UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Introducción de soporte para redes neuronales en GAIA

Trabajo Fin de Máster

Máster Universitario en Ingeniería Informática

AUTOR/A: Blanco Villorejo, Saúl

Tutor/a: Mollá Vayá, Ramón Pascual

CURSO ACADÉMICO: 2022/2023

Para mi padre, mi madre,
mi hermano y mi tío

Agradecimientos

Gracias Dr. Ramón Mollá Vallá por darme la oportunidad de trabajar en este proyecto y por guiarme cuando el ambiente estaba nublado.

Gracias Triple Cherry y especialmente Eneko Ridruejo Fernández, por enseñarme a trabajar en equipo, a enfrentarme a nuevos desafíos con una mentalidad más abierta y a pensar que todo se puede hacer, sea más fácil o más difícil.

Resumen

El mercado de los videojuegos es un mercado en crecimiento con más de 3200 millones de jugadores a nivel mundial. Parte de los videojuegos contienen caracteres controlados por inteligencia artificial, la mayoría de los cuales están programados utilizando una de las dos técnicas más famosas, las máquinas de estados finitos o los árboles de comportamiento. El problema de estas dos técnicas radica en la dificultad de crear comportamientos complejos similares a los que tendría una persona. Por ese motivo, se están investigando otras tecnologías que puedan tomar el relevo, como las redes neuronales. Las redes neuronales son capaces de aprender comportamientos y adaptarse al nivel de los jugadores para que su experiencia sea más satisfactoria. Este trabajo pretende incorporar el soporte de redes neuronales en el asset de inteligencia artificial de GAIA, con el fin de facilitar su acceso a los usuarios de Unity.

Palabras clave: videojuegos, inteligencia artificial, máquina de estados finitos, árbol de comportamiento, redes neuronales, comportamiento realista, Unity, GAIA

Abstract

The video game market is a growing market with over 3200 million players worldwide. Some video games contain characters controlled by artificial intelligence, most of which are programmed using one of two famous techniques, finite state machines or behavior trees. The problem with these two techniques lies in the difficulty of creating complex behaviors similar to those a person would have. For this reason, other technologies are being researched that can take over, such as neural networks. Neural networks are capable of learning behaviors and adapting to the players' level to make their experience more satisfying. This work aims to incorporate support for neural networks into GAIA's artificial intelligence asset in order to make it more accessible to Unity users.

Key words: video games, artifcial intelligence, finite state machine, behavior tree, neural networks, realistic behavior, Unity, GAIA

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

CAPÍTULO 1
Introducción

La inteligencia artificial (IA) es una rama de la informática cuyo objetivo es la creación de máquinas que puedan imitar el comportamiento de la inteligencia humana. Existen muchas técnicas de inteligencia artificial distintas, entre las cuales se encuentran la lógica difusa [1], los sistemas expertos [2], los modelos ocultos de Markov [3], los algoritmos genéticos [4], los sistemas basados en agentes [5] y las redes neuronales [6]. Por nombrar algunos de los usos de esta rama y entender su potencial, estos son los bots de asistencia al cliente, recomendadores de contenido, detector de situaciones peligrosas como robos, los asistentes virtuales, la optimización en procesos de fabricación y la conducción autónoma.

Las técnicas de inteligencia artificial también son usadas en la industria de los videojuegos para crear comportamientos realistas de los personajes y monstruos. Desde la creación de los primeros videojuegos hasta la actualidad, los desarrolladores han tenido que crear NPCs (Non Player Character) y dotarles de comportamientos que parezcan inteligentes. Estos comportamientos hacen que un videojuego no sea aburrido para los jugadores. Es más, lo que se intenta es que sea desafiante y mantenga al jugador en un estado de flow, en el que siente que sus habilidades van mejorando y que esas mejoras le ayudan a superar retos cada vez más difíciles [7].

Hasta el momento, existen dos métodos principales para la creación de esta inteligencia artificial: las máquinas de estados finitos [8] (Capítulo 9) (Finite State Machine o FSM) y los árboles de comportamiento [9] (Behavior Tree o BT). Para conocer su importancia, basta con nombrar que estas dos técnicas vienen implementadas en dos de los motores de videojuegos más populares a nivel mundial: Unity[1] (máquina de estados finitos) y Unreal Engine[2] (árbol de comportamiento). El problema de estas técnicas es que, aunque son muy prácticas y sencillas de programar, cuando se quiere implementar un comportamiento inteligente complejo, la red de nodos y aristas que las definen complican su comprensión y por ende su modificación y mantenimiento. Por otro lado, el comportamiento de los NPCs llega a ser predecible. A modo de ejemplo, cuando el avatar del jugador se acerca a menos de 5 metros del enemigo, éste siempre le detecta y empieza a seguirlo. El enemigo siempre se va a comportar de la misma manera, no importa quién esté jugando. La dificultad del juego no varía en cuanto al comportamiento del enemigo, aunque sí que puede modificarse su precisión, el daño que inflige o su velocidad de avance.

[1] https://unity.com/es
[2] https://www.unrealengine.com

Una posible solución al problema de las técnicas mencionadas es el uso de redes neuronales para modelar el comportamiento de los NPCs. Este trabajo pretende incorporar el soporte para redes neuronales que puedan controlar la respuesta de un NPC en un videojuego dentro del asset de inteligencia artificial para Unity, GAIA[3]. Una vez incorporado, se pretende entrenar las redes neuronales con los comportamientos de FSMs o BTs creados por los programadores o bien haciendo que una persona asuma el papel de un NPC y entrenar con dicho comportamiento humano a la red neuronal. El objetivo es que las redes neuronales repliquen cualquier comportamiento sea de la fuente que sea. Además, las redes podrán seguir aprendiendo para que su comportamiento cada vez se acerque más al deseado.

1.1 Motivación

Como jugador ocasional de los videojuegos e informático profesional, buscaba un tema que pudiera abarcar ambos campos. Por supuesto que el trabajo debía de estar relacionado con el campo de la informática, pero si también podía aportar mi granito de arena al mundo de los videojuegos, lo haría encantado.

Cuando se me presentó el tema de este trabajo, decidí no perder la oportunidad de crear un programa que pudiera mejorar la jugabilidad de los juegos, ya que esto podría significar más diversión, realismo y entretenimiento para todos los jugadores. Al fin y al cabo, ¿a quién no le gusta divertirse?

No solo eso, si no que también pensé en las puertas que me abriría en la industria de los videojuegos realizar un trabajo sobre este tema. Un ingeniero informático que ha realizado su trabajo de final de máster sobre videojuegos e inteligencia artificial, que puede cambiar la cantidad de realismo y diversión de un videojuego para bien, puede entrar más fácilmente en empresas del sector.

Por todos los motivos expuestos en esta sección, acepté investigar el tema del trabajo.

1.2 Objetivos

La meta del proyecto es la extensión del asset de inteligencia artificial para Unity, GAIA, incluyendo soporte a la creación de redes neuronales a través de una especificación XML, entrenarlas para que imiten un comportamiento ya implementado, que puedan seguir aprendiendo una vez se acercan lo suficiente al comportamiento esperado, medir el grado de consecución de sus objetivos y compararlos entre sí.

El asset es el producto del trabajo y no tiene dependencia de ningún otro proyecto más que de las tecnologías que se vayan a utilizar para implementarlo.

A continuación se define la meta mencionada como un conjunto de objetivos SMART [4] que el producto debería cumplir. Definir los objetivos permite comprobar si se ha cum-

[3] https://github.com/tetracube/GAIA/tree/master
[4] https://asana.com/es/resources/smart-goals

plido con lo que se esperaba al finalizar el trabajo.

Todos los objetivos se plantean con fecha de fin el día de la entrega del trabajo, por lo que están acotados en el tiempo.

- Escribir una memoria de mínimo 80 páginas que refleje el flujo de pensamiento y trabajo llevado a cabo para implementar el asset.
- Definir la arquitectura del asset.
- Definir el diseño de clases del asset.
- Implementar el diseño de clases.
- Implementar un entorno de aprendizaje en el que una red neuronal aprende de una FSM ya preexistente cuyo comportamiento se quiere imitar.
- Entrenar a una red neuronal hasta que su comportamiento sea un porcentaje mínimo, especificado por el diseñador del videojuego, el comportamiento de una FSM.
- Implementar un entorno de aprendizaje en el que una red neuronal se enfrenta a una FSM y se hace más eficiente consiguiendo su objetivo.
- Crear un entorno donde una red neuronal y una FSM son enfrentadas con el mismo objetivo y comprobar quién cumple más veces con él.
- Crear un entorno donde dos redes neuronales son enfrentadas con el mismo objetivo y comprobar quién cumple más veces con él.

1.3 Impacto esperado

El asset puede ser utilizado por toda la comunidad de Unity. Aquellas personas que quieran implementar un juego en Unity y necesiten inteligencia artificial para programar el comportamiento de algún NPC, son posibles usuarios del producto. Se incluyen programadores de videojuegos aficionados y profesionales. Una empresa con un videojuego ya implementado podría migrar su sistema de inteligencia artificial a redes neuronales y vender su título como “implementado con redes neuronales”.

También los jugadores de videojuegos se verán afectados si los desarrolladores de los mismos utilizan este asset, ya sea tal cual se termine o con adaptaciones a las necesidades de los estudios, para crear la inteligencia artificial de sus juegos. Podrían crear una FSM o un BT sencillos, que sería aprendido por una red neuronal para que ésta no empiece de cero. Después, la red neuronal seguiría aprendiendo de los testers del juego. Cada cierto tiempo, se podría guardar el estado de la red neuronal para crear diferentes niveles. Cuanto más haya aprendido la red neuronal, mejor conseguirá su objetivo y más difícil se lo pone al jugador. Finalmente, una vez el juego esté siendo vendido y las personas lo jueguen, la red neuronal puede seguir aprendiendo de los jugadores para adaptarse a sus habilidades.

1.4 Metodología

Para realizar el trabajo se siguen una serie de pasos que llevan desde la idea inicial hasta su implementación y pruebas.

Primero de todo, un estudio de las técnicas de inteligencia artificial en el mundo de los videojuegos ha sido llevado a cabo. Este paso es importante para conocer el valor que aporta el proyecto a la tecnología existente.

A continuación, se ha realizado una especificación de requisitos que sirve de guía para el desarrollo posterior.

Una vez están claros los requisitos del producto, se analizan cuáles son los posibles riesgos y se preparan respuestas a los más importantes para cumplir con los objetivos en el tiempo disponible.

El siguiente paso ha sido analizar posibles tecnologías que sirven de soporte a GAIA para implementar redes neuronales y escoger una de ellas.

Como penúltimo paso, se ha definido la arquitectura y el diseño de clases del software a crear, para pasar a continuación a su implementación.

El último paso es la realización de las pruebas y el análisis de los resultados obtenidos.

1.5 Estructura

El capítulo 1 de la memoria, Introducción, es un contexto que ayuda a entender la motivación de este trabajo. En él, se explica la importancia y el impacto que podría suponer desarrollar el producto. Se trata de un capítulo de toma de contacto con el tema propuesto que establece unos objetivos claros que se usarán para comprobar si el trabajo ha sido finalizado con éxito.

En el capítulo 2, Estado del arte, se exponen las técnicas de inteligencia artificial más usadas en la industria de los videojuegos, explicando cuáles son sus ventajas e inconvenientes. Para finalizar el capítulo, se presenta una propuesta de tecnología que incorpora valor al estado del arte actual.

A continuación, en el capítulo 3, Especificación de requisitos software, se especifican los requisitos funcionales del sistema mediante diagramas de clase, se indican cuáles son los potenciales usuarios del asset, y los límites del proyecto, ya que los recursos son limitados.

El siguiente capítulo es el 4, Análisis del problema, donde se hace un análisis de los posibles riesgos para tenerlos bajo control y evitar que su materialización no impida terminar el trabajo a tiempo. Además, se analizan las posibles soluciones al problema plan-

teado y se elige la considerada más oportuna.

El capítulo 5, Solución Propuesta, contiene el plan de trabajo llevado a cabo, junto con la arquitectura y diseño del software a implementar. En este punto donde la tecnología a emplear ya ha sido elegida, se plantea con detalle la solución y se explica cómo ha sido el desarrollo de la misma.

Posteriormente, en el capítulo 6, Implantación, se explica cómo adquirir el software, su instalación y las pruebas que se han realizado con él junto con los resultados obtenidos.

Finalmente, el capítulo 7, Conclusiones, se explica si se han conseguido los objetivos y mi experiencia con el desarrollo del asset.

CAPÍTULO 2
Estado del arte

Existen técnicas de inteligencia artificial que son utilizadas actualmente para modelar el comportamiento de los NPCs. En este capítulo se repasan las técnicas más populares a día de hoy en este sector y se finaliza con una crítica al estado de las técnicas actuales y una propuesta de mejora.

2.1 Máquinas de estados finitos

Desde hace muchos años que se vienen utilizando las FSM para modelar el comportamiento de NPCs en videojuegos [10], [11], [12] y [13]. Se trata de una técnica que sigue en uso a día de hoy, consiguiendo captar la atención y el tiempo de los jugadores. Como se detalla en el Libro Blanco del Desarrollo Español de Videojuegos 2022[1], a la industria del videojuego no le ha ido nada mal en 2022, con unos ingresos de más de 180 mil millones de euros utilizando, en muchos casos, FSM para modelar los comportamientos de los NPCs.

Es tan importante esta técnica, que incluso uno de los motores de videojuegos más conocidos, Unity, la tiene implementada para que los diseñadores la utilicen a la hora de gestionar las animaciones de todos los elementos que precisen de ellas. Y por si no es suficiente el uso de FSMs clásicas, existen varios tipos de máquinas [10] que logran una amplia gama de comportamientos. Entre ellas se encuentran las máquinas jerárquicas [14], las máquinas probabilistas [15], las máquinas basadas en pilas [16], las máquinas inerciales [16] y las máquinas concurrentes [16].

La FSM clásica o determinista, también llamada autómata finito se define como como la quíntupla $(Q, \sum, \delta, q_0, F)$ [17] donde

- Q es el conjunto finito y no vacío de estados.
- $\sum$ es el alfabeto de entrada.
- $\delta : Qx\sum \rightarrow Q$ es la función de transición.
- q_0 es el estado inicial.
- $F \subseteq Q$ es el conjunto de estados finales.

[1] https://www.dev.org.es/es/publicaciones/libroblancodev2022

La máquina de estados siempre se encuentra en un estado q perteneciente a Q. Inicialmente, el estado de la máquina es q_0. Desde un estado cualquiera q y dado un símbolo a contenido en el alfabeto de entrada $\sum$, la máquina de estados cambia al estado s si y solamente si $s = \delta(q, a)$. Finalmente, el conjunto de estados finales se utiliza para las máquinas aceptoras, diseñadas para aceptar un lenguaje. Si al acabar de procesar una palabra, la máquina de estados se encuentra en un estado final, entonces la palabra pertenece al lenguaje aceptado por la máquina.

Otra manera de representar las FSMs es mediante un diagrama de transición entre estados, como se muestra en la Figura 2.1. Un diagrama de transición entre estados es un grafo dirigido donde los vértices son los estados de la máquina y las aristas son las transiciones entre dichos estados, etiquetadas con el símbolo necesario para cambiar entre los dos estados que une. Además, el estado inicial se indica con una flecha entrante que no viene de otro estado, como la que tiene el estado S2, y los estados finales se muestran con dos círculos concéntricos con un diámetro similar, como es el estado S3.

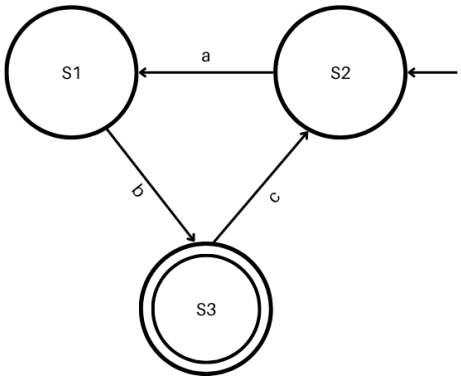


Figura 2.1: Diagrama de transción de estados de una FSM determinista simple. Los vertices son los estados, las aristas son las transiciones entre los estados y las etiquetas de las aristas son los símbolos que producen las transiciones, respectivamente.

Desde un punto de vista práctico enfocado a la implementación de inteligencia artificial para un NPC, una FSM es una colección de estados y transiciones entre esos estados. Los estados representan las diferentes situaciones en las que un NPC se puede encontrar. Además, el NPC realiza acciones en función del estado en el que se encuentra. Las transiciones indican cuándo es necesario cambiar de un estado a otro. Un ejemplo de máquina de estados que implementa un comportamiento similar al de Blinky, el fantasma rojo de Pac-Man, en la versión de Pac-Man 256 para dispositivos móviles, se muestra en la figura 2.2. Quien esté interesado en saber más sobre el comportamiento de los fantasmas de este juego debe saber que cada fantasma de Pac-Man tiene su comportamiento único que puede ser encontrado fácilmente en Internet.

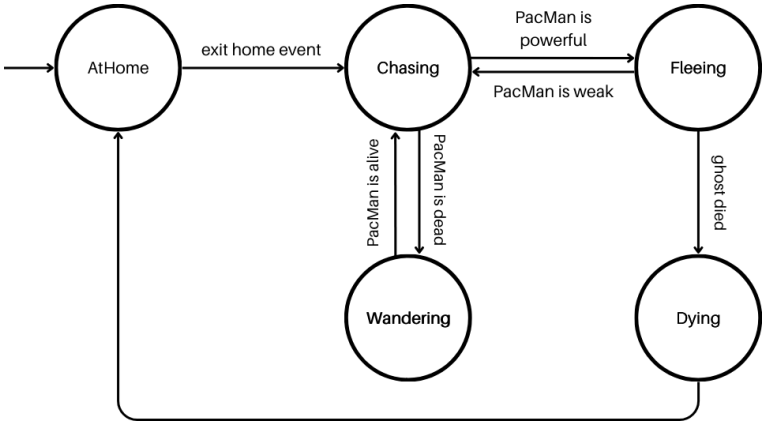


Figura 2.2: Diagrama de transción de estados de una FSM que implementa la inteligencia artificial de Blinky, el fantasma rojo de Pac-Man 256.

El diagrama de la Figura 2.2 puede leerse de la siguiente manera. Blinky comienza en el estado *AtHome* y se mantiene en él hasta que se da un evento que le indica que debe salir de su zona de aparición. Este evento podría darse con un temporizador, cuando haya pasado un cierto tiempo establecido. Una vez sale de su zona de aparición, cambia de estado a *Chasing*. En este estado, Blinky va a moverse por el mapa en busca de PacMan. En el estado *Chasing* pueden ocurrir dos cosas. Que Blinky se coma a PacMan o que PacMan se vuelva poderoso al comerse una píldora de poder. En caso de que Blinky se coma a PacMan, cambiará su estado a *Wandering*. En este estado, Blinky se mueve libremente por el mapa hasta que PacMan revive y vuelve al estado *Chasing*. Si desde el estado *Chasing*, PacMan se come una píldora de poder, Blinky pasará al estado *Fleeing* y huirá de PacMan para que no lo devore. El fantasma vuelve al estado *Chasing* cuando el efecto de la píldora se acaba (PacMan está débil). También puede pasar al estado *Dying* si PacMan se lo come. Desde este estado, el fantasma se convierte en dos ojos y se mueve por el mapa de vuelta a la zona de reaparición. Cuando llega allí, pasará al estado *AtHome*, donde esperará a que le llegue un evento que le indique que puede moverse.

Aunque en el ejemplo mostrado la FSM es suficiente para modelar a Blinky, las FSM tienen algunos inconvenientes. Entre ellos se encuentran la predictibilidad y el rápido aumento de estados cuando la complejidad aumenta. Un ejemplo de máquina de estados finitos más grande en Unity puede verse en la Figura 2.3. Es por ello que se hace necesario buscar nuevas técnicas que permitan modelar el comportamiento de un NPC más fácilmente.

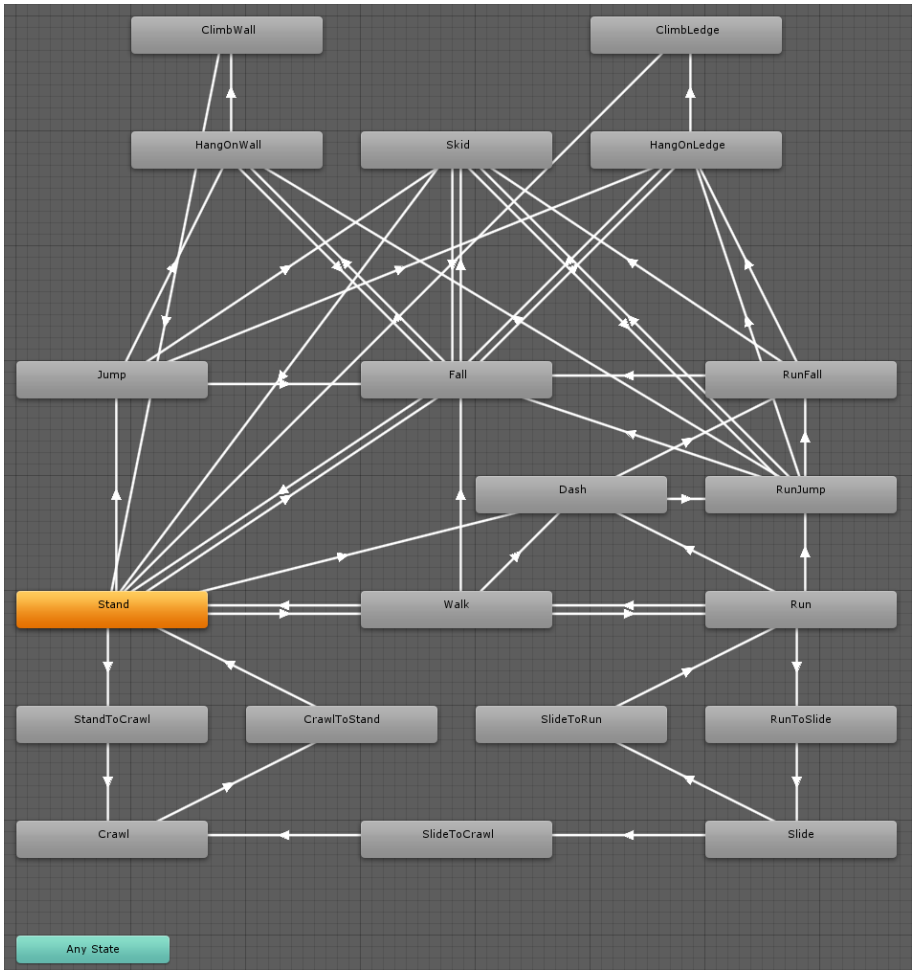


Figura 2.3: Posible máquina de estados finitos[2] utilizada para implementar la animación de un personaje que puede correr, dashear, colgarse en bordes y escalar paredes entre otras mecánicas, sin incluir mecánicas de ataque.

Una búsqueda de herramientas que facilitan la creación de máquinas de estados en Unity lleva a los siguientes resultados.

- **Mecanim**[3] (gratuito). El sistema de animaciones de Unity. Mecanim viene integrado en el editor y tiene una interfaz gráfica que ayuda a visualizar la FSM. Desde la propia interfaz es posible crear estados y transiciones entre los estados en función de una condición. Las condiciones de transición van a depender del valor de ciertas variables de la propia FSM que van a ser actualizadas por la entidad en Unity que tenga la FSM asociada. Un ejemplo es la FSM de la Figura 2.2. Cada estado de la FSM podría tener una animación asignada. Además, la FSM tendría una variable *pacmanPowerful* con dos posibles valores (verdadero y falso). Cuando el fantasma de Pac-Man se entera de que Pac-Man se ha comido una píldora de poder, actualizará la variable de su FSM a verdadero. De esa manera, en el siguiente paso de ejecución, la FSM cambiará del estado *Chasing* al estado *Fleeing*. Cuando el efecto de la píldora se acaba, el fantasma vuelve a actualizar la variable a falso y volverá al estado *Chasing*. Una de las características importantes en Mecanim, que es necesaria para FSMs que gestionan las animaciones, es la posibilidad de mezclar dos

[2] https://answers.unity.com/questions/354357/how-complex-should-mecanim-state-machines-be.html

[3] https://docs.unity3d.com/Manual/AnimationOverview.html

estados mientras se transita de uno a otro. Esto hace que las transiciones sean fluidas y no haya saltos entre ellas. Otra funcionalidad útil es el soporte de crear FSMs jerárquicas, donde un estado puede ser toda una FSM. Esta funcionalidad permite hilar más fino. Por ejemplo, hace posible crear un estado que sea correr y, dentro de él, en función de si el personaje está corriendo hacia delante, hacia detrás o hacia los lados, se ejecutará una animación u otra. Existen más detalles interesantes que se pueden ver en la documentación de Unity.

- **UnityHFSM**[4] (gratuito). Es un asset de código abierto que se puede añadir a cualquier proyecto en unos cuantos clics. Como su nombre indica, es una implementación de una FSM jerárquica. Es simple, intuitiva y fácil de utilizar. Tan solo es necesario crear una FSM y añadirle sus estados y transiciones. Los estados tienen tres funciones que se ejecutan en diferentes momentos, cuando se entra, mientras se está dentro y al salir del estado. Una manera de crear los estados es pasando referencias a esas tres funciones en el constructor de un estado. Para aportar más flexibilidad, UnityHFSM permite crear estados personalizados propios sin necesidad de "ensuciar" el fichero donde se crea la FSM. Esto se hace extendiendo la funcionalidad de un estado base. Por otro lado, las transiciones se crean pasando una función que, al ejecutarse, su resultado es sí o no, indicando si la transición debe realizarse. Además, permite transiciones donde el estado de origen puede ser cualquiera.

- **Finite State Machine System**[5] (de pago). Otro asset de Unity pero, en este caso, de pago y accesible a través de la Unity Asset Store. En este caso, no cuenta con soporte para la implementación de máquinas de estados jerárquicas y, aunque también es sencillo de utilizar, tiene menos flexibilidad que UnityHFSM. Para crear un nuevo estado, es necesario crear una nueva clase. Las condiciones no son funciones, sino clases que implementan la función *CheckCondition*. No es posible crear transiciones y añadirlas a la FSM, sino que hay una función con el largo nombre *CreateFSMStateToAnotherFSMStateTransition* a la que hay que pasarle los nombres del estado origen y destino junto con la condición de transición. No obstante, el resultado práctico es el mismo, y también permite crear transiciones desde cualquier estado.

- **GAIA**[6] (gratuito). GAIA es un asset de Unity creado en la Universitat Politècnica de Valencia [10] que podemos utilizar para la creación de máquinas de estados. Esta vez, en contraste con las demás herramientas, en GAIA es posible escribir en lenguaje XML la especificación de las FSMs, por lo que no es necesario hacerlo por código. Casi todo está gestionado por etiquetas. El nombre de los estados, las acciones de entrada, actualización y salida de los estados, el nombre de las transiciones y el nombre de los eventos que habilitan las transiciones son todo etiquetas. En el código, las etiquetas se traducen a enumeraciones y con ellas se gestiona la máquina. En GAIA, las transiciones entre estados también se hacen cuando se cumplen ciertas condiciones. La diferencia con otras herramientas es que, el cumplimiento de las condiciones genera eventos que llevan a un cambio de estado. Esto se hace para poder especificar las condiciones fácilmente en XML como etiquetas. Otra característica de GAIA es que permite la definición de varios tipos de máquinas de estados: determinista, probabilista, inercial, basada en pilas o de estados concurrentes.

[4] https://github.com/Inspiaaa/UnityHFSM
[5] https://assetstore.unity.com/packages/tools/ai/finite-state-machine-system-108051
[6] https://github.com/tetracube/GAIA

2.2 Árboles de comportamiento

Un árbol de comportamiento (BT) es un árbol cuyos nodos internos son *nodos de control de flujo* y los nodos hoja son *nodos de ejecución* [9, 18]. Cuando el BT es ejecutado, lo hace desde su nodo raíz hacia los nodos hoja, a través de una señal llamada *tick* que se ejecuta y se propaga hacia los hijos con una frecuencia determinada. Una vez un nodo empieza su ejecución, envía a su padre una señal de *Running*, indicando que se ha iniciado su ejecución. Cuando termina, envía una señal de *Success* o *Failure* en función de si el nodo se ha ejecutado con éxito o no.

Típicamente se utilizan cuatro tipos de nodos de control de flujo y dos tipos de nodos de ejecución, aunque el árbol de comportamientos puede expandirse. En la Tabla 2.1 se resumen los distintos tipos de nodos.

Un nodo Fallback redirige la señal de *tick* a sus hijos de izquierda a derecha. Si alguno de sus hijos responde *Running* o *Success*, el nodo Fallback devuelve la misma respuesta a su nodo padre. En este caso, el nodo Fallback deja de enviar *ticks* a sus hijos. Cuando un hijo responde *Failure*, el nodo Fallback continúa enviando señales *tick* a sus hijos en orden.

Un nodo Sequence redirige la señal de *tick* a sus hijos de izquierda a derecha hasta que uno de sus hijos responde *Running* o *Failure*. En ese momento, el nodo Sequence redirige la respuesta de su hijo hacia su padre y no continúa enviando *ticks*. Cuando un hijo responde *Success*, el nodo Sequence sigue enviando señales de *tick* a sus hijos en orden, hasta que todos los hijos respondan *Success*, momento en que el nodo Sequence hará lo mismo.

Un nodo Parallel redirige la señal de *tick* a todos sus hijos. Después de que todos los hijos hayan hecho su trabajo, Parallel responde *Success* si al menos M hijos responden *Success*, donde $M < N$ es un límite que determina el diseñador y N es el número de hijos del nodo. Responde *Failure* si $N - M + 1$ hijos devuelven *Failure*. En cualquier otro caso, responde *Running*.

Un nodo Action, al recibir un *tick*, ejecuta código y responde *Success* o *Failure* en función de si la acción se ha completado correctamente o no. Mientras se ejecuta, el nodo responde *Running*.

Un nodo Condition que recibe un *tick*, comprueba si una proposición es verdadera o falsa. Responde *Success* si es verdadera y *Failure* si es falsa.

Un nodo Decorator es un nodo de control de flujo con un único hijo. Este nodo puede modificar la señal que devuelve su nodo hijo o decidir cuándo el nodo hijo debe recibir la señal *tick*. Por ejemplo, si es una tarea que puede realizarse cada varios *ticks*, el nodo Decorator puede pasarle la señal de *tick* cada 5 *ticks* recibidos del nodo padre.

Además de los diferentes nodos, una característica que suelen tener los BT es la *pizarra*. La pizarra es un espacio en memoria con información que comparten los nodos del árbol. Los nodos pueden modificar esas variables o leerlas. Por ejemplo, un nodo de un NPC puede calcular la posición del jugador y guardarla en la pizarra mientras que otro

Tabla 2.1: Tipos de nodos de un árbol de comportamiento.

Tipo de nodo	Símbolo	Éxito	Fracaso	Ejecutando
Fallback	?	Si un hijo tiene éxito	Si todo los hijos fracasan	Si un hijo devuelve Running
Sequence	→	Si todos los hijos tienen éxito	Si un hijo fracasa	Si un hijo devuelve Running
Parallel	⇉	Si por lo menos M hijos tiene éxito	Si más de N - M hijos fracasan	En otro caso
Action	text	Cuando termina	Si no se completa	Mientras se ejecuta
Condition	text	Si es verdadera	Si es falsa	Nunca
Decorator	◊	Personalizado	Personalizado	Personalizado

nodo lee esa información para mover al NPC hacia el jugador.

Poniendo el mismo ejemplo que en la Sección 2.1, podríamos implementar la inteligencia artificial de Blinky con el BT de la Figura 2.4

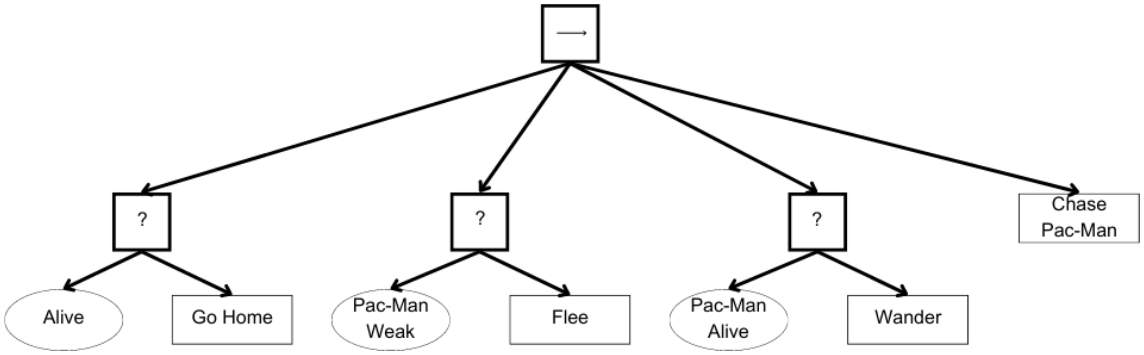


Figura 2.4: Posible BT que implementa una inteligencia artificial similar a la de Blinky, el fantasma rojo del juego Pac-Man 256.

Este BT se lee de la siguiente manera. El nodo raíz es un nodo Sequence que va a ir enviando *ticks* a sus nodos hijos de izquierda a derecha. Primero, a su hijo situado más a la izquierda, un nodo Fallback. Este hijo envía la señal de *tick* a su hijo izquierdo Alive, que comprueba si el fantasma que tiene el BT asociado está vivo o no. Si el fantasma está vivo, entonces el nodo Fallback responde con *Success* y el nodo raíz envía una señal de *tick* a su siguiente nodo Fallback. El segundo nodo Fallback comprueba si Pac-Man está débil. Ésto lo hace dirigiendo la señal de *tick* a su nodo Condition Pac-Man Weak. Este nodo comprueba si Pac-Man está débil. Si Pac-Man no está débil (se ha comido una píldora de poder), el nodo Condition responde *Failure*, y su nodo padre dirige la señal de *tick* al siguiente nodo hijo, el nodo de tipo Action llamado Flee. Este nodo ejecuta código que aleja al fantasma de Pac-Man, hace que huya para no ser devorado. Así, podemos continuar leyendo el árbol de izquierda a derecha.

También podemos leer el árbol desde un nivel más alto de abstracción. El fantasma, lo primero que hace es comprobar si está vivo. Si no lo está, vuelve a casa. Si está vivo, comprueba si Pac-Man está débil. Cuando Pac-Man se ha comido una píldora de poder, entonces huye de él. Si Pac-Man está débil, comprueba si Pac-Man está vivo. Si Pac-Man está muerto, el fantasma pasea por el mapa sin rumbo. Si está vivo, le persigue. Desde

este nivel de abstracción, el comportamiento se entiende en menos tiempo y es más sencillo recordarlo.

Para más información sobre cómo crear y estructurar árboles de comportamiento, Damian Isla, encargado de implementar la inteligencia artificial del conocido videojuego Halo 2, ha escrito un artículo llamado *GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI*[7]. En este artículo, Damian enseña los cuatro principios que utilizó para crear la inteligencia artificial de Halo 2, y es un ejemplo del potencial que tiene esta técnica para aplicarse a juegos AAA.

De nuevo, algunas de las alternativas diseñadas para crear BTs en Unity son la siguientes.

- **Behavior Designer**[8] (de pago). Es un paquete de Unity que puede comprarse en la Asset Store por aproximadamente 90€. Tiene mucha documentación y ejemplos de cómo usarse. Viene con una interfaz gráfica que puede ser abierta dentro de Unity, donde se muestra el BT en creación de forma visual. Ahí se pueden visualizar los nodos de control de flujo, los nodos de ejecución y sus conexiones. Behavior Designer viene con un conjunto considerable de nodos de control de flujo y ejecución implementados que se pueden utilizar sin escribir código. Además, está integrado con Unity y se pueden seleccionar nodos que ejecutan funcionalidad ya implementada en el motor. Y si los nodos que trae no fueran suficientes, es posible extender la funcionalidad creando nuevos nodos que se ajusten a las necesidades del comportamiento. Otra característica interesante es que la interfaz muestra el BT creado en ejecución mientras el juego está activo. Los nodos que se ejecutan se marcan de color verde, mientras que aquellos que no se están ejecutando serán pintados de color gris. Existen más detalles interesantes que pueden explorarse en su página web oficial.
- **Bonsai Behaviour Tree**[9] (gratuito). Herramienta de código abierto disponible en GitHub. Para utilizarlo, simplemente se descarga el repositorio y se copia el directorio raíz al proyecto. Es similar a Behavior Designer en cuanto a la interfaz, pero viene con menos nodos predefinidos. Aunque no existen nodos que ejecutan funcionalidad de Unity, también permite crear nodos personalizados extendiendo la funcionalidad de un nodo base. Desde su propia interfaz, se pueden editar los árboles de comportamiento, añadiendo y eliminando nodos de forma gráfica, además de poder ver la ejecución del árbol mientras el juego se ejecuta (aunque esta última funcionalidad no se ejecuta correctamente todo el tiempo).
- **Fluid BT**[10] (gratuito). Otra herramienta de código abierto disponible en GitHub. A diferencia de los dos mencionados, los árboles se crean por código, así que no existe un editor gráfico. No obstante, es posible visualizar el árbol mientras se está ejecutando el juego. Fluid BT viene con algunos nodos predefinidos, aunque estos son pocos. La ventaja que tiene es que no es necesario crear una clase que extiende de la clase nodo para crear una nueva funcionalidad. Si se desea añadir un nodo Action personalizado, es suficiente con llamar a una función a la que se le pasa como argumento el código personalizado. No obstante, también es posible crear nodos personalizados como clases extendiendo de los nodos base.

[7] https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai
[8] https://opsive.com/assets/behavior-designer
[9] https://github.com/luis-l/BonsaiBehaviourTree
[10] https://github.com/ashblue/fluid-behavior-tree

- **NodeCanvas**[11] (de pago). Igual que Behavior Designer, se encuentra en la Unity Asset Store por aproximadamente 90€. Este paquete incluye la opción de crear tanto árboles de comportamientos como FSMs y árboles de diálogos. Como Behavior Designer, NodeCanvas está integrado en Unity y a la hora de crear BTs, existen nodos predefinidos que ejecutan funcionalidad del motor. Por ejemplo, existen nodos para mover objetos o aplicar fuerzas sobre ellos. También incluye una interfaz gráfica intuitiva y agradable visualmente que sirve para crear los árboles y ver cómo se ejecutan cuando el juego está activo. No solo eso, sino que también permite modificar los árboles de comportamiento mientras el juego se está ejecutando y ver los cambios en tiempo real. El cambio se mantendrá al terminar la ejecución.

- **Panda BT Free**[12] (gratuito). Utilizado por GAIA para implementar los árboles de comportamiento. En Panda BT, los árboles de comportamiento se especifican en un fichero de texto que puede ser modificado desde cualquier editor de texto. La herramienta también viene con algunos nodos predefinidos, aunque son pocos. Panda BT tampoco está integrado con el motor de Unity. Para ayudar a organizar el árbol, cada nodo puede ser un árbol, por lo que es posible desarrollar los niveles de un árbol profundo por separado, entendiendo el árbol a medida que se baja por sus ramas. A diferencia de otras herramientas, los nodos se crean como métodos de un script de C# que se escriben en el objeto que contiene el árbol. Solamente hay que marcar el método con el atributo[13] [Task] y estará disponible en el árbol. Sin una interfaz gráfica, Panda BT muestra la ejecución del BT mientras el juego está activo mediante un código de colores sobre el texto que define el árbol.

- **GAIA** (gratuito). Además de poder crear máquinas de estados finitos, GAIA también tiene soporte para la creación de BTs. La implementación de árboles de comportamiento en GAIA se hace sobre Panda BT. Añade un parser de XML para procesar un documento con la estructura definida por los desarrolladores y lo transforma en texto interpretable por Panda BT, que es el que hace el trabajo.

Aunque se pueda modelar inteligencia artificial muy compleja con BT, todavía es predecible. Un jugador avanzado puede aburrirse si siempre se enfrenta a los mismos enemigos y un jugador novato se frustra si no puede vencer ni al primer enemigo que se encuentra porque es muy difícil. Para modificar la dificultad, pueden crearse enemigos con más puntos de vida, más defensa o más ataque, manteniendo su comportamiento. También puede modificarse su comportamiento creando diferentes BTs para cada dificultad, una tarea ardua. Pero un NPC con un BT asignado, siempre se va a comportar de la misma manera ante las mismas situaciones. ¿Qué alternativas existen, entonces, para que un NPC se comporte de forma menos predecible y adaptándose a su entorno? En resumen, de manera más realista.

2.3 Redes neuronales artificiales

Para solventar las carencias de los métodos mencionados, es posible utilizar redes neuronales. El comienzo de las redes neuronales se remonta a 1943, cuando Warren McCulloch y Walter Pitts publicaron la estructura de una célula del cerebro [19], la neurona (figura 2.5). La neurona McCulloch-Pitts (MCP) es muy sencilla. Ésta recibe varias señales de entrada y, en función de la intensidad de esas señales, emite una señal eléctrica

[11] https://assetstore.unity.com/packages/tools/visual-scripting/nodecanvas-14914
[12] https://assetstore.unity.com/packages/tools/ai/panda-bt-free-33057
[13] https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/

o no. El cálculo realizado para determinar si una neurona transmite una señal o no es una suma ponderada de las diferentes entradas por sus pesos asociados, que indican la importancia de cada conexión de entrada. Finalmente, a ese resultado se le aplica una función de activación. La función de activación más sencilla es la función escalón (2.1). Si el valor de la suma ponderada es mayor o igual a 0, devuelve 1. Devuelve 0 en caso contrario.

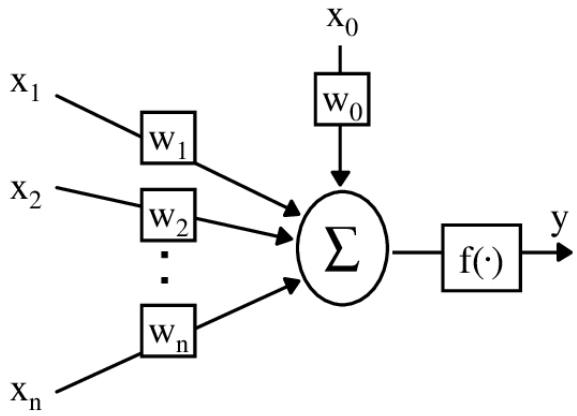


Figura 2.5: Neurona McCulloch-Pitts (MCP).

$$a(x) = \begin{cases} 1, \, si \, x \geq 0 \\ 0, \, si \, x < 0 \end{cases} \tag{2.1}$$

Años más tarde, en 1958, Frank Rosenblatt publicó el primer modelo de red neuronal y lo llamó *Perceptrón* [20]. El psicólogo neoyorquino basó su método en las ideas de McCulloch y Pitts sobre el funcionamiento de las neuronas. Un perceptrón es una máquina capaz de aprender de su entorno, asociando ciertas respuestas a ciertos estímulos de entrada.

La idea básica de un perceptrón se refleja en la Figura 2.6. Se trata de una red neuronal de una sola capa usada en el aprendizaje supervisado de clasificadores binarios, que clasifican una cierta entrada en una clase entre dos posibles.

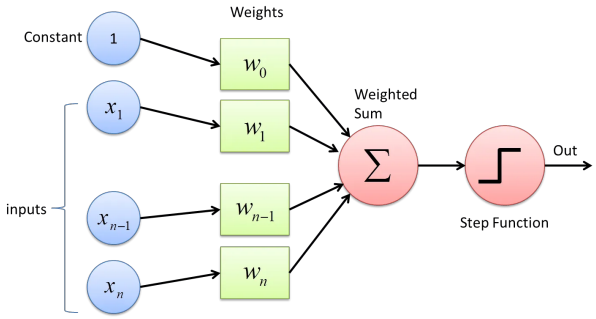


Figura 2.6: Estructura de un perceptrón simple. Fuente: https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53.

Al comparar la Figura 2.5 con la Figura 2.6, se puede ver que son prácticamente iguales. Un perceptrón (el que se conoce hoy en día, no el que describe Rosenblatt en [20]) no es más que una neurona MCP entrenada de manera supervisada para una clasificación

binaria. El entrenamiento consiste en la modificación del vector de pesos w para optimizar la clasificación. Como se ha mencionado anteriormente, una neurona MCP emite o no una señal eléctrica en función de las señales de entrada que recibe. Esto es lo mismo que una clasificación binaria. Poniendo como ejemplo la clasificación de correo spam, si la neurona emite señal, podríamos decir que el mensaje de entrada pertenece a la clase de correo spam, mientras que si no emite señal alguna, no es considerado spam.

En el ejemplo de un detector de correo spam, la entrada al perceptrón podría ser un vector que contiene, por cada elemento, cuántas veces aparece una palabra en el mensaje. Cada palabra está codificada por el índice del elemento del vector de entrada que se le asocia. El índice 0 del vector de entrada podría representar la palabra "Maravillosa". En el procesamiento del mensaje, lo primero que se hace es multiplicar las veces que aparece cada palabra en el texto por el peso correspondiente a esa palabra, w_i. Cuando todas las palabras se han multiplicado por su peso, se suman y se les aplica una función de escalón (2.1). Si el valor del resultado es positivo, la salida es 1 (spam). Si, en cambio, el valor del resultado es negativo, la salida es 0 (no spam). En este caso, cada uno de los pesos indica la importancia de cada palabra para las clases. Cuanto más grande sea el peso, más importancia tiene la palabra. Si el peso es positivo, la palabra es importante en la clase *spam*, mientras que si el peso es negativo, la palabra es importante en la clase *no spam*.

Aunque el modelo surgió en la década de los 60, un avance que ha permitido la evolución y la expansión de las redes neuronales ha sido el algoritmo de aprendizaje para redes neuronales llamado *Backpropagation* [21] y su versión más eficiente *Backpropagation Through Time* (BTT) [22]. Este algoritmo permite calcular la actualización de los pesos de una red neuronal desde las neuronas de la capa de salida hacia detrás. De ahí su nombre.

En cuanto a las herramientas de las que nos provee Unity, actualmente existen varios assets para crear redes neuronales. Entre ellos se encuentran los siguientes.

- **ANN Perceptrón**[14] (de pago). Este asset permite crear, mediante código C#, redes neuronales con dos métodos de aprendizaje: entrenamiento por imitación y generación aleatoria. En el primer caso, la red neuronal aprende de un comportamiento externo, mientras en el segundo caso, aprende a base de realizar acciones de forma aleatoria. Es posible personalizar la red con diferentes parámetros como el número de entradas, salidas, capas, neuronas por capa, velocidad de aprendizaje y error máximo esperado, entre otros. Además, tiene una pequeña interfaz integrada con Unity que permite configurar la red creada y ver su estado, indicando el valor de las entradas, salidas y pesos de las diferentes capas.
- **Artificial Intelligence and Machine Learning Prototyping Kit**[15] (de pago). Este asset está implementado para facilitar la creación de redes neuronales y su entrenamiento mediante algoritmos genéticos. Desde la propia interfaz de Unity se pueden seleccionar los parámetros de entrenamiento del algoritmo genético y también se puede indicar la estructura de la red neuronal mediante el número de entradas, salidas, capas y neuronas por capa. Como punto negativo, no tiene documentación sobre cómo se guardan y se usan las redes neuronales entrenadas.
- **Unity MLAgents**[16] (gratuito). Se trata de un asset de código abierto desarrollado por Unity y que utiliza tanto C# como Python. MLAgents permite la creación y el

[14] https://assetstore.unity.com/packages/tools/ai/ann-perceptron-117766
[15] https://assetstore.unity.com/packages/tools/ai/artificial-intelligence-and-machine-learning-prototyping-k
[16] https://unity.com/products/machine-learning-agents

entrenamiento de agentes inteligentes que utilizan redes neuronales como modelo de aprendizaje. El entrenamiento puede hacerse utilizando *Reinforcement Learning* (aprende por refuerzo) o *Imitation Learning* (aprende de un comportamiento específico, que puede ser un ser humano u otra IA). El asset permite especificar en un fichero de configuración el entrenamiento que va a realizarse, sus hiperparámetros y la arquitectura de la red neuronal. De la arquitectura de la red, se pueden especificar el número de capas, el número de neuronas por capa y, en caso de utilizar una red neuronal recurrente, el tamaño de su estado oculto y el número de experiencias que se utilizan como única entrada a la red neuronal. Un aspecto interesante de esta herramienta es que permite el entrenamiento en la tarjeta gráfica, a diferencia del resto.

- **Noedify - Easy Neural Networks**[17] (gratuito). Se trata de un asset creado por el grupo de ingenieros *TinyAngleLabs*. Está implementado como una librería de enlace dinámico (Noedify.dll). Es sencillo de utilizar y tiene una documentación detallada de su uso. Con Noedify se pueden crear redes neuronales convolucionales o totalmente conectadas en pocas líneas de código, creando una red y añadiendo las capas que se quieran una a una. Se trata de un asset flexible al permitir indicar, para cada capa, su tipo (entrada, salida, totalmente conectada, convolucional, pooling), el número de neuronas y la función de activación de éstas. Las redes neuronales pueden entrenarse en Unity mientras se ejecuta el código o bien, pueden entrenarse en Python con Tensorflow Keras e importarse posteriormente. El entrenamiento en Noedify se realiza utilizando el sistema de trabajos de Unity, que permite ejecutar código en paralelo desde el procesador.

2.4 Más técnicas

Además de las técnicas vistas, a día de hoy existen otras técnicas más avanzadas que permiten crear comportamientos más complejos de Inteligencia Artificial (IA) en varias áreas de un videojuego. De las diez áreas identificadas en [23], la que guarda una relación más estrecha con este trabajo es el comportamiento de los NPC. Las técnicas de IA más utilizadas a la hora de modelar el comportamiento de un NPC son las siguientes.

- **Computación Evolutiva** (Primaria). Se parte de una población inicial de NPCs que van evolucionando y obteniendo mejor rendimiento en la tarea que se les ha encomendado.
- **Aprendizaje por Refuerzo** (Primaria). Las decisiones tomadas por el modelo tienen castigos y recompensas que vienen dadas por el efecto de las acciones sobre el entorno en el que se encuentra la red. Normalmente, el entrenamiento no cesa hasta que el modelo ha llegado a un estado de equilibrio.
- **Aprendizaje Supervisado o asociativo** (Secundaria). Dado un conjunto de datos y sus correspondientes etiquetas, se entrena un modelo que clasifique con el menor error posible. Se diferencia del aprendizaje por refuerzo en que, en este caso, las etiquetas, que indican cuáles deben ser las salidas del modelo dadas unas entradas específicas, tienen un valor correcto conocido, mientras que en el aprendizaje por refuerzo, el *feedback* que recibe la red es un valor numérico que indica cuánto de buena o de mala han sido las acciones que ha realizado.

[17] https://assetstore.unity.com/packages/tools/ai/noedify-easy-neural-networks-161940

Estas técnicas no tienen por qué ser disjuntas, se pueden utilizar varias al mismo tiempo. Un caso usual es la creación de redes neuronales que son entrenadas mediante algoritmos evolutivos.

También, en el artículo se mencionan las técnicas de IA más utilizadas para modelar el comportamiento de un jugador.

- **Computación Evolutiva** (Primaria).
- **Aprendizaje Supervisado** (Primaria).
- **Aprendizaje no Supervisado** (Primaria). Se trabaja con datos sin respuesta conocida y se pretende conocer alguna propiedad de ellos. Por ejemplo, la técnica de clustering agrupa los datos de manera que los elementos del mismo grupo sean más parecidos entre ellos que los elementos de diferentes grupos.

2.5 Crítica al estado del arte

Como ya se ha mencionado, a día de hoy existen bastantes técnicas de inteligencia artificial, y algunas de ellas se utilizan en la industria de los videojuegos. Uno de los usos en el mundo de los videojuegos es el modelado de comportamientos de NPCs. En este tema, las técnicas más utilizadas son las FSMs y los BTs. No obstante, en los últimos años, las redes neuronales han ido cogiendo fuerza y se han ido introduciendo en el mundo de los videojuegos.

2.5.1. Tecnología de redes neuronales en Unity

Ya se han mencionado cuatro tecnologías de redes neuronales para Unity en la sección 2.3. De entre todas ellas, la más avanzada y con más funcionalidad es MLAgents, que como ventaja con respecto al resto, permite el entrenamiento de redes neuronales haciendo uso de algoritmos de aprendizaje por refuerzo. Este entrenamiento hace posible que un NPC aprenda a base de recompensas y castigos sin tener especificado un comportamiento determinista que deba seguir. De este modo, el entrenamiento se hace guiado por un objetivo, y un NPC puede cambiar su comportamiento para volverse más eficiente en la consecución de ese objetivo, que puede ser múltiple. No obstante, su instalación y puesta en marcha es retorcida, ya que requiere de dos entornos, uno en Unity y otro en Python, que se comunican entre ellos, con unas versiones específicas de sus dependencias.

2.5.2. Tecnología desarrollada en la UPV

En cuanto a la tecnología desarrollada en la Universitat Politècnica de València (UPV), en Septiembre de 2014, José Alapont [10] implementó las bases de GAIA, diseñando cuidadosamente su arquitectura y facilitando el uso de FSMs en Unity mediante especificaciones XML. Años más tarde, en Abril de 2021, Jorge Andreu [24] incorporó en GAIA los BTs. De la misma forma que para las FSMs, se crea un XML en el que se especifica la estructura del BT. Este XML es procesado para crear el BT en tiempo de ejecución. Y hasta hoy, este es el estado del asset.

Por lo tanto, GAIA soporta las dos técnicas de inteligencia artificial más utilizadas mundialmente a la hora de modelar el comportamiento de NPCs. Podría ser suficiente para crear todo tipo de comportamientos, pero sería incluso mejor si tuviese soporte para redes neuronales. Esto permitiría que un NPC utilizase una red neuronal para decidir qué acciones llevar a cabo en función de su entorno. Y no solo eso, si no que también sería capaz de aprender nuevas estrategias para comportarse de manera óptima a base de jugar con jugadores humanos, entendiendo "óptima" por aquella que cumple los objetivos que los desarrolladores diseñan. Así, los jugadores disfrutarían más de los juegos porque el comportamiento de los NPCs sería más realista, más improvisado, sin ser muy fácil ni muy difícil y manteniendo al jugador en un estado de concentración y con la sensación de que forma parte del videojuego por más tiempo.

2.6 Propuesta

Lo que pretende este trabajo, es añadir el soporte de redes neuronales en GAIA. Las redes neuronales se especificarán mediante ficheros XML y podrán aprender de otros comportamientos artificiales programados o del comportamiento de un ser humano controlando una entidad. Esto va a permitir partir de una implementación sencilla del comportamiento que se desea simular y obtener un modelo más complejo, que será capaz de aprender el estilo de juego de un jugador y adaptarse a sus habilidades. La idea detrás del proyecto es facilitar la creación de inteligencia artificial basada en redes neuronales en Unity.

Para lograrlo, se hará uso de las tecnologías que existen de redes neuronales y se implementará una capa de abstracción sobre ellas, de forma que pueda cambiarse la tecnología sobre la cual se ayudará GAIA. No se pretende inventar una tecnología, sino facilitar su uso.

CAPÍTULO 3
Especificación de requisitos software

Lo que va a poder hacer un usuario con el asset desarrollado se describe en este capítulo mediante una especificación de requisitos basada en el conocido estándar ISO/IEC/IEE 29148:2018 [25].

A través de la especificación de requisitos software, se establece un enlace entre dos dominios, el del cliente/usuario y el del proveedor/desarrollador. Aquí es donde las necesidades se transforman en los requisitos que el producto va a cumplir y en cuáles son sus limitaciones desde el punto de vista de la solución a las necesidades. En este caso, las necesidades son los objetivos. Con la especificación de requisitos establecida, en caso de existir un cliente que haya firmado el documento, no podrá pedir cambios en el sistema si los desarrolladores no los aceptan. Para este proyecto, no existe un cliente, y la especificación servirá como guía en las siguientes fases del proyecto. Además, sirve a los futuros usuarios del producto a entender cuál es su funcionalidad y sus restricciones.

3.1 Propósito

El propósito del producto a desarrollar es satisfacer la necesidad de soportar redes neuronales para modelar comportamientos de NPCs en GAIA.

Actualmente, GAIA tiene soporte para FSMs y BTs. No obstante, la tecnología sigue avanzando y surgen nuevas técnicas. Se pretende actualizar GAIA para que sus actuales y futuros usuarios tengan acceso a una nueva tecnología de forma sencilla.

Este complemento va a soportar el entrenamiento de una red neuronal a partir de comportamientos ya implementados o de un ser humano. Una vez el entrenamiento ha terminado y la red cumple con el comportamiento que se desea imitar en un porcentaje establecido, un usuario podrá programar que la red neuronal siga aprendiendo de los jugadores humanos. Por lo tanto, el sistema creará redes neuronales a partir de una especificación, las entrenará basándose en la información que recibe y en el resultado que deberían devolver, y las ejecutará para modelar el comportamiento de un NPC.

3.2 Ámbito del sistema

GAIA es un asset de inteligencia artificial para Unity. Añadir soporte para redes neuronales significa que los usuarios de GAIA podrán crear comportamientos para NPCs que sean controlados por un modelo de inteligencia artificial complejo sin necesidad de conocer en profundidad esta técnica.

Utilizar redes neuronales para modelar comportamientos permite crear NPCs más inteligentes, donde inteligentes significa realistas, más cercanos a un comportamiento humano, aleatorios y que consiguen su objetivo más eficazmente. Tener NPCs más inteligentes en un videojuego supone una mejora en la experiencia de los usuarios que sentirán más entretenimiento y aprenderán más jugando.

3.3 Características de los usuarios

Los usuarios de GAIA son usuarios del motor de videojuegos Unity que necesiten implementar técnicas de inteligencia artificial para modelar el comportamiento de NPCs. Los usuarios pueden estar comenzando en el mundo de los videojuegos o ser expertos en él y necesitan un conocimiento básico de redes neuronales para sacar el máximo potencial de la herramienta. Por lo tanto se trata de un nicho a nivel mundial y en un sector (videojuegos) con altos ingresos.

3.4 Restricciones

Las restricciones impuestas sobre el sistema son:

- La frecuencia y el tiempo de respuesta de una red neuronal permiten mantener una tasa de refresco de la pantalla de mínimo 30 imágenes por segundo.
- El entrenamiento se podrá hacer tanto en la CPU como en la GPU.
- La conversión de entradas y salidas de una máquina de estados finitos a una red neuronal no es automática, la hace el usuario.
- El usuario no puede especificar el número de neuronas en cada capa individualmente ni puede especificar el tipo de cada una de las capas de la red neuronal.
- El usuario no tendrá acceso al estado de la red neuronal durante el entrenamiento. No verá cuáles son los pesos de los enlaces entre las neuronas ni las entradas y salidas de cada neurona.

3.5 Asunciones y dependencias

Para el correcto funcionamiento de GAIA se asume que:

- El ordenador de un usuario no se apaga durante el entrenamiento de una red neuronal.
- El ordenador de un usuario cumple con los requisitos mínimos para ejecutar el editor de Unity y un juego, tanto en el editor como exportado.

3.6 Requisitos funcionales

Para describir los requisitos funcionales, se procede a utilizar diagramas de casos de uso mediante el lenguaje de modelado UML (*Unified Modeling Language*) [26].

Primero, se exponen cuáles son los actores que interactúan con el sistema y después se pasa a enunciar los casos de uso mediante los diagramas UML y con la descripción de cada uno en formato tabla.

3.6.1. Actores

Los actores del sistema son aquellas entidades no pertenecientes a este pero que interactúan con él. Los actores especifican los roles que toman las entidades a la hora de la interacción. A continuación se describen todos los actores.

Cualquier usuario de GAIA va a poder realizar las mismas operaciones con la herramienta, por lo que no existen diferentes roles a la hora de interactuar con ella. Entonces, solo tenemos un único actor.

Tabla 3.1: Actor - Usuario de Unity.

Actor	ACT. 1 Usuario de Unity
Descripción	Persona que utiliza Unity

3.6.2. Casos de uso

Los casos de uso de la nueva funcionalidad de GAIA vienen representados visualmente en la Figura 3.1.

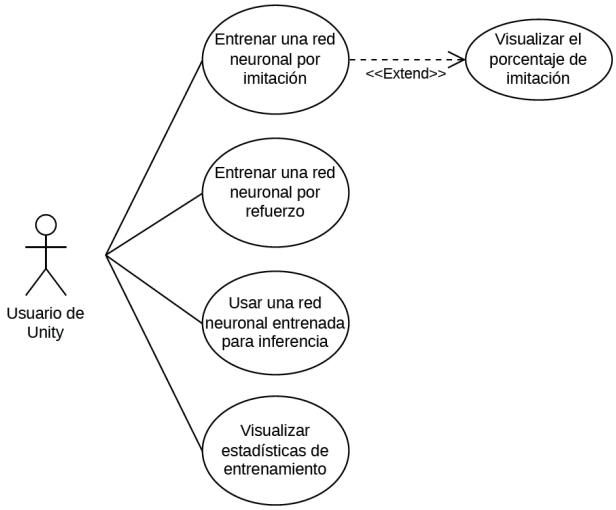


Figura 3.1: Diagrama de casos de uso.

Ahora, para entender mejor qué hace cada caso de uso, se escriben sus descripciones en tablas con toda la información necesaria para saber cuál va a ser el funcionamiento del sistema.

Tabla 3.2: Caso de uso - Entrenar una red neuronal por imitación.

Caso de uso	CU.1 Entrenar una red neuronal por imitación
Actor	ACT.1 Usuario de Unity
Descripción	Un Usuario de Unity entrena una red neuronal por imitación
Flujo básico	1. **Iniciar entrenamiento**. El caso de uso se inicia cuando un usuario le da al botón de Play en Unity después de crear un nivel preparado para llevar a cabo un entrenamiento por imitación y de especificar los parámetros del entrenamiento. 2. **Verificar especificación**. El sistema lee la especificación del usuario y comprueba que su estructura y sus valores son correctos. 3. **Crear red neuronal**. El sistema crea una red neuronal a partir de un nombre que el usuario ha indicado con anterioridad y que debe coincidir con el nombre que aparece en la especificación. 4. **Ejecutar entrenamiento**. El sistema ejecuta el nivel y entrena la red neuronal creada en base a la información que recibe como entrada y la acción que debería de haber ejecutado, aquella que se desea imitar. El entrenamiento cesa cuando el porcentaje de imitación del comportamiento no mejora durante x iteraciones del entrenamiento, siempre y cuando el porcentaje máximo obtenido haya superado el umbral, y las x iteraciones también lo hayan hecho. La ejecución también puede ser finalizada por el usuario.
Flujo alternativo	1. **Error en la verificación de la especificación**. En el paso 2 del flujo básico, el sistema no encuentra algunos de los campos en la especificación o encuentra un error en sus datos. El sistema utiliza los parámetros de entrenamiento por defecto. El sistema va al paso 4 del flujo básico con los datos por defecto. 2. **Comportamiento no encontrado**. En el paso 3 del flujo básico, el sistema no encuentra ninguna especificación con el nombre que el usuario ha indicado. El sistema muestra por pantalla un mensaje indicando al usuario lo sucedido.
Pre-condiciones	-
Post-condiciones	Se crea una red neuronal entrenada en el sistema de ficheros del usuario

Tabla 3.3: Caso de uso - Visualizar porcentaje de imitación.

Caso de uso	CU.2 (extiende CU.1) Visualizar porcentaje de imitación
Actor	ACT.1 Usuario de Unity
Descripción	Un Usuario de Unity visualiza el porcentaje de imitación del entrenamiento que se está llevando a cabo
Flujo básico	1. **Mostrar porcentaje de entrenamiento**. El caso de uso se inicia durante el paso 2 del flujo básico del caso de uso 1. El sistema, mientras entrena la red neuronal, muestra por pantalla y escribe en un fichero de texto cuál es el porcentaje de imitación junto con la hora en que se ha calculado.
Flujo alternativo	-
Pre-condiciones	-
Post-condiciones	-

Tabla 3.4: Caso de uso - Entrenar una red neuronal por refuerzo.

Caso de uso	CU.3 Entrenar una red neuronal por refuerzo
Actor	ACT.1 Usuario de Unity
Descripción	Un Usuario de Unity entrena una red neuronal por refuerzo
Flujo básico	1. **Iniciar entrenamiento**. El caso de uso se inicia cuando un usuario le da al botón de Play en Unity después de crear un nivel preparado para llevar a cabo un entrenamiento por refuerzo y de especificar los parámetros del entrenamiento. 2. **Verificar especificación**. El sistema lee la especificación del usuario y comprueba que su estructura y sus valores son correctos. 3. **Crear red neuronal**. El sistema crea una red neuronal a partir de un nombre que el usuario ha indicado con anterioridad y que debe coincidir con el nombre que aparece en la especificación. 4. **Ejecutar entrenamiento**. El sistema ejecuta el nivel y entrena una red neuronal en base a la información que recibe como entrada y a las recompensas y castigos que recibe de su entorno. El entrenamiento cesa cuando el usuario detiene la ejecución.
Flujo alternativo	1. **Error en la verificación de la especificación**. En el paso 2 del flujo básico, el sistema no encuentra algunos de los campos en la especificación o encuentra un error en sus datos. El sistema utiliza los parámetros de entrenamiento por defecto. El sistema va al paso 4 del flujo básico con los datos por defecto. 2. **Comportamiento no encontrado**. En el paso 3 del flujo básico, el sistema no encuentra ninguna especificación con el nombre que el usuario ha indicado. El sistema muestra por pantalla un mensaje indicando al usuario lo sucedido.
Pre-condiciones	-
Post-condiciones	Se crea una red neuronal entrenada en el sistema de ficheros del usuario

Tabla 3.5: Caso de uso - Utilizar red neuronal entrenada para inferencia.

Caso de uso	CU.4 Utilizar red neuronal entrenada para inferencia
Actor	ACT.1 Usuario de Unity
Descripción	Un Usuario de Unity utiliza una red neuronal entrenada para controlar el comportamiento de un NPC
Flujo básico	1. **Iniciar inferencia**. El caso de uso se inicia cuando un usuario le da al botón de Play en Unity después de crear un nivel con un NPC que va a ser controlado por una red neuronal y de especificar el nombre de la red neuronal a utilizar. 2. **Ejecutar inferencia**. El sistema busca la red neuronal cuyo nombre ha sido especificado por el usuario y la utiliza para tomar decisiones en base a la información que recibe de su entorno.
Flujo alternativo	1. **Red neuronal no existe**. En el paso 2 del flujo básico, el sistema no encuentra la red neuronal cuyo nombre ha sido especificado por el usuario. El sistema muestra por pantalla un mensaje indicando al usuario lo sucedido.
Pre-condiciones	La red neuronal que el usuario indica existe
Post-condiciones	-

Tabla 3.6: Caso de uso - Visualizar estadísticas de entrenamiento.

Caso de uso	CU.5 Visualizar estadísticas de entrenamiento
Actor	ACT.1 Usuario de Unity
Descripción	Un Usuario de Unity visualiza las recompensas obtenidas por un agente a lo largo de su entrenamiento ya sea mientras entrena o una vez finalizado el entrenamiento.
Flujo básico	1. **Visualizar recompensas**. El caso de uso se inicia cuando un usuario indica al sistema que quiere visualizar las estadísticas del entrenamiento. El sistema lee las estadísticas guardadas sobre el entrenamiento que el usuario desea y se las muestra.
Flujo alternativo	-
Pre-condiciones	-
Post-condiciones	-

3.7 Atributos del sistema

3.7.1. Disponibilidad

GAIA es un asset que se encuentra en un repositorio abierto de GitHub[1]. Una vez finalizado el proyecto, se subirá al repositorio y cualquier persona con conexión a Internet, podrá descargarlo y utilizarlo. Una vez descargado, GAIA puede utilizarse sin tener conexión a Internet. Como la disponibilidad de los servidores de GitHub es prácticamente del 100 %, GAIA también lo es.

[1] https://github.com/tetracube/GAIA

3.7.2. Seguridad

GAIA se mantiene protegido de modificación y destrucción al estar subido a un repositorio de GitHub. Si alguien desea modificarlo, tiene que pedir permiso al creador del repositorio, que es el doctor Ramón Mollá Vallá. La seguridad de GAIA es tan buena como la seguridad de GitHub.

3.7.3. Portabilidad

Al tratarse Unity de un motor de videojuegos multiplataforma, GAIA también podrá utilizarse en las plataformas soportadas por Unity. Estas plataformas son Windows, Mac y Linux.

CAPÍTULO 4
Análisis del problema

Antes de comenzar con la implementación de la solución, es importante tener en cuenta los riesgos del proyecto para evitar sobrepasar la línea base del tiempo que se dispone. Es por ello que en este capítulo se hace un análisis de los posibles riesgos y de las acciones que se van a tomar para mantenerlos controlados. Además, se analizan las diferentes tecnologías de redes neuronales descritas en la sección 2.3 y se elige una de ellas como tecnología base que será utilizada por GAIA para implementar el soporte de redes neuronales.

4.1 Riesgos

Un riesgo es una condición con un grado de aparición incierta que, en caso de materializarse, afecta a los objetivos del proyecto. Los riesgos, bajo la definición dada, pueden ser positivos o negativos, dependiendo de su impacto. Un riesgo podría ser, por ejemplo, una inyección de capital en el proyecto por parte de un inversor que acelera la implementación en un 10 %. En este caso, podríamos decir que el riesgo es positivo. A este tipo de riesgos se les denomina oportunidades, mientras que a los negativos se les llama amenazas.

El objetivo de la gestión de riesgos es controlar la incertidumbre, mejorando la toma de decisiones y aumentando las probabilidades de éxito. Para ello, es necesario identificar los riesgos, evaluarlos y controlarlos.

En este capítulo se definen los riesgos del proyecto para evitar fracasar. Se define cada uno en base al estándar de gestión de proyectos Prince2[1]: causa, evento y efecto. También se categorizan los riesgos siguiendo la RBS (Risk Breakdown Structure) de la guía del PMBok [27] pero utilizando únicamente el primer nivel.

- **Riesgos técnicos**. Los relacionados directamente con el producto. Incluyen los requisitos, la tecnología, la calidad, el rendimiento y la disponibilidad, entre otros.
- **Riesgos externos**. Son externos al producto y a la empresa. Incluyen riesgos relacionados con el mercado, los clientes, las condiciones climatológicas, las leyes, los proveedores y empresas subcontratadas.
- **Riesgos de la organización**. Son riesgos que vienen de la empresa donde se realiza el proyecto, como la dependencia con otros proyectos, los recursos y la priorización.

[1] https://prince2.wiki/es/

- **Riesgos de gestión del proyecto**. Los que tienen que ver con la gestión de proyectos. Incluyen riesgos en las estimaciones, la planificación, el control y la comunicación.

Quedando explicado cómo se van a identificar los riesgos, a continuación se listan todos los que se han tenido en cuenta.

Tabla 4.1: Riesgo - Falta de funcionalidad en la tecnología utilizada.

ID	1
Tipo	Amenaza
Categoría	Técnico
Causa	Es difícil preveer si la tecnología que se va a utilizar es suficiente para cumplir con los objetivos del proyecto
Evento	La tecnología que se decide utilizar no provee toda la funcionalidad necesaria
Efecto	Retraso en la finalización del proyecto

Tabla 4.2: Riesgo - La tecnología a utilizar es fácil de entender.

ID	2
Tipo	Oportunidad
Categoría	Técnico
Causa	Los diseñadores de la tecnología crearon una interfaz sencilla de comprender y utilizar
Evento	Se necesita menos tiempo de lo previsto para entender la tecnología y utilizarla
Efecto	Adelanto en la finalización del proyecto

Tabla 4.3: Riesgo - La tecnología a utilizar es complicada de entender.

ID	3
Tipo	Amenaza
Categoría	Técnico
Causa	Los diseñadores de la tecnología no crearon una interfaz sencilla de comprender y utilizar
Evento	Se necesita más tiempo de lo previsto para entender la tecnología y utilizarla
Efecto	Retraso en la finalización del proyecto

Tabla 4.4: Riesgo - Se hacen estimaciones de tiempo por encima del tiempo necesitado realmente.

ID	4
Tipo	Oportunidad
Categoría	Gestión del proyecto
Causa	Poca experiencia planeando e implementando proyectos
Evento	Se emplea menos tiempo del necesario para terminar tareas, sobrando tiempo hasta el día de finalización
Efecto	Adelanto en la finalización del proyecto

Tabla 4.5: Riesgo - Se hacen estimaciones de tiempo por debajo del tiempo necesitado realmente.

ID	5
Tipo	Amenaza
Categoría	Gestión del proyecto
Causa	Poca experiencia planeando e implementando proyectos
Evento	Se emplea más tiempo del necesario para terminar tareas, hasta que el plan pasa la fecha de finalización
Efecto	Retraso en la finalización del proyecto

Tabla 4.6: Riesgo - Las tareas se priorizan incorrectamente.

ID	6
Tipo	Amenaza
Categoría	Gestión del proyecto
Causa	No se dedica suficiente tiempo a pensar en la importancia de cada tarea
Evento	Se emplea tiempo en tareas que no son prioritarias para terminar el proyecto o se emplea demasiado tiempo en tareas que no son críticas
Efecto	Retraso en la finalización del proyecto

Tabla 4.7: Riesgo - La potencia del ordenador sobre el cual se llevan a cabo las pruebas hace que el entrenamiento de las redes neuronales sea rápido.

ID	7
Tipo	Oportunidad
Categoría	Técnico
Causa	El ordenador sobre el que se realizan los entrenamientos es potente
Evento	El entrenamiento de las redes neuronales es rápido
Efecto	Adelanto en la finalización del proyecto

Tabla 4.8: Riesgo - La potencia del ordenador sobre el cual se llevan a cabo las pruebas hace que el entrenamiento de las redes neuronales sea lento.

ID	8
Tipo	Amenaza
Categoría	Técnico
Causa	El ordenador sobre el que se realizan los entrenamientos es poco potente
Evento	El entrenamiento de las redes neuronales es lento
Efecto	Retraso en la finalización del proyecto

Tabla 4.9: Riesgo - Mal diseño de las pruebas de entrenamiento.

ID	9
Tipo	Amenaza
Categoría	Técnico
Causa	Poca experiencia trabajando con redes neuronales
Evento	Las pruebas de entrenamiento suponen más tiempo del esperado para cumplir con los objetivos
Efecto	Retraso en la finalización del proyecto

Tabla 4.10: Riesgo - Me pongo enfermo.

ID	10
Tipo	Amenaza
Categoría	Organización
Causa	En cualquier momento, una persona puede resfriarse o le puede sentar mal una comida
Evento	Tengo que descansar varios días para recuperarme
Efecto	Retraso en la finalización del proyecto

Ahora que los riesgos asociados al proyecto ya están definidos, se evalúan sus probabilidades e impacto. Tanto la probabilidad como el impacto van a ser medidas de manera cualitativa, haciendo uso de la siguiente escala: muy bajo, bajo, medio, alto y muy alto. Para visualizarlo, se utiliza la matriz del riesgo con la frecuencia en horizontal y el impacto en vertical, tal y como muestra la Figura 4.1.

Matriz del riesgo					
Impacto	Probabilidad				
	Muy alta	Alta	Media	Baja	Muy baja
Muy alto				1	
Alto		7, 9	6, 8	3	
Medio		4			
Bajo		5	4		
Muy bajo					

Figura 4.1: Matriz de riesgos cualitativa del proyecto donde se evalúan los riesgos por impacto y probabilidad en la escala: muy alto, alto, medio, bajo y muy bajo.

En la Figura 4.1, el color de las celdas indica la importancia de los riesgos, siendo rojo el más importante y verde el que menos. Cuanto más importante es un riesgo, más control hay que poner sobre él. Aunque se podrían preparar acciones a tomar sobre todos los riesgos, en este proyecto se van a preparar para los riesgos rojos y naranjas. La respuesta a los demás riesgos son aceptar, en el caso de las amenazas y rechazar, en el caso de las oportunidades. Esto es lo mismo que no realizar ninguna acción.

Las respuestas para los riesgos también van a ser etiquetadas. Primero, según sean proactivas o reactivas. Las respuestas proactivas son aquellas que se anticipan a la materialización del riesgo, mientras que las reactivas se ejecutan una vez el riesgo se ha materializado. Otra etiqueta a utilizar es el tipo de acción, tal y como se propone en Prince2.

Para las amenazas, Prince2 propone seis tipos de respuesta.

- **Evitar**. Reducir a 0 el impacto o la probabilidad.
- **Reducir**. Reducir en la medida de lo posible el impacto o la probabilidad.
- **Contingencia**. Reducir el impacto de la amenaza una vez se ha materializado.
- **Transferir**. Transferir el riesgo a otra parte.
- **Compartir**. El impacto se comparte entre varias partes.

- **Aceptar**. No hacer nada.

Para las oportunidades, en cambio, proponen cuatro tipos de respuesta.

- **Aprovechar**. Aumentar el impacto una vez la oportunidad se materializa.
- **Incrementar**. Aumentar en la medida de lo posible la probabilidad o el impacto.
- **Compartir**. El impacto se comparte entre varias partes.
- **Rechazar**. No hacer nada

Conociendo los tipos de respuesta, la Tabla 4.11 muestra una lista con las respuestas a los riesgos más importantes.

Tabla 4.11: Respuestas a los riesgos más importantes.

ID	Evento de riesgo	Respuesta	Proactiva/Reactiva	Tipo
1	La tecnología que se decide utilizar no provee toda la funcionalidad necesaria	Buscar otra tecnología que abarque la funcionalidad	Reactiva	Contingencia
6	Se emplea tiempo en tareas que no son prioritarias para terminar el proyecto o se emplea demasiado tiempo en tareas que no son críticas	Al terminar cada tarea, recapacitar sobre cuál es el orden de prioridad de las tareas que faltan	Proactiva	Reducir
7	El entrenamiento de las redes neuronales es rápido	Dedicar el tiempo restante a realizar más pruebas para que la memoria esté más completa	Reactiva	Aprovechar
8	El entrenamiento de las redes neuronales es lento	Entrenar con más ordenadores al mismo tiempo	Proactiva	Reducir
9	Las pruebas de entrenamiento suponen más tiempo del esperado para cumplir con los objetivos	Entrenar con más ordenadores para probar más parámetros al mismo tiempo	Proactiva	Reducir

4.2 Posibles soluciones

En el Capítulo 2, se han visto tres de las técnicas de inteligencia artificial más utilizadas en el desarrollo de comportamientos de un NPC en videojuegos. Al hacerlo, el foco

de la tecnología utilizada se lo ha llevado Unity. La razón es, que se trata de una herramienta extensamente utilizada y con pruebas de lo que se puede lograr con ella. Además, GAIA es un asset de Unity, y es coherente aumentar su funcionalidad con la introducción de redes neuronales en el mismo entorno.

Como ha sido mencionado en la Sección 2.6, no se pretende inventar lo ya inventado, sino utilizar tecnología ya existente que facilite la creación y el entrenamiento de redes neuronales. Lo que se va a hacer es diseñar un lenguaje de especificación de redes neuronales para que un usuario pueda detallar la estructura de las mismas. Posteriormente, se creará la red neuronal especificada y se utilizará un algoritmo de aprendizaje que permitirá entrenar a partir de otra inteligencia artificial o una persona controlando la entidad. La red neuronal creada también podrá entrenarse a partir de unos objetivos, sin la necesidad de imitar otro comportamiento. Una vez la red neuronal esté entrenada, se guardará en un fichero para poder utilizarla cuando se quiera. Finalmente, se necesita una forma de cargar la red neuronal y que un NPC la utilice para decidir las acciones a realizar o incluso para continuar entrenando.

De las diferentes tecnologías disponibles para llevar a cabo el trabajo, se analizan en esta sección las más populares, que son aquellas descritas en el Capítulo 2, Sección 2.3. Para ello, la Tabla 4.12 muestra un resumen de las características de las que dispone cada tecnología.

Las características analizadas son las siguientes.

- **Creación**. Permite crear redes neuronales.
- **Capas ocultas**. Permite especificar el número de capas ocultas de las redes neuronales.
- **Neuronas por capa**. Permite especificar el número de neuronas por capa oculta. Todas las capas ocultas tienen el mismo número de neuronas.
- **Neuronas por capa individual**. Permite especificar el número de neuronas por capa oculta de manera individual. A diferencia del punto anterior, cada capa puede tener diferente número de neuronas.
- **Imitación**. Entrenar redes neuronales imitando el comportamiento de una IA preexistente o de una persona controlando una entidad.
- **Refuerzo**. Entrenar redes neuronales utilizando aprendizaje por refuerzo. En vez de imitar un comportamiento, la red neuronal es capaz de aprender por sí sola en base a las recompensas y castigos que recibe de su entorno.
- **Aleatorio**. Entrenar creando redes neuronales con pesos aleatorios y comprobando su desempeño en el objetivo para el que han sido diseñadas.
- **Almacenaje**. Almacenar redes neuronales entrenadas en el sistema de ficheros del ordenador en el que se entrenan.
- **Inferencia**. Utilizar redes neuronales entrenadas para inferir las acciones de un NPC.
- **Continuar aprendizaje**. Seguir entrenando una red neuronal entrenada. Este entrenamiento puede ser de cualquiera de los dos tipos descritos, por imitación o por objetivos.

- **Estado de la red**. Permite visualizar el estado de la red neuronal mientras se ejecuta. Esto es, las entradas y salidas de las neuronas así como los pesos de las conexiones.
- **Estado del aprendizaje**. Permite visualizar métricas sobre la calidad del aprendizaje.
- **Modificación gráfica**. Permite modificar la red neuronal a través de una interfaz gráfica.
- **Entrenamiento en GPU**. El entrenamiento puede realizarse en la GPU.
- **Librería externa necesaria**. Se necesita otra herramienta externa a Unity para realizar el entrenamiento de la red neuronal. La red neuronal resultante puede ser importada en Unity para ser utilizada por un NPC.
- **Librería externa opcional**. Existe la posibilidad de utilizar una herramienta externa a Unity para realizar el entrenamiento de la red neuronal. La red neuronal resultante puede ser importada en Unity para ser utilizada por un NPC.
- **Soporte activo**. La herramienta recibe actualizaciones frecuentemente para solucionar errores.
- **Documentación necesaria**. Contiene la documentación necesaria para entender cómo funcionar con la herramienta y sus posibilidades.
- **Documentación extensa**. Además de contener la documentación necesaria, contiene una lista de ejemplos para ver cuáles son las posibilidades de la herramienta.
- **Foro**. La herramienta dispone de un foro en el que proponer dudas. En él, se pueden encontrar soluciones a problemas que salgan a la luz al implementar código.
- **Es gratuita**. La tecnología no tiene coste alguno.

Característica \Tecnología	ANN Perceptrón	AIMLPK	MLAgents	Noedify
Creación	✓	✓	✓	✓
Capas ocultas	✓	✓	✓	✓
Neuronas por capa	✓	✓	✓	✓
Neuronas por capa individual	✓			✓
Imitación	✓		✓	✓
Refuerzo			✓	
Aleatorio	✓			
Algoritmos genéticos		✓		
Almacenaje	✓		✓	✓
Inferencia	✓	✓	✓	✓
Continuar aprendizaje	✓		✓	✓
Estado de la red	✓			
Estado del aprendizaje	✓		✓	
Modificación gráfica	✓			
Entrenamiento en GPU			✓	✓
Librería externa necesaria			✓	
Librería externa opcional				✓
Soporte activo			✓	
Documentación necesaria	✓		✓	✓
Documentación extensa			✓	
Foro			✓	
Es gratuita			✓	✓

Tabla 4.12: Comparación de características entre tecnologías de Unity para la creación de IA basada en redes neuronales.

De las cuatro tecnologías descritas, Artificial Intelligence and Machine Learning Prototyping Kit es la que menos funcionalidad trae consigo, y queda descartada ya que no permite realizar aprendizaje por imitación ni almacenar las redes entrenadas para continuar el entrenamiento desde donde se dejó o realizar inferencia desde ese punto.

Otra tecnología que queda descartada es ANN Perceptrón por varios motivos. Primero, se trata de un asset de pago que no podría utilizarse en GAIA de manera abierta. Por otro lado, los aprendizajes que permite hacer son por imitación y aleatorio. El aprendizaje por imitación es útil para el trabajo, pero el aprendizaje aleatorio es menos eficaz que un entrenamiento por refuerzo, en el que la red neuronal evoluciona hacia un objetivo que impone el diseñador. Además, toda la interfaz gráfica que permite modificar la red neuronal y visualizar su estado no es necesaria.

Entre los dos candidatos restantes, el vencedor es MLAgents. Aunque Noedify tiene más control sobre las neuronas que hay en cada capa oculta, solo provee de aprendizaje por imitación, que no permite continuar aprendiendo una vez se ha imitado el comportamiento deseado. La forma de seguir aprendiendo podría ser especificando programáticamente para cada entrada de la red, una salida que se ajuste al comportamiento que se desea. Pero si son conocidas las acciones que debería de llevar a cabo, ¿para qué se entrenaría una red neuronal? Además, una de las restricciones del proyecto es la posibilidad de realizar un entrenamiento utilizando la GPU, y Noedify no está preparado para ello. MLAgents también trae consigo el aprendizaje por refuerzo, que permite seguir apren-

diendo a una red neuronal a base de señales de recompensa y castigo. Por otro lado, MLAgents viene con una herramienta de visualización del entrenamiento en la web, tiene una documentación extensa con ejemplos de uso en la que poder buscar información, tiene un soporte activo de la comunidad y un foro en el que los desarrolladores escriben sus preguntas y otros desarrolladores de la comunidad de Unity pueden contestar.

CAPÍTULO 5
Solución Propuesta

En esta sección se describe la solución que se ha desarrollado para cumplir con los objetivos del proyecto. Se incluyen el plan de trabajo, la arquitectura, el diseño de clases, la tecnología utilizada con las dependencias que tienen y la explicación de la implementación llevada a cabo.

5.1 Plan de trabajo

Para desarrollar el producto, el trabajo ha sido dividido en diferentes fases tal y como indica la Tabla 5.1.

Con una dedicación aproximada de quince horas a la semana, se puede visualizar en el diagrama de Gantt de la Figura 5.1 cómo se ha repartido el trabajo a lo largo de cinco meses y medio.

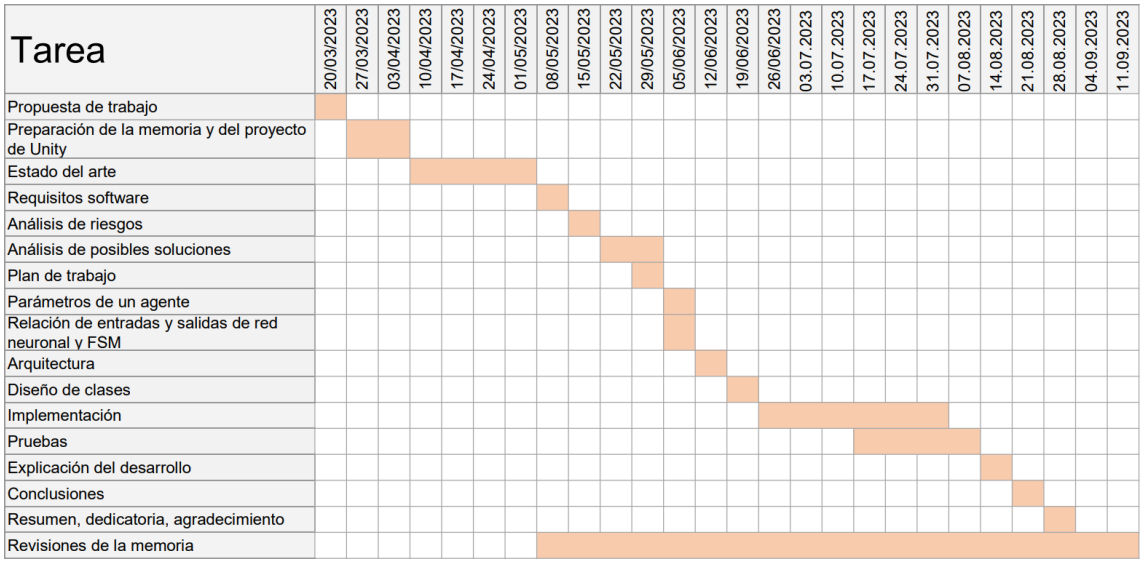


Figura 5.1: Diagrama de Gantt del desarrollo de la solución y la realización de las pruebas.

5.2 Diseño

A la hora de implementar la solución, se deben de tener en cuenta varios puntos. El primero son los parámetros que van a poder especificarse por XML sobre el entrena-

Tabla 5.1: Plan de trabajo. El trabajo se ha dividido en tres fases con un total de 140 horas, entre el diseño, la implementación las pruebas y las conclusiones.

Fase	Horas	Descripción
Preparación de la propuesta del trabajo	15	Buscar información sobre inteligencia artificial y cómo se aplica a los videojuegos
Preparación de la memoria y del proyecto de Unity	25	Crear la estructura de la memoria en latex y subirla a un repositorio de GitHub. Bajar el proyecto de GAIA de GitHub, hacerlo funcionar y crear un proyecto nuevo para trabajar sobre él.
Investigación del estado del arte	60	Investigación y escritura del estado de las técnicas de inteligencia artificial más utilizadas a día de hoy, su impacto y sus inconvenientes.
Especificación de requisitos software	15	Describir el funcionamiento del plugin formalmente, explicando las restricciones, asunciones, los requisitos funcionales y los atributos del sistema.
Análisis	35	Buscar los riesgos asociados al proyecto, evaluarlos según su impacto y probabilidad y preparar respuestas para evitar superar la línea base del tiempo. También, valorar las diferentes alternativas de tecnologías que podrían utilizarse para implementar el soporte de redes neuronales en GAIA.
Diseño	70	Describir el flujo de trabajo, los subsistemas, las clases que va a haber y la relación entre ellas.
Implementación	70	Implementar el código necesario para cumplir con los requisitos siguiendo el diseño.
Explicación del desarrollo	15	Escribir en la memoria el desarrollo de la solución implementada.
Pruebas	30	Diseño, implementación y ejecución de pruebas.
Conclusiones	10	Concluir los resultados obtenidos en las pruebas junto con una conclusión del trabajo realizado.
Resumen, agradecimiento, dedicatoria	10	Escribir el abstract de la memoria junto con los agradecimientos y la dedicatoria.
Revisiones de la memoria	40	Revisar la memoria para eliminar erratas, mejorar la comprensión y adaptar el contenido a lo que se requiere.

miento a realizar. El segundo, cómo relacionar las entradas y las salidas de una FSM y una red neuronal, puesto que son los dos modelos que van a utilizarse para las pruebas. Finalmente, el tercer punto es la persistencia de los pesos de una red neuronal.

5.2.1. Parámetros de entrenamiento de un agente

Existen una serie de parámetros que van a poder especificarse a la hora de crear una red neuronal.

- **Algoritmo de aprendizaje por refuerzo**. Algoritmo de aprendizaje a utilizar. Para la primera versión del proyecto solo habrá posibilidad de un algorítmo: Proximal Policy Optimization (PPO).

- **Velocidad de aprendizaje** (learning rate). Velocidad de aprendizaje utilizada en el algoritmo de descenso de gradiente por la red neuronal que modela el comportamiento del agente. Indica la fuerza con la que la red neuronal modifica sus pesos en cada paso de aprendizaje.
- **Tamaño de lote** (batch size). Número de acciones que toma el agente antes de aplicar un paso de aprendizaje.
- **Número de entradas**. Cantidad de entradas que tiene la red neuronal. Cada entrada es un número en coma flotante.
- **Salidas discretas**. Cantidad y rango de las salidas discretas. Las salidas discretas son números enteros dentro del rango [A, B].
- **Salidas continuas**. Cantidad y rango de las salidas continuas. Las salidas continuas son números en coma flotante dentro del rango [A, B].
- **Parámetros de la red neuronal del agente**. Red neuronal que modela el comportamiento del agente.
  - **Número de capas ocultas**. Número de capas ocultas de la red. Las capas ocultas son las que no son de entrada ni de salida.
  - **Neuronas ocultas por capa**. Cuántas neuronas tiene cada capa oculta de la red.
  - **Tamaño de memoria**. Utilizada para redes Long Short-Term Memory (LSTM). Tamaño del estado oculto de la red LSTM.
  - **Tamaño de secuencia**. Utilizada para redes LSTM. Indica el número de experiencias que se usan como una única entrada a la red neuronal. Tiene que ser múltiplo de 2.
- **Fuerza de recompensas extrínsecas**. Valor entre 0 y 1 por el cual se multiplican las recompensas recibidas del entorno.
- **Gamma de recompensas extrínsecas**. Cuando en el entrenamiento por refuerzo se calcula la función de coste, se utiliza una estimación de la recompensa que se obtendrá en el futuro si en el presente se toma una decisión. La importancia de esta estimación se define mediante Gamma. Si el valor es alto, se tienen muy en cuenta las recompensas futuras. Si, en cambio, Gamma es bajo, la recompensa del presente coge más fuerza.
- **Fuerza de recompensas con Curiosidad**. Valor entre 0 y 1 por el cual se multiplican las recompensas recibidas mediante la estrategia de Curiosidad. Esta estrategia consiste en tomar decisiones que el agente ha tomado pocas veces con anterioridad.
- **Gamma de recompensas con Curiosidad**. Igual que el parámetro Gamma en las recompensas extrínsecas.
- **Parámetros de la red neuronal con Curiosidad**. Red neuronal que modela el comportamiento de la estrategia de Curiosidad. Son los mismos parámetros que los de la red neuronal del agente.
- **Velocidad de aprendizaje de la red con Curiosidad**. Velocidad de aprendizaje de la red neuronal utilizada en la estrategia de Curiosidad.

La manera de especificar estos parámetros en XML para ser utilizados por GAIA está especificado en el Apéndice A.

5.2.2. Relación entre las entradas y las salidas de una FSM y una red neuronal

Es posible construir una red neuronal que simule el comportamiento de una FSM. Además, el número de neuronas necesario utilizando la red neuronal de [19] para dicha tarea está acotado entre $\Omega((m\ log\ m)^{1/3})$ y $O(m^{3/4})$ [28]. Aunque la demostración aportada en [28] hace uso de máquinas de Mealy, FSMs deterministas con alfabeto de entrada y de salida de únicamente dos símbolos, es extrapolable a máquinas con alfabetos más amplios.

Las máquinas de estados en videojuegos son distintas a las definiciones formales que se encuentran en los artículos y libros. Pero no es difícil encontrar la manera de que sean más similares. Se puede formalizar la FSM de un videojuego haciendo que el alfabeto de entrada se genere a partir de las condiciones que se dan para hacer un cambio de estado en la FSM. Si para el cambio de estado se tienen en cuenta los valores de puntos de vida, distancia al jugador y naturaleza del enemigo, el alfabeto de entrada de la FSM puede tener tamaño $rangos\_de\_vida * rangos\_de\_distancia * numero\_de\_naturalezas$, donde cada combinación de estos valores se convierte en un símbolo del alfabeto.

Aunque se ha demostrado que el comportamiento de una FSM puede ser imitado por una red neuronal, en este trabajo no se van a utilizar los descubrimientos de [28] ni de otros autores formalmente, si no que se va a comprobar experimentalmente el resultado. Para ello, primero es necesaria una forma de relacionar las entradas y las salidas de una FSM y una red neuronal.

En cuanto a las entradas de la red neuronal, éstas podrían ser los eventos de cambio de estado de la FSM o los valores de las variables que utiliza la máquina de estados para determinar si ejecutar o no los eventos de cambio entre estados. Un ejemplo es la distancia al jugador. Este dato es información que la FSM utiliza para computar el comportamiento de un NPC, entonces también es útil que la red neuronal tenga acceso a esa información para tomar decisiones. Además de los datos que utiliza la FSM, se puede proveer a la red neuronal con más información sobre el entorno en el que se encuentra para que tome decisiones más informadas y pueda aprender comportamientos más complejos. Entre estas dos posibilidades, se utilizará la segunda, ya que tener acceso al estado del juego es tener más información que si solo se tiene acceso a eventos que se generan cuando dicho estado toma unos valores determinados. Por otro lado, no depender de los eventos de la FSM permite a una red neuronal seguir aprendiendo mediante aprendizaje por refuerzo sin necesitar los eventos de la FSM como entrada.

También es sencillo relacionar las salidas de la FSM con las salidas de la red neuronal. La FSM, al encontrarse en un estado determinado, ejecuta las acciones asociadas a dicho estado. Estas acciones son las salidas de la red neuronal. Si un enemigo en el estado *Atacando*, elige entre AtacarConEspada y AtacarConPistola, la red neuronal debe poder tener como salida esas dos acciones. La dificultad de definir las salidas adecuadas radica en decidir su codificación. ¿Es la salida un vector de ceros y unos donde cada elemento indica si la acción con índice i se ejecuta? ¿O es la salida un vector de números reales entre 0 y 1 que indica la fuerza con la que la red ejecutaría cada acción?. Además ¿puede la red sacar como salida dos acciones al mismo tiempo? De esta manera, el enemigo podrá atacar mientras camina o podrá curarse mientras se defiende. Por suerte, MLAgents abstrae estas decisiones.

No existe una única manera de relacionar las entradas y salidas de una FSM y de una red neuronal. Entonces, será trabajo del diseñador de la red neuronal el decidir, no solo sus capas ocultas, sino también sus entradas y sus salidas. Estos diseños pueden ser guiados por los resultados obtenidos al entrenar las redes. No obstante, para facilitar esta relación, se describe a continuación un algoritmo que, a partir de las entradas y salidas de una FSM, se consiguen las entradas y salidas de una red neuronal. Este algoritmo funciona para cualquier tipo de FSM que se han mencionado hasta ahora.

Algoritmo 5.1 Algoritmo que encuentra la relación entre las entradas y salidas de una FSM y una red neuronal.

for all *transition* en transiciones de la FSM **do**
  vars ← *transition.condition_variables*
  for all *var* en *vars* **do**
    añadir *vars* como neurona de entrada a la red neuronal
for all *state* en estados de la FSM **do**
  actions ← *state.actions*
  for all *action* en *actions* **do**
    añadir *action* como neurona de salida de la red neuronal

5.2.3. Pesos y persistencia de una red neuronal

En cuanto a la persistencia de los pesos de las redes neuronales, esta depende de la tecnología que se esté utilizando por debajo de GAIA. Como esta primera versión va a utilizar MLAgents, la persistencia de las redes neuronales la llevará a cabo esta tecnología de Unity.

Unity MLAgents exporta los modelos de redes neuronales utilizando el formato ONNX[1] (Open Neural Network Exchange). Este es un formato de código abierto respaldado por The Linux Foundation[2] cuyo objetivo es conseguir interoperabilidad entre diferentes tecnologías. El objetivo se alinea con el diseño de este trabajo, que pretende permitir el uso de diferentes tecnologías bajo la misma interfaz, así que no tiene inconveniente alguno.

A partir de la especificación que recibe MLAgents, éste crea una red neuronal añadiendo tantas capas como necesite para cumplir con lo indicado y obtener buenos resultados. Un ejemplo de red neuronal vista desde el editor de Unity se muestra en la Figura 5.2.

[1] https://onnx.ai/
[2] https://www.linuxfoundation.org/

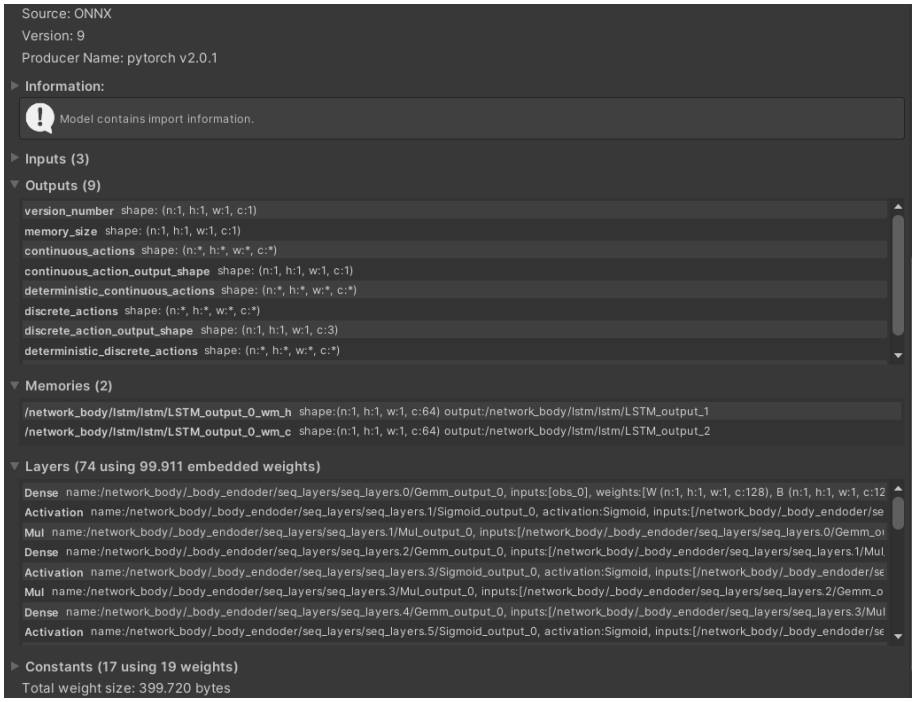


Figura 5.2: Imagen que muestra la estructura de una red neuronal creada por MLAgents en formato ONNX y visualizada desde el editor de Unity. La red neuronal mostrada tiene 4 capas ocultas y memoria a través de una capa LSTM.

Como se puede observar, el número de capas de la red neuronal es de 74, más de las que podrías esperar cuando en la especificación se ha indicado que se querían 4. Esta diferencia se debe a varios factores. Primero, una capa oculta en la especificación no equivale a una capa de la red neuronal resultante. En la Figura 5.2, las tres primeras filas del apartado *Layers* se repiten 4 veces. Esas son las 4 capas ocultas indicadas en la especificación. La primera es una capa totalmente conectada que calcula la suma de las multiplicaciones de las entradas por sus respectivos pesos. La segunda capa simplemente aplica una función de activación a las salidas de la capa anterior. La tercera capa multiplica las salidas de las funciones de activación por un valor escalar. Además de esta variación, se necesitan varias capas para modificar las salidas de estas capas ocultas y poder utilizarlas en una capa LSTM, que permite a la red "recordar" lo que ha pasado anteriormente. También existen más capas que transforman la salida de la LSTM y le aplica las recompensas extrínsecas, que también se modelan con una red neuronal. Finalmente, el resultado se convierte para generar las salidas discretas y continuas, por lo que se añaden más capas a la red. Para más información sobre el funcionamiento de las diferentes capas, visitar la documentación de la página oficial de PyTorch[3].

Una vez se tiene el modelo en formato ONNX, debe de transformarse al formato utilizado por Barracuda, el paquete de Unity para trabajar con redes neuronales. Esto no es ningún problema ya que el mismo paquete ofrece un conversor que se puede utilizar en el proyecto.

[3] https://pytorch.org/docs/stable/nn.html

5.3 Arquitectura

En esta sección se verán cómo los diferentes componentes del sistema interactúan entre sí.

Primero de todo, y aunque se va a abstraer la funcionalidad, al trabajar con ML-Agents, se va a repasar cuál es su arquitectura (Figura 5.3). En ML-Agents hay dos subsitemas. Primero de todo está el entorno de aprendizaje que se ejecuta en Unity. En él los agentes interactúan con su entorno mediante acciones que han sido decididas por su comportamiento. En segundo lugar está la interfaz de Python, que crea y utiliza redes neuronales para decidir qué acción debe tomar el agente. El comportamiento es un conjunto de parámetros que determinan el número de observaciones (entradas), el número y tipo de acciones (salidas) y el tipo de comportamiento, entre otras cosas. Y es en Python donde se ejecutan los algoritmos de aprendizaje.

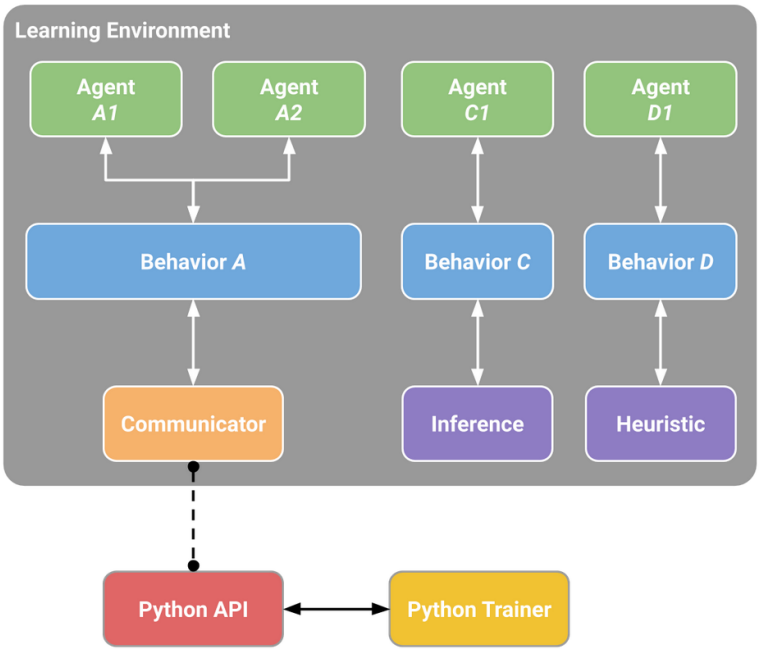


Figura 5.3: Arquitectura de ML-Agents[4]. El área gris, llamado entorno de aprendizaje, se encuentra en Unity y está compuesto por los agentes, los comportamientos y un comunicador. Los agentes se encargan de generar las observaciones (entradas a la red neuronal), de la ejecución de las acciones que recibe y de la asignación de recompensas. El comportamiento recibe las observaciones y las recompensas del agente y devuelve acciones para que el agente las ejecute. El comunicador se encarga de establecer la comunicación entre el comportamiento y la interfaz de Python, encargada de realizar el entrenamiento.

GAIA va a ser una capa de abstracción sobre ML-Agents Toolkit, y va a interactuar tanto con el entorno de Unity como con el de Python. En caso de que se incorpore una nueva tecnología, la interfaz de programación permite introducirla fácilmente así como cambiar entre las tecnologías disponibles. La arquitectura propuesta se puede visualizar en la Figura 5.4.

[4] https://unity-technologies.github.io/ml-agents/ML-Agents-Overview

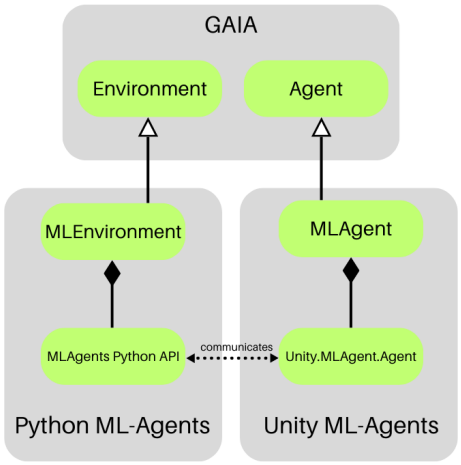


Figura 5.4: Arquitectura del sistema incorporado a GAIA.

De esta forma, están las interfaces Environment y Agent con las que cualquier usuario de GAIA va a trabajar. A través de Environment, podrá configurar el entorno de aprendizaje, mientras que con Agent se le pasarán las observaciones y las recompensas a la red neuronal que prepara el entorno. Esto permite que un usuario no tenga que saber qué tecnología se encuentra por debajo y solamente necesita aprender cómo funciona la interfaz propuesta en GAIA. Además, si en cualquier momento se desea incorporar una nueva tecnología, un usuario podrá hacer uso de ella sin modificar su código, solamente la especificación XML indicando la tecnología que quiere utilizar.

5.4 Diseño detallado

En esta sección se vuelve a utilizar el lenguaje UML, en vez de para representar los requisitos funcionales mediante diagramas de casos de uso, para representar el diseño del asset mediante un diagrama de clases.

En las Figuras 5.5 y 5.6 se representan las clases necesarias a implementar para llevar a cabo el presente proyecto. En el diagrama, no aparecen todos los atributos y métodos de las clases, solo los esenciales para entender la nueva funcionalidad de GAIA. No se considera necesario, por ejemplo, incluir aquella información relacionada con las FSM y los árboles de comportamientos que ya ha sido explicada con más detalle en sus respectivos trabajos. Tampoco se considera necesario incorporar clases que son de librerías externas y que no aportan sino complejidad, como VectorSensor, que es una implementación específica para los agentes de Unity MLAgents.

En los diagramas se ha utilizado un código de tres colores. Primero, el color verde representa las interfaces y clases de GAIA que son genéricas, que no dependen de ninguna tecnología. Así pues, está la interfaz *IAgent* que representa un agente con la habilidad de aprender un comportamiento mediante el algoritmo de aprendizaje por refuerzo utilizando redes neuronales. Esta interfaz es implementada por la clase *MLAAgent*, que extiende de la clase *Agent* (de MLAgents Toolkit) y que implementa el código específico para tener un agente que usa la librería de Unity MLAgents. De esta forma, si se quisiese tener una implementación de un agente utilizando otra tecnología, sería suficiente crear la clase del agente específica que implemente la interfaz. El resto del código no cambiaría.

En segundo lugar está el color naranja que, a diferencia de la genericidad del color verde, representa clases específicas que implementan la funcionalidad de las interfaces utilizando una tecnología determinada. En caso de querer añadir agentes utilizando Noedify, se podría crear una nueva clase *NoedifyAgent* que implementase la funcionalidad de la interfaz *IAgent*.

Finalmente, el color azul se ha utilizado para representar clases que pertenecen a librerías externas. En el diagrama, solo hay una clase con este color para evitar añadir complejidad innecesaria al diagrama. Esta clase es *Agent*, disponible en la librería de Unity MLAgents con mucha funcionalidad implementada que facilita el trabajo.

Algunos puntos a destacar del primer diagrama son las dos interfaces *IAgent* y *IEnvironment*. Estas son interfaces clave que muestran la capa de abstracción más alta. A través de ellas, un usuario podría utilizar redes neuronales sin saber qué tecnología hay por debajo. Las dos interfaces *IAgent* y *IEnvironment* hacen uso de dos estructuras que representan sus configuraciones, *AgentConfig* y *EnvironmentConfig* respectivamente. Estas clases son las que crea y rellena el procesador de ficheros XML y que sirven para crear tantos agentes y entornos como se necesiten manteniendo la misma configuración. Para terminar de hablar del primer diagrama, la clase *Utils* ha sido creada con el fin de mantener funcionalidad potencialmente común a diferentes partes del código.

En el segundo diagrama, en cambio, quedan plasmadas dos clases que son las encargadas de gestionar los entrenamientos por imitación (*ImitationManager*) y las batallas entre tanques (*WinManager*). Ambas funcionalidades se explican con más detalle en la Sección 5.6.

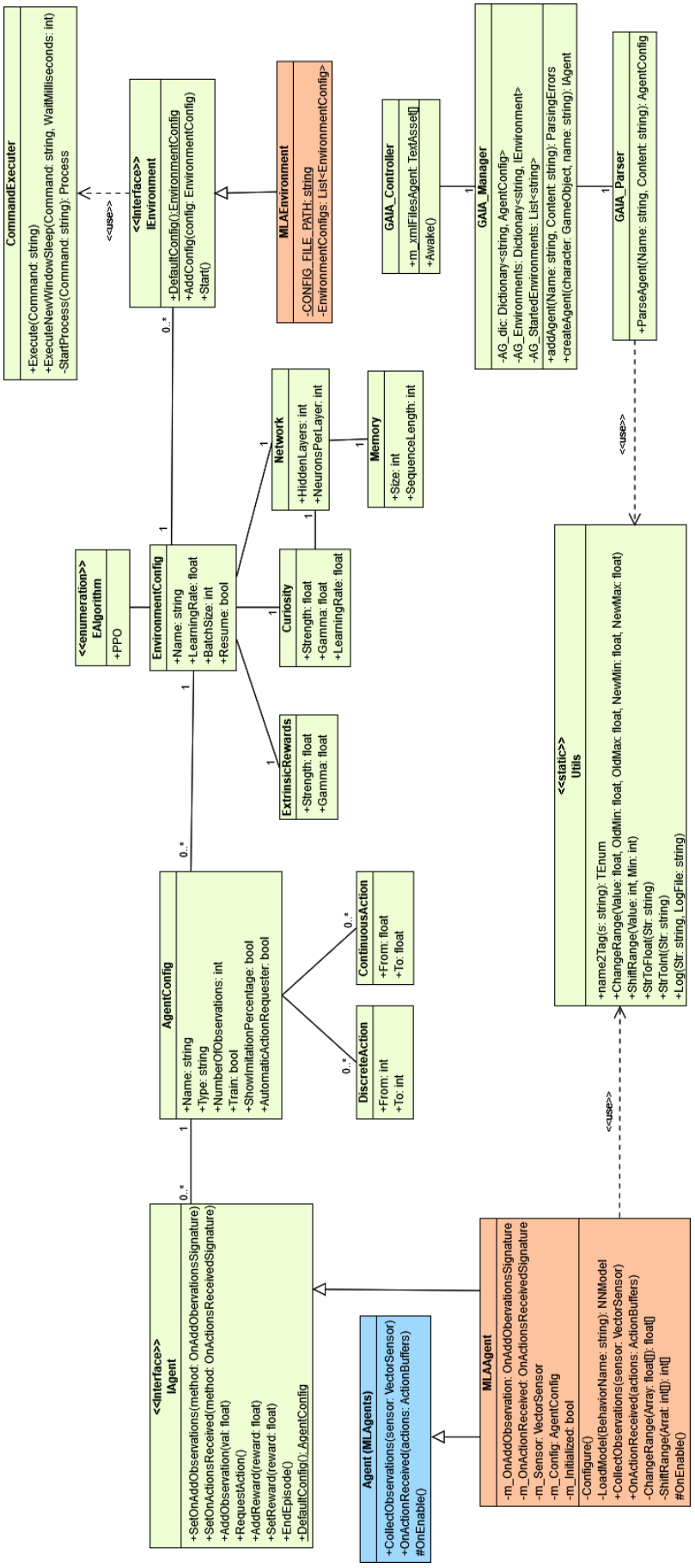


Figura 5.5: Diagrama de clases UML.

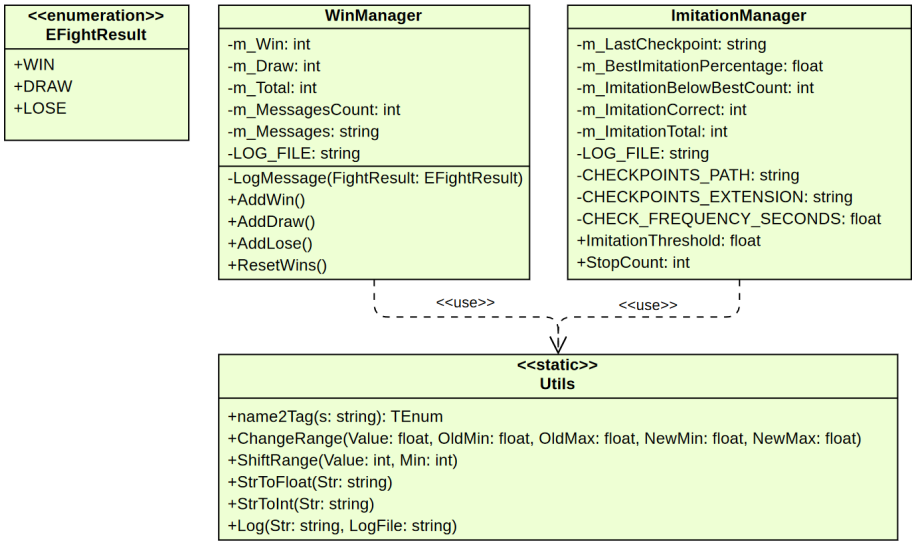


Figura 5.6: Continuación del diagrama de clases UML.

5.5 Tecnología

En este punto ya está decidido que la tecnología de redes neuronales sobre la que se va a soportar GAIA es Unity MLAgents. Por lo tanto, ya existen diferentes decisiones tomadas. Primero, el uso del motor de videojuegos Unity, puesto que es en el que se ha basado GAIA tanto para las FSM como para los árboles de comportamiento. Al estar ligado a Unity, el proyecto va a ser programado utilizando el lenguaje de programación C#.

Existen otras variables importantes. Para utilizar MLAgents es necesario tener instalado Python y algunas librerías de este lenguaje. La versión del proyecto de MLAgents que se va a utilizar es la Release 20, la más actualizada actualmente. En el repositorio de GitHub donde se encuentra alojado el proyecto, hay una página con información sobre cada una de las realeses. Las recomendaciones de los desarrolladores para las dependencias de la Release 20[5] quedan reflejadas en la Figura 5.7.

Package	Version
com.unity.ml-agents (C#)	v2.3.0-exp.3
com.unity.ml-agents.extensions (C#)	v0.6.1-preview
ml-agents (Python)	v0.30.0
ml-agents-envs (Python)	v0.30.0
gym-unity (Python)	v0.30.0
Communicator (C#/Python)	v1.5.0

Figura 5.7: Versiones de las dependencias de MLAgents recomendadas por sus desarrolladores a utilizar con la Release 20.

[5] https://github.com/Unity-Technologies/ml-agents/releases/tag/release_20

Siguiendo las recomendaciones de los desarrolladores de MLAgents y utilizando versiones compatibles de las diferentes librerías, las dependencias de las diferentes tecnologías que van a ser instaladas en GAIA se recogen en la Tabla 5.2.

Tabla 5.2: Dependencias de MLAgents con las versiones utilizadas en GAIA.

Librería	**Versión**
Unity	2022.3.4f1
AI Navigation (Unity)	1.1.4
CUDA (GPU)	11.7
cuDNN (GPU)	8.9.4
com.unity.ml-agents (C#)	2.3.0-exp.3
Python (Python)	3.9.12
pip (Python)	23.2
torch (Python)	2.0.1+cu117
torchaudio (Python)	2.0.1+cu117
torchvision (Python)	0.15.2+cu117
mlagents (Python)	0.30.0
mlagents-envs (Python)	0.30.0
protofub (Python)	3.20.3
onnx (Python)	1.14.0

Con respecto a la elección de la versión de Unity, en la página de la Release 20 de MLAgents aparece la versión mínima de Unity soportada, que es, 2021.3. Con el motivo de continuar con la misma versión que se estaba utilizando en GAIA y evitar conflictos con lo que ya estaba implementado, Unity 2022.3.4f1 ha sido la elegida. Y como dependencia de Unity está AI Navigation, utilizada para implementar el movimiento de los NPCs en el juego. La versión utilizada es la 1.1.4, la más actualizada a fecha actual.

Para que el entrenamiento pueda ser realizado en la tarjeta gráfica, se necesitan tener instaladas tanto CUDA (Compute Unified Device Architecture) como cuDNN (CUDA Deep Neural Network), ambas de NVIDIA. Esto es así puesto que las tarjetas gráficas de las que se disponen son las dos de la marca NVIDIA. Las versión escogida para CUDA es una de las que soporta torch para ejecutar el entrenamiento en la tarjeta gráfica. La versión de cuDNN es la última versión compatible con la instalada de CUDA.

La versión de Python escogida ha sido tal que ha permitido la instalación de la versión 0.30.0 de mlagents y de mlagents-env. La versión debe de ser mayor o igual a la 3.7.2 y menos que la 3.10, así que la 3.9.12 es la elegida. La versión de pip, el gestor de paquetes de Python, es la más actualizada.

Además de las dependencias de las tecnologías, los entornos en los que se han llevado a cabo las pruebas son dos ordenadores cuyas especificaciones están recogidas en la Tabla 5.3.

Tabla 5.3: Entornos en los que se han ejecutado las pruebas de entrenamiento de este trabajo.

Característica	PC	Portátil
Sistema operativo	Windows 10 Pro 21H2	Windows 11 Pro 22H2
Memoria	16GB DDR4 3200MHz	32GB DDR4 3200MHz
Procesador	AMD Ryzen 7 3700X	AMD Ryzen 7 5800H
Tarjeta gráfica	NVIDIA GeForce GTX 970	NVIDIA GeForce RTX 3060 Laptop

Personalmente, he trabajado con Unity en el pasado y mi nivel de conocimiento del motor y de C# es bajo. Aún así, no será una novedad descargar e instalar Unity, crear un nuevo proyecto, navegar por la interfaz, crear escenas e implementar scripts en C#.

Con respecto a los entornos utilizados, los dos utilizan un sistema operativo Windows debido a que son los que tenía instalados y configurados. Además, se trata del sistema operativo de referencia en el mundo de los videojuegos y es tiempo ahorrado al no tener que instalar y configurar un nuevo sistema operativo en alguno de los ordenadores.

5.6 Desarrollo

Habiendo especificado los requisitos, la arquitectura y el diseño de las clases, se pasa al desarrollo de la solución. Lo primero que hay que hacer es instalar el entorno en el que se va a desarrollar el producto. La instalación se puede ver en el Capítulo 6.

5.6.1. Conexión entre los entornos de Unity y Python

Una vez instalado el entorno de trabajo, una decisión que se tuvo que tomar es el orden y la forma en que se ejecutaría tanto el entorno de Unity como el de Python con MLAgents. La forma que aparece en los tutoriales consiste en ejecutar el programa *mlagents-learn* mediante la línea de comandos para que el entorno de Python quede a la espera de escuchar la comunicación que se inicia cuando se crea el primer agente en Unity. Para abstraer esta secuencia de pasos, ya que se ha buscado la posibilidad de añadir nuevas tecnologías de redes neuronales en el futuro y no todas las tecnologías tienen por qué tener los mismos pasos, se diseñó una nueva manera de realizar los mismos pasos sin tener que ejecutar manualmente *mlagents-learn*. La idea es que un desarrollador solamente necesite escribir un fichero de configuración XML, código para ejecutar las acciones recibidas por la red neuronal, código para enviar las entradas a la red neuronal y código para definir las recompensas y castigos que harán que el agente aprenda.

Para comenzar, ha sido necesario sobrescribir la funcionalidad de un agente de MLAgents para que no intente establecer una conexión con el entorno de Python nada más crearse. Para ello, se ha sobrescrito el método *OnEnable* que está heredado de la clase *MonoBehavior* de Unity. La primera vez que cualquier agente llama a *OnEnable*, se establece la conexión entre el entorno de Unity y el de Python, así que es importante que esto se haga una vez se haya puesto a la escucha el entorno de Python. Además, es en este momento cuando se crea el *cerebro* del agente a partir de sus atributos. El *cerebro* incluye la configuración de las entradas y salidas de la red neuronal, que será utilizada para comunicar al entorno de Python lo que espera enviar y recibir. Si los atributos del agente no son los correctos en este momento, el *cerebro* será incorrecto y el entrenamiento fallará. La

nueva funcionalidad de este método hace una llamada al método *OnEnable* de la clase de la que hereda, el agente de MLAgents, únicamente cuando los atributos ya se han actualizado.

La actualización de los atributos de un agente se hace mediante el método *Configure* disponible en todos los agentes. Este método se ejecuta cuando se le asigna una nueva configuración al agente y lo que hace es configurar los atributos necesarios. Es en este método donde un agente *MLAAgent* de GAIA asigna los atributos del agente del que hereda, permitiendo que su *cerebro* se cree con los valores correctos. Una vez se han asignado los valores, se llama al método *OnEnable* para que se cree el *cerebro* y se establezca la conexión con el entorno de Python. Esto quiere decir que el primer agente que se configura va a forzar que la comunicación entre Unity y Python comience, y antes de eso el entorno de Python tiene que estar escuchando. Finalmente, mediante un atributo de la instancia, se indica que el agente se ha inicializado.

Antes de que la comunicación se haya establecido, el entorno de Python se debe haber iniciado. Para ello existen la clase *MLAEnvironment* y la estructura *EnvironmentConfig*. En MLAgents puede haber varios comportamientos con diferentes configuraciones. La configuración de un comportamiento incluye la estructura de la red neuronal, la velocidad de aprendizaje y el algoritmo de aprendizaje por refuerzo a utilizar, entre otros parámetros. Las configuraciones se especifican en un fichero *yaml*, cuya localización en el sistema de ficheros del ordenador se indica posteriormente al programa *mlagents-learn*. Este proceso, que puede hacerse de forma manual, se ha automatizado en GAIA. *MLAEnvironment* es la clase encargada de recibir las configuraciones de los diferentes comportamientos (*EnvironmentConfig*) y de escribirlas en el fichero *yaml*. Finalmente, tiene un método *Start* que es el encargado de ejecutar el script que inicia *mlagents-learn* mediante el uso del *CommandExecuter* (Sección 5.6.4).

Para que el entorno de Python se inicie una vez todas las configuraciones hayan sido escritas en el fichero *yaml*, el gestor de GAIA, *GAIA_Manager*, es el encargado de gestionar los tiempos. Lo primero que ocurre al lanzar un juego, es que el controlador, *GAIA_Controller*, mira qué ficheros XML se le han asignado y éste le manda al gestor que lea estos ficheros y guarde la información necesaria para posteriormente crear las FSMs, los BTs y los agentes. Cuando el primer XML relativo a un agente se lee, el controlador crea un entorno para el tipo de agente leído (que actualmente solo puede ser un agente de MLAgents). Y por cada agente leído, añade la configuración al entorno.

Una vez ya se han leído y procesado todos los ficheros XML, se crean las FSMs, BTs y agentes, que lo hace el gestor bajo petición de los diferentes scripts de la escena. Cuando el primer agente de un tipo (*MLAAgent*) es creado, se inicia el entorno de ese tipo (*MLAEnvironment*) llamando al método *Start*. Como se ha dicho, éste método ejecuta *mlagents-learn* diciéndole dónde se encuentra el fichero de configuración que ha creado. Hasta que no se ha iniciado la escucha del entorno de Python, no se continúa con la ejecución, para evitar que un agente intente conectarse antes de que esté preparada su escucha. La creación de los agentes siguientes es más rápida, puesto que no requiere volver a iniciar el entorno de Python. El diagrama de flujo de los pasos mencionados puede verse en la Figura 5.8.

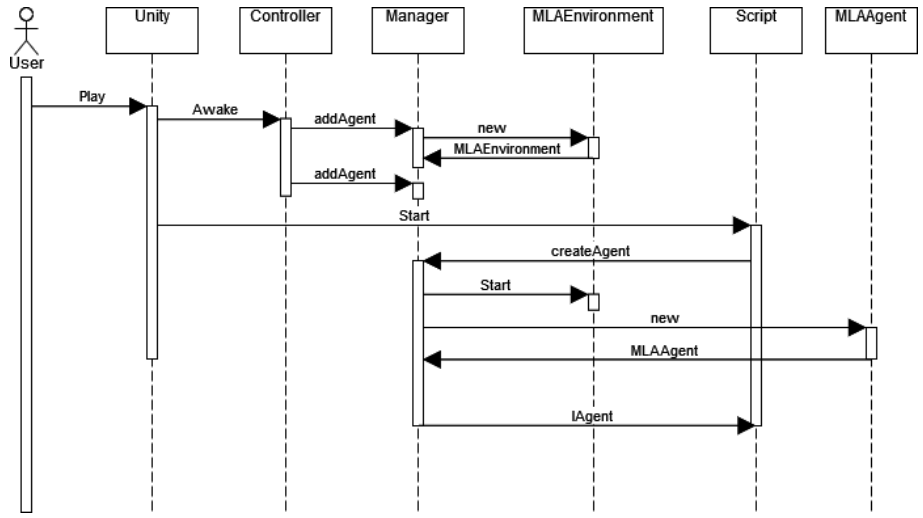


Figura 5.8: Diagrama de secuencia simplificado del flujo de ejecución desde que se inicia el juego hasta que se crea un agente.

5.6.2. Procesado del fichero XML

En la Figura 5.8 se muestra cómo el *Controller* se comunica con el *Manager* a través de su método *addAgent* para pedirle que procese un XML y guarde la información leída para la posterior creación de agentes. El procesamiento del fichero lo hace *GAIA_Parser* con ayuda de la funcionalidad de C# incluida en el espacio de nombres *System.Xml*, que facilita el acceso a las diferentes etiquetas y valores de su estructura.

La estructura de un fichero XML que define a un agente se muestra en el Apéndice A. Para el procesamiento de la especificación, se ha implementado en *GAIA_Parser* una función llamada *ParseAgent* que devuelve una instancia del objeto *AgentConfig* con la configuración leída. El procesamiento del fichero no tiene por qué seguir un orden. Una vez se lee su contenido y se crea un *XmlDocument*, es posible acceder a las etiquetas y a sus valores de forma aleatoria. No obstante, para facilitar la comprensión, el procesamiento se hace partiendo de los niveles más superficiales a los niveles más profundos de la jerarquía. Así pues, se comienza por el nombre del agente (*Name*), su tipo (*Type*), las observaciones (*Observations*), la petición automática de acciones (*AutomaticRequester*) y si es un entrenamiento o una inferencia (*Train*). Después se procesan las acciones, primero las continuas (*ContinuousActions*) y seguidamente las discretas (*DiscreteActions*). Finalmente se procesa el entorno (*Environment*). Por cada etiqueta con posibles hijos, se tiene un método distinto que la procesa. De esta manera, se tienen ocho métodos distintos que son los siguientes.

- **ParseAgent**. Método que procesa todo el documento, ayudándose de otras funciones de procesamiento.
- **ParseContinuousActions**. Procesa las acciones continuas.
- **ParseDiscreteActions**. Procesa las acciones discretas.
- **ParseEnvironment**. Procesa todo un entorno, incluyendo la red neuronal y las recompensas, tanto extrínsecas como las producidas por el módulo *Curiosity*.
- **ParseNetwork**. Procesa una red neuronal.

- **ParseMemory**. Procesa la información relacionada con la memoria de una red neuronal.
- **ParseExtrinsicRewards**. Procesa las recompensas extrínsecas.
- **ParseCuriosity**. Procesa las recompensas del módulo *Curiosity*.

En la Figura 5.9 se representa la jerarquía de llamadas que se realizan al procesar el fichero XML de un agente. En la misma, se ha omitido el prefijo *Parse* de todas las funciones para mejorar la claridad de lectura.

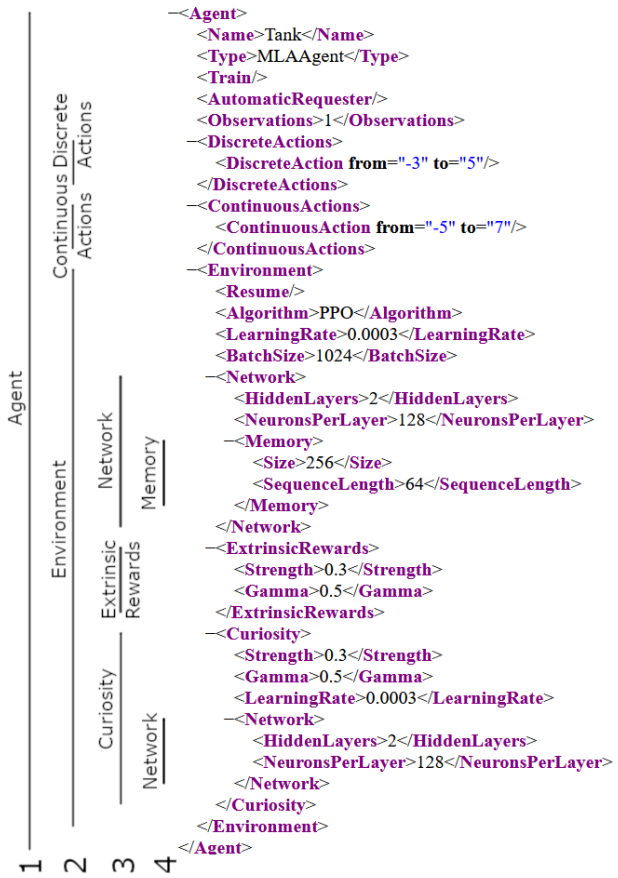


Figura 5.9: Jerarquía de llamadas al procesar la especificación de un agente.

En la Figura 5.9 vemos los diferentes niveles de llamadas, desde el 1 hasta el 4. Dentro del procesamiento de un agente en el nivel 1, se hace una llamada a todas las funciones del nivel 2. Esto quiere decir que, al procesar un agente, también se procesarán sus acciones discretas, acciones continuas y entorno. Dentro del nivel 2, el procesamiento del entorno hace llamadas a las funciones del nivel 3. Finalmente, el procesamiento del módulo *Curiosity* y de la red neuronal del entorno llaman a funciones del nivel 4. En caso de que la red neuronal de módulo *Curiosity* sea LSTM, también se le puede añadir información sobre sus características y la función *ParseNetwork* del nivel 4, crearía un quinto nivel llamando a *ParseMemory*.

5.6.3. Entornos de ejecución en Unity

Para poder realizar las pruebas, es necesario crear entornos de ejecución en los que un agente aprenda del comportamiento de FSMs, donde puedan aprender haciendo uso del aprendizaje por refuerzo y donde utilicen la inferencia para comprobar la calidad de

su aprendizaje.

Al disponer de un entorno previamente implementado en GAIA, este ha servido como base de los creados para el entrenamiento. El entorno de GAIA es un juego en que el jugador conduce un tanque que puede desplazarse por un mapa cerrado con obstáculos y disparar. En el mapa hay otros dos tanques, uno implementado mediante una FSM y otro con un BT, ambos utilizando la tecnología de GAIA. El objetivo del juego es reducir los puntos de vida de los tanques enemigos a 0 para ganar una ronda tras otra. El entorno puede visualizarse en la Figura 5.10

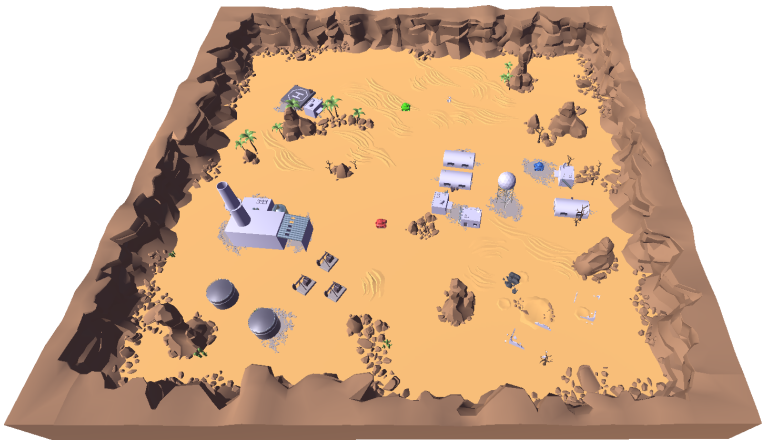


Figura 5.10: Imagen sacada del juego implementado en GAIA.

5.6.3.1. Imitación de una FSM

Este entorno de aprendizaje es una adaptación del entorno de GAIA de la Figura 5.10. Los cambios más notables han sido la eliminación de los tanques controlados por el jugador y por un BT, el duplicado del tanque controlado por una FSM con un script modificado, la eliminación de sonidos del juego, la eliminación de algunos elementos decorativos del mapa, la eliminación de la interfaz gráfica, la finalización de una ronda no solo cuando solo queda un tanque con puntos de vida sino también cuando se ha llegado a un límite de tiempo en segundos, el duplicado del mapa varias veces para que el entrenamiento sea más rápido y la adición de un nuevo script que gestiona el entrenamiento por imitación (Sección 5.6.5).

De entre los cambios, uno de los importantes es la modificación del script que utiliza una FSM para gestionar las acciones que toma un tanque. La idea que se ha llevado a cabo es, imitar a la FSM mediante el aprendizaje por refuerzo que viene implementado en MLAgents. Para ello, se comparan las salidas de la FSM con las salidas de la red neuronal en proceso de aprendizaje. Si las salidas son iguales, entonces se le aporta una recompensa. Si, en cambio, las salidas difieren, se le aporta un castigo. La relación entre las entradas y salidas de la FSM y la red neuronal, se ha hecho con ayuda del Algoritmo 5.1. Primero se han buscado aquellas variables utilizadas en los cambios entre estados. El código que genera estos eventos de cambio de estado en el tanque controlado por una FSM es el del Fragmento de código 5.1.

```
1  public List<int> BuscarEventos()
2  {
3    FSMevents.Clear();
```

```
4
5     if (distance < 5)
6     {
7       FSMevents.Add((int)Utils.EventTags.DEMASIADO_CERCA);
8     }
9
10    if (distance > 17)
11    {
12      FSMevents.Add((int)Utils.EventTags.DISTANCIA_NO_OK);
13    }
14
15    if (distance <= 17 && distance >= 5)
16    {
17      FSMevents.Add((int)Utils.EventTags.DISTANCIA_OK);
18    }
19
20    if (FSMevents.Count == 0)
21    {
22      FSMevents.Add((int)Utils.EventTags.NULL);
23    }
24
25    return FSMevents;
26  }
```

Fragmento de código 5.1: Método *BuscarEventos* que genera eventos de cambio de estado de la FSM que controla a un tanque en GAIA.

Sin tener en cuenta la última condición que se utiliza para el correcto funcionamiento de la máquina cuando no se da ningún evento, las tres anteriores utilizan el valor de una única variable para generar los eventos. Esa variable es *distance*, y contiene el valor actualizado de la distancia entre un tanque y su contrincante. El valor de esta variable va a ser, entonces, la única entrada a la red neuronal. Como con MLAgents no es necesario especificar las neuronas de entrada, sino las observaciones que se van a enviar, el número de neuronas de entrada es decidido por la herramienta. No obstante, la entrada podría ser una única neurona con el valor de la distancia.

Siguiendo con el Algoritmo 5.1, se buscan las salidas de la red neuronal. Para ello, se busca en el método *ExecuteActions* del script que gestiona el tanque. El código del método se muestra en el Fragmento de código 5.2.

```
1   public void ExecuteAction(int actionTag)
2   {
3     switch (actionTag)
4     {
5       case (int)Utils.ActionTags.APARTAR:
6       Aim();
7       MoveBack();
8       Shoot();
9       break;
10
11      case (int)Utils.ActionTags.DISPARAR:
12      Aim();
13      Shoot();
14      break;
15
16      case (int)Utils.ActionTags.MOVERSE:
17      Move();
```

```
18          break;
19      }
20  }
```

Fragmento de código 5.2: Método *ExecuteActions* que ejecuta las acciones del estado en el que se encuentra la FSM que controla a un tanque en GAIA.

Como se puede observar, la FSM puede realizar tres acciones, que son APARTAR, DISPARAR y MOVERSE. Esas van a ser las tres salidas de la red neuronal. De igual manera que con las entradas, como en MLAgents no se especifica la cantidad de neuronas de salida, lo que se hace es indicar el número de salidas que tiene y su tipo. En este caso, al ser acciones binarias, es decir, que pueden estar activas o no, una opción podría ser crear una acción discreta cuyo valor esté en el rango [0, 3] y relacionar cada uno de los cuatro posibles valores con una acción y uno de ellos con la acción nula, para darle la opción a la red neuronal de no estar ejecutando ninguna acción. Este método, no obstante, no permite que varias acciones se ejecuten al mismo tiempo. Otra opción, por la que se ha optado, es tener tres salidas en el rango [0, 1]. De esta forma, si la salida 0 es un 1, significa que la red neuronal ha decidido ejecutar la acción 0. En cambio, si su valor es un 0, la red neuronal ha decidido no ejecutarla.

Una vez se han establecido las entradas y salidas de una red neuronal, se implementan las recompensas y castigos. Cada vez que se le pida a la red neuronal una salida, se compara esa salida con la de la FSM. Si son iguales, se premia a la red neuronal con una recompensa. Si son diferentes, se le castiga. El código encargado de este proceso, junto con la creación del agente y el suministro de la entrada está en en Fragmento de código 5.3.

```
m_Agent = manager.createAgent(gameObject, "Tank");
m_Agent.SetOnActionsReceived(RewardFromFSMActions);
m_Agent.SetOnAddObservations(AddObservations);

void AddObservations()
{
  m_Agent.AddObservation(m_distance);
}

void RewardFromFSMActions(float[] ContinuousActions, int[]
    DiscreteActions)
{
  int CorrectImitations = 0;
  int TotalImitations = 0;


  if (IsCorrectAction(DiscreteActions[0],
      (int)Utils.ActionTags.APARTAR))
  {
    m_Agent.AddReward(1f);
    CorrectImitations++;
  }
  else
  {
    m_Agent.AddReward(-1f);
  }
  TotalImitations++;

```

```
27    if (IsCorrectAction(DiscreteActions[1],
          (int)Utils.ActionTags.DISPARAR))
28    {
29      m_Agent.AddReward(1f);
30      CorrectImitations++;
31    }
32    else
33    {
34      m_Agent.AddReward(-1f);
35    }
36    TotalImitations++;
37
38    if (IsCorrectAction(DiscreteActions[2],
          (int)Utils.ActionTags.MOVERSE))
39    {
40      m_Agent.AddReward(1f);
41      CorrectImitations++;
42    }
43    else
44    {
45      m_Agent.AddReward(-1f);
46    }
47    TotalImitations++;
48
49    if (m_ImitationManager)
50    {
51      m_ImitationManager.AddImitationSample(CorrectImitations,
            TotalImitations);
52    }
53  }
54
55  private bool IsCorrectAction(int RNNAction, int FSMActionTag)
56  {
57    return (RNNAction == 1 && FSMactions.Contains(FSMActionTag)) ||
58    (RNNAction == 0 && !FSMactions.Contains(FSMActionTag));
59  }
```

Fragmento de código 5.3: Configuración de una red neuronal que aprende del comportamiento de una máquina de estados finitos.

Primero de todo se crea un agente y se configura llamando a los métodos *SetOnActionsReceived* y *SetOnAddObservations* que le dicen al agente lo que tiene que hacer cuando recibe una acción de la red neuronal y cuando necesita recolectar observaciones o entradas a la red neuronal.

A continuación se define el método *AddObservations*. Lo que hace es añadir como observación del agente la distancia al jugador. Finalmente, se define el método *RewardFromFSMActions*, llamado cuando la red neuronal se ha ejecutado para obtener una salida. En ese momento, se comparan las acciones que está ejecutando la FSM con las acciones de la red neuronal. Por cada una de las posibles acciones, si la acción de la FSM y la red neuronal coinciden, se añade una recompensa de valor 1 al agente. Si las acciones no coinciden, se añade un castigo de valor -1. El objetivo del aprendizaje por refuerzo es que la red neuronal obtenga el valor de $recompensas + castigos$ más elevado posible, por lo que tenderá a entrenar para que sus salidas sean iguales a las salidas de la FSM y obtener un valor alto de recompensas y un valor bajo de castigos en valor absoluto.

En el Fragmento de código 5.3, también aparecen dos variable, *CorrectImitations* que guarda el número de imitaciones correctas, y *TotalImitations* que almacena el total de acciones que puede imitar el agente. Por cada una de las posibles acciones, cuando la red neuronal y la FSM coinciden, se aumenta en uno el valor de la variable. Después de comprobar todas las acciones, se añade la información de las imitaciones correctas y del total de acciones, lo que permitirá al gestor del entrenamiento (Sección 5.6.5) comprobar el porcentaje de imitación en cada momento de la ejecución.

Para visualizar la diferencia entre el nivel del entorno de aprendizaje y del entorno base, se pueden observar en las Figuras 5.10 y 5.11. En el caso del entorno de aprendizaje, el mapa se repite cuatro veces, y en cada mapa hay dos tanques controlados por la misma FSM. Cada uno de los tanques, tiene un agente que monitoriza las acciones de la FSM y se conecta con la red neuronal que está siendo entrenada en Python, a la cual le envía las observaciones y las señales de recompensa y castigo. También es posible observar la eliminación de las dunas del mapa y de algunos obstáculos innecesarios.

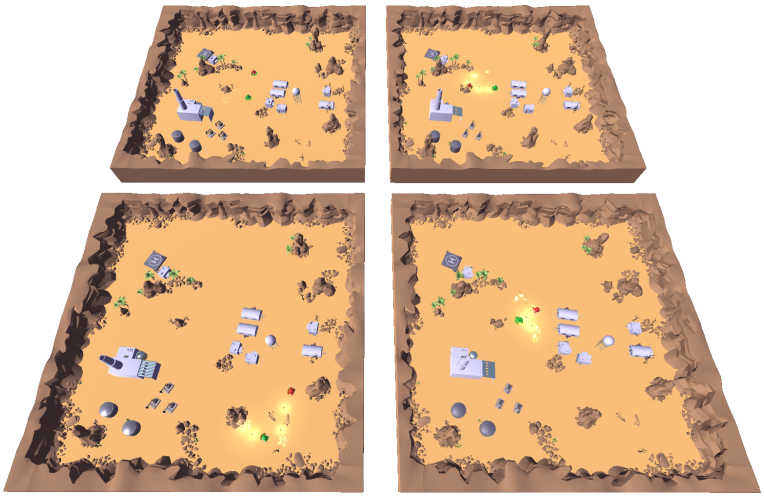


Figura 5.11: Imagen sacada del entorno de aprendizaje creado en Unity para que un agente aprenda por imitación de FSMs.

5.6.3.2. Entrenamiento contra una FSM

Una vez está la red neuronal entrenada por imitación a una FSM, se quiere que la red neuronal siga entrenando para modificar su comportamiento y optimizar la consecución de sus objetivos. En el caso del juego que se ha decidido utilizar para probar el comportamiento de las redes neuronales, el objetivo del agente es reducir los puntos de vida de su contrincante a 0. Este entrenamiento es similar al de la sección anterior, salvo por los detalles que se comentan a continuación.

En este caso, no se utiliza el *ImitationManager* que solo es necesario en el entrenamiento por imitación. Por otro lado, el script del agente ya no se encarga únicamente de enviar al entorno de Python las observaciones y las recompensas, si no que también se encarga de ejecutar las acciones. Finalmente, las recompensas son distintas que en el entrenamiento por imitación, donde las recompensas y castigos se dan acorde a si las acciones de la red neuronal coinciden con las de la FSM o no.

Como el objetivo de este entrenamiento es reducir los puntos de vida del tanque enemigo a 0 en el mínimo tiempo posible y recibiendo el mínimo daño, las recompensas que se han decidido utilizar son las de la Tabla 5.4.

Tabla 5.4: Recompensas de un agente que entrena con el objetivo de reducir los puntos de vida de su enemigo a 0 lo más rápido posible, recibiendo la menor cantidad de daño y utilizando el menor número de proyectiles posible.

Momento	Valor	Objetivo
Disparar	-0.5	Reducir el número de disparos
Ganar una ronda	500	Reducir los puntos de vida del enemigo a 0
Perder una ronda	-500	Evitar que el enemigo reduzca los puntos del vida del agente a 0
Cálculo de físicas	-0.1	Reducir el tiempo que tarda en terminar una ronda
Los puntos de vida del enemigo disminuyen	Puntos de vida restados	Restar los puntos de vida del enemigo
Los puntos de vida del agente disminuyen	-(Puntos de vida restados)	Evitar que el agente reciba daño

5.6.4. Ejecutor de comandos

El ejecutor de comandos o *CommandExecuter* es una clase que ha sido creada para suplir la necesidad de MLAgents de iniciar el entorno de Python mediante la llamada a un programa por línea de comandos. La solución planteada ha sido la creación de un pequeño script que es llamado desde Unity a través del *CommandExecuter* y que, a su vez, llama a *mlagents-learn* pasando los parámetros necesarios. De esta forma, el entorno de MLAgents, *MLAEnvironment*, puede iniciar el entorno de Python tal y como muestra el Fragmento de código 5.4.

```
1  public void Start()
2  {
3    GAIA.Utils.Log("Executing learn.bat", LOG_FILE);
4
5    if (Resume)
6    {
7      CommandExecuter.ExecuteNewWindowSleep("learn.bat " + ModelName
           + " resume" , 6000);
8    }
9    else
10   {
11     CommandExecuter.ExecuteNewWindowSleep("learn.bat " +
           ModelName , 6000);
12   }
13 }
```

Fragmento de código 5.4: Método start de la clase *MLAEnvironment* que ejecuta el entorno de Python mediante el *CommandExecuter*.

El método estático *ExecuteNewWindowSleep*, lo que hace es ejecutar el comando que se le pasa como primer argumento en un proceso en segundo plano y espera el número

de milisegundos especificado en el segundo parámetro antes de continuar. Esta espera se hace porque existe un lapso de tiempo entre que se llama al script hasta que el entorno de Python está preparado para escuchar. Cuando el método *Start* del entorno se llama, se abre una consola como la mostrada en la Figura 5.12.

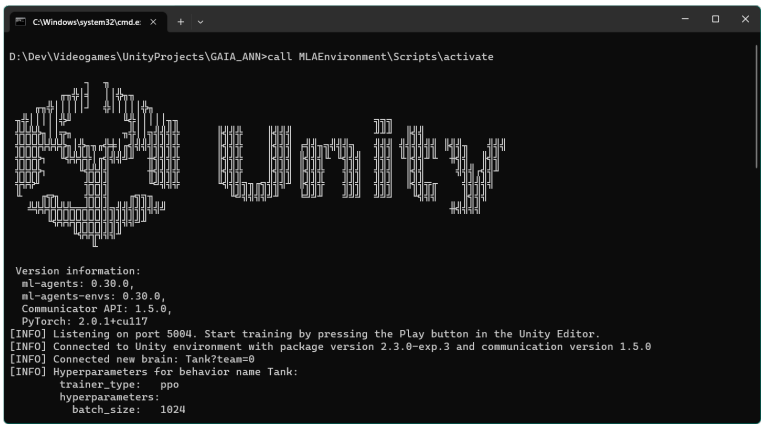


Figura 5.12: Consola de Windows que se abre al ejecutar el método *Start* de *MLAEnvironment*.

Una vez *mlagents-learn* se ha ejecutado y en la consola aparece el texto "[INFO] Listening on port 5004. Start training by pressing the Play button in the Unity Editor", significa que el entorno de Python está listo para comenzar la conexión con Unity.

Durante la ejecución del entrenamiento, la consola se mantendrá abierta y se podrá ver cuándo el entrenamiento ha llegado a un *checkpoint* y también el momento en el que se guarda información de estadísticas para mostrar en Tensorflow. Cuando guarda las estadísticas, el entorno de Python muestra por la consola de la Figura 5.12 cuál es la media de las recompensas obtenidas en los episodios junto con la desviación estándar. Lo mejor es conseguir un valor alto de recompensa y uno bajo de desviación estándar, lo que significa que el agente está consiguiendo su objetivo (valor alto de recompensa) y que lo hace a menudo (valor bajo de desviación estándar). En el caso de una imitación, su significado es que el agente está tomando las mismas decisiones que el comportamiento a imitar.

5.6.5. Gestor del entrenamiento por imitación

Un problema que emerge del entrenamiento con MLAgents es determinar cuándo se guarda el estado de la red neuronal y calcular el porcentaje de imitación que ha conseguido la red neuronal entre dos guardados del estado o *checkpoints*.

Desde el fichero de configuración *yaml* es posible determinar el número de experiencias de una red neuronal antes de que haya un guardado de su estado. También se puede determinar el número de experiencias antes de que se haga un guardado de las estadísticas del entrenamiento, que podrán ser visualizadas desde Tensorflow. No obstante, no es posible monitorizar desde Unity el número de experiencias para saber cuándo está realizando un *checkpoint*. Tampoco es posible enviar una petición al entorno de Python para que responda con ese número. Python tampoco comunica a Unity cuándo se está guardando el estado.

Una solución sencilla es controlar el directorio donde se guarda el estado de la red neuronal. Cuando se crea un nuevo fichero en ese directorio quiere decir que se ha guardado el estado de la red neuronal. Ese es el trabajo del gestor del entrenamiento. Cada segundo, el gestor comprueba cuál es el *checkpoint* más reciente. Si es diferente al que tenía guardado como más reciente, es que se ha vuelto a guardar el estado de la red neuronal. En ese momento, el gestor calcula el porcentaje de imitación a partir de las muestras que ha recibido hasta ese momento y reinicia las muestras para comenzar a calcular el porcentaje de imitación del siguiente *checkpoint* desde cero. El gestor se configura con un porcentaje de imitación límite y un número entero, llamémosle X, de manera que el aprendizaje cesa cuando el porcentaje de imitación está por encima del límite y no ha mejorado en los últimos X *checkpoints*. Por cada nuevo *checkpoint*, el gestor del entrenamiento hace un *Log* (ver Sección 5.6.7) que muestra por la consola de Unity y guarda en un fichero de texto el porcentaje de imitación.

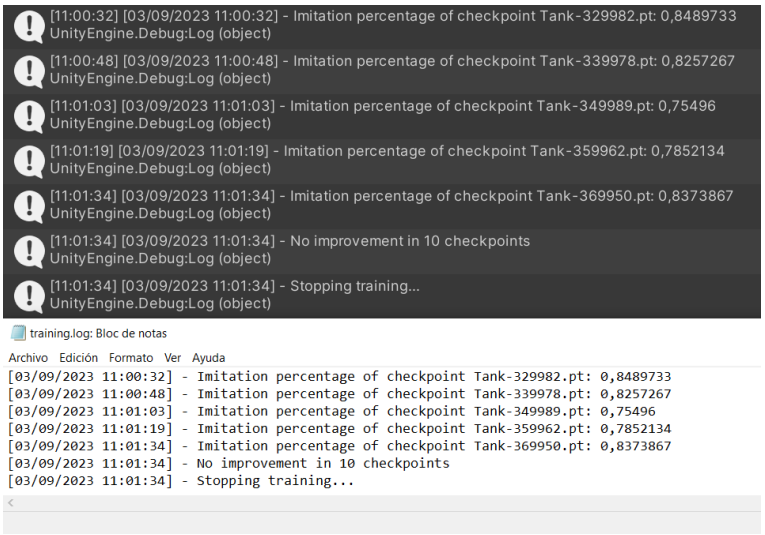


Figura 5.13: Mensajes por la consola de Unity y fichero de logs que muestran la hora y el porcentaje de imitación de un entrenamiento.

5.6.6. Gestor de batallas

Otra funcionalidad implementada para llevar a cabo las pruebas ha sido el gestor de batallas. Su objetivo es llevar la cuenta de cuántas veces uno de los dos agentes en batalla gana una ronda para posteriormente calcular cuál de los dos agentes cumple mejor su objetivo de derrotar a su contrincante. Así, se tiene una forma de medir la calidad de los agentes. El gestor de batalla, a medida que recoge resultados de las rondas, muestra por la consola de Unity y escribe en un fichero de texto estos resultados. El porcentaje de rondas ganadas, tiene en cuenta únicamente las rondas que ha ganado o perdido. Las rondas que acaban en empate no se tienen en cuenta para no obtener un resultado bajo. De hecho, el gestor muestra un mensaje cada 20 partidas en las que el agente gana o pierde, sin tener en cuenta las empatadas. Se puede ver cómo se muestra el porcentaje de victorias en Unity y en el fichero de texto en la Figura 5.14. En la misma Figura, es posible observar cómo una ronda empatada no modifica el número de rondas ganadas ni el total, manteniendo el porcentaje de victorias intacto.

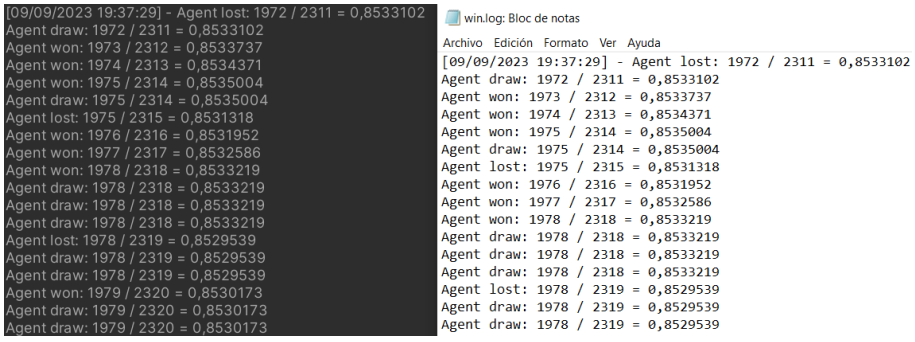


Figura 5.14: Mensajes por la consola de Unity y el fichero de logs que muestran la hora y el porcentaje de imitación de un entrenamiento.

5.6.7. Utilidades

Otra de las características que se han añadido en GAIA es una clase llamada *GAIA_Utils* con métodos estáticos que pueden utilizarse en diversas situaciones o desde diferentes clases. Los seis métodos que contiene son los siguientes.

- **name2Tag**. Convierte un string en un tipo enumerado que se utiliza en las FSMs.
- **ChangeRange**. Dado un valor decimal entre el rango [a, b], devuelve cuál sería su valor en el rango [c, d].
- **ShiftRange**. El trabajo que realiza es muy sencillo y se ha pasado a una función porque así se entiende qué es lo que se pretende hacer. Desplaza un número entero dentro del rango [0, b] a otro rango [c, d]. Para ello, devuelve la suma de los dos números enteros que recibe como argumentos.
- **StrToFloat**. Convierte un string en un número decimal donde las separaciones decimales son puntos. Se utiliza en el procesamiento de un XML de un agente.
- **StrToInt**. Convierte un string en un número entero. Se utiliza en el procesamiento de un XML de un agente.
- **Log**. Muestra un mensaje por la pantalla de Unity y lo guarda en un fichero de texto. Es una forma de centralizar los mensajes que genera GAIA para poder modificarlos o desactivarlos rápidamente.

5.6.8. Problema con los sonidos

Uno de los problemas encontrados en la realización de este proyecto fue que el entrenamiento se quedaba bloqueado. Cuando se ejecutaba un entorno de aprendizaje para que un agente aprendiera a imitar a una FSM, cada cierto tiempo, Unity se paralizaba y volvía a responder al cabo de unos segundos. Esto es un problema no solo porque el entrenamiento puede tardar más tiempo en realizarse, si no porque el entorno de Python de MLAgents se cierra cuando deja de recibir información durante un tiempo. Y de vez en cuando, el tiempo de parálisis de Unity era más largo que el tiempo de espera de Python, por lo que el entrenamiento cesaba repentinamente.

Para solventar el problema, se probó a eliminar las corrutinas de Unity que se estaban utilizando para gestionar las rondas. El problema continuó. En aquel momento, todavía no se estaba utilizando la GPU para realizar el entrenamiento. Quizá el problema podría

estar en que el procesador no tenía suficiente potencia (aunque su porcentaje de uso no superaba el 40 %). Se probó con la instalación de CUDA en Windows y las librerías necesarias de Python para que el entrenamiento se realizara en la GPU. Este cambio tampoco solucionó el problema. Otro intento fue dejar de instanciar los proyectiles de los tanques, que podrían estar generando un exceso de consumo de la memoria o que el juego se quedara bloqueado porque las instancias de los proyectiles no se estuvieran eliminando correctamente al crearse muchos al mismo tiempo. Tampoco hubo suerte. La siguiente prueba fue crear un ejecutable del juego en vez de ejecutarlo en el propio editor. Quizá alguna característica del editor estaba entorpeciendo el entrenamiento, pero no fue el caso. Finalmente se encontró la solución cuando, cansado de escuchar todo el tiempo los sonidos de los tanques a la hora de entrenar, se decidió eliminar los efectos de sonido del motor y del movimiento del tanque.

CAPÍTULO 6
Implantación

Para poder hacer uso del asset de GAIA con el soporte para redes neuronales, es necesario instalar sus dependencias. Para facilitar el proceso, se ha creado un script que instala automáticamente todas las dependencias de Python después de comprobar que las versiones de Python y Pip son las soportadas. El código del script se puede leer en el apéndice B. Los pasos a seguir para su instalación se especifican a continuación.

1. Instalar Unity Hub.
2. Instalar la versión 2022.3.4f11 de Unity a través de Unity Hub.
3. Crear un nuevo proyecto de Unity.
4. Instalar el paquete *AI Navigation* de Unity a través de su gestor de paquetes.
5. Descargar la Release 20 de *MLAgents* del repositorio de GitHub[1].
6. Añadir el paquete recién descargado de *MLAgents* al proyecto de Unity.
7. Copiar el directorio de GAIA[2] en la carpeta Assets del proyecto de Unity.
8. Copiar el script de instalación de dependencias del apéndice B en el directorio raíz del proyecto de Unity.
9. Ejecutar el script de instalación.

Una vez se han completado todos los pasos, GAIA con el soporte para redes neuronales estará listo para utilizarse en el proyecto.

6.1 Pruebas

En esta sección de la memoria se procede a explicar las diferentes pruebas a realizar para comprobar el funcionamiento del software implementado.

[1] https://github.com/Unity-Technologies/ml-agents/releases/tag/release_20
[2] https://github.com/tetracube/GAIA/tree/master/GAIA/Assets/GAIA

6.1.1. Prueba 1. Dar valores a los parámetros

El objetivo de esta prueba es encontrar valores para los diferentes parámetros de aprendizaje que consigan un porcentaje de imitación de al menos un 80 %. Para diferentes valores que den resultados por encima del 80 %, el criterio de selección es escoger el que mejor porcentaje de imitación obtenga. Entre dos resultados con porcentajes de imitación similares, se escogerán los valores que generan una red neuronal más sencilla, donde sencilla es con menos pesos para entrenar. Los parámetros de aprendizaje a variar y los valores que van a tomar son los siguientes.

- **Tamaño de lote**. 128, 256, 512, 1024.
- **Velocidad de aprendizaje**. 0.0001, **0.0003**, 0.0005, 0.001, 0.005, 0.01, 0.05.
- **Número de capas ocultas**. 1, **2**, 4, 8.
- **Número de neuronas por capa oculta**. 32, 64, **128**, 256, 512.
- **Tamaño de la memoria de la red neuronal**. **128**, 256, 512.
- **Longitud de la secuencia de la memoria de la red neuronal**. 32, **64**, 128.

Como se puede observar, comparando con el total de parámetros que se pueden especificar (Sección 5.2.1), no se prueban diferentes valores de los parámetros relacionados con las recompensas extrínsecas ni aquellos relacionados con el módulo de Curiosidad. El módulo de Curiosidad se ha desactivado para esta prueba. La decisión se ha tomado porque cuando se quiere realizar una imitación, las únicas recompensas que se quieren tener en cuenta son aquellas que el agente obtiene al imitar correctamente, o no, los comportamientos de la FSM. No se quiere que el agente reciba recompensas positivas al realizar acciones que no realiza a menudo.

Si se trataran de probar todas las combinaciones, serían 5040 diferentes, y el tiempo de entrenamiento sería de 1680 horas suponiendo que cada entrenamiento es de 20 minutos y si se realizan secuencialmente.

Por este motivo, en vez de probar todas las posibles combinaciones, se darán valores por defecto a todos los parámetros y se irán variando los parámetros de arriba a abajo tal y como aparecen en la lista hasta obtener los resultados esperados. Los valores por defecto de cada uno de los parámetros aparece en negrita en la lista de arriba. Cuando se vaya a probar un parámetro, se usarán, para los parámetros anteriores, los mejores valores obtenidos. De esta forma, se tendrán que ejecutar 21 entrenamientos, que serían 7 horas si cada ejecución es de 20 minutos.

Se dejará de entrenar la red cuando el porcentaje de imitación no aumente a lo largo de 10 *checkpoints*.

6.1.2. Prueba 2. Enfrentamiento de un agente entrenado al 80 % y una máquina de estados finitos

Con el objetivo de comparar la eficiencia de un agente con un porcentaje de imitación superior al 80 % recién entrenado y un agente que, después de imitar un mínimo de 80 % ha continuado entrenando mediante aprendizaje por refuerzo, primero se pondrán

a combatir al agente recién salido de la imitación contra una máquina de estados finitos, los dos con el mismo objetivo: bajar a 0 los puntos de vida del contrincante. El agente que realiza la imitación utiliza los valores de los parámetros obtenidos en la prueba anterior.

El enfrentamiento se dejará ejecutar durante 1 hora y se obtendrá el porcentaje de veces que un agente vence a una máquina de estados finitos. Con este valor, será posible hacer comparaciones con el valor obtenido en la prueba 6.1.3.

6.1.3. Prueba 3. Enfrentamiento de un agente entrenado con aprendizaje por refuerzo y una máquina de estados finitos

La tercera prueba consiste en entrenar a un agente, una vez su grado de imitación contra una máquina de estados finitos es de mínimo 80 %, mediante aprendizaje por refuerzo, durante 1, 2, 4 y 8 horas. Cada uno de los 4 agentes entrenados, se enfrentará a la máquina de estados finitos y se calculará el porcentaje de veces que el agente reduce los puntos de vida de su oponente a 0 antes de que lo haga el otro.

Junto con la prueba dos, esta prueba dará a conocer si las redes neuronales implementadas utilizando la tecnología de MLAgents son capaces de superar a una máquina de estados finitos en la tarea de conseguir su objetivo una vez han aprendido lo suficiente.

6.1.4. Prueba 4. Enfrentamiento de una agente de imitación y otro de aprendizaje por refuerzo

Finalmente, para comprobar no solo cómo se comporta el agente entrenado mediante aprendizaje por refuerzo contra una máquina de estados finitos, si no también contra otro agente cuyo comportamiento sea una imitación de mínimo el 80 % de una máquina de estados finitos, se pondrán a combatir a los agentes resultado de las pruebas 2 y 3. El agente con mejores resultados de la prueba 6.1.3 será el elegido para esta prueba.

Se calculará el porcentaje de veces que el agente de la prueba 3 consigue bajar los puntos de vida del agente de la prueba 2 a 0 de todos los enfrentamientos que tengan en 15 minutos.

6.2 Resultados

A continuación, se muestran y describen los resultados obtenidos de las pruebas descritas en la Sección 6.1

6.2.1. Resultados de la Prueba 1

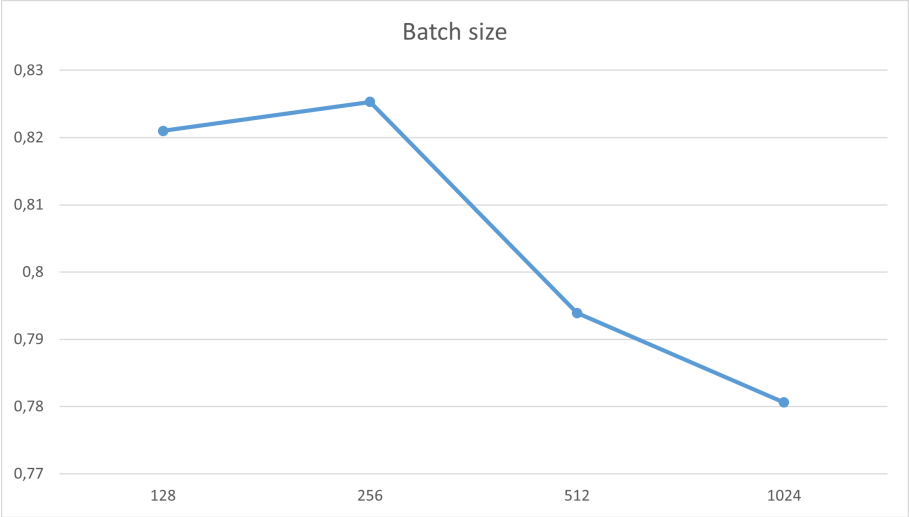
Para esta prueba, se han probado valores de diferentes parámetros del agente para ver su comportamiento. Los diferentes valores se muestran en la Figura 6.1. El número del tanque es el que se utiliza en las gráficas durante esta sección. De esta forma, cuando se esté observando la gráfica del tanque *Tank3*, se sabrá que los valores utilizados para el entrenamiento han sido 512, 0.0003, 2, 128, 128 y 64.

	batch size	learning rate	hidden layers	neurons per layer	memory size	memory length	imitation
1	128	0.0003	2	128	128	64	0.821
2	256	0.0003	2	128	128	64	0.8253
3	512	0.0003	2	128	128	64	0.7939
4	1024	0.0003	2	128	128	64	0.7806
5	256	0.0001	2	128	128	64	0.7795
6	256	0.0005	2	128	128	64	0.7988
7	256	0.001	2	128	128	64	0.7984
8	256	0.005	2	128	128	64	0.6684
9	256	0.01	2	128	128	64	0.67
10	256	0.05	2	128	128	64	0.597
11	256	0.0003	1	128	128	64	0.809
12	256	0.0003	4	128	128	64	0.8567
13	256	0.0003	8	128	128	64	0.8564
14	256	0.0003	4	32	128	64	0.8453
15	256	0.0003	4	64	128	64	0.8664
16	256	0.0003	4	256	128	64	0.8204
17	256	0.0003	4	512	128	64	0.7906
18	256	0.0003	4	64	256	64	0.8864
19	256	0.0003	4	64	512	64	0.8673
20	256	0.0003	4	64	256	32	0.8804
21	256	0.0003	4	64	256	128	0.8764

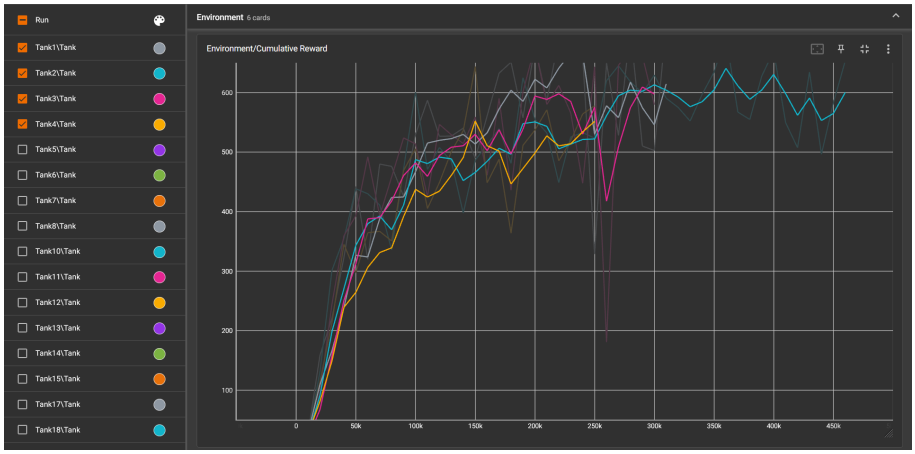
Figura 6.1: Valores que han tomado los diferentes parámetros a probar en el aprendizaje de un agente que imita el comportamiento de una FSM.

Manteniendo los valores por defecto de todos los parámetros y probando los diferentes valores propuestos para el tamaño de lote, el resultado con mejor imitación obtenida es utilizando 256, tal y como se muestra en la Figura 6.2a. Este será, entonces, el valor utilizado para las pruebas con los demás parámetros. Cabe mencionar que el valor del porcentaje de imitación mostrado en esta sección se ha calculado como la media de los últimos 10 porcentajes obtenidos en el entrenamiento, para que el valor obtenido no solo tenga en cuenta lo que haya podido pasar en el último *checkpoint*, sino que tenga en cuenta los últimos 10 por debajo del máximo obtenido.

Además, la Figura 6.2b muestra cuál es el valor medio de la recompensa obtenida en cada episodio a lo largo del tiempo. Un episodio equivale a una ronda del juego, hasta que uno de los tanques llega a tener cero puntos de vida o hasta que se ha sobrepasado un tiempo límite. Así pues, con episodios más cortos en los que las FSM acaban con su contrincante rápidamente, hay menos tiempo de ronda y por lo tanto menos recompensa. Como puede observarse, las redes neuronales aprenden deprisa hasta cierto punto alrededor de los 550 puntos de recompensa. A partir de ahí, las recompensas oscilan hacia arriba y hacia abajo. Es un punto en que el grado de imitación lleva sin aumentar varios ciclos. Si se continúa el aprendizaje desde ese punto, el comportamiento de la recompensa se mantiene y no logra incrementar. Por esa razón y porque no es necesario que la red neuronal consiga comportarse como la FSM al 100 %, es un buen punto para dejar de aprender.



(a) Porcentaje de imitación para diferentes valores del tamaño de lote.

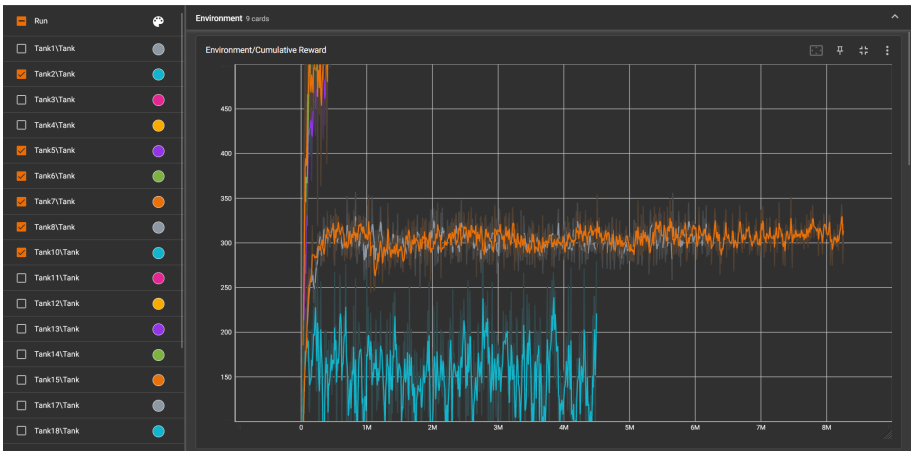


(b) Valor acumulado de la recompensa para diferentes valores del tamaño de lote.

Figura 6.2: Resultados del entrenamiento para diferentes valores del tamaño de lote.

El siguiente parámetro que se ha probado es la velocidad de aprendizaje, obteniendo unos porcentajes de imitación que se muestran en la Figura 6.3a. Como se ve en la figura, cuando el valor de la velocidad de aprendizaje es elevado, por encima de 0.005, el porcentaje de imitación disminuye considerablemente por debajo del 70 %. Al no llegar al 80 % establecido en el *ImitationManager*, el entrenamiento no cesa y se ha terminado manualmente después de varias horas. La Figura 6.3b es un claro ejemplo de lo que pasa con el valor de la recompensa a lo largo del tiempo, oscila subiendo y bajando alrededor de un valor constante que se puede representar como una recta en el plano. Vistos los resultados, se escoge el valor 0.0003 como valor de la velocidad de aprendizaje para las siguientes pruebas, ya que es el que mejor porcentaje de imitación reporta y es el elegido por defecto en MLAgents.

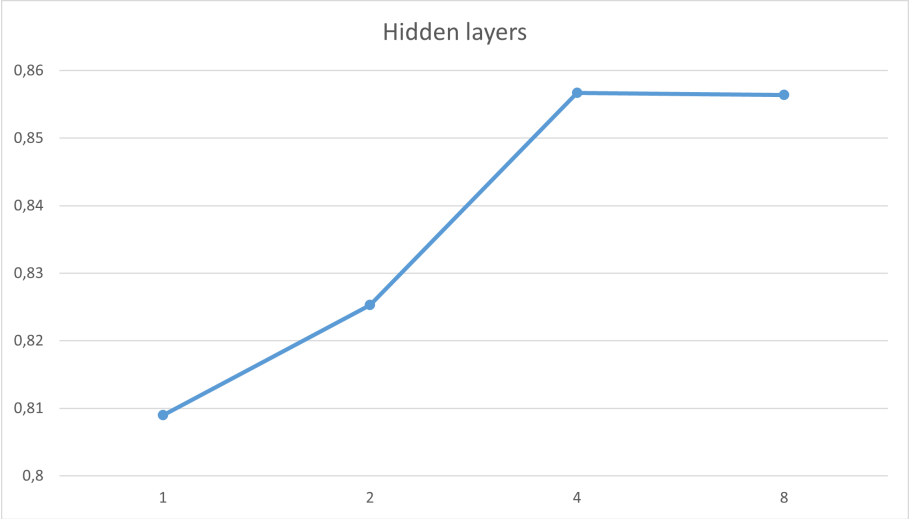
(a) Porcentaje de imitación para diferentes valores de la velocidad de aprendizaje.



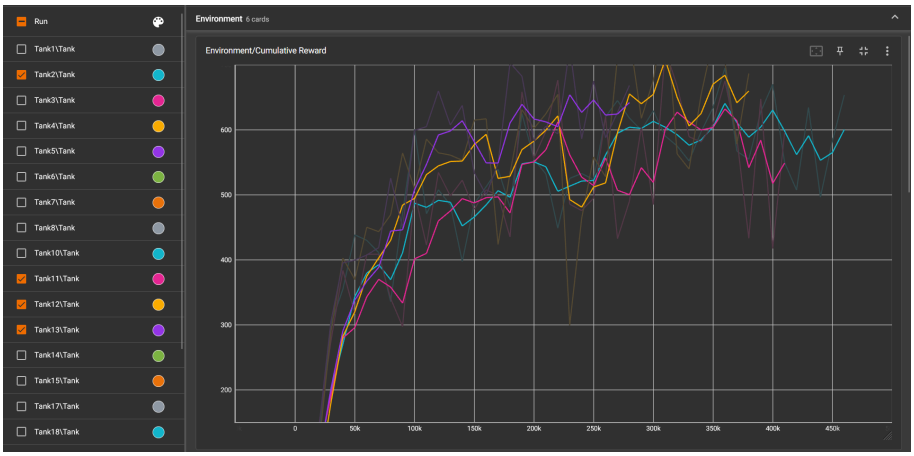
(b) Valor acumulado de la recompensa para diferentes valores de la velocidad de aprendizaje.

Figura 6.3: Resultados del entrenamiento para diferentes valores de la velocidad de aprendizaje.

A continuación, se ha probado con el número de capas ocultas de la red neuronal, obteniendo los resultados de las Figuras 6.4a y 6.4b. En ellas se puede observar que, con una y dos capas ocultas, el resultado es un porcentaje de imitación inferior al obtenido con 4 y 8 capas ocultas. La elección entre 4 y 8 capas ocultas se ha hecho pensando en la complejidad de la red neuronal. Cuantas menos capas ocultas, más sencilla es la red neuronal, con menos pesos que modificar y menor tiempo de inferencia. Es por ello que se ha decido elegir el valor 4 del número de capas ocultas para las siguientes pruebas.



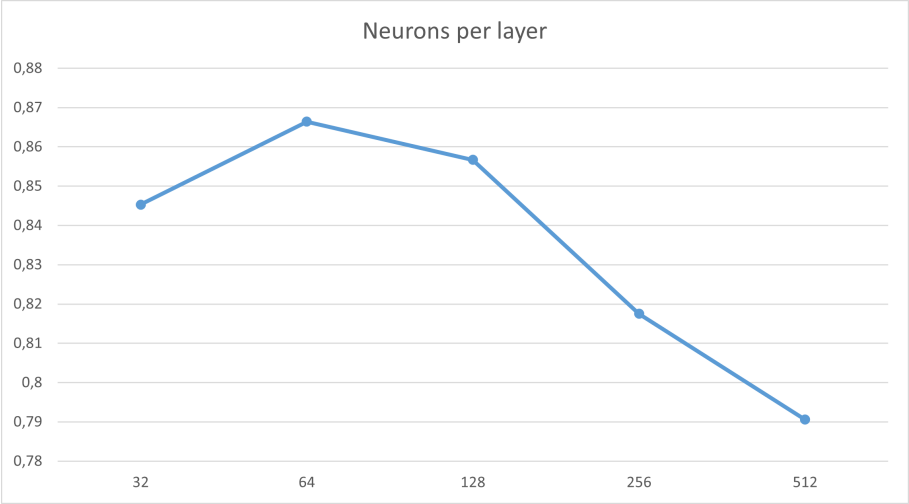
(a) Porcentaje de imitación para diferentes valores del número de capas ocultas.



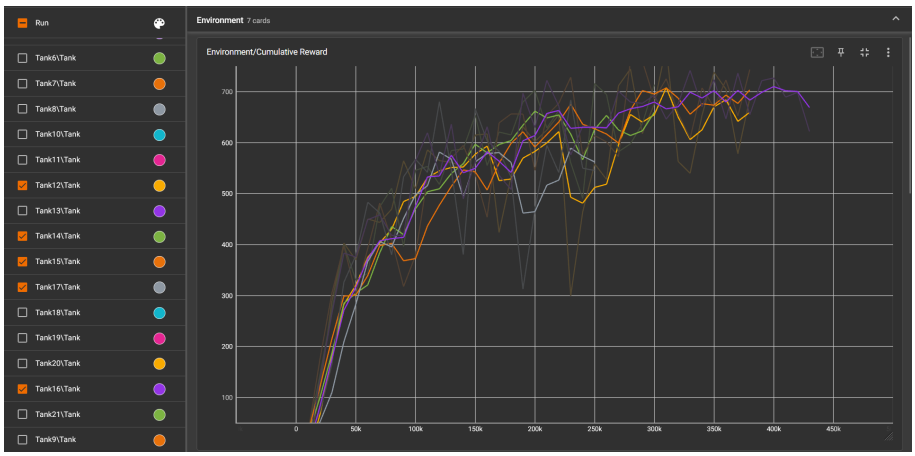
(b) Valor acumulado de la recompensa para diferentes valores del número de capas ocultas.

Figura 6.4: Resultados del entrenamiento para diferentes valores del número de capas ocultas.

El número de neuronas por capa oculta ha sido el siguiente parámetro a probar, y los resultados se muestran en las Figuras 6.5a y 6.5b. Al pasar de 32 a 64 neuronas por capa, el porcentaje de imitación aumenta, pero al seguir incrementando este valor a 128, 256 y 512, el resultado es cada vez peor y además la complejidad de la red neuronal aumenta. El valor escogido es el que consigue mejor porcentaje de imitación, 64 neuronas por capa.



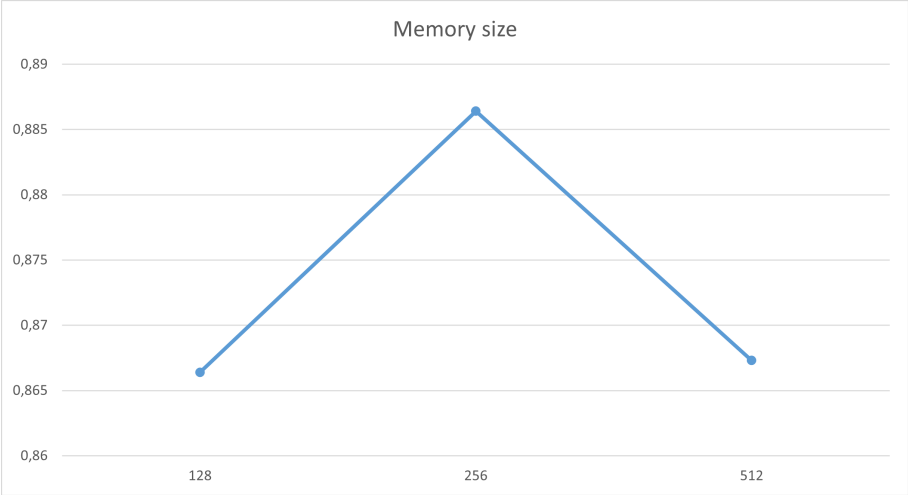
(a) Porcentaje de imitación para diferentes valores del número de neuronas por capa oculta.



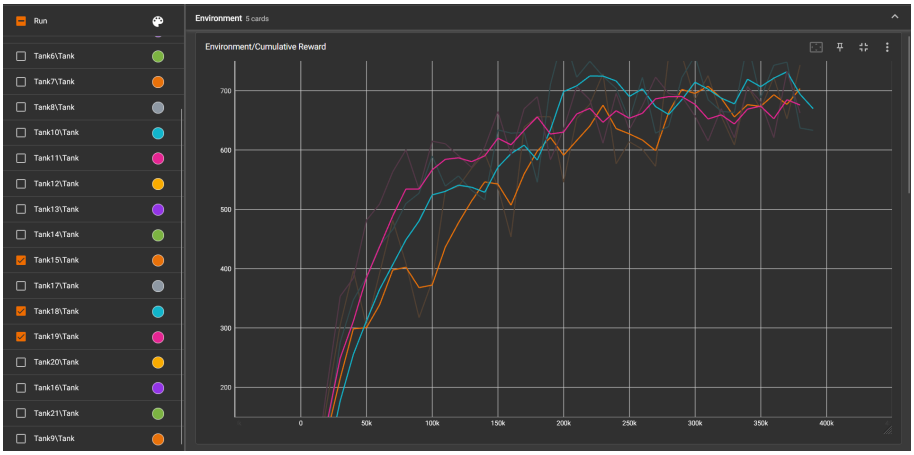
(b) Valor acumulado de la recompensa para diferentes valores del número de neuronas por capa oculta.

Figura 6.5: Resultados del entrenamiento para diferentes valores del número de neuronas por capa oculta.

Los últimos dos parámetros a probar están relacionados con la memoria de la red neuronal LSTM, que es la que se está utilizando para los agentes. Primero, se han probado diferentes valores de tamaño del estado oculto, obteniendo los resultados de las Figuras 6.6a y 6.6b. De entre los tres valores probados, 256 es el que mejores resultados ha obtenido con diferencia y por eso ha sido el valor elegido para las pruebas de la longitud de la secuencia.



(a) Porcentaje de imitación para diferentes valores del tamaño del estado oculto de la red neuronal.

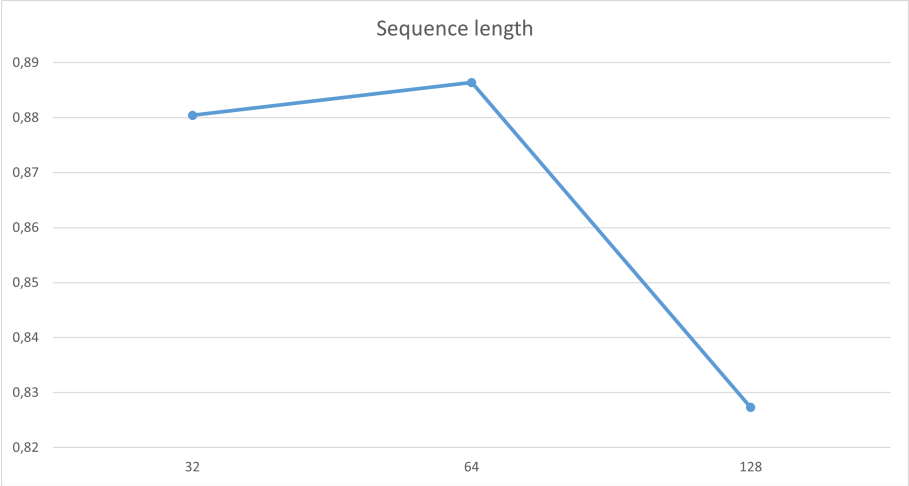


(b) Valor acumulado de la recompensa para diferentes valores del tamaño del estado oculto de la red neuronal.

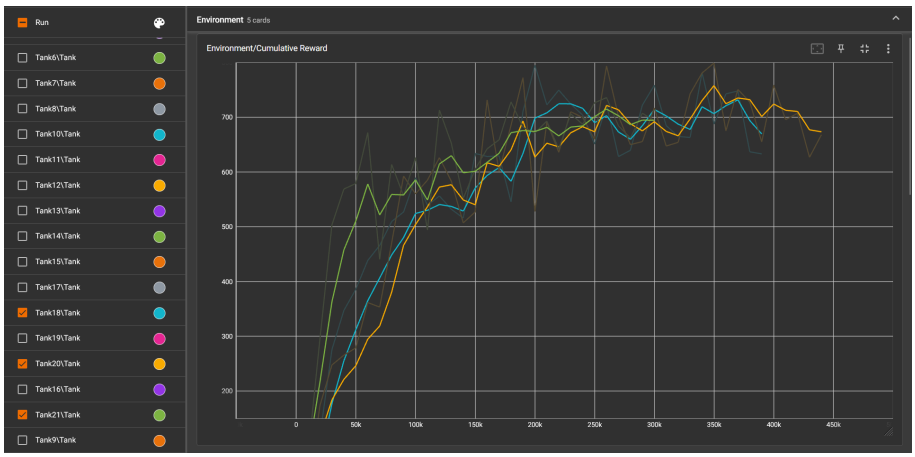
Figura 6.6: Resultados del entrenamiento para diferentes valores del tamaño del estado oculto de la red neuronal.

Finalmente, los resultados del último parámetros probado, la longitud de la secuencia, se muestran en las Figuras 6.7a y 6.7b. Como en otros casos, se ha decidido optar por el valor que reporta mejor porcentaje de imitación, 64. Con un valor de 128, el porcentaje de imitación se reduce, indicando que tener en cuenta muchos valores pasados de la distancia al jugador no es importante para predecir el comportamiento de la FSM.

Como conclusión, el tanque elegido para las siguientes pruebas es el tanque 18 de la Figura 6.1.



(a) Porcentaje de imitación para diferentes valores del tamaño de la longitud de la secuencia de la red neuronal.



(b) Valor acumulado de la recompensa para diferentes valores del tamaño de la longitud de la secuencia de la red neuronal.

Figura 6.7: Resultados del entrenamiento para diferentes valores del tamaño de la longitud de la secuencia de la red neuronal.

6.2.2. Resultados de la Prueba 2

El enfrentamiento del tanque 18 de la Figura 6.1 contra una máquina de estados finitos durante 1 hora da como resultado un porcentaje de victoria del **43.7 %**. Este porcentaje no incluye las partidas empatadas (se ha acabado el tiempo), únicamente las ganadas y las perdidas, lo que quiere decir que gana menos de la mitad de las rondas que no terminan por haber superado el tiempo límite. La conclusión es que, para este juego, una red neuronal entrenada para imitar a una FSM no mejora la eficiencia en la consecución de su objetivo.

6.2.3. Resultados de la Prueba 3

Una vez el agente utilizado en la Prueba 2 ha sido entrenado utilizando las recompensas indicadas en la Sección 5.6.3.2 durante 1, 2, 4 y 8 horas, el porcentaje de victorias contra una FSM se muestra en la Tabla 6.1.

Tabla 6.1: Porcentajes de victorias obtenidos al enfrentar al agente de la Prueba 2 entrenado por refuerzo durante 1, 2, 4 y 8 horas contra una FSM.

Horas	Porcentaje de victorias
1	67.10 %
2	77.52 %
4	79.65 %
8	85.3 %

Como se observa, el entrenamiento da sus frutos y el agente cada vez consigue mejor su objetivo de reducir los puntos de vida de la FSM a 0 a partir de las recompensas y castigos que se han establecido. No obstante, el comportamiento del agente entrenado tiende a ser quedarse quieto y disparar sin parar, un comportamiento que podría percibirse como poco inteligente o poco humano. Este comportamiento puede modificarse cambiando los valores de los castigos y las recompensas de forma que estar parado castigue al agente y que disparar, tenga un valor negativo más grande que el utilizado.

6.2.4. Resultados de la Prueba 4

Al enfrentar el agente de la Prueba 2 y el agente de la Prueba 3 cuyo entrenamiento por refuerzo se ha mantenido durante 8 horas, el resultado es el esperado, que el agente entrenado por refuerzo consigue un porcentaje de victorias mayor que el agente que recién sale de una imitación. Este porcentaje obtenido es del **82.48 %**, parecido al conseguido contra una FSM (85.3 %).

6.2.5. Resultados extra

Para aquellos lectores que quizá se hayan preguntado qué pasaría si se enfrentan dos agentes con el mismo comportamiento, el del agente que ha entrenado 8 horas después de haber llevado un proceso de imitación de una FSM, la respuesta es la más lógica. El porcentaje de victorias de uno de los agentes es del **49.60 %**. La mitad de las rondas las gana uno de los agentes y la otra mitad el otro.

CAPÍTULO 7
Conclusiones

Este proyecto se considera completado satisfactoriamente al haber cumplido la meta y los objetivos especificados en la Sección 1.2. Además, los resultados de las pruebas muestran que una red neuronal entrenada utilizando la tecnología de MLAgents, puede ser más eficiente que una FSM consiguiendo sus objetivos, lo que se esperaba. Esos objetivos pueden volver a la red neuronal más realista, sin que un programador tenga que implementar ese comportamiento complejo manualmente, que podría volverse una tarea difícil.

Mi experiencia con el proyecto ha sido enriquecedora, a la vez que complicada. Mi conocimiento de Unity, de la programación de videojuegos y de redes neuronales era pequeño, por lo que el proyecto ha sido la puerta que me ha abierto a investigar estos tres temas que me han parecido interesantes. Mis conocimientos sobre MLAgents eran nulos antes de comenzar con el proyecto, no conocía su existencia. Durante la realización del TFM, he afianzado mis conocimientos sobre FSMs y BTs aplicados a los videojuegos, he aprendido las bases de PyTorch para redes neuronales y he aprendido las bases y algunos detalles de implementación de MLAgents. No obstante, considero que el nivel de conocimiento obtenido es la superficie del gran mundo de los videojuegos.

En cuanto a la redacción de la memoria, ha sido de gran ayuda disponer de la memoria de mi TFG. Parte de la investigación de cómo utilizar Latex y de algunos de los apartados de la memoria, ya la había llevado a cabo y ha ahorrado tiempo de trabajo, que he podido dedicar al contenido y no tanto a la presentación del mismo. Y aunque ya llevo dos trabajos escritos en Latex, todavía siento que sé poco. Existen una cantidad de paquetes enorme, cada uno con su funcionalidad, sus dependencias y sus incompatibilidades que es difícil de conocer cómo funcionan.

Siento que el reto del TFM, al mismo tiempo que realizando prácticas en empresa, es una tarea ardua y he aprendido a que no hay nada imposible, aunque en algunos momentos parezca que no hay esperanza. Cada persona es distinta, y necesitamos conocernos para ser felices. Este reto grande y, en momentos, estresante me ha hecho ver que sentirme presionado no me ayuda a realizar aquello que quiero y que en los momentos difíciles en los que te sientes sin fuerza para continuar, un respiro es bienvenido y me ayuda a retomar lo que quiera que esté haciendo con energía.

Profesionalmente, el TFM me ha ayudado a aprender a organizar mejor mis tareas y tiempo, además de gestionar mejor el estrés y mejorar mi capacidad de resolver problemas, viéndolos desde diferentes perspectivas.

CAPÍTULO 8
Mejoras futuras

Durante la realización y al haber terminado el proyecto, surgen posibles mejoras que podrían ser aplicadas para mejorar la funcionalidad del asset o para que su uso sea más cómodo. Algunas propuestas de mejoras son las siguientes.

- Permitir la especificación de múltiples agentes en el mismo fichero XML de modo que los usuarios puedan elegir cómo estructurar sus especificaciones. Podrían incluir varias especificaciones de un mismo agente para conseguir diferentes niveles de dificultad, y no tener diferentes ficheros, uno por cada especificación.
- Permitir la especificación del número de neuronas en cada capa individual. Aunque en MLAgents no es posible, en otras tecnologías que puedan implementar la interfaz de GAIA puede que lo sea y da más flexibilidad para los usuarios que lo requieran.
- Crear un script que reciba como entrada una especificación XML de GAIA para una FSM o un BT y genere una plantilla de especificación para una red neuronal que pueda imitar su comportamiento.
- Automatizar el proceso de aprendizaje por imitación para que el usuario no tenga que escribir las recompensas y castigos de imitar correctamente o no a una FSM o un BT.
- Incluir una nueva tecnología de redes neuronales.
- Modificar los valores de recompensas y castigos de la red neuronal para que su comportamiento sea más real cuando aprende enfrentándose a una FSM.
- Probar el comportamiento de la FSM y de la red neuronal con humanos y evaluar la percepción que tienen en cuanto al realismo de su comportamiento.

CAPÍTULO 9
Glosario

- **NPC (Non-Player Character)**. Personaje dentro de un videojuego que no es controlado por una persona.
- **Asset**. Cualquier recurso utilizado para el desarrollo de un videojuego. Incluye modelos 3D, modelos 2D, texto, sonidos, animaciones, código, etc.
- **Script**. Fichero con instrucciones escritas en un lenguaje de programación. Se utiliza para referirse a instrucciones que proveen de funciones adicionales, se utilizan para la integración de sistemas o como comunicación entre otros lenguajes de programación.
- **XML**[1] **(eXtensible Markup Language)**. Se trata de un lenguaje de marcado general en el que el usuario define sus propias etiquetas, en vez de estar ya definidas como pasa con HTML (HyperText Markup Language).
- **Dashear**. Movimiento rápido y repentino de un personaje en un videojuego, tanto vertical como horizontal. Puede tener diferentes funciones como avanzar más deprisa, saltar más alto, embestir a enemigos, pasar un túnel repleto de pinchos y mucho más.
- **Videojuego AAA**. Se llama videojuego AAA o Triple A a un videojuego creado por una desarrolladora importante y vendido por una distribuidora también importante. Estos videojuegos cuentan con un gran presupuesto y su coste es elevado. Son videojuegos que requieren de varios años de trabajo por equipos grandes para terminarse y suelen tener altas ventas.
- **C#**. Lenguaje de programación orientado a objetos desarrollado y mantenido por Microsoft.
- **Reinforcement learning (aprendizaje por refuerzo)**. Es una técnica de aprendizaje en la que un modelo de inteligencia artificial aprende en base a las recompensas y castigos que recibe. El modelo aprende a decidirse por las acciones que le reportan recompensas y evitar aquellas con castigos asociados.
- **Librería de enlace dinámico**. Fichero de código ejecutable que se carga en memoria bajo petición. Son independientes a los programas que las utilizan, y en caso de no existir en el ordenador, el programa que la usa podría fallar. Facilitan la reutilización y la reducción del tamaño de los programas.

[1] https://developer.mozilla.org/en-US/docs/Web/XML/XML_introduction

Bibliografía

[1] J. K. Peckol, *Introduction to fuzzy logic*. wiley, 7 2021.

[2] L. Bolc and M. J. Coombs, *Expert System Applications*. Springer Berlin Heidelberg, 1988.

[3] W. Zucchini, I. L. Macdonald, and R. Langrock, *Hidden Markov models for time series: An introduction using R, second edition*. CRC Press, 1 2017.

[4] S. N. Sivanandam and S. N. Deepa, *Introduction to genetic algorithms*. Springer Berlin Heidelberg, 2008.

[5] J. C. d. R. G. D Giarratano, *Expert Systems Principles and Programming Fourth Edition*. 2005.

[6] N. Silaparasetty, *Machine Learning Concepts with Python and the Jupyter Notebook Environment: Using Tensorflow 2.0*. Springer International Publishing, 1 2020.

[7] M. Csikszentmihalyi, "Toward a psychology of optimal experience," *Flow and the Foundations of Positive Psychology: The Collected Works of Mihaly Csikszentmihalyi*, pp. 209–226, 4 2014.

[8] D. M. Bourg and G. Seemann, "Ai for game developers," p. 373, 2004.

[9] M. Colledanchise and P. Ögren, "Behavior trees in robotics and ai: An introduction," *Behavior Trees in Robotics and AI*, 8 2017.

[10] J. A. Luján, "Api de gestión de inteligencia artificial basada en las máquinas de estados finitos en c#," Master's thesis, Universitat Politècnica de València, septiembre 2014.

[11] D. Jagdale, "Finite state machine in game development," *International Journal of Advanced Research in Science, Communication and Technology*, vol. 10, no. 1, 2021.

[12] I. Smolyakov and S. Belyaev, "Design of the software architecture for starcraft video game on the basis of finite state machines," pp. 356–359, 01 2019.

[13] K. Fathoni, R. Y. Hakkun, and H. A. T. Nurhadi, "Finite state machines for building believable non-playable character in the game of khalid ibn al-walid,"

[14] A. Girault, B. Lee, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 742–760, 1999.

[15] E. Vidal, F. Thollard, C. de la Higuera, F. Casacuberta, and R. C. Carrasco, "Probabilistic finite-state machines - part i," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, pp. 1013–1025, 2005.

[16] B. Schwab, *AI Game Engine Programming*. 2008.

[17] A. Enrique, M. Alfonseca, and M. Roberto, *Teoría de autómatas y lenguajes formales*. McGraw-Hill Interamericana de España S.L.

[18] S. Rabin, *AI Game Programming Wisdom 4 (AI Game Programming Wisdom (W/CD))*, vol. 2nd. 2008.

[19] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, 1943.

[20] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychological review*, vol. 65, pp. 386–408, nov 1958.

[21] P. J. Werbos, "Backpropagation: Past and future," pp. 343–353, 1988.

[22] P. J. Werbos, "Backpropagation Through Time: What It Does and How to Do It," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[23] G. N. Yannakakis and J. Togelius, "A panorama of artificial and computational intelligence in games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 4, pp. 317–335, 2015.

[24] J. A. Royo, "Ampliación del asset game artificial intelligence para unity incorporando behavior trees," Master's thesis, Universitat Politècnica de València, abril 2021.

[25] "Iso/iec/ieee 29148:2018 - systems and software engineering — life cycle processes — requirements engineering."

[26] J. Arlow and I. Neustadt, *UML 2.0 and the Unified Process: Practical Object-Oriented Analysis and Design (2nd Edition)*. Addison-Wesley Professional, 2005.

[27] P. M. Institute, "The standard for project management and a guide to the project management body of knowledge (pmbok guide).," p. 274.

[28] N. Alon, A. K. Dewdney, and T. J. Ott, "Efficient simulation of finite automata by neural nets," *Journal of the ACM (JACM)*, vol. 38, pp. 495–514, 4 1991.

APÉNDICE A

Plantilla XML de especificación de un agente

```xml
<?xml version="1.0" encoding="utf-8" ?>

<Agent>
  <!-- Name of the agent -->
  <Name>Tank</Name>
  <!-- Technology to be used -->
  <Type>MLAgents</Type>
  <!-- Use it when training -->
  <Train/>
  <!-- Use it if you want the agent to take actions without you
     requesting them -->
  <AutomaticRequester/>

  <!-- Number of inputs to the neural network -->
  <Observations>1</Observations>

  <!-- Integer ranges the agent can output. Add at will -->
  <DiscreteActions>
    <!-- Action is in integer range [-3, 5] -->
    <DiscreteAction from="-3" to="5"></DiscreteAction>
  </DiscreteActions>

  <!-- Decimal ranges the agent can output. Add at will -->
  <ContinuousActions>
    <!-- Action is in decimal range [-5, 7] -->
    <ContinuousAction from="-5" to="7" />
  </ContinuousActions>

  <!-- Learning parameters -->
  <Environment>
    <!-- Use it to continue the learning from where it was left
       off -->
    <Resume/>
    <!-- Reinforcement learning algorithm to be used -->
    <Algorithm>PPO</Algorithm>
    <!-- Learning rate (speed of learning) -->
    <LearningRate>0.0003</LearningRate>
    <!-- Number of experiences to take into consideration each
       gradient descent step -->
    <BatchSize>256</BatchSize>
    <!-- Network structure managing the agent behavior -->
```

```
39      <Network>
40        <!-- Number of hidden layers -->
41        <HiddenLayers>2</HiddenLayers>
42        <!-- Number of neurons per layer -->
43        <NeuronsPerLayer>128</NeuronsPerLayer>
44        <!-- LSTM settings for the neural network -->
45        <Memory>
46          <!-- If set, the agent will use an LSTM. It indicates
                the size of the hidden state -->
47          <Size>256</Size>
48          <!-- How many experiences the LSTM will take into
                consideration each step. The bigger, the more the
                neural network knows -->
49          <SequenceLength>64</SequenceLength>
50        </Memory>
51      </Network>
52
53      <!-- Rewards from the environment -->
54      <ExtrinsicRewards>
55        <!-- Factor by which the reward from the environment is
              multiplied -->
56        <Strength>0.99</Strength>
57        <!-- Discount factor for future rewards. Larger means it
              does not care about future -->
58        <Gamma>0.95</Gamma>
59      </ExtrinsicRewards>
60
61      <!-- Rewards from Curiosity module that makes the agent to
           explore new alternatives -->
62      <Curiosity>
63        <!-- Factor by which the reward is multiplied -->
64        <Strength>0.80</Strength>
65        <!-- Discount factor for future rewards. Larger means it
              does not care about future -->
66        <Gamma>0.90</Gamma>
67        <!-- Learning rate of the neural network manging Curiosity
              -->
68        <LearningRate>0.0005</LearningRate>
69        <!-- Network structure managing Curiosity -->
70        <Network>
71          <!-- Number of hidden layers -->
72          <HiddenLayers>2</HiddenLayers>
73          <!-- Number of neurons per layer -->
74          <NeuronsPerLayer>256</NeuronsPerLayer>
75          <!-- LSTM settings for the neural network -->
76          <Memory>
77            <!-- If set, the agent will use an LSTM. It indicates
                  the size of the hidden state. -->
78            <Size>128</Size>
79            <!-- How many experiences the LSTM will take into
                  consideration each step. The bigger, the more the
                  neural network knows -->
80            <SequenceLength>32</SequenceLength>
81          </Memory>
82        </Network>
83      </Curiosity>
84    </Environment>
85  </Agent>
```

APÉNDICE B

Script de instalación de dependencias Python

```
@ for /f "delims=" %%i in ('python --version') do @ set
    pythonVersion=%%i
@ echo %pythonVersion% | find "Python 3.9.12" >nul
@ if errorlevel 1 (
echo Python version must be 3.9.12
set /p nul="Press any key to continue..."
exit /b 0
)

@ for /f "delims=" %%i in ('pip --version') do @ set
    pipVersion=%%i
@ echo %pipVersion% | find "pip 23.2" >nul
@ if errorlevel 1 (
echo Pip version must be 23.2
set /p nul="Press any key to continue..."
exit /b 0
)

python -m venv MLAEnvironment
call MLAEnvironment\Scripts\activate

pip install torch==2.0.1 torchvision torchaudio --index-url
    https://download.pytorch.org/whl/cu117
pip install mlagents==0.30.0
pip install mlagents-envs==0.30.0
pip install protobuf==3.20.3
pip install onnx==1.14.0

echo:
set /p nul="Installation completed successfully. Press any key
    to continue..."
```

Fragmento de código B.1: Script de instalación de dependencias Python.

APÉNDICE C

Objetivos de Desarrollo Sostenible

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

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

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**			X	
ODS 2. **Hambre cero.**			X	
ODS 3. **Salud y bienestar.**			X	
ODS 4. **Educación de calidad.**			X	
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**			X	
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**				X
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.				X
ODS 12. **Producción y consumo responsables.**			X	
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

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

Unity es un motor de videojuegos que puede ser utilizado no solo para desarrollar videojuegos sino también para montar simulaciones. En las simulaciones, existen agentes con un comportamiento programado para cumplir los objetivos que se hayan establecido. Al utilizar redes neuronales para modelar comportamientos, los agentes de las simulaciones pueden aprender a realizar mejor su trabajo, por lo que sirve de ayuda para simular y predecir el futuro en muchos campos que están incluidos en los Objetivos de Desarrollo Sostenible. Algunos ejemplos se indican a continuación.

Meta 1.2. Se podría crear un entorno en Unity que simulase a las personas que viven pobreza extrema y a los gobiernos que pueden tomar decisiones para que esta situación cambie. A través de redes neuronales simulando el comportamiento de gobiernos, sería posible determinar cuáles son las decisiones más efectivas para reducir la pobreza mundial.

Meta 2.3. También podría utilizarse GAIA para simular el proceso de gestión del cultivo para mejorar su productividad al predecir cuál será la producción a lo largo de varios años. En este caso, una red neuronal podría modelar el comportamiento de un gestor encargado de decidir en qué tierras se va a cultivar, la cantidad de cultivo, de dónde va a proceder el agua, cuándo se sembrar y a cosechar, cómo reinvertir los beneficios para obtener mejores resultados, etc.

Meta 3.6. Una simulación podría crearse para detectar causas escondidas de accidentes de tráfico y utilizar una red neuronal que tome decisiones con respecto a qué medidas de seguridad tomar para reducir el número de accidentes. El agente podría tomar decisiones sobre dónde situar rotondas, pasos de peatones y señales de tráfico, sobre cómo concienciar a la población del peligro de las carreteras para que las respeten pero que no les tengan miedo y el material a enseñar en las autoescuelas, entre otras decisiones.

Meta 4.a. La construcción de instalaciones educativas que tengan en cuenta las necesidades de cada niño puede incluir juegos cuyo nivel de dificultad se adapta de forma automática al nivel cada niño, permitiendo mantener a cada uno de ellos entretenido y divirtiéndose, aprendiendo y evitando así que los niños se aburran y no quieran estudiar. Para adaptar el nivel de dificultad a cada niño, se puede utilizar en Unity una red neuronal que modele el comportamiento del gestor de niveles, para que cada niño tenga su plan personalizado y su educación sea de mejor calidad.

Meta 6.4. Con el objetivo de mejorar el uso eficiente de los recursos hídricos, se puede crear una simulación en Unity que copie el comportamiento del uso actual de estos recursos por parte de los seres humanos. Una vez se haya hecho, se crea un agente

ETS Enginyeria Informàtica
Camí de Vera, s/n. 46022. València
T +34 963 877 210
F +34 963 877 219
etsinf@upvnet.upv.es - www.inf.upv.es

modelado por una red neuronal que toma decisiones que afectan al entorno con el objetivo de reducir el número de personas que sufren falta de agua. El agente sería capaz de aprender, en base a las decisiones que toma y a los resultados que obtiene, a tomar decisiones correctas que podrían aplicarse a la vida real.

Meta 12.8. Una forma de dar a conocer información a todo el mundo es a través de los videojuegos. En ellos se pueden contar historias de cualquier tipo, incluyendo el desarrollo sostenible y estilos de vida en armonía con la naturaleza. Para que un videojuego sea jugado por muchas personas y su mensaje se retenga en las mentes de aquellos que lo juegan, debe de ser entretenido y mantener a los jugadores en su estado de flow. Una forma de hacerlo es mediante el uso de redes neuronales para adaptar el nivel de dificultad del juego al jugador, que sentirá que forma parte del videojuego y hará suya la misión de mejorar el mundo.

El nivel en que la tecnología desarrollada afecta a cumplir los ODS es baja, debido a que es una ayuda indirecta. Sin los entornos para realizar las simulaciones y los resultados para poder tomar decisiones acertadas y efectivas, el soporte de redes neuronales en GAIA con Unity no es suficiente.

ETS Enginyeria Informàtica
Camí de Vera, s/n. 46022. València
T +34 963 877 210
F +34 963 877 219
etsinf@upvnet.upv.es - www.inf.upv.es