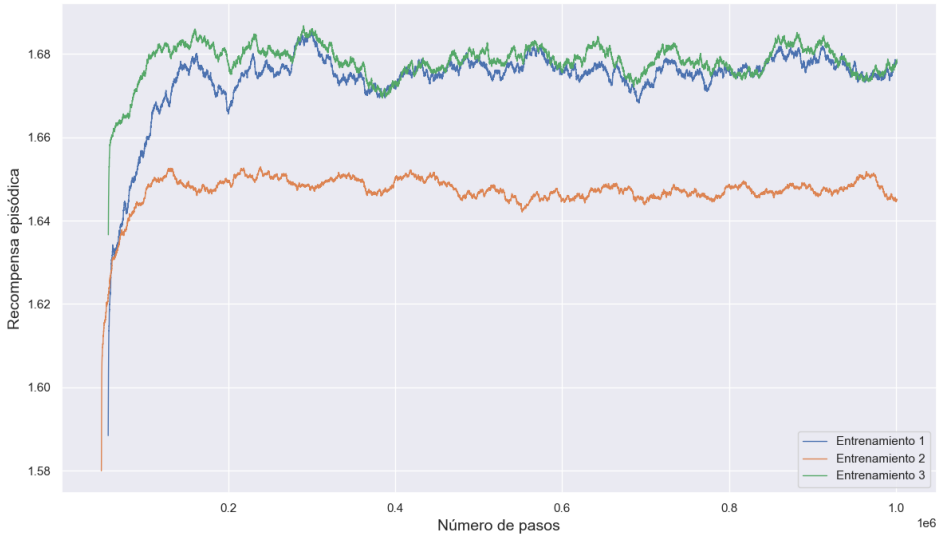

Figura 10.1: Avance de la recompensa conforme a los pasos en PPO

Una forma más visual de apreciar el progreso del entrenamiento es mediante la gráfica de la Figura 10.1. En ella, se pueden ver tres entrenamientos diferentes del agente utilizando el algoritmo PPO. En todos ellos se ven los mismos patrones de progreso, con una recompensa media que aumenta en gran medida durante los primeros 150000 pasos y se estabiliza a partir de ese momento. El primer entrenamiento, en azul, es el que se ha descrito en este apartado. Los otros dos, en naranja y verde, son entrenamientos adicionales con los mismos hiperparámetros, pero con ciertas diferencias en el entorno.

En el entrenamiento en naranja se desactivó la aleatoriedad del tamaño

de la cuadrícula, así como la del porcentaje de movimiento. Esta aleatoriedad es un concepto que se ideó para mejorar la robustez del agente, ya que le obliga a adaptarse a diferentes situaciones. Funciona de la siguiente forma:

- En cada episodio, el tamaño de la cuadrícula se elige aleatoriamente entre 200 y 8000. Esto se puede entender como el tamaño de la pantalla del juego. A modo de ejemplo, una pantalla Full HD tiene un tamaño de 1920x1080 píxeles. Esto quiere decir, que el tamaño de la pantalla (en píxeles) puede ir desde 200 hasta 8000 tanto de forma vertical como horizontal.

- Asimismo, en cada episodio se elige un porcentaje de movimiento aleatorio entre 0.01 y 0.05. Este porcentaje se aplica al tamaño de la cuadrícula, y representa cuántos píxeles se mueve el agente en cada paso. Por ejemplo, si el tamaño de la cuadrícula es de 1000×1000 píxeles, y el porcentaje de movimiento es de 0.01, el agente se moverá 10 píxeles en cada paso.

Esto implica que el agente se enfrenta a diferentes tamaños de pantalla y velocidades de movimiento en cada episodio, justo como sería en un entorno real, ya que cada jugador puede tener un tamaño de pantalla diferente o incluso tener el juego en modo ventana en lugar de en pantalla completa.

Como se puede apreciar en la figura, eliminar esta aleatoriedad tiene dos efectos principales. Por un lado, la recompensa media disminuye, lo que puede deberse a la forma en la que la función de recompensa asigna puntos en función del número de pasos del episodio. Por otro lado, se aprecia una mayor estabilidad en la recompensa, lo que tiene sentido puesto que el agente se enfrenta a situaciones más predecibles.

En el entrenamiento en verde, se eliminó la recompensa negativa por "desapuntar" al objetivo una vez ya estaba en el centro de la cuadrícula (0,0). Esta recompensa negativa se ideó para que el agente se mantuviera

en el centro de la cuadrícula una vez llegara. Como se puede apreciar, este cambio no tiene mucho efecto en el agente final, pero sí que parece que el agente mejora su recompensa episódica más rápidamente al principio del entrenamiento que en el entrenamiento por defecto. Por lo tanto, podría ser una buena idea eliminar esta recompensa negativa por completo.

10.1.2. Entrenamiento del agente con A2C

Hiperparámetro	**Valor**
Número de entornos	16
Número total de pasos en el tiempo	2.5×10^7
Política	"MlpPolicy"
Coeficiente de entropía	3.27×10^{-4}
Norma máxima del gradiente	0.3
Número de pasos de actualización	512
Factor de descuento para GAE	0.92
Coeficiente de función de valor	0.71
Factor de descuento	0.99
Uso de RMSProp	Verdadero
Normalización de la ventaja	Verdadero
Tasa de aprendizaje	lin_0.00015
Función de activación	Tanh
Arquitectura de la red	[256, 256]
Inicialización de desviación estándar	-2
Inicialización ortogonal	Verdadero

Tabla 10.2: Hiperparámetros usados durante el entrenamiento del agente mediante A2C.

Tras realizar la optimización de hiperparámetros y entrenamiento del agente A2C utilizando las mismas condiciones de tiempo que para PPO, se obtuvieron resultados muy inferiores. Por esta razón se decidió incrementar en gran medida tanto el número de combinaciones a probar en la optimización de hiperparámetros, como el número de pasos a realizar en el entrenamiento. En lugar de 15 combinaciones, se probaron 250, lo que tardó alrededor de 8 horas. Por su parte, durante el entrenamiento se realizaron 25 millones de pasos (en lugar de un millón), lo que tomó alrededor de 4 horas y media. La velocidad de entrenamiento fue de 9000 iteraciones por segundo, lo que agilizó considerablemente el proceso. La función de activación de la red fue Tanh, que aunque es algo más sencilla que ReLU, sigue siendo una función de activación muy popular en las redes neuronales profundas. La arquitectura de la red fue de dos capas ocultas de 256 neuronas cada una.

En el caso de A2C, las métricas que se mostraban durante el entrenamiento eran mucho más dispares entre sí que cpn PPO. Por ejemplo, para dos actualizaciones consecutivas después de 200 iteraciones (1638400 pasos), se obtuvieron valores de longitud media de episodios de 67.8 y 50001, y recompensas medias de 0.961 y 0.106, respectivamente. Estos valores tan dispares indican que el agente no estaba aprendiendo de manera consistente. Es decir, al final del entrenamiento, donde el agente debería estar resolviendo el entorno, todavía estaba teniendo problemas para hacerlo (llegando al tope de pasos de 50000). La varianza explicada, con valores de 0.0015 y 0.00063 para esas mismas actualizaciones, también indicaba que el agente no conseguía aprender de forma consistente.

La gráfica del avance de la recompensa del algoritmo A2C presenta este mismo escenario. En la Figura 10.2 se ven los resultados de tres entrenamientos diferentes del agente con este algoritmo. En el entrenamiento 1, en azul, se pueden ver los resultados de aquel primer intento utilizando las mismas condiciones que con PPO. Las recompensas con este agente son especialmen-

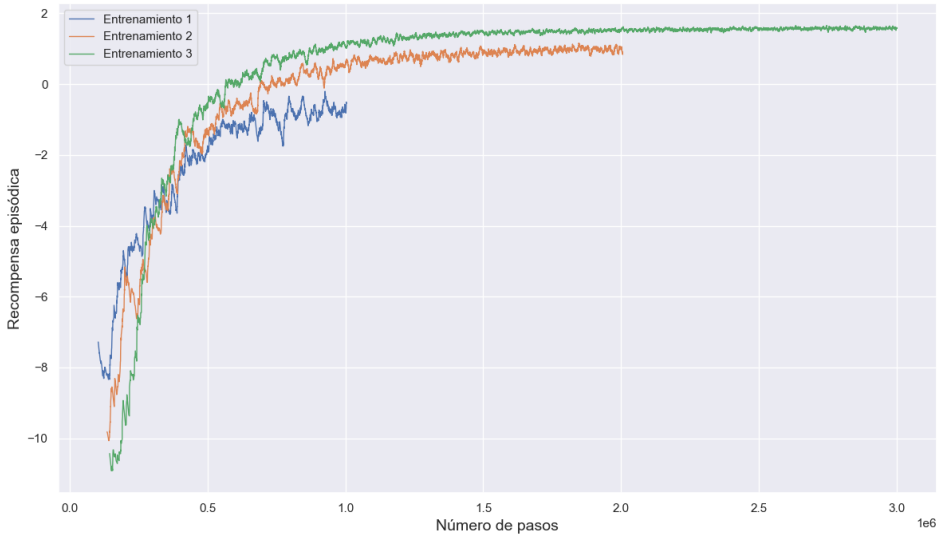

Figura 10.2: Avance de la recompensa conforme a los pasos en A2C

te erráticas y, al acercarse al final del entrenamiento después de un millón de pasos, se aprecia que el agente parece estar estabilizándose en recompensas negativas. Por lo que este agente no había aprendido a resolver el entorno.

En el entrenamiento número 2, en naranja, se utilizaron los hiperparámetros finales de la tabla anterior. Sin embargo, dado que el entorno aún estaba en proceso de desarrollo, la distancia entre puntos se estaba calculando de forma incorrecta. Esto provocaba que el agente no pudiera aprender correctamente, y luego de dos millones de episodios se estancó en una recompensa que casi no llegaba a 1. Esto sigue siendo un mal resultado, ya que como mínimo la recompensa total del episodio debe ser de 1.5 dada la forma en la que está diseñada la función de recompensa.

Por último, en el entrenamiento número 3, en verde, se corrigió el cálculo de la distancia entre puntos, y se volvió a entrenar el agente con los hiperparámetros finales. Aunque el entrenamiento duró 25 millones de pasos, en la gráfica se ha cortado en 3 millones para poder apreciar mejor el progreso. Aún así, se aprecia una gran constancia en la recompensa que no cambia

mucho en los 22 millones restantes. Incluso consiguiendo una recompensa media cercana a la del algoritmo PPO, en secciones posteriores se verá que su rendimiento real sigue sin acercarse debido a la longitud media de sus episodios.

10.2. Acciones extendidas

Uno de los experimentos más interesantes realizados durante el desarrollo del proyecto fue el de ampliar las posibilidades de movimiento del agente. En lugar de tener únicamente cuatro direcciones de movimiento, se le permitió moverse en otras cuatro direcciones oblicuas. Dicho de otra manera, si viéramos las direcciones de movimiento por defecto como "Norte, Sur, Este y Oeste", las nuevas direcciones serían "Noreste, Noroeste, Sureste y Suroeste". Este cambio se hizo para ver si el agente era capaz de aprender a moverse de forma más eficiente cambiando solamente sus posibles acciones.

Aunque los hiperparámetros optimizados de la tabla anterior fueron conseguidos con Optuna mediante las 15 combinaciones de valores, el agente final no se acabó entrenando con ellos, sino con los del algoritmo PPO por defecto. La razón de esto es que el entrenamiento pasaba de tener una velocidad de unas 4500 iteraciones por segundo a unas 400, lo que resultaba en un entrenamiento mucho más lento. Esto principalmente se debe a dos hiperparámetros: el tamaño del lote y el número de épocas. En el algoritmo PPO por defecto, el tamaño del lote es de 256 y el número de épocas es de 5, mientras que en el PPO extendido, el tamaño del lote es de 32 y el número de épocas es de 20. Un tamaño de lote menor significa que el modelo actualiza sus ponderaciones con más frecuencia, pero con menos muestras en cada paso. Esto puede dar lugar a más iteraciones y, por tanto, a un entrenamiento más lento en general (ivanbgd, 2018). Por otro lado, aumentar el número de épocas significa que el modelo pasará por más ciclos de entrenamiento, lo que aumenta el tiempo total de entrenamiento.

Hiperparámetro	**Valor**
Número de entornos	16
Número total de pasos en el tiempo	1×10^6
Política	"MlpPolicy"
Número de pasos de actualización	32
Tamaño del lote	32
Factor de descuento para GAE	0.9
Factor de descuento	0.99
Número de épocas	20
Coeficiente de entropía	1.98×10^{-7}
Tasa de aprendizaje	0.0089
Rango de recorte	0.2
Norma máxima del gradiente	0.8
Coeficiente de función de valor	0.55
Función de activación	ReLU
Arquitectura de la red	pi=[64], vf=[64]

Tabla 10.3: Hiperparámetros iniciales de PPO extendido.

Es decir, reducir el tamaño del lote aumenta el número de lotes por época, y aumentar el número de épocas conlleva un aumento del número de ciclos de entrenamiento. Ambos cambios provocan actualizaciones más frecuentes, lo que ralentiza el proceso de entrenamiento. Por esta razón, se decidió no utilizar estos hiperparámetros, y en su lugar se probaron a utilizar los del algoritmo PPO con solo cuatro direcciones de movimiento.

En este pequeño experimento, usando los hiperparámetros de un agente en otro parecido, se obtuvieron resultados positivos, ya que el modelo extendido conseguía aprender tan rápido como lo hacía el agente por defecto. En la Figura 10.3 se puede ver el avance de su recompensa en el tiempo.

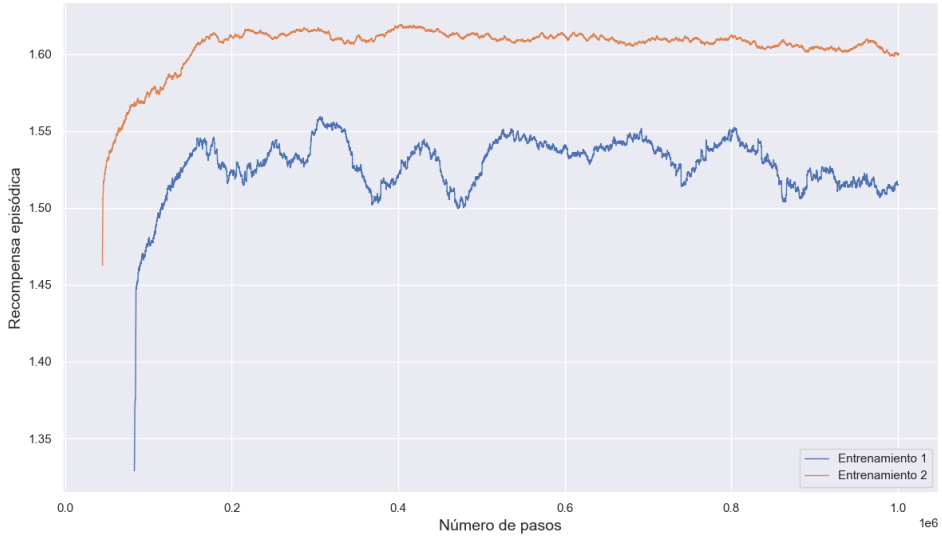

Figura 10.3: Avance de la recompensa conforme a los pasos en PPO extendido

En esta figura se muestran los entrenamientos de dos agentes diferentes. El primero de ellos, entrenado con los hiperparámetros que encontró Optuna para el PPO extendido, y el segundo, entrenado con los hiperparámetros del PPO por defecto. Es evidente que el agente es mucho más estable en el segundo caso, siendo además su recompensa media más alta. Por lo tanto, se puede concluir que fue una buena decisión no utilizar los hiperparámetros

optimizados para el PPO extendido.

10.3. Comparación entre PPO y PPO extendido

En esta sección, se compararán los resultados de ambos agentes entrenados con PPO, cuya única diferencia es el espacio de acción del entorno. Tanto la función de recompensa como el espacio de observación son los mismos, así como los hiperparámetros utilizados en el entrenamiento y la longitud de este son idénticos. Por lo tanto, cualquier diferencia en el rendimiento de los agentes se deberá únicamente a la diferencia en las acciones posibles.

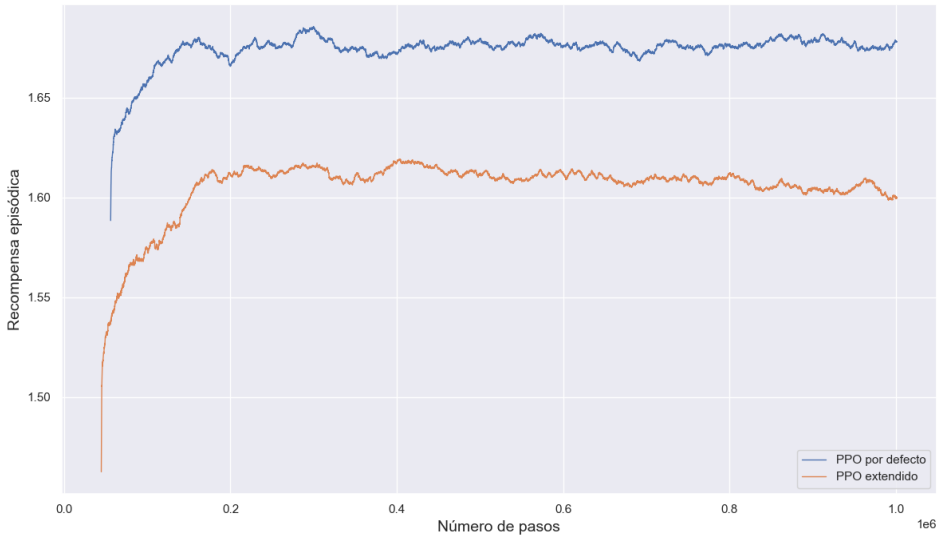

Figura 10.4: Comparación de recompensa episódica entre PPO y PPO extendido

En la Figura 10.4 se aprecia una gran similitud entre ambos agentes, habiendo una única y clara diferencia: la recompensa media del agente con acciones extendidas es ligeramente inferior a la del agente con acciones por defecto. A primera vista se podría pensar que el agente con acciones extendidas funciona peor, pero hay que tener en cuenta que la recompensa depende del número de pasos.

En la Sección 7.1.2.3 se mencionó que la función de recompensa asigna puntos positivos o negativos en cada paso del episodio. Si en un paso el agente se acerca hacia el objetivo, su recompensa es de 0.1; si se aleja, su recompensa es de -0.1. Esto, por lo tanto, tiene una correlación directa con la longitud de los episodios. Si un episodio exitoso es más largo que otro igualmente exitoso que requiere de menos pasos para llegar al mismo sitio, el primero tendrá una recompensa mayor que el segundo.

Si se tiene en cuenta que ambos agentes se entrenaron con la misma semilla, y por lo tanto, con los mismos episodios, se puede concluir que el agente con acciones extendidas es más eficiente que el agente con acciones por defecto. Esto se puede ver en la Figura 10.5, donde se muestra la longitud de los episodios de ambos agentes. En ella se puede apreciar que el agente con acciones extendidas tiene episodios más cortos que el agente con acciones por defecto, lo que significa que es capaz de llegar al objetivo en menos pasos.

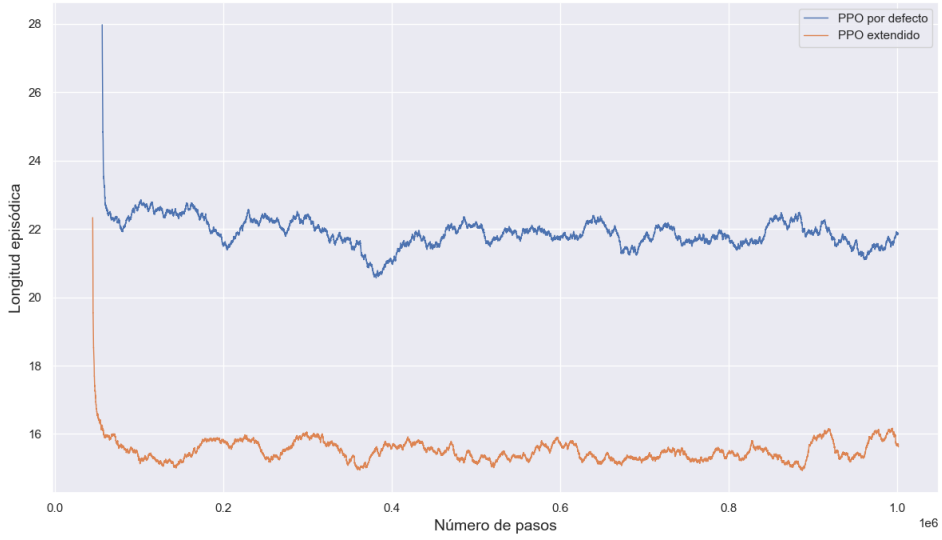

Figura 10.5: Comparación de longitud de los episodios entre PPO y PPO extendido

Para aclarar por completo cuál es el algoritmo que genera una mayor

cantidad de recompensa en función de la longitud de sus episodios, se ha diseñado una última gráfica. La figura 10.6 presenta esta misma situación. Al dividir la recompensa obtenida por el número de pasos del episodio, se obtiene un valor que se puede interpretar como la cantidad de recompensa el agente genera cada paso. Es aquí donde se puede ver claramente que este valor es mayor cuando se usa el algoritmo PPO en el entorno con acciones extendidas.

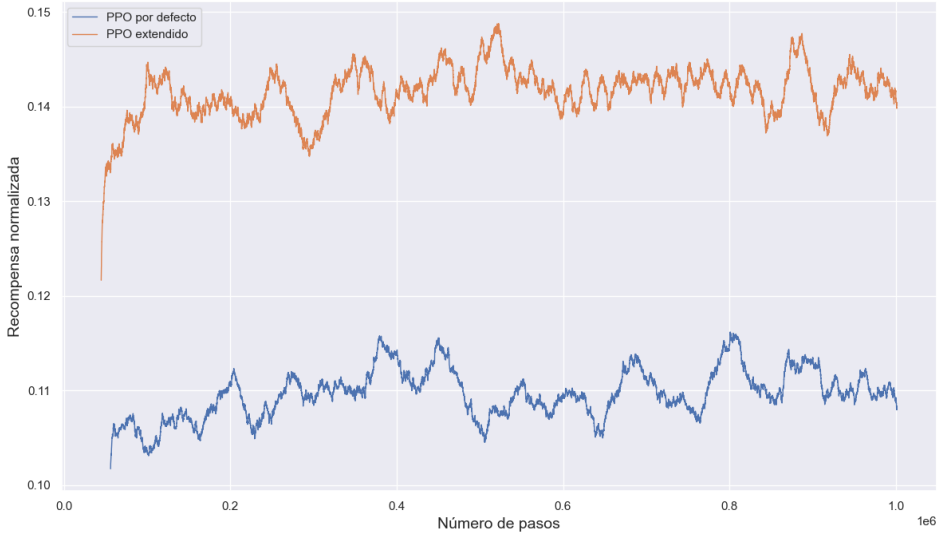

Figura 10.6: Comparación de recompensa normalizada entre PPO y PPO extendido

10.4. Experimentos con PPO

Por último, se realizaron una serie de experimentos con el agente por defecto del proyecto, entrenado con PPO y cuatro direcciones de movimiento. Estos experimentos se realizaron para evaluar las consecuencias de hacer ciertos cambios en el entorno, tales como la aleatoriedad del tamaño de la cuadrícula o del porcentaje de movimiento que se mencionó en la Sección 10.1.1. Se generaron en total siete agentes diferentes, cada uno con diferentes

alteraciones en el entorno (excepto el primero, que es el agente de control). Estos son los cambios de cada uno:

- Agente 1: Control, sin cambios.
- Agente 2: Sin aleatoriedad en el tamaño de la cuadrícula ni en el porcentaje de movimiento.
- Agente 3: Sin recompensa negativa por desapuntar al objetivo.
- Agente 4: Valores de las recompensas divididos por 10 (excepto la recompensa final y la de disparo).
- Agente 5: Valores de las recompensas aumentados en 0.25 (excepto la recompensa final y la de disparo).
- Agente 6: Sin recompensa negativa (-0.1) por alejarse del objetivo.
- Agente 7: Todos los valores de las recompensas multiplicados por 10.

En la siguiente Figura 10.7, se puede ver el avance de la recompensa de los siete agentes entrenados, contando con los tres ya mencionados en la sección anterior. Aunque no se puede apreciar con mucho detalle la recompensa media de todos los entrenamientos debido a la gran disparidad de su funciones de recompensa, se puede ver que la mayoría de ellos tienen una recompensa media positiva y constante. Asimismo, todos ellos consiguieron aprender a resolver el entorno. El único experimento que impidió por completo el entrenamiento es el cuarto, que ni siquiera aparece en la tabla debido a que no generó datos útiles. Esto evidencia que tener recompensas demasiado pequeñas, como -0.01 por alejarse del objetivo o 0.01 por acercarse, genera problemas en el entrenamiento. Por lo tanto, estos experimentos parecen indicar que ciertos cambios en los valores de las recompensas no afectan significativamente al rendimiento del agente, al menos para esta combinación en concreto de algoritmo y entorno.

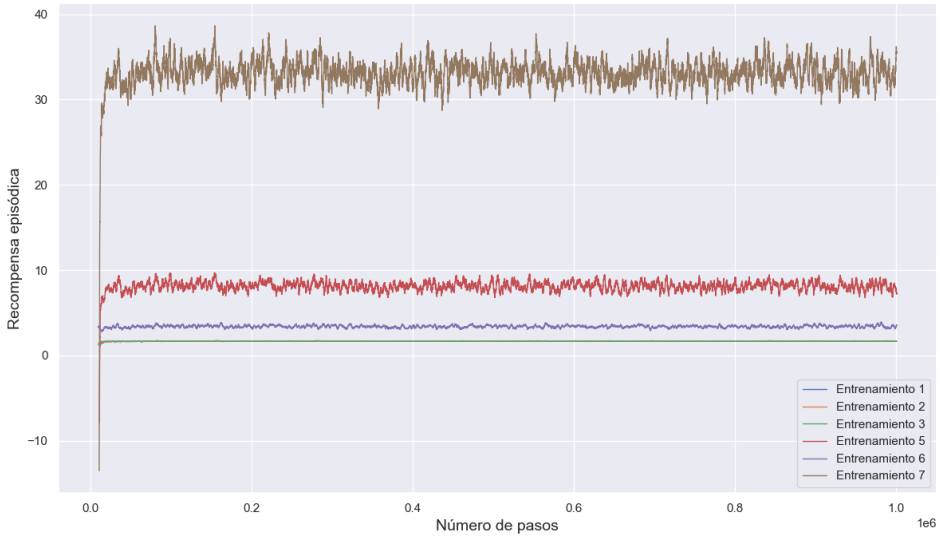

Figura 10.7: Avance de la recompensa episódica conforme a los pasos en los experimentos de PPO

En este caso, el gráfico con la longitud episódica de los agentes puede resultar más útil, ya que estos datos no se vieron tan afectados por la función de recompensa. En la Figura 10.8 se puede ver que la mayoría de los agentes tienen episodios de longitud parecida.

De aquí se destacarían dos entrenamientos anómalos, el 2 y el 6, que presentan una longitud de episodios menor o mayor que el resto, respectivamente. La anomalía del entrenamiento 2 puede deberse a que los parámetros por defecto del entorno crearan situaciones en las que el objetivo requería de menor cantidad de pasos para ser alcanzado. En el caso del entrenamiento 6, la falta de la recompensa negativa por alejarse del objetivo puede haber influido en que el agente se enfocase en centrar este con menor premura, lo que resulta en episodios más largos.

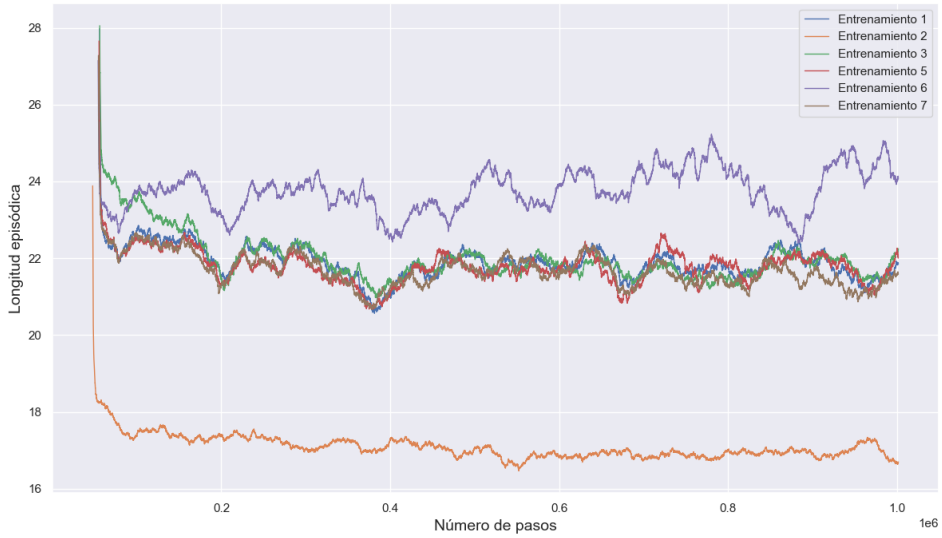

Figura 10.8: Avance de la longitud episódica conforme a los pasos en los experimentos de PPO

Capítulo 11

Análisis del rendimiento de los agentes en Unreal Engine

Tras presentar los resultados del entrenamiento de cada algoritmo y agente en el entorno virtual de Python, en este capítulo se presenta el rendimiento de los agentes en el entorno de Unreal Engine. Para ello, se han ejecutado 10 rondas del minijuego de la galería de tiro con el mejor agente de cada uno de los algoritmos. En una ronda del minijuego el jugador o agente tiene 60 segundos para disparar al mayor número de objetivos como le sea posible. En todo momento sólo existe un objetivo activo en pantalla, y cuando este es abatido aparece el siguiente de forma aleatoria y se suma un punto al marcador. Los fallos no penalizan la puntuación. Para cada ronda, se contaron la puntuación total (número de objetivos abatidos) y el número de fallos. Estos resultados se presentan en las Figuras 11.1 y 11.2.

La Figura 11.1 es una gráfica Boxplot y Swarmplot que muestra la puntuación total obtenida en cada una de las rondas para los 3 agentes. El agente que obtiene la mayor puntuación es el PPO extendido, con una media de 33 puntos. Le sigue el agente PPO por defecto, con una media de alrededor de 27 puntos. Esto concuerda con los resultados observados en la Figura 10.5 del capítulo anterior, donde se observó que el agente PPO extendido cuenta

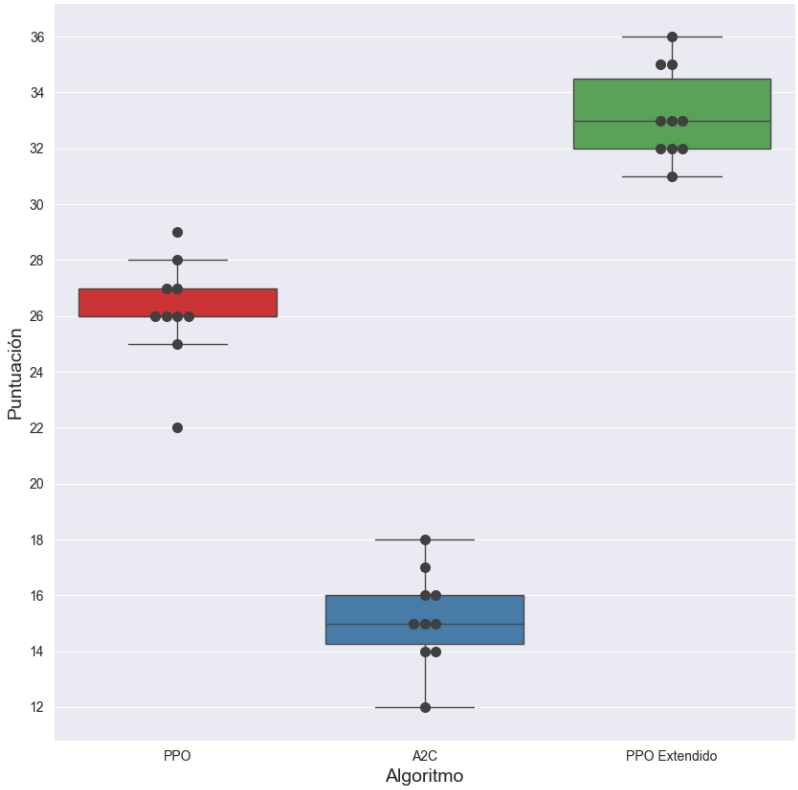

Figura 11.1: Comparación de la recompensa en 10 rondas entre los modelos A2C, PPO y PPO extendido

con una menor longitud episódica, lo que le permite obtener más puntos en el mismo periodo de tiempo que el agente PPO por defecto. En último lugar, el agente A2C obtiene una puntuación media de 15 puntos. La razón por la que el agente A2C obtiene una puntuación tan baja se ve de una forma más clara en este vídeo comparativo https://youtu.be/GV67p5SrTTc. A partir del minuto 1, se observa cómo en ocasiones el agente A2C "desapunta" al objetivo una vez está en el centro de la cuadrícula (0,0), es decir, mueve la cámara dentro del objetivo sin llegar a disparar. Esto es improbable que se

deba a un problema de falta de entrenamiento, ya que el agente A2C ha sido entrenado durante más episodios que los otros dos agentes. Lo que podría estar ocurriendo es que la penalización por llegar al objetivo y no disparar no sea lo suficientemente grande. Actualmente, si el agente llega al centro del objetivo y no se aleja de él, sino que se ronda el objetivo sin disparar, sólo recibe una recompensa negativa de -0.1 en cada paso. También es posible que en algunas instancias, el agente esté recibiendo una recompensa positiva al mostrar este comportamiento, debido a pequeñas inexactitudes que hagan que se detecte un ínfimo acercamiento al objetivo. En este caso, quizás se podría solucionar requiriendo que la diferencia de distancias entre la anterior y la actual sea mayor a un umbral (por ejemplo, 5 %) para verificar que se ha acercado al objetivo. En cualquier caso, el agente A2C no es capaz de abatir objetivos tan rápidamente como los otros dos agentes, lo que le impide obtener una alta puntuación.

Otra forma de evaluar el rendimiento de los agentes es contando la cantidad de errores cometidos en cada ronda, entendiendo un error como un disparo que no acierta en el objetivo. La Figura 11.2 muestra la cantidad de errores cometidos en las mismas 10 rondas de la figura anterior. Curiosamente, el agente que comete menos errores (de hecho, ninguno) es el A2C, que era el que obtenía la menor puntuación. Los agentes con el algoritmo PPO cometen algunos errores, siendo PPO extendido el que comete más errores. Por ejemplo, hubo una ronda en la que cometió 4 errores, que es el máximo de errores cometidos en una ronda por cualquiera de los agentes. En el vídeo https://youtu.be/NkENKC2DAao se muestran ejemplos de fallos cometidos por el agente PPO extendido y en la Figura 11.3 se muestra una captura de uno de estos errores.

Dado que este proyecto tiene como objetivo generar una IA que suponga un desafío para los jugadores humanos, se realizó un experimento para comparar el rendimiento de los agentes con el de una persona. Para ello,

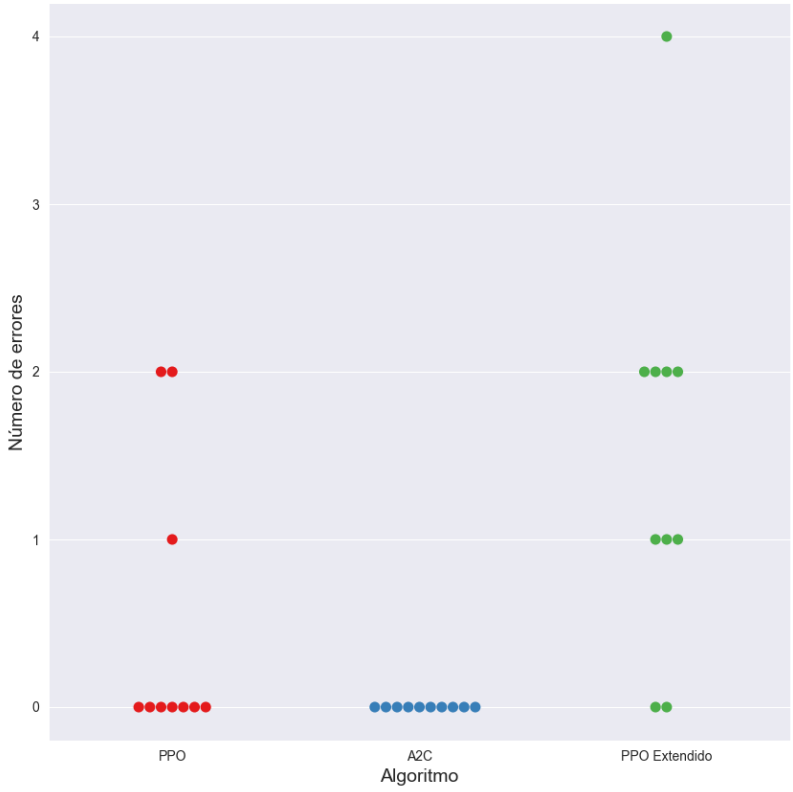

Figura 11.2: Comparación de la cantidad de errores en 10 rondas entre los modelos A2C, PPO y PPO extendido

un jugador completó 10 rondas del minijuego, obteniendo una media de 65 puntos y cometiendo un promedio de 4 errores por ronda. Esto quiere decir que el mejor agente, el PPO extendido, obtiene la mitad de la puntuación que podría conseguir un jugador humano. Por tanto, este modelo podría utilizarse como IA contra la que competir solamente en un modo de dificultad fácil. Para mejorar la puntuación de los agentes hasta niveles equiparables a jugadores humanos, sería necesario aumentar la velocidad en la que pueden tomar acciones. Dada la arquitectura actual del proyecto, esta velocidad no

Figura 11.3: Captura de un error cometido por el agente PPO extendido en el que no dispara al objetivo

puede ser aumentada. No obstante, si el juego se ejecutara en una máquina más potente o se optimizara la transmisión de datos entre Unreal Engine y Python, se podría mejorar en gran medida la velocidad de estas acciones y, por lo tanto, la puntuación de los agentes.

Capítulo 12

Discusión de los resultados

En este capítulo se analizan los puntos fuertes y débiles de los algoritmos PPO y A2C en el aprendizaje por refuerzo aplicado a videojuegos, destacando la superioridad de PPO y los desafíos de comportamiento errático. Además, se evalúa la viabilidad de Unreal Engine para estas técnicas.

A lo largo de este manual técnico se han presentado dos opciones de las muchas disponibles para entrenar un modelo de aprendizaje por refuerzo. Tanto Proximal Policy Optimization como Advantage Actor Critic han demostrado la capacidad de aprender planes de acción que pueden ser trasladados a un entorno virtual, como un videojuego. Aunque con eficiencias diferentes, ambos algoritmos han logrado mejorar sus puntuaciones a lo largo de las iteraciones. Sin embargo, es necesario destacar que los dos agentes entrenados con PPO han sido capaces de aprender esta tarea mucho más rápido que el agente entrenado con A2C. Puede que esto se deba al uso de la región de confianza en la política, que evita fluctuaciones muy grandes entre iteraciones, o puede que para este problema concreto, por la casuística de sus recompensas, espacios de acción y observación, el algoritmo PPO simplemente sea más adecuado. De cualquier forma, queda claro que con el suficiente tiempo de optimización de parámetros y entrenamiento, los dos algoritmos son útiles para el propósito establecido.

Uno de los desafíos a los que se ha de enfrentar cuando se utilizan este tipo de algoritmos es la inexplicabilidad de algunos de los comportamientos generados por sus agentes. Aunque se han establecido recompensas y penalizaciones para guiar el aprendizaje, en ocasiones los agentes han aprendido comportamientos que no estaban previstos. En un proyecto de estas características, no sólo puede fallar una parte del código específica con una función concreta, sino que el propio comportamiento que se quiere moldear puede estar diseñado de forma incorrecta. En este sentido, la ingeniería de recompensas puede determinar en gran parte el éxito o el fracaso del agente. En este Trabajo de Fin de Grado, por ejemplo, existe un comportamiento erróneo que, si bien se ha reducido en gran medida, no se ha podido eliminar por completo. La existencia de comportamientos erráticos por parte del agente, debidos a una definición inadecuada de la función de recompensa, es un problema común en el aprendizaje por refuerzo. Es aconsejable tenerlo en cuenta a la hora de plantear el entorno del problema.

Este proyecto tenía como uno de sus objetivos averiguar si Unreal Engine es una herramienta que favorece el uso del aprendizaje por refuerzo para crear comportamientos en videojuegos. Tras haber trabajado en este tema durante varios meses, se puede afirmar que actualmente UE no es una plataforma que facilite en gran medida la utilización de estas técnicas de aprendizaje. No obstante, esto no significa que sea una tarea imposible. Este proyecto es un ejemplo de que, aunque no se disponga de software específico como un equivalente a ML-Agents de Unity, se puede hacer uso de otras herramientas y bibliotecas para conseguir un resultado similar. Quizás más adelante, cuando otros proyectos que pretenden integrar estas técnicas en UE estén más avanzados, sea un momento adecuado para retomar este proyecto y mejorarlo. Pero por ahora, se puede afirmar que aunque no es sencillo, es posible.

Los algoritmos de aprendizaje por refuerzo permiten generar comporta-

mientos emergentes que no han sido previamente establecidos por el programador, por lo que muestran el potencial de llegar a ser herramientas útiles para la creación de situaciones inmersivas dentro de videojuegos. Con la suficiente guía y heurística, estos algoritmos podrían ser moldeados de tal forma que superen las limitaciones de técnicas como los árboles de comportamiento, que son un estándar en la industria actual. Sin embargo, el mayor muro al que se enfrentan estos algoritmos es la volatibilidad inherente al proceso de desarrollo de videojuegos, que puede hacer que el agente aprenda comportamientos que puedan ser irrelevantes dentro de un entorno en constante cambio.

Quizás crear agentes que aprendan a través de la experiencia del ensayo y el error acabe siendo no sólo la siguiente evolución de la IA dentro de los videojuegos, sino de la IA en general (Silver et al., 2021). O puede que el aprendizaje por refuerzo sea una técnica que se quede en el camino. Lo que es seguro es que contar con una recompensa propiamente ideada siempre es signo de un buen comienzo.

Capítulo 13

Conclusiones y futuras mejoras

Este Trabajo de Fin de Grado trata de la creación de un prototipo de videojuego que une Unreal Engine y el aprendizaje por refuerzo. Un producto que demuestre que dicha combinación es viable y que puede formar una vía de investigación atractiva. Asimismo en todo momento se ha buscado que el proyecto sea una oportunidad de aprendizaje sobre una materia tan compleja e interesante como es la inteligencia artificial y el aprendizaje automático.

A modo de revisión final al contenido del proyecto, se volverán a repasar tanto los objetivos específicos como los didácticos para realizar una retrospectiva de los logros alcanzados.

13.1. Objetivos Específicos

El diseño, desarrollo e implementación de un pequeño juego de disparos en 3D como entorno de pruebas se ha logrado tomando el diseño de Resident Evil 4 y la base de Lyra. Se ha creado una galería de tiro que funciona tanto como minijuego para un jugador humano como campo de entrenamiento para la IA, ofreciendo al jugador humano opciones adicionales como el uso de diferentes armas y la posibilidad de lanzar granadas, mientras que la IA tiene un sistema capaz de interpretar sus movimientos. En cuanto al

diseño, desarrollo e implementación de una red neuronal capaz de jugar al juego, aunque no se ha desarrollado una red neuronal de forma directa, se han ajustado varias redes al entrenar la IA. PPO y A2C utilizan dos redes neuronales para optimizar su política y función de valor, por lo que en total se han optimizado y ajustado seis redes neuronales distintas (PPO, PPO Extendido y A2C). Este objetivo refleja una mejora en la comprensión de la metodología del aprendizaje por refuerzo. Al principio se pensaba que sería necesario diseñar una red neuronal, cuando realmente lo que se haría es ajustar una red preexistente a un entorno de entrenamiento.

El objetivo de diseñar, desarrollar e implementar un programa adyacente al juego para gestionar el entrenamiento y testeo de la red neuronal se ha cumplido. Este programa está dividido en dos partes: una escrita en C++ que gestiona el juego en UE y otra en Python que gestiona el entrenamiento de la IA. El análisis de datos y el establecimiento de parámetros determinantes para el funcionamiento óptimo de la IA también se han realizado. El análisis se ha expuesto en este documento y el ajuste de hiperparámetros se ha llevado a cabo mediante Optuna, logrando en gran medida el éxito de los agentes, aunque con algunas limitaciones en el funcionamiento de la IA.

Finalmente, la validación y corrección del sistema se ha centrado en entregar un sistema lo más eficiente posible. Desde el principio se ha buscado mejorar el rendimiento del servidor local y minimizar errores, probando diferentes bibliotecas de corutinas. Se considera que el sistema funciona correctamente y con la mayor eficiencia posible dentro del tiempo disponible.

13.2. Objetivos didácticos

En cuanto a los objetivos didácticos, se planteaba el aprendizaje de varias herramientas. El uso de Unreal Engine como herramienta de desarrollo de videojuegos ha requerido un esfuerzo significativo en documentación y búsqueda de conocimientos, debido a la complejidad de los sistemas utilizados.

Esto ha permitido un aprendizaje profundo de cómo funcionan y se pueden modificar estos sistemas, aunque ha reducido el tiempo para crear contenido propio. El aprendizaje de C++ y Python se ha enfocado en su aplicación en el ámbito del aprendizaje por refuerzo, construyendo sobre una buena base de conocimientos previos.

Stable Baselines, como biblioteca de aprendizaje por refuerzo, ha sido ampliamente estudiada y utilizada, recurriendo a la documentación oficial y a la comunidad para resolver dudas y problemas. De manera similar, Gymnasium se ha utilizado para crear el entorno de entrenamiento de la IA, alcanzando un nivel adecuado de conocimiento. Aunque Pytorch no se ha utilizado extensamente, ha sido útil en aspectos específicos, como las definiciones de las funciones de activación necesarias para describir los hiperparámetros del entrenamiento de los agentes.

La redacción de este documento en LaTeX ha facilitado el aprendizaje y el uso futuro de esta herramienta en trabajos académicos. Además, se han adquirido múltiples conceptos de aprendizaje automático y redes neuronales, destacando el curso de David Silver que ayudó a entender los conceptos básicos del aprendizaje por refuerzo. La redacción de documentos técnicos ha mejorado gracias a la orientación de los codirectores del proyecto, aprendiendo a redactar correctamente este tipo de documentos.

El estudio de la aceleración de software mediante hardware gráfico se realizó al principio del proyecto, seleccionando bibliotecas que utilizan CUDA para acelerar el entrenamiento de la IA. Este proceso ha fomentado la comprensión de cómo se puede acelerar el entrenamiento. En general, todo el Trabajo de Fin de Grado ha sido una gran oportunidad de aprendizaje en ingeniería informática, destacando la importancia de una organización mensual bien definida y la gestión del alcance del proyecto.

13.3. Futuras mejoras

Como futuras mejoras, se podrían añadir más opciones de movimiento para el agente, como recargar manualmente y lanzar granadas. Aunque esto complicaría el entrenamiento de la IA al aumentar el espacio de acción y la complejidad de la función de recompensa, permitiría un comportamiento más realista.

Por otro lado, en trabajos futuros podría ser interesante crear más niveles de dificultad, añadiendo maniquíes móviles y objetivos pequeños, lo que mejoraría la rejugabilidad y demostración de adaptabilidad de la IA.

Por último, para mejorar la eficiencia del entrenamiento, se podría seguir dos enfoques:

- Crear un sistema de avisos para que Python reciba notificaciones cuando haya nuevos datos, evitando comprobaciones continuas.
- Implementar la paralelización del entrenamiento, utilizando varias galerías de tiro y agentes en un mismo mapa para mejorar la exactitud de la IA.

Finalmente, se podría rehacer la IA con el *plugin* oficial de Epic Games para agentes entrenados con aprendizaje por refuerzo ("Learning Agents"), que aún no ha sido publicado, lo que aumentaría la eficiencia del sistema y mejoraría las capacidades del agente.

Bibliografía

Kaplan, A., & Haenlein, M. (2019). Siri, Siri, in my hand: Who's the fairest in the land? On the interpretations, illustrations, and implications of artificial intelligence. *Business Horizons*, *62*(1), 15-25. https://doi.org/10.1016/j.bushor.2018.08.004

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115-133. https://doi.org/10.1007/BF02478259

Turing, A. M. (1950). I.—Computing machinery and intelligence. *Mind*, *LIX*(236), 433-460. https://doi.org/10.1093/mind/LIX.236.433

Goldsmith, J. T. T., & Ray, M. E. (1948, 14 de diciembre). *Cathode-ray tube amusement device* (pat. estadounidense N.º 2455992A).

McDonald, E. (2023, 17 de mayo). *Newzoo's video games market size estimates and forecasts for 2022* [Newzoo]. https://newzoo.com/resources/blog/the-latest-games-market-size-estimates-and-forecasts

Wikipedia, C. (2022, 18 de octubre). AAA (industria del videojuego) [Page Version ID: 146720870]. En *Wikipedia, la enciclopedia libre*. https://es.wikipedia.org/w/index.php?title=AAA_(industria_del_videojuego)&oldid=146720870

Wikipedia, C. (2023a, 8 de mayo). Diseño de juegos [Page Version ID: 151025854]. En *Wikipedia, la enciclopedia libre*. https://es.wikipedia.org/w/index.php?title=Dise%C3%B1o_de_juegos&oldid=151025854

Russell, S., & Norvig, P. (2020, 28 de abril). *Artificial Intelligence: A Modern Approach*.

Sekhavat, Y. (2017). Behavior Trees for Computer Games. *International Journal on Artificial Intelligence Tools*, *26*. https://doi.org/10.1142/S0218213017300010

Jagdale, D. (2021). Finite State Machine in Game Development, 384-390. https://doi.org/10.48175/IJARSCT-2062

Coursera, S. (2024, 9 de febrero). *Machine learning vs. AI: Differences, uses, and benefits* [Coursera]. https://www.coursera.org/articles/machine-learning-vs-ai

Song, L. (2021). *Figure 3. relationship between artificial intelligence, machine...* [ResearchGate]. https://www.researchgate.net/figure/Relationship-between-artificial-intelligence-machine-learning-neural-network-and-deep_fig3_354124420

Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects [Publisher: American Association for the Advancement of Science]. *Science*, *349*(6245), 255-260. https://doi.org/10.1126/science.aaa8415

IBM, C. (2024). *What are Neural Networks? | IBM*. https://www.ibm.com/topics/neural-networks

Elkan, C. (2012). Reinforcement Learning. https://scholar.google.com/citations?view_op=view_citation&hl=en&user=im5aMngAAAAJ&cstart=100&pagesize=100&citation_for_view=im5aMngAAAAJ:evX43VCCuoAC

StableBaselines3, E. d. (2021). *Reinforcement Learning Tips and Tricks — Stable Baselines3 2.3.2 documentation*. https://stable-baselines3.readthedocs.io/en/master/guide/rl_tips.html

Nicholson, C. V. (2023). *A beginner's guide to deep reinforcement learning* [Pathmind]. http://wiki.pathmind.com/deep-reinforcement-learning

Narvekar, S., Peng, B., Leonetti, M., Sinapov, J., Taylor, M. E., & Stone, P. (2020, 17 de septiembre). Curriculum Learning for Reinforcement Learning Domains: A Framework and Survey. https://doi.org/10.48550/arXiv.2003.04960

Hill, A. (2023, 10 de julio). *Stable Baselines* [original-date: 2018-07-02T14:28:59Z]. https://github.com/hill-a/stable-baselines

OpenAI, C. (2023, 10 de julio). *gym* [original-date: 2016-04-27T14:59:16Z]. https://github.com/openai/gym

ASALE, R., & RAE. (2024). *Videojuego* [«Diccionario de la lengua española» - Edición del Tricentenario]. https://dle.rae.es/videojuego

Plante, C. (2015, 4 de marzo). *Why video game engines may power the future of film and architecture* [The verge]. https://www.theverge.com/2015/3/4/8150057/unreal-engine-4-epic-games-tim-sweeney-gdc-2015

Skinner, G., & Walmsley, T. (2019). Artificial Intelligence and Deep Learning in Video Games A Brief Review. *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*, 404-408. https://doi.org/10.1109/CCOMS.2019.8821783

Duong, V. (2021, 12 de junio). *Artificial intelligence in gaming - technology insider* [Section: AI - Machine Learning]. https://technologyinsider.asia/ai-machine-learning/artificial-intelligence-in-gaming/

Anderson, E. (2003). Playing Smart - Artificial Intelligence in Computer Games.

Yannakakis, G. N. (2012). Game AI revisited. *Proceedings of the 9th conference on Computing Frontiers*, 285-292. https://doi.org/10.1145/2212908.2212954

Statt, N. (2019, 6 de marzo). *How artificial intelligence will revolutionize the way video games are developed and played* [The verge]. https:

//www.theverge.com/2019/3/6/18222203/video-game-ai-future-procedural-generation-deep-learning

Helsinki, U. (2024). *A free online introduction to artificial intelligence for non-experts* [Elements of AI]. https://course.elementsofai.com/

Campbell, M., Hoane, A. J., & Hsu, F.-h. (2002). Deep blue. *Artificial Intelligence*, *134*(1), 57-83. https://doi.org/10.1016/S0004-3702(01)00129-1

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning [Number: 7553 Publisher: Nature Publishing Group]. *Nature*, *521*(7553), 436-444. https://doi.org/10.1038/nature14539

Emu, S. (1995). *Stella* [original-date: 2016-10-01T12:56:15Z]. Stella (Atari 2600 emulator). https://github.com/stella-emu/stella

Justesen, N., Bontrager, P., Togelius, J., & Risi, S. (2019, 18 de febrero). Deep Learning for Video Game Playing. https://doi.org/10.48550/arXiv.1708.07902

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search [Number: 7587 Publisher: Nature Publishing Group]. *Nature*, *529*(7587), 484-489. https://doi.org/10.1038/nature16961

OpenAI, Berner, C., Brockman, G., Chan, B., Cheung, V., Dębiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., Pinto, H. P. d. O., Raiman, J., Salimans, T., ... Zhang, S. (2019, 13 de diciembre). Dota 2 with Large Scale Deep Reinforcement Learning. https://doi.org/10.48550/arXiv.1912.06680

Toftedahl, M. (2019, 30 de septiembre). *Which are the most commonly used game engines?* https://www.gamedeveloper.com/production/which-are-the-most-commonly-used-game-engines-

Wikipedia, C. (2023b, 23 de junio). Convolutional neural network [Page Version ID: 1161617479]. En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Convolutional_neural_network&oldid=1161617479

Wikipedia, C. (2024, 11 de mayo). Steam (service) [Page Version ID: 1223329270]. En *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Steam_(service)&oldid=1223329270

Obedkov, E. (2022, 22 de marzo). *5 game studios that switched to unreal engine for flexibility and new hiring opportunities* [Game world observer] [Section: Game Development]. https://gameworldobserver.com/2022/03/22/5-game-studios-that-switched-to-unreal-engine-for-flexibility-and-new-hiring-opportunities

OpenAI. (2018a, 1 de septiembre). *Part 1: Key Concepts in RL — Spinning Up documentation*. https://spinningup.openai.com/en/latest/spinningup/rl_intro.html

OpenAI. (2018b, 1 de septiembre). *Part 2: Kinds of RL Algorithms — Spinning Up documentation*. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

GeeksforGeeks. (2021, 23 de septiembre). *Bellman equation* [GeeksforGeeks] [Section: AI-ML-DS]. https://www.geeksforgeeks.org/bellman-equation/

StableBaselines3. (2021a). *PPO — Stable Baselines3 2.4.0a0 documentation*. https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html

StableBaselines3. (2021b). *A2C — Stable Baselines3 2.4.0a0 documentation*. https://stable-baselines3.readthedocs.io/en/master/modules/a2c.html

Farama. (2022). *Gymnasium documentation*. https://gymnasium.farama.org/main/content/basic_usage/

Raffin, A. (2019). *DLR-RM/rl-baselines3-zoo* [original-date: 2020-05-05T05:53:27Z]. DLR-RM. https://github.com/DLR-RM/rl-baselines3-zoo

Eschmann, J. (2021). Reward function design in reinforcement learning. En B. Belousov, H. Abdulsamad, P. Klink, S. Parisi & J. Peters (Eds.), *Reinforcement learning algorithms: Analysis and applications* (pp. 25-33). Springer International Publishing. https://doi.org/10.1007/978-3-030-41188-6_3

StableBaselines3. (2021c). *Logger — Stable Baselines3 2.4.0a1 documentation*. https://stable-baselines3.readthedocs.io/en/master/common/logger.html

StableBaselines3. (2021d). *Base RL Class — Stable Baselines3 2.4.0a1 documentation*. https://stable-baselines3.readthedocs.io/en/master/modules/base.html#stable_baselines3.common.on_policy_algorithm.OnPolicyAlgorithm

Donges, N. (2024). *What are recurrent neural networks (RNNs)? | built in*. https://builtin.com/data-science/recurrent-neural-networks-and-lstm

Krishnamurthy, B. (2024). *ReLU activation function explained | built in*. https://builtin.com/machine-learning/relu-activation-function

ivanbgd. (2018, 3 de febrero). *Answer to "What is the trade-off between batch size and number of iterations to train a neural network?"* [Cross Validated]. https://stats.stackexchange.com/a/326663/342730

Silver, D., Singh, S., Precup, D., & Sutton, R. S. (2021). Reward is enough. *Artificial Intelligence*, *299*, 103535. https://doi.org/10.1016/j.artint.2021.103535