UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería del Software

Trabajo Fin de Grado

Inteligencia Artificial de Comportamiento de Agentes no jugadores en un RPG clásico con Unity y TensorFlow

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería del Software

Trabajo Fin de Grado

Inteligencia Artificial de Comportamiento de Agentes no jugadores en un RPG clásico con Unity y TensorFlow

Autor: Pablo Expósito Palomo

Tutor: Antonio M. Silva Luengo

ÍNDICE GENERAL DE CONTENIDOS

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

RESUMEN

Este proyecto se basa en el estudio del aprendizaje automático de los personajes no jugadores (**NPC**) de un videojuego 3D. Para ello se ha desarrollado un juego RPG que servirá como herramienta para dicho estudio.

Para dotar de inteligencia los personajes no jugadores haremos uso de algoritmos heurísticos y uso de redes neuronales basados en el conjunto de herramientas para Unity que hace uso de la librería TensorFlow.

Las técnicas que hemos usado en el desarrollo tienen un amplio campo de aplicación, no solamente en el ámbito de lo videojuegos, si no en otros campos como medicina, economía, seguridad, etc.

La librería usada está especificada para el desarrollo de la inteligencia artificial aplicada a los videojuegos.

ABSTRACT

This project is based on the study of the automatic learning of the non-player characters (**NPC**) of a 3D video game. For it, an RPG game has been developed that will serve as a tool for this study.

To provide intelligence for non-player characters we will use heuristic algorithms and use of neural networks based on the Unity toolkit that makes use of the TensorFlow library.

The techniques we have used in development have a wide field of application, not only in the field of video games, but in other fields such as medicine, economics, security, etc.

The library used is specified for the development of artificial intelligence applied to video games.

1. INTRODUCCIÓN

El uso de inteligencia artificial en distintos tipos de proyectos, hoy en día se está convirtiendo en una necesidad ya que muchos de los problemas actuales deben resolver tareas complejas. Hablamos de campos como economía, medicina, astronomía… y una larga lista de disciplinas donde podemos encontrar este tipo solución, que mediante otras técnicas sería muy costoso o imposible resolver.

En este documento hablaremos sobre el uso de inteligencia artificial, desde un enfoque en el desarrollo de videojuegos. El desarrollo de videojuegos se espera que entre 2017 y 2021 sea uno de los tres motores principales de la industria del entretenimiento, situándose por encima del cine.

Para llevar a cabo las especificaciones que veremos más adelante, haremos uso de algoritmos heurísticos y uso de redes neuronales. Sobre este último recae todo el peso de este documento. Para ello haremos uso de librerías matemáticas como TensorFlow.

Se utilizará un motor gráfico de actualidad, para la implementación de la herramienta (videojuego), en la que aplicaremos los algoritmos y donde se realizará el estudio.

Para realizar el entrenamiento de nuestros personajes no jugadores haremos uso del algoritmo PPO (Proximal Policy Optimal).

Dado que el objetivo es dotar a los personajes no jugadores con un comportamiento de varias acciones posibles, en el estudio para comprobar que el entrenamiento ha sido exitoso o no, veremos el aprendizaje para cada tarea individual. Gracias a ello podemos reforzar las conclusiones obtenidas en el entrenamiento global.

Para realizar el entrenamiento, haremos uso de un entorno especial, que se asemeja al entorno donde el personaje no jugador va a tener que trabajar.

2. OBJETIVOS

El objetivo principal del proyecto es la implementación de los comportamientos que deben tener los NPC de un juego de rol, mediante el uso de algoritmos de aprendizaje automático basados en redes neuronales modernas.

Se pretende de este modo introducir al lector los fundamentos sobre los que se basan las redes neuronales que, aunque en este caso se aplique al desarrollo de videojuegos, puedan aplicarse a otros problemas.

Otros objetivos secundarios, por una parte, es el desarrollo de un videojuego RPG (juego de rol), con los aspectos más importantes que tiene dicho desarrollo y por otra parte conocer el funcionamiento de las herramientas (TensorFlow y Unity) que se han usado en el desarrollo del proyecto.

3. MOTIVACIÓN

En el mundo de la ciencia ficción, la inteligencia artificial ha tenido un papel importante, con aplicaciones que hemos implementando, aunque todavía otras muchas son inviables. Actualmente se encuentra con un amplio crecimiento y con un peso no solo en la ciencia ficción, sino en la sociedad actual, que por tanto le da a esta disciplina una imagen mucho más llamativa.

Una de las razones por la que elegí este grado fue por la posibilidad de trabajar con el campo de la inteligencia artificial, por tanto, no podía excluirlo en el trabajo de fin de grado.

Otro interés de igual peso es el desarrollo de videojuegos, que actualmente juega un papel muy importante es nuestra sociedad, no solo a nivel de entretenimiento, sino económico.

Estos dos campos son a los que me quiero dedicar profesionalmente y dado que la Universidad de Extremadura me ha dado la oportunidad de ampliar conocimientos en estos dos campos, he decidido realizar este proyecto.

4. ANTECEDENTES / ESTADO DEL ARTE

El campo de la inteligencia artificial es un campo de investigación establecido que sigue en crecimiento y con un rápido desarrollo. Concretamente el subcampo de aprendizaje automático (también conocido como "*Machine Learning*"). Aplicado al campo de los videojuegos, existe un gran número de algoritmos y métodos, pero actualmente el que ha ofrecido mejores resultados es la neuro-evolución.

1. NEURO-EVOLUCIÓN

La neuro-evolución es una técnica de generación de redes neuronales (con sus pesos y topología) usando algoritmos genéticos. Esta técnica ha tenido éxito en tareas diversas como el control de robots, generación de música y modelado de fenómenos biológicos entre otros.

La idea principal de la neuro-evolución es entrenar la red neuronal con un algoritmo evolutivo, es decir, una clase de métodos de búsqueda estocástico basados en la población, inspirados en la evolución Darwinista.

Llegados a este punto podemos preguntarnos por qué usar neuro-evolución:

- **Rendimiento**, para algunos problemas, neuro-evolución ofrece el mayor rendimiento frente a otros métodos de aprendizaje. Encuentra soluciones usando menos intentos que otros algoritmos.
- **Amplia aplicabilidad**, puede usarse en tareas de aprendizaje supervisado, no supervisadas y de refuerzo.
- **Escalable**, a diferencia de otros tipos de aprendizaje de refuerzo, la NE permite manejar muy bien grandes espacios de acciones/estados, especialmente en la selección de acciones.
- **Diversidad**, al usar algoritmos evolutivos, podemos recurrir a métodos de preservación de la diversidad, permitiendo que la neuro-evolución obtenga diversidad en sus resultados teniendo así diferentes estrategias, controladores, modelos y contenido significativamente diferente.
- **Aprendizaje abierto**
- **Permite nuevos tipos de juego**

Aunque la neuro-evolución tiene muchas propiedades atractivas, no siempre es el mejor método ya que para problemas particulares son otros.

El problema principal de las redes neuronales evolutivas es que tienden a ser una "caja negra", es decir, para un programador le resultará muy difícil lo que hace. Esto recae en la garantía de calidad del producto, ya que se hace muy **compleja** la **depuración** del comportamiento aprendido. Otro problema relacionado con la neuro-evolución en línea es la **predicción de su comportamiento**, ya que resultará muy **complejo** predecirlo, algo que puede chocar con los principios de diseño tradicionales de los juegos comerciales.

Una vez aclarados cuando debemos usar esta técnica, podemos decir que los tres aspectos más importantes que abarca la neuro-evolución en los videojuegos son:

- Enseñar a jugar a los NPC (para evaluar los estados o las acciones).
- Generación procedural
- Predecir experiencias o preferencias de los jugadores

A continuación, una tabla con un resumen del uso de la NE en diferentes tipos de juego con los tipos de métodos usados.

The Role of Neuroevolution in Selected Games. ES = evolutionary strategy, GA = genetic algorithm, MLP = multi-layer perceptron, MO = multiobjective, TP = third-person (input not tied to a specific frame of reference, e.g. number edible ghosts) , UD = user-defined network topology, PA = performance alone

NE Role (Section III)	Game	ANN Type (Section IV)	NE Methods (Section V)	Fitness Evaluation (Section VI)	Input Representation (Section VII)
State/action evaluation	**Checkers** [32]	MLP	UD, GA	Coevolution	TP (piece type)
	Chess [32]	MLP	UD, GA	PA (positional values)	TP (piece type)
	Othello [79]	MLP	Marker-based [34]	Cooperative coevolution	TP (piece type)
	Go (7×7) [38]	CPPN (MLP)	HyperNEAT	PA (score+board size)	TP (piece type)
	Ms. Pac-Man [71]	MLP	UD, ES	PA (average score)	Path-finding
	Simulated Car Racing [74]	MLP	UD, ES	PA (waypoints visited)	Speed, pos, waypoints
Direct action selection	**Quake II** [85, 86]	MLP	UD, GA	PA (kill count)	Visual Input (14×2)
	Unreal Tournament [135]	Recurrent, LSTM	UD, GA, NSGA-II	MO (damage&accuracy)	Pie-slice, way point, etc.
	Go (7×7) [118]	MLP	NEAT	Transfer Learning	Roving Eye (3×3)
	Simulated Car Racing [124]	MLP	UD, ES	Incremental Evolution	Rangefinders, waypoints
	Keepaway Soccer [122]	MLP	NEAT	Transfer Learning	Distances
	Battle Domain [105]	MLP	NEAT, NSGA-II	MO+Incremental	Angle, straight line
	NERO [119]	MLP	NEAT	Interactive Evolution	Rangefinders, pie-slice
	Ms. Pac-Man [106]	Modular MLP	NEAT, NSGA-II	MO (pills&ghosts eaten)	Path-finding
	Simulated Car Racing [29]	MLP	UD, GA	PA (distance)	Roving Eye (5×5)
	Atari [48]	CPPN (MLP)	HyperNEAT	PA (game score)	Raw input (16×21)
	Creatures [44]	Modular MLP	GA	Interactive Evolution	TP (e.g. type of object)
Selection between strategies	**Keepaway Soccer** [142, 143]	MLP	NEAT	PA (hold time)	Angle and distance
	EvoCommander [56]	MLP	NEAT	Interactive Evolution	Pie-slice, rangefinder
Modelling opponent strategy	**Texas Hold'em Poker** [66]	MLP	NEAT	PA (%hands won)	TP (e.g. size of pot, cost of a bet, etc.)
Content generation	**GAR** [46]	CPPN (MLP)	NEAT	Interactive Evolution	Model
	Petalz [97]	CPPN (MLP)	NEAT	Interactive Evolution	Model
Modelling player experience	**Super Mario Bros** [87]	MLP, Perceptron	UD, GA	PA (player preference)	TP (e.g. gap width, number deaths, etc.)

Ilustración 1. Rol de la neuroevolución en distintos juegos.

Tomado de Neuroevolution in Games: State of the Art and Open Challenges, Sebastian Risi and Julian Togelius, https://arxiv.org/pdf/1410.7326.pdf

5. MATERIAL Y MÉTODO

Para el desarrollo de este proyecto se han utilizado diferentes tecnologías y herramientas. Hacemos uso de un motor gráfico moderno, una librería para dicho motor que ofrece algoritmos de aprendizaje automático y un juego implementado desde cero.

1. UNITY

Unity es un motor de videojuegos multiplataforma, disponible para las plataformas Windows, OS X y Linux. Puede usarse junto con programas de modelado, como Blender, 3ds Max, Adobe Photoshop, etc. Por tanto, es muy versátil en cuanto al uso de modelos procedentes de diferentes plataformas.

El **motor gráfico** que usa Unity está basado principalmente en OpenGL (Windows, Mac y Linux), Direct3D (Windows), OpenGL ES (Android y iOS) e interfaces propietarias (por ejemplo, para Wii). Así que podemos desarrollar para varias plataformas, reduciendo costes (dinero y tiempo) en producción para plataformas específicas, pudiendo portar de Android a Windows o viceversa.

Tiene un soporte integrado con NVIDIA ya que usa su **motor de física** PhysX, dando una mayor aceleración hardware cuando se hace uso de dichas tarjetas gráficas.

Para la creación de **shaders** (sombreadores) se utiliza el lenguaje ShadersLab, pudiendo escribir shaders de tres formas diferentes: Surface shaders, Vertex y Fragment shaders. Un shader puede incluir diferentes versiones junto a una especificación de reserva, permitiendo a Unity detectar la mejor versión de la tarjeta gráfica y si no fueran compatibles, usar un shader alternativo para dar una mayor compatibilidad.

El **scripting** se basa en Mono, una implementación de código abierto de .NET. Aunque soporta varios lenguajes como JavaScript (proceso de desuso) y Boo (en desuso), el lenguaje más usado es C#.

Un aspecto bastante importante en cualquier desarrollo software es el uso de un **controlador de versiones**, dado que los más comunes son usados para texto plano no resultan eficientes para la plataforma de Unity (muchos ficheros binarios), por ello, en las últimas versiones del motor han desarrollado *Unity Assets Server* que permite el control de versiones, que usa PostgreSQL como *backend*.

Unity tiene un **sistema de audio** construido con la biblioteca FMOD, pudiendo reproducir audio comprimido Ogg Vorbis, soporta la reproducción de **video** haciendo uso de Theora.

Además, cuenta con un motor de terreno y vegetación con soporte de *billboarding*, determinación de cara oculta, funciones de iluminación *lightmapping* y global, redes multijugador y una función de búsqueda de camino en mallas de navegación.

¿Por qué usamos Unity? Hagamos una comparativa de Unity con otro motor gráfico de última generación y muy conocido, como Unreal Engine.

Unreal Engine es uno de los motores más populares e importantes que hace la competencia con Unity. Unreal Engine es un motor de juegos de PC y consolas. La versión actual (Unreal Engine 4) es multiplataforma. Esta versión fuera la primera en ser gratuita. Esto hace que en cuanto a la comunidad sea más reducida que en Unity, por tanto, a la hora de buscar asistencia en línea con problemas recurrentes, Unity ofrece un amplio abanico.

A diferencia que Unity, que usa C# como lenguaje de programación, Unreal Engine utiliza lenguaje de programación C++ siendo un motor más exigente a la hora de las técnicas de programación, aunque también hace uso de *Blueprints* (visual scripting). A la hora de escribir código, resulta más fácil y rápido haciéndolo en Unity, permitiendo un prototipado rápido y dado que queremos hacer un estudio de Inteligencia Artificial en NPCs, Unity nos dará más rapidez en crear una herramienta para dicho estudio.

Hoy en día Unreal Engine está considerado como un motor más profesional, principalmente para empresas que se dedican al desarrollo de juegos triple A[1], ya que gráficamente tiene un motor bastante potente, aunque esto no quiere decir que con Unity no podamos conseguir resultados iguales.

Por tanto, como el estudio de este documento no se centra en el desarrollo de un videojuego, sino en el estudio de cómo dotar de inteligencia a los NPC el motor que vamos a usar es Unity ya que nos va a proporcionar rapidez en la implementación y

[1] *Juego triple A* es una clasificación informal para videojuegos de compañías que tienen un amplio presupuesto para su desarrollo.

solución rápida de problemas recurrentes, pudiendo centrarnos más en el objetivo de este documento.

2. CONJUNTO DE HERRAMIENTAS PARA AGENTES DE APRENDIZAJE AUTOMÁTICO EN UNITY

Mejor conocido como *ML-Agents Toolkit*, es un plugin de código abierto que permite a los juegos y simulaciones que funcionen como entornos para el entrenamiento de agentes inteligentes.

Tiene amplias funciones de entrenamiento basadas en: aprendizaje por refuerzo, aprendizaje por imitación, neuro-evolución u otros métodos de aprendizaje automático a través de una API implementada en Python. Esta librería también dispone de algoritmos de última generación para facilitar a los desarrolladores de videojuegos entrenar agentes inteligentes para juegos 2D, 3D, VR (Realidad Virtual) y AR (Realidad Aumentada).

La librería tiene bastante documentación asociada y tiene una integración directa con el motor gráfico Unity, lo que nos dota de una mayor facilidad a la hora de implementar el comportamiento de nuestros agentes.

Este conjunto de herramientas se basa en la librería de código abierto TensorFlow.

El estudio que se realiza en este trabajo se realiza con la librería de código abierto TensorFlow para aprendizaje automático. Esta librería ha sido desarrollada por Google. Se considera como sistema de aprendizaje automático de segunda generación. Una ventaja es que permite correr en múltiples CPUs y GPUs (pudiendo hacer uso de extensiones de CUDA).

Las operaciones de TensorFlow se representan con grafos de flujo con estado (*stateful dataflow graphs*). Su nombre deriva de las operaciones que las redes neuronales realizan sobre vectores multidimensionales de datos. Esto vectores se denominan tensores (*ver* ***Anexo Matemáticas Aplicadas***).

Una vez definido qué es esta herramienta, debemos empezar a hablar de su arquitectura. El conjunto de herramientas *ML-Agents* es un complemento de Unity que contiene tres componentes de alto nivel:

- **Entorno de Aprendizaje**: contiene la escena de Unity y todos los personajes del juego.

- **API de Python**: contiene todos los algoritmos de aprendizaje utilizados en el entrenamiento (fuera del entorno de Unity).
- **Comunicador Externo**: conecta el entorno de aprendizaje con la API de Python (dentro del entorno de aprendizaje).

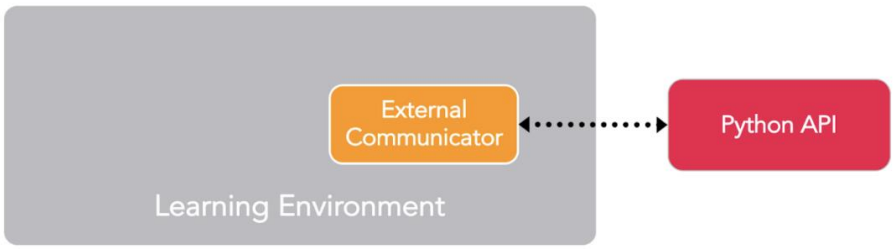

Ilustración 2. Diagrama de componentes del conjunto de herramientas.

Tomado de Unity ML-Agents Toolkit Documentation, apartado ML-Agents Toolkit Overview, https://github.com/Unity-Technologies/ml-agents/tree/master/docs

El entorno de aprendizaje consta de tres componentes que ayudan a organizar la escena de Unity:

- **Agentes**: será cualquier personaje dentro de una escena que queramos dotar con inteligencia. Cada agente está asociado con un cerebro.
- **Cerebros**: encapsula la lógica para la toma de decisiones del Agente. El cerebro determina qué acciones debe tomar el agente en cada instancia. Recibe las observaciones, recompensa al agente (negativa o positivamente) y devuelve una acción.
- **Academia**: organiza el proceso de observación y toma de decisiones. Permite especificar parámetros como la calidad de representación y velocidad de ejecución.

Cada Entorno de Aprendizaje siempre tendrá una Academia Global y un Agente para cada personaje. Los Agentes que tengan un mismo comportamiento pueden estar asociados a un mismo cerebro.

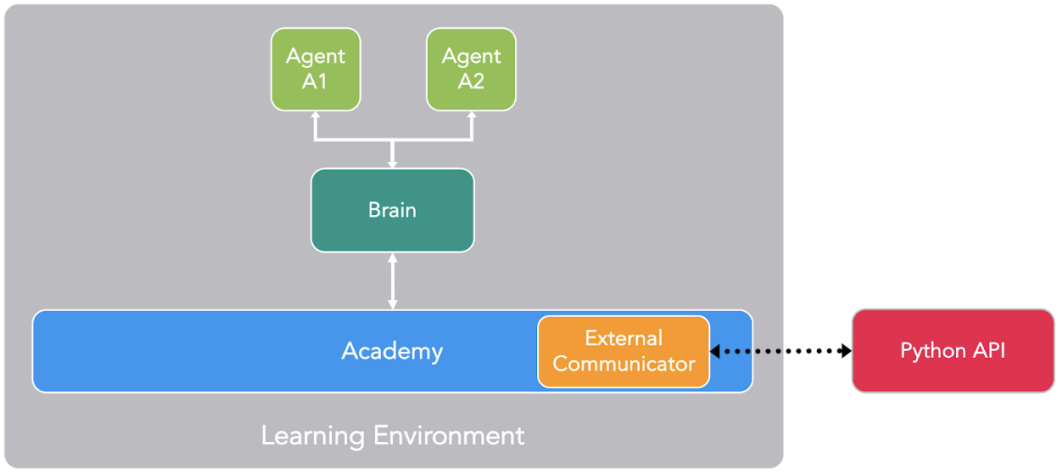

Ilustración 3. Diagrama del entorno de aprendizaje en la escena.

Tomado de Unity ML-Agents Toolkit Documentation, apartado ML-Agents Toolkit Overview, https://github.com/Unity-Technologies/ml-agents/tree/master/docs

Existen cuatro tipos de cerebro:

- **Externo**: las decisiones que toma el cerebro se basan usando la API de Python.
- **Interno**: las decisiones se toman usando un modelo integrado de *TensorFlow*.
- **Jugador**: las decisiones se toman usando la entrada real de un controlador. Un jugador humano controla el agente.
- **Heurístico**: las decisiones se toman usando un comportamiento codificado.

Dicho conjunto de herramientas nos permite una amplia gama de entrenamiento de nuestros sistemas inteligentes, a continuación, definamos los tipos de entrenamiento:

Entrenamiento Integrado

En este modo, se usan las implementaciones de los algoritmos de última generación para el entrenamiento de agentes inteligentes.

El Cerebro se establece como Externo durante el entrenamiento e Interno durante la inferencia. Durante el entrenamiento, todos los Agentes envían sus observaciones a la API de Python a través del Comunicador Externo. La API de Python procesa estas observaciones y devuelves las acciones para cada Agente. Durante el entrenamiento, las acciones son exploratorias, para que la API de Python aprenda el mejor comportamiento para cada Agente.

Una vez finalizado el entrenamiento, ese comportamiento aprendido podemos exportarlo para usarlo en la fase de inferencia. En esta fase los Agentes continúan generando observaciones, pero ya no son enviados a la API de Python, si no que alimentan a su modelo interno, para tomar la decisión más óptima en cada momento.

Entrenamiento Personalizado

Esta librería permite el uso de algoritmos personalizados para el entrenamiento de sistemas inteligentes. El uso y funcionamiento es parecido anterior, pero en este caso los algoritmos para el entrenamiento son los creados por el usuario y el Cerebro se establecería como Externo tanto en la fase de entrenamiento como en la fase de inferencia, por tanto, el comportamiento de los Agente será controlado dentro de Python.

Aprendizaje Curricular

Este modo es una extensión del Entrenamiento Integrado. El aprendizaje curricular consiste en entrenar de forma gradual, empezando por los aspectos más simples y ampliando la dificultad. Esta idea es la misma en la que los humanos normalmente aprendemos. Las habilidades y el conocimiento aprendido con anterioridad sirven como base para conocimiento más complejo.

Un ejemplo visual sería el aprendizaje matemático.

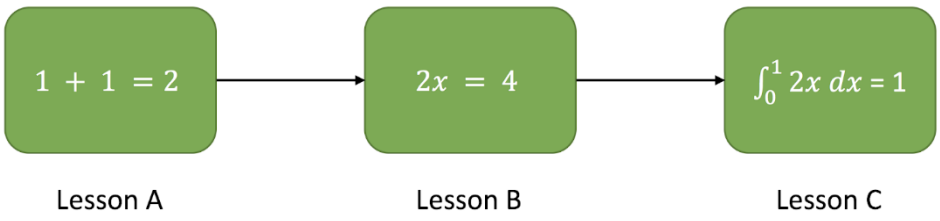

Ilustración 4. Aprendizaje matemático por lecciones.

Tomado de Unity ML-Agents Toolkit Documentation, apartado ML-Agents Toolkit Overview, https://github.com/Unity-Technologies/ml-agents/tree/master/docs

La forma en la que aprende el Agente en un aprendizaje por refuerzo es por medio de señales de recompensa. El punto de partida para entrenar a un Agente es un comportamiento aleatorio. Al inicio, el Agente estará ejecutando dicho comportamiento en bucle, y probablemente nunca, o muy raramente, logre dicha señal de recompensa en entornos complejos. Por ello debemos simplificar el entorno en las fases iniciales del entrenamiento.

Aprendizaje de Imitación

A veces resulta más intuitivo demostrar al Agente el comportamiento que queremos que obtenga, en lugar de intentar que aprenda por métodos de prueba y error. En este modo, el tipo de Cerebro se establece en Reproductor y todas las acciones realizadas con el controlador se registran y envían a la API de Python. El algoritmo de aprendizaje de imitación hace uso de estas dos observaciones y acciones del jugador humano.

Hemos hablado de los modos de entrenamiento que nos permite dicha librería, pero no sólo incluye está característica, sino que además permite diferentes escenarios para el aprendizaje:

- **Single-Agent**. Es la forma tradicional de aprendizaje de un Agente, el cual está vinculado a un solo Cerebro con su propia señal de recompensa.
- **Simultaneous Single-Agent**. Es una versión paralela del escenario anterior, que permite acelerar el proceso de aprendizaje. Múltiples Agentes independientes con señales de recompensa individuales vinculadas a un mismo Cerebro.
- **Adversarial Self-Play**. Dos agentes interactúan con señales inversa vinculadas a un único Cerebro. En juegos de dos jugadores, un juego contradictorio permite que un Agente cada vez sea más habilidoso para estar a la altura de su contrincante. Dicha estrategia fue la utilizada en entrenamientos como AlphaGO y más recientemente utilizada por OpenAI[2] para entrenar agentes del juego Dota 2.
- **Cooperative Multi-Agent**. Especial para juegos en los que los Agentes tienen información parcial y deben compartir entre ellos dicha información. Múltiples Agentes interactúan con señales compartidas vinculadas a uno o varios Cerebros. Deben trabajar juntos para lograr una tarea que no puede hacer un individuo solo.

[2] *OpenAI* es una compañía sin ánimo de lucro que pretende desarrollar y promover el uso de la Inteligencia Artificial para beneficiar a la humanidad en su conjunto sin hacer mal uso de ella.

- **Competitive Multi-Agent**. Todos los juegos de equipo deben usar este escenario. Múltiples Agentes con señales de recompensa inversa vinculadas a uno o más cerebros diferentes. Los Agentes deben competir entre sí para ganar.
- **Ecosystem**. Perfecto para simulaciones. Múltiples Agentes interactúan con señales de recompensa independientes vinculados a uno o varios Cerebros diferentes.

A parte de los escenarios de entrenamiento disponibles en esta librería, incluye características adicionales que incrementan su flexibilidad:

- **Tomas de decisiones bajo demanda**: podemos forzar que los Agentes tomen decisiones solo cuando sea necesario, en lugar de solicitar decisiones en base al entorno. Permite hacer un entrenamiento para juegos por turnos, donde los Agentes deben reaccionar a eventos o juegos donde los Agentes puedan realizar acciones de duración variable.
- **Agentes con memoria mejorada**: útil para almacenar decisiones que el Agente tomó en el pasado y que le pueden servir para la toma de decisiones. Se basa en la implementación de Long Long-Term Memory. Esta implementación se basa en la memoria a largo plazo.
- **Monitoreo de toma de decisiones del agente**: dentro del propio entorno de Unity, la librería ofrece la capacidad de mostrar aspectos del Agente, como por ejemplo la percepción de los propios Agentes de qué tan bien lo están haciendo. Útil a la hora de depurar los comportamientos de los Agentes.
- **Observaciones visuales complejas**: a diferencia de otras plataformas, donde la observación puede estar limitada a un solo vector o imagen, el kit permite que se utilicen varias cámaras para las observaciones del Agente.
- **Broadcasting**: como explicamos más arriba, un Cerebro Externo envía observaciones a la API de Python de forma predeterminada. Útil para el entrenamiento o la inferencia, por tanto, el *broadcasting* es una función que puede habilitarse para los otros tres modos, donde las observaciones y acciones del Agente también se envían a la API de Python (a pesar de que no está controlado por la API).

- Configuración e integración con Docker, AWS (Amazon Web Services) y Microsoft Azure.

3. JUEGO RPG DESARROLLADO

Juego RPG está basado en la idea de los videojuegos Diablo para ordenador y Dungeon Hunter para dispositivo móvil.

Personajes

El jugador podrá elegir entre dos personajes (guerrero o mago). El personaje empieza al nivel 1 y según vaya matando enemigos gana experiencia para subir dicho nivel, permitiendo desbloquear habilidades nuevas, aumentar sus estadísticas.

Cada uno tiene un estilo de juego diferente usando armas (hachas y espadas o magia), con habilidades únicas.

Las habilidades pueden cambiar para diferentes situaciones del juego, por ejemplo, una que sirve para hacer daño a un solo objetivo u otra para hacer daño a varios. Para poder usar las habilidades cada personaje usa un tipo de recurso especial: ira o maná.

Para darle complejidad al desarrollo del personaje existen estadísticas que benefician a cada personaje (fuerza o intelecto). Dichas estadísticas multiplican el daño que le dará el arma que usa. Una estadística que beneficia a todos es la armadura, que se sumará a la armadura base de cada personaje, permitiendo reducir el daño recibido. La velocidad de movimiento también se incluye en estadísticas general.

Por lo tanto, hablamos de daño infligido, armadura y velocidad de movimiento.

Objetos

Por dar una sensación de avance, existirán tres tipos de armas: comunes, raros y épicos. Para diferenciar los objetos se usan diferentes colores para el nombre del objeto. (verde, azul y morado).

Existen objetos para curar al personaje o recuperar ira o maná, que los enemigos dejarán caer al morir.

Inventario

Para poder almacenar los objetos el personaje tendrá un inventario limitado basado en una cuadrícula, donde cada objeto ocupa un espacio. Este aspecto es muy importante

en muchos desarrollos, por tanto, hay que prestar bastante atención en la optimización del inventario.

Misiones

Para obtener las misiones se debe interactuar con los NPC amistosos, los cuales al completarla te dará una recompensa (armas o nivel).

Las misiones se desarrollan en mapas procedurales (autogenerados). En estos mapas habrá NPC enemigos, que tendrán su propia IA.

Mapa

Habrá una zona principal diseñada manualmente donde están los NPC amistosos.

Existirán zonas autogeneradas, donde se desarrollarán las misiones y estarán los NPC enemigos. Seguirán un patrón aleatorio, se basan en salas ya definidas con sus elementos, cofres, enemigos, etc.

NPC (Non-Player Character)

Los NPC amistosos estarán en la zona principal, con los que se pueden interaccionar para obtener misiones, no tienen acciones ni se rigen por una IA. Servirán para controlar el progreso del personaje en la historia.

Los NPC enemigos estarán en las zonas secundarias, los cuales tendrán una IA que maneja su comportamiento. Al morir dejarán caer armas o pociones de recurso.

Controles

Para controlar el juego se requiere de ratón y teclado.

El movimiento se hace clicando en la posición que queremos a la que vaya el personaje. Para usar las habilidades, podemos hacer clic encima del icono de la habilidad o usar el teclado (tecla 1, tecla 2, etc.).

Para atacar a un enemigo el comportamiento es igual que al moverse, clicando en el enemigo. Si somos una clase que ataca a distancia no se mueve, si somos una clase que ataca a corta distancia se moverá hacia el enemigo.

Para interactuar con los NPC amistosos sigue la misma lógica que la anterior, sin tener en cuenta el tipo de clase que seas, siempre va hacia el NPC.

Si queremos recoger los objetos que ha soltado un enemigo bastará con pasar por encima y si tiene espacio en el inventario lo cogerá.

Inteligencia Artificial Heurística

Para que los enemigos actúen en el juego se implementará una IA de estados finitos, usando el **Patrón de diseño *Delegate*** (Delegación). Trabajaremos pues con **Estados**, **Decisiones**, **Transiciones** y **Acciones**.

Estados: la acción que está realizando.

Decisiones:

- Parar.
- Seguir al Objetivo.
- Volver a la posición inicial.
- Entrar en modo combate.
- Atacar.
- Usar beneficio.

Acciones:

- Quieto.
- Seguir al objetivo.
- Volver a la posición inicial.
- Atacar.
- Usar beneficio.

El movimiento usará la técnica de *pathfinding* usando el algoritmo A*, ya que da unos resultados óptimos a la hora de encontrar los caminos.

Para la toma de decisiones existen 3 niveles de dificultad (fácil, medio, difícil) que puede establecer el jugador al inicio de la partida. **NOTA**: a parte de la dificultad de la IA también tendrá incremento de vida y daño.

Los enemigos tendrán una serie de atributos que usarán para la toma de decisiones: distancia de visión, radio de visión, distancia de ataque, posición del jugador, posición inicial del enemigo y vida del enemigo.

Las estadísticas serán mayores o menores, por ejemplo, la distancia de visión de la dificultad difícil es mayor que la dificultad medio o la velocidad de escaneo en fácil es menor que en medio En este punto recae la mayor parte de la dificultad de la IA ya que se basa en las estadísticas usadas para manejarla.

Inteligencia Artificial Con Redes Neuronales

El comportamiento de los NPC será el mismo que con la Inteligencia Artificial Heurística. El estudio realizado en este documento trata sobre este apartado, el cual veremos con más detalle en el apartado de Implementación y Desarrollo. En este caso hablamos de cinco posibles resultados que nos puede proporcionar la red neuronal, siendo valores discretos los cuales representan una acción u otra:

- Quieto.
- Seguir al Objetivo.
- Volver a la Posiciones Inicial.
- Usar beneficio.
- Atacar al jugador.

6. IMPLEMENTACIÓN Y DESARROLLO

1. REDES NEURONALES

Las redes neuronales son un modelo de computación basado en un conjunto de unidades neuronales, basados en el comportamiento en los axones de las neuronas de los cerebros biológicos. Las unidades básicas son las neuronas, organizadas en capas.

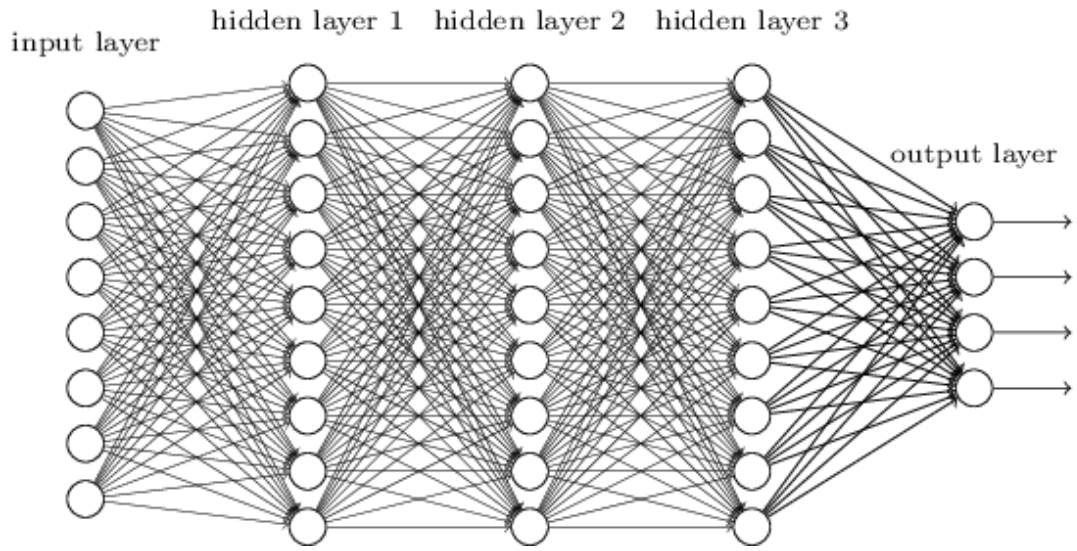

Ilustración 5. Red Neuronal: capa de entrada, capas ocultas y capa de salida.

Tomado de, Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015 http://neuralnetworksanddeeplearning.com

Una red neuronal es un modelo simplificado que simula cómo el cerebro humano procesa la información, es decir, con un número elevado de unidades de procesamiento (neuronas) interconectadas entre sí.

Estas unidades de procesamiento se organizan en capas. Existen tres partes en una red neuronal: capa de entrada, son las unidades que representan la entrada de la información; una o varias capas ocultas; y una capa de salida, en la que puede haber una o más unidades que representa la salida del procesamiento de dicha información. La red neuronal propaga los valores desde las unidades de la entrada, hasta las unidades de la siguiente capa, llegando finalmente a la unidad de salida.

La red aprende examinando los registros individuales, generando una predicción para cada registro y ajustando las ponderaciones. Este proceso se repite y la red mejora sus predicciones.

TIPOS DE NEURONA

Perceptrones

Los perceptrones fueron desarrollados en los años 50 y 60 por Frank Rosenblatt [3]. Hoy en día es más común usar otro modelo de neurona artificial. En la mayoría de los trabajos sobre redes neuronales el modelo más usado es la llamada Neurona Sigmoide. Pero antes de hablar del modelo más moderno debemos aclarar cómo funciona un perceptrón.

Un perceptrón toma muchas entradas binarias y produce una única salida binaria. Tal como podemos ver en el siguiente dibujo.

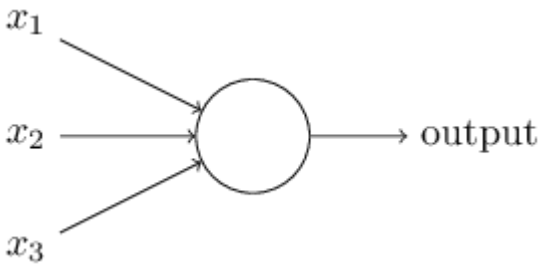

Ilustración 6. Perceptrón con varias entradas y una salida.

Tomado de, Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015 http://neuralnetworksanddeeplearning.com

La salida, 0 o 1, está determinada si el peso FORMULA es menor o mayor que un valor límite.

$$output = \begin{cases} 0\ si\ \sum_j w_j x_j\ \leq límite \\ 1\ si\ \sum_j w_j x_j\ > límite \end{cases}$$

De esta manera los perceptrones funcionan. Por tanto, una manera de pensar cómo funcionan es tomando decisiones en base a un peso.

Neurona Sigmoide

Al igual que los perceptrones este modelo tiene varias entradas, $w_1, w_2, \ldots, w_n$, pero en vez de ser binarias (0 o 1) son cualquier número real (0.689, 1.0, 0.35, etc.).

[3] Frank Rosenblatt (11 de julio de 1928-11 de julio de 1971) fue un psicólogo americano con un gran peso en la inteligencia artificial.

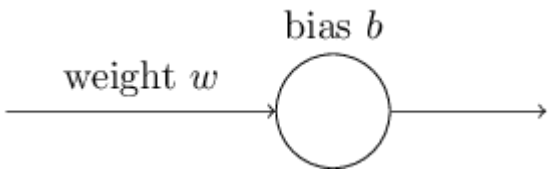

Ilustración 7. Neurona Sigmoide con pesos y "bias".

Tomado de, Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015 http://neuralnetworksanddeeplearning.com

Al igual que el perceptrón, tiene pesos por cada entrada, como ya dijimos con anterioridad, y un conjunto de "bias" (este concepto es similar al de un umbral, aunque no debemos considerarlo como tal), b . Aunque en este modelo la salida no es 0 o 1, la salida está determinada por $\sigma(w \cdot x + b)$ donde σ es la función sigmoide, w es el peso, x es la entrada y b es el "bias", y dicha función sigmoide está definida por:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}}$$

Lo importante real de esta fórmula no es la forma exacta de la función, sino su contorno (la "suavidad" del mismo); queremos que la función varíe suavemente, sin saltos brucos. Podemos representarlo de la siguiente manera:

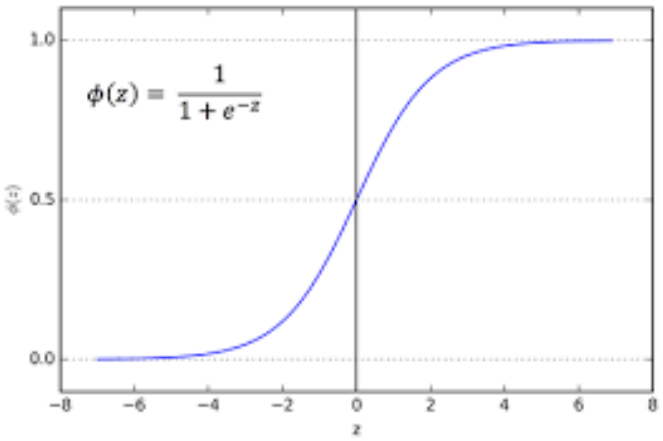

Ilustración 8. Función Sigmoide.

Tomado de https://www.quora.com/What-does-an-x-axis-represent-in-a-logistic-regression-sigmoid-function

APRENDIZAJE

Una vez definidas las redes neuronales y visto estos dos tipos de neuronal artificial, la siguiente pregunta que deberíamos responder sería ¿cómo aprenden las redes neuronales?

Cuando decimos que la red neuronal aprende, es que ajusta los pesos (w) y los "bias" (b) de cada neurona individual, como dijimos con anterioridad una forma de pensar cómo funcionan las redes neuronales es tomando una decisión en base a pesos (y "bias"), por tanto, el entrenamiento se basa en este ajuste. Para ello nos basamos en la búsqueda del mínimo de la función C (función de coste) con dichos parámetros.

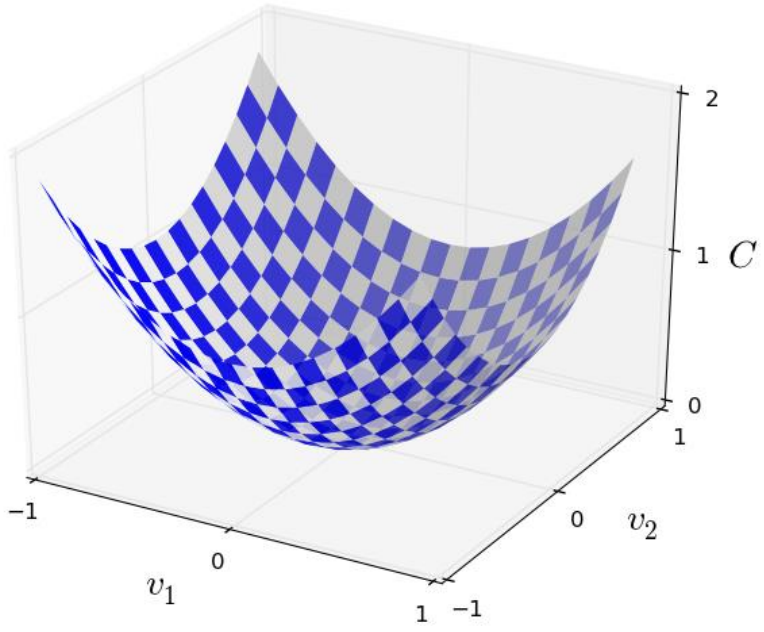

Ilustración 9. Función de coste C.

Tomado de, Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015 http://neuralnetworksanddeeplearning.com

En la ilustración anterior usamos la notación v_1 *y* v_2 en lugar de w y b, ya que de esta manera podemos generalizar a cualquier problema de optimización, sin estar en el contexto de las redes neuronales. Podemos ver que la función es muy simple, y visualmente es fácil reconocer el mínimo de la función, pero en muchos casos será una función de muchas variables y bastante compleja, y no será tan fácil encontrar el mínimo.

Existes varios algoritmos para el aprendizaje, los más usados son el *Backpropagation* (propagación hacia atrás) y *Gradient Descent* (descenso de gradiente), los cuales nos permiten encontrar el gradiente de la función de coste.

El algoritmo de *Backpropagation* tiene como objetivo calcular las derivadas parciales $\partial C/\partial w$ y $\partial C/\partial b$ de la función C con respecto cualquier peso o "bias". De este modo obtenemos el gradiente de la función de coste. Optimizando de esta manera los valores de w y b.

El algoritmo más popular y estándar de aprendizaje en las redes neuronales es el método conocido como *Gradient Descent Stochastic*, el cual permite obtener el valor mínimo de una función, basándose en una función convexa, que ajusta sus parámetros de forma iterativa. Básicamente busca los parámetros de la función (w y b) que minimizan la función de coste lo máximo posible (*ver* ***Anexo Matemáticas Aplicadas***).

2. PREPARACIÓN E INTEGRACIÓN

Preparación del Entorno

Trabajaremos con la distribución Anaconda para tener nuestro entorno de aprendizaje, donde se ejecutarán los entrenamientos que diseñemos en Unity.

Para crear un entorno haremos uso del siguiente comando:

```
conda create -n ml-agents python=3.6
```

Con la opción `-n` establecemos el nombre del entorno y establecemos que la versión de Python que vamos a usar es la 3.6 (actualmente es la soportada para el plugin de Unity).

Una vez ejecutado el comando, preguntará sobre la instalación de nuevos paquetes los cuales debemos aceptar, ya que, sin estos paquetes el entorno no funcionará correctamente.

Para iniciar el entorno basta con ejecutar el comando

```
activate ml-agents
```

para el cual debemos usar el nombre del entorno creado con anterioridad.

Una vez dentro del entorno debemos instalar la librería TensorFlow ya que será el núcleo de todo el trabajo que vayamos a realizar. Para ello tenemos dos versiones, la versión CPU y la versión GPU. Lo ideal es trabajar con la versión GPU, ya que

trabajaremos con operaciones matemáticas complejas, para las cuales son más óptimas las GPUs. Por tanto, ejecutamos el siguiente comando:

```
pip install tensorflow-gpu==1.7.1
```

Una vez instalado TensorFlow, debemos clonar el repositorio ML-Agents Toolkit (el conjunto de herramientas) de GitHub (necesitamos tener instalado Git), ejecutando el siguiente comando:

```
git clone https://github.com/Unity-Technologies/ml-agents.git
```

La carpeta ml-agents que obtendremos será esencial a la hora de realizar los entrenamientos a los modelos. A continuación, debemos cambiar la ruta a `..\ml-agents\python` y ejecutar el siguiente comando para instalar todos los paquetes necesarios para poder correr el conjunto de herramientas:

```
pip install .
```

Debemos aclarar que si vamos a usar tensorflow en la versión GPU sería interesante tener instalado CUDA (si nuestra tarjeta gráfica es compatible) ya que incluye librerías para acelerar la GPU, herramientas de optimización y el compilador C/C++ necesaria para ejecutar el conjunto de herramientas ML-Agents. La versión recomendada es la 9.0.176. Para optimizar el trabajo de redes neuronales profundas, debemos instalar la librería cuDNN, la cual debemos tener la versión 9.0.

Integración con el proyecto de Unity

El proyecto clonado anterior de GitHub, debemos abrirlo desde Unity, ya que dicho proyecto contiene lo necesario para que funcionen los entornos de aprendizaje e inferencia.

Una vez dentro, necesitamos integrar el Juego RPG implementado con el conjunto de herramientas para el aprendizaje automático. Para ello debemos copiar toda la carpeta del juego realizado (la carpeta Assets) en la carpeta Examples del proyecto Unity-Enviroment (el que acabamos de abrir) y poner la carpeta StreamingAssets (perteneciente al juego) como hijo de la carpeta Assets del proyecto Unity-Enviroment.

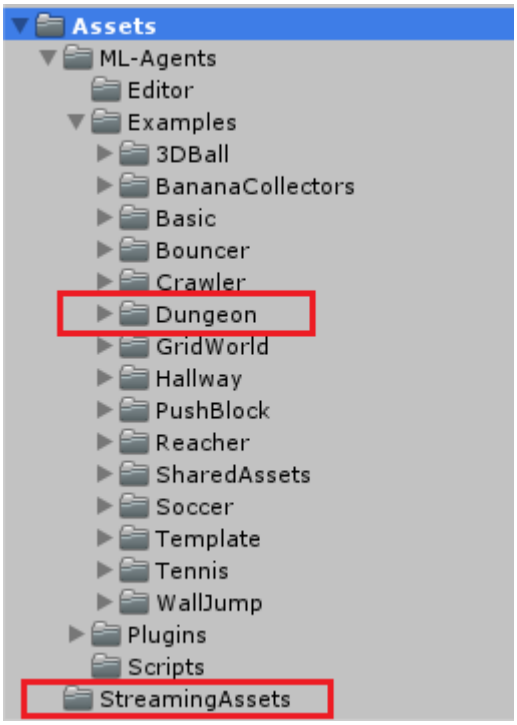

Ilustración 10. Estructura del proyecto para el aprendizaje automático de agentes.

De esta manera tendremos el proyecto configurado, sin tener que importar ningún plugin ni configurar parámetros y tenemos una biblioteca de ejemplos de uso más amplia, para futuros estudios o modificaciones.

Aun así, si queremos configurar un proyecto único debemos hacer los siguientes cambios en la configuración del proyecto.

Una vez dentro del Proyecto que vamos a configurar debemos abrir el menú ***Edit*** > ***Project Settings*** > ***Player***.

En el apartado *Resolution & Presentation* debemos comprobar que la opción *Run In Background* está marcada, ya que nos permite tener un mayor control del entrenamiento ya que lo estaremos lanzando desde la consola de comandos de Anaconda, y desde allí debemos seguir el proceso de entrenamiento. La opción *Display Resolution Dialog* debe estar desactivada, ya que esta propiedad será controlada por la Academia que debemos tener en nuestro entorno. La desactivamos para que cada vez que lancemos el modo entrenamiento no nos pregunte qué resolución de pantalla o calidad queremos ya que nuestra Academia la sobrescribirá.

Esta configuración es interesante sólo para la fase de entrenamiento, en el momento que vayamos a compilar el juego para un despliegue real, las configuraciones pueden cambiar dependiendo de las necesidades del juego que se esté desarrollando.

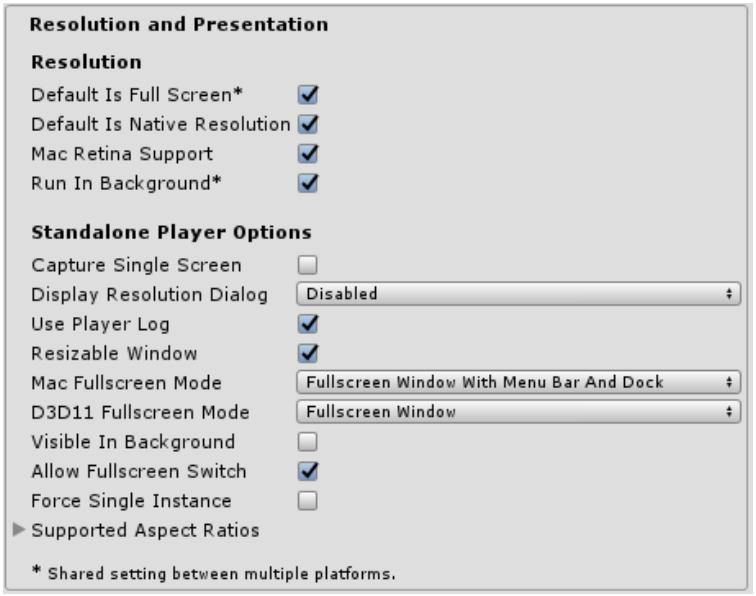

Ilustración 11. Configuración del proyecto para aprendizaje automático de agentes.

La siguiente configuración es necesaria tanto para la fase de entrenamiento como la fase de inferencia, ya que son necesarios para poder integrar el conjunto de herramientas con Unity. En el apartado *Other Setting* debemos buscar el subapartado *Configuration*. Una vez allí tenemos las siguientes propiedades, como aparecen en la captura siguiente.

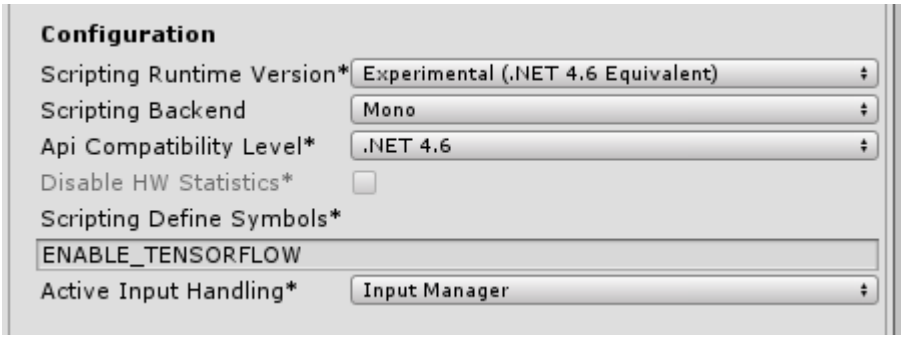

Ilustración 12. Integración de tensor flow para Unity.

Las dos propiedades importantes son: *Scripting Runtime Version* que debemos poner con la versión .NET 4.6 Equivalent o .NET 4.x Equivalent, y la propiedad *Scripting Define Symbols*, donde debemos escribir ENABLE_TENSORFLOW para poder integrar el plugin TensorFlowSharp, que permitirá integrar TensorFlow con Unity.

Una vez configurados estos parámetros debemos importar el plugin TensorFlowSharp, para ello bastará con clicar dos veces sobre el plugin descargado y comprobar que lo vamos a importa en ***Assets*** > ***ML-Agents*** > ***Plugins*** > ***Computer***.

Aclarados estos pasos ya estamos listos para pasar a la fase de entrenamiento de nuestros Agentes. En el siguiente apartado veremos toda la fase de entrenamiento.

3. ENTRENAMIENTO

Antes de explicar el procedimiento seguido para el entrenamiento de la red neuronal debemos seguir buenas prácticas de diseño para entornos.

- Inicialmente debemos empezar con entornos donde exista una versión simple del problema, para asegurarnos de que el agente pueda aprender. Podemos lograrlo de forma manual o mediante Aprendizaje Curricular (es el que usaremos).
- Al crear un entorno, resulta útil asegurarnos de que podemos completar la tarea nosotros mismo usando un Cerebro de tipo Jugador.
- Para que en entrenamiento sea lo más óptimo y rápido conviene hacer copias del agente y correr todas las simulaciones al mismo tiempo bajo el mismo cerebro, recogiendo de esta forma más información de todos los agentes.

A la hora de recompensar la red neuronal, la recompensa no debe ser mayor de 1.0, para asegurar un proceso de aprendizaje más estable, ya que nos interesa trabajar con valores entre 0 y 1.

Las recompensas positivas son más útiles para ayudar a establecer el comportamiento de un agente que las recompensas negativas.

Si existen tareas en la que hay un movimiento (por ejemplo, un NPC que se mueve hacia el personaje), típicamente se usa una pequeña recompensa (+0.1) para la velocidad de avance.

Para los agentes que requieran tareas con una optimización de tiempo, es útil añadir una penalización en cada paso (-0.05) en el que no complete la tarea.

Debemos evitar las recompensas excesivamente grandes, ya que pueden causar comportamientos indeseables en nuestros agentes, evitando acciones con dicha recompensa que puedan llevar a una recompensa positiva.

El vector de las observaciones debe incluir todas las variables relevantes para que el Agente tome la decisión de manera óptima. Por ejemplo, si queremos que el NPC siga al jugador, sería recomendable que una observación fuse la posición del jugador.

Si queremos que nuestro agente tenga memoria para que recuerde acciones, debemos aumentar el valor *Stacked Vectors* para permitir que el Agente use observaciones pasadas. Esto puede ser útil por ejemplo en un Agente que debe tomar la decisión de ir por un camino u otro, y este ya tuvo la misma experiencia en el pasado, de este modo tomará una decisión con las observaciones actuales y pasadas, eligiendo la mejor opción.

Las variables categóricas, como tipos de objeto, deben codificarse de manera única.

No sólo debemos codificar valores no numéricos, sino que todas las entradas deben estar normalizadas para estar en el rango 0 a 1 (o de -1 a 1).

Para normalizar nuestros valores debemos usar la siguiente ecuación, que tiene en cuenta los valores mínimos y máximos (que debemos conocer):

$$normalizedValue = \frac{currentValue - minValue}{maxValue - minValue}$$

La información de la posición de los GameObjects debe codificarse en coordenadas relativas siempre que sea posible. De este modo nos aseguramos de que aprenda de forma generalizada y no para posiciones particulares ya que trabajar con posiciones relativas nos permite saber dónde estamos con respecto a otra posición independiente de las coordenadas. Siempre debemos tener en cuenta que lo óptimo es que estén normalizadas las coordenadas.

Para trabajar con posiciones relativas basta con restar las posiciones de los vectores con los que estamos trabajando:

$$relativePosition = positionA - positionB$$

Arquitectura de la Escena

Los objetos que entran en juego en la escena (independientemente de los objetos del propio juego) son: la **Academia**, el **Cerebro** y un **Agente**. En la definición del plugin ya hemos hablado sobre estos aspectos, por tanto, con este apartado busco refrescar al lector sobre cómo debemos organizar la escena.

En la siguiente ilustración podemos ver que el Cerebro debe ser hijo de la Academia, y el Agente puede estar en cualquier otra jerarquía dentro de la escena del juego. Estas consideraciones son válidas tanto para la fase de entrenamiento como la fase de inferencia.

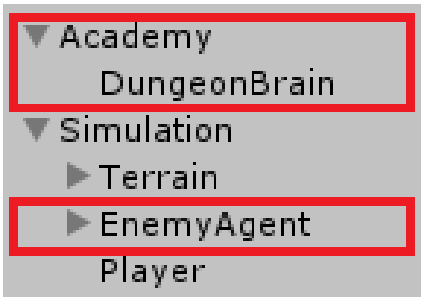

Ilustración 13. Arquitectura de la escena para el funcionamiento de los agentes.

Una vez especificados los aspectos para optimizar el entrenamiento y recordado cual es la estructura de objetos necesarios para correr el entorno veamos el procedimiento llevado a cabo para dotar a nuestros NPC de inteligencia artificial.

Procedimiento

Una vez definidos estos aspectos para la optimización del entrenamiento hablemos del entrenamiento en nuestro proyecto. Haremos uso del Aprendizaje Curricular, que ya lo definimos. Recordemos que debemos ir añadiendo dificultad al Agente en diferentes situaciones, para que cuando esté en el entorno real sepa que acciones debe realizar.

El agente debe tener un script que herede de la clase Agent, ya que la librería nos proporciona funciones como la captura de observaciones o ejecución de acciones. Al heredar de la clase Agent tenemos las siguientes propiedades para configurar.

El cerebro (***Brain***) que más adelante definiremos y hablaremos de él. Encapsula la lógica del agente, es el que tomará las decisiones que debe ejecutar el Agente.

El agente puede tener cámaras asociadas (***Agent Cameras***), que sirven como observaciones visuales. En nuestro proyecto no será necesario el uso de observaciones visuales, ya que las entradas de la red neuronal se basan en números reales.

Número máximo de pasos (***Max Step***) que el Agente, independiente del especificado en la Academia, debe realizar hasta que se resetee. En este caso podríamos usar el valor 0, para que sea la Academia quien controle dicha propiedad, aunque por ser algo redundante uso el valor 1.

La propiedad ***Reset On Done*** nos permite decidir si el Agente debe resetear cuando determine la tarea como finalizada, de esta manera podemos tener código en la función de reseteo para el entrenamiento y en la fase de inferencia bastará con marcar o desmarcarla para que no se resetee.

La propiedad ***On Deman Decisions*** es útil cuando queremos que el Agente tome decisiones cuando nosotros se lo especifiquemos. En algunos entrenamientos o juegos nos interesa que este modo este activado, por ejemplo, en juegos por turnos.

La frecuencia de decisiones (***Decision Frecuency***) sólo estará presente cuando la propiedad anterior este desmarcada, ya que determina cuantos pasos han de pasar para que tome una decisión nuestro agente. El valor elegido de 1, ya que como el número máximo de pasos es 1, la frecuencia de decisión debe ser de 1, si fuera superior el agente tardaría bastante en su aprendizaje ya que sólo tiene permitido un único intento por situación.

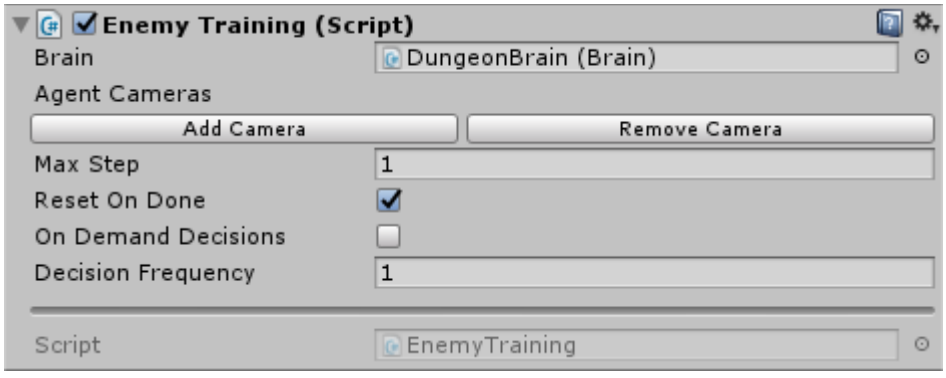

Ilustración 14. Configuración del componente de entrenamiento para el Agente.

La clase Agent tiene definidas muchas funciones que pueden resultar útiles, pero nosotros nos centraremos especialmente en tres funciones: ***CollectObservations()***, ***AgentAction()*** y ***AgentReset()***. A continuación, explicaré la implementación y decisiones tomadas en cada función.

Con la función **CollectObservations**, enviamos al Cerebro todas las observaciones que queremos que tenga por entrada la red neuronal, por tanto, podemos considerar esta función como la entrada a la red neuronal. Podemos trabajar con datos continuos o discretos; trabajar con valor continuos nos permite tomar cualquier número real mientras que los discretos nos limitan a los números naturales.

Las observaciones que vamos a tomar son de tipo continuo, ya que necesitamos conocer distancias o porcentaje de vida, que están representados como números reales.

Para abordar las especificaciones del juego en cuanto al comportamiento de los NPC debemos dar cinco observaciones como entrada a la red neuronal. A continuación, vemos las cinco observaciones.

En nuestro caso, al tener un comportamiento de seguir al personaje, podríamos pensar que debemos pasarle la posición del jugador, pero existen dos consideraciones: su comportamiento consiste en seguir al personaje siempre que este a rango, en caso contrario regresará a su posición inicial y el movimiento está determinado por un algoritmo de *pathfinding*, el A* (debemos tener en cuenta que nuestra inteligencia artificial toma sólo decisiones). Por tanto, la primera observación será la **distancia entre la posición inicial y el personaje**. Aclarar que, si queremos que el personaje no use el algoritmo A* para encontrar el mejor camino, deberíamos usar las posiciones de los objetos que entran en juego, siempre posiciones relativas.

La segunda observación que vamos a pasar será el **rango para seguir al personaje**, ya que la toma de decisiones se basa en distancias y no en posiciones. De este modo dependiendo de dicho valor obtendremos un comportamiento u otro. Resulta útil para reforzar la primera observación. La red neuronal debe conocer este valor ya que si tenemos una distancia entre el punto inicial y el personaje de 5.25 unidades y el rango para seguir al personaje es de 6.0 unidades el Agente debe seguir al jugador, pero si desconocemos el rango para seguir al personaje, resulta imposible obtener un aprendizaje para nuestro requisito.

La tercera observación que recibirá será la **vida normalizada (de 0 a 1) del enemigo**, ya que uno de los comportamientos que queremos que tengan los NPC es que al estar a punto de morir éste usará un beneficio para recibir menos daño del personaje.

La cuarta observación que recibe la red neuronal por entrada es **la distancia entre el enemigo y el personaje**, ya que para los valores oportunos de esta entrada el Agente debe atacar o no al jugador.

La quinta y última observación que recibe nuestra red neuronal es **la distancia entre el enemigo y su posición inicial**. De manera similar que la observación anterior, para ciertos valores de esta entrada debe tomar la decisión de pararse o no.

La función **AgentAction** se encarga de ejecutar las acciones que el agente va a realizar, recibe por parámetro la salida de la red neuronal, en un vector de números. Al igual que en la entrada, podemos decidir si usar acciones discretas o continuas.

En nuestro caso haremos uso de valores discretos, ya que queremos que la red neuronal nos devuelva las acciones que debe realizar en todo momento, pudiendo hacer un mapeo valor-acción, como los recogidos en la siguiente tabla.

Tabla 1. Mapeo entre valores discretos y acciones

Valor	Acción
0	Pararse
1	Seguir al Jugador
2	Volver a la posición inicial
3	Atacar al jugador
4	Usar beneficio

Este método es muy importante a la hora del entrenamiento, ya que no sólo podemos usarlo para ejecutar las acciones, sino que podemos usarlo para dar una recompensa (positiva o negativa) por acción realizada.

Para dar recompensa hacemos uso de la función ***AddReward()*** que recibe por parámetro un número real. Como ya explicamos en las buenas prácticas, las recompensas deben tomar valores entre -1.0 y 1.0, teniendo en cuenta que con las recompensas positivas nos acercaremos más al comportamiento deseado.

Para cumplir el requisito de nuestra inteligencia artificial, los condicionantes para dar recompensa positiva o negativa son los siguiente.

```
switch (action)
{
	case 0:
		enemyBehaviour.StopEnemy();
		Debug.Log("Pararse");
		if (distanceBetweenEnemyAndSpawn <= 0.25f &&
		distanceBetweenPlayerAndSpawn > enemyBehaviour.distanceToFollow)
		{
			AddReward(1.0f);
			Done();
			return;
		}
	break;
	case 1:
		enemyBehaviour.MoveToPlayer();
		Debug.Log("Seguir al Jugador");
		if (distanceBetweenPlayerAndSpawn <=
		enemyBehaviour.distanceToFollow && !(distanceBetweenPlayerAndEnemy
		<= 1.25f))
		{
			AddReward(1.0f);
			Done();
			return;
		}
	break;
	case 2:
		enemyBehaviour.MoveToSpawnPoint();
		Debug.Log("Volver a la posición inicial");
		if (distanceBetweenPlayerAndSpawn >
		enemyBehaviour.distanceToFollow && !(distanceBetweenEnemyAndSpawn
		<= 0.25f))
		{
			AddReward(1.0f);
			Done();
			return;
		}
	break;
	case 3:
		enemyBehaviour.AttackPlayer();
		Debug.Log("Atacar al Jugador");
		if (distanceBetweenPlayerAndEnemy <= 1.25f)
		{
			AddReward(1.0f);
			Done();
			return;
		}
	break;
	case 4:
		enemyBehaviour.BuffEnemy();
		Debug.Log("Beneficio");
		if (percentHealth <= 0.25f)
		{
			AddReward(1.0f);
			Done();
			return;
		}
	break;
}
AddReward(-0.35f);
Done();
```

Como podemos observar, hemos definido las cinco posibles acciones que puede realizar el NPC.

Si el NPC ejecuta la acción 0 (pararse), la distancia entre su posición y la posición inicial es menor o igual a 0.25 y el jugador no está a rango, el Agente es recompensando con 1.0, este valor es el máximo que puede recibir, por tanto, lo recompensamos con el mayor valor y damos por finalizada la tarea.

Si el NPC ejecuta la acción 1 (seguir al jugador), el jugador está a rango y la distancia entre en NPC y el jugador no es menor o igual a 0.25, el Agente será recompensando con el valor máximo de recompensa, 1.0 y finalmente la tarea finaliza.

Si el NPC ejecuta la acción 2 (volver a la posición inicial), la distancia entre el jugador y la posición inicial del NPC es mayor que la distancia establecida para seguir del NPC al jugador y no se encuentra al lado de su posición inicial, el Agente recibirá una recompensa de 1.0 y la tarea finaliza.

Si el NPC ejecuta la acción 3 (atacar al jugador) y, la distancia entre el enemigo y el jugador es menor o igual a 1.25, el Agente recibirá la máxima recompensa, 1.0 y la tarea se dará por finalizada.

Si el NPC ejecuta la acción 4 (usar beneficio) y, se cumple que el porcentaje de su vida es menor o igual que 0.25 el Agente obtendrá la recompensa positiva más alta, es decir, 1.0 y la tarea finaliza.

Hemos visto que las recompensas son de 1.0 (la más alta), esto se debe a que el agente sólo tiene una oportunidad para abordar la situación en el entrenamiento y la recompensa al hacerlo bien debe ser la máxima.

Si no se cumple ninguna de las condiciones anteriores el NPC será penalizado con una recompensa negativa, en este caso será una recompensa de -0.35. El valor tomado se basa en la experiencia, ya que obtenemos unos comportamientos más cercanos a los requisitos.

Aclarar que no se puede determinar con exactitud qué valor de recompensa debemos dar por cada acción tomada ya que no existe una fórmula matemática para determinarlo. Por ello uno de los trabajos más costosos a la hora de entrenar una red neuronal recae en cómo debemos premiarla para obtener los resultados más óptimos.

La experiencia en este campo nos permite diseñar entrenamientos con recompensas más óptimos.

La función **AgentReset** se ejecuta sólo cuando el Agente ha realizado la tarea (haciendo uso de la función ***Done()***, podemos indicarle que la ha completado). Sirve para resetear al agente (u objetos dependientes de él) al terminar la tarea. En el entrenamiento para poder realizar un entrenamiento más genérico, en esta función cambiaremos las posiciones del Agente, Jugador y Posición Inicial del Agente, de este modo podemos abarcar más situaciones proporcionando un ajuste mayor de la red neuronal.

Este método tiene cierta importancia para nosotros ya que como vamos a hacer uso de Aprendizaje Curricular, debemos cambiar los objetos en la escena en base a las especificaciones de cada lección. A continuación, veremos cómo trabajar con las lecciones.

Una vez comentado cómo funciona nuestro proyecto, debemos comprobar que nuestro objeto Cerebro tiene el componente *Brain* y tiene sus parámetros correctamente. En la siguiente foto podemos comprobarlo.

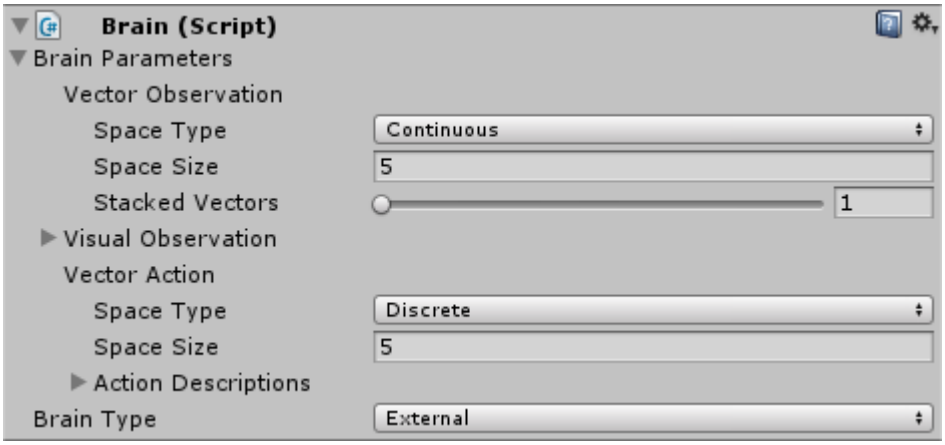

Ilustración 15. Configuración del Cerebro en la fase de entrenamiento.

Podemos ver que el Vector de Observaciones es de tipo continuo, tiene un tamaño de 5 (cinco observaciones que hemos comentado) y el vector para usar decisiones pasadas a 1 ya que no nos interesa este parámetro.

En cuanto al Vector de Observaciones Visuales no hablaremos de ello ya que nuestra red neuronal no necesita de información visual.

Podemos observar que el Vector de Acciones es de tipo discreto y tiene un tamaño de 5 (las posibles acciones que puede realizar).

Lecciones para el aprendizaje

Para que nuestra red neuronal aprenda de manera más óptima debemos ir añadiendo dificultad a la hora de realizar las tareas en el entrenamiento. Para ello hacemos uso de un fichero con el siguiente formato en JSON:

```
{
    "measure": "reward",
    "thresholds": [0.5, 0.65, 0.8, 0.85, 1],
    "min_lesson_length": 1,
    "signal_smoothing": true,
    "parameters":
    {
        "enemySpawnpoitnDistance": [1, 2, 0, 2, 1, 0],
        "playerSpawnpoitnDistance": [4, 7, 6, 5, 8, 6],
        "enemyHealthPoints": [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
    }
}
```

La variable *measure* determina qué vamos a medir durante el proceso de aprendizaje y en qué vamos a basarnos para aumentar la dificultad de la lección. Podemos basarnos en recompensas (*rewards*) o simplemente en el progreso total (*progress*). En nuestro caso queremos basarnos en las recompensas, ya que estamos entrenando a nuestra red mediante recompensas negativas o positivas. Si tomamos como medida el progreso total, se basaría en el número de pasos máximo que tiene por límite nuestra red.

La variable *thresholds* es un vector de números reales que determina los puntos, dependiendo del tipo de medida, en el que la lección debe incrementarse. En el tipo *reward* 0.5 significa que ha recibido un valor de 0.5. Mientras que en el tipo de medida *progress* 0.5 significa que va por la mitad del número de pasos máximo.

La variable *min_lesson_length* es un número natural que determina cuantas veces seguidas debe darse la condición anterior (*threshold*) para que la lección se considere superada. Aumentando este número podemos afianzar el aprendizaje de la lección, pero no siempre ya que si ponemos valores muy altos la lección pasada podría no estar reflejada en su modelo de aprendizaje.

La variable *signal_smoothing* es un valor booleano, determina si debemos tener en cuenta los valores obtenidos con anterioridad. Si es verdadero, el peso será el siguiente: 0.75 para el nuevo y 0.25 para el antiguo. De esta manera tenemos en cuenta los aprendizajes pasados para los valores actuales.

La variable *parameters* es un diccionario (clave: cadena, valor: vector de números reales) que corresponde con la Academia definida en nuestro proyecto, son los valores a los que se receta en cada lección. Dado que estos valores son por cada lección y los valores del *threshold* son entre lección, el tamaño de cada parámetro debe ser siempre uno mayor que el de los límites.

Para configurar la Academia debemos tener en cuenta las siguientes propiedades.

- ***Max Steps***. Determina el número máximo de pasos que puede realizar el Agente hasta ser reseteado. Mientras estemos entrenando he decidido que el número máximo de pasos sea 1 ya que, gracias a los parámetros usados permiten poner al agente en todas las situaciones posibles y que tome una única acción por cada situación. En la fase de inferencia usaremos el valor 0 ya que no queremos que el Agente se resetee.

La configuración tanto la de entrenamiento como la de inferencia son configuraciones correspondientes con el motor gráfico para cambiar la calidad gráfica o la velocidad del motor.

- ***Width***. Determina el ancho de la ventana en el modo entrenamiento o inferencia. Su medida es en pixeles. Para entornos que requieran visualización debemos elegir un tamaño apropiado.
- ***Height***. Determina el alto de la ventana en el modo entrenamiento o en el modo de inferencia. Su medida es en pixeles. Para entornos que requieran visualización debemos elegir un tamaño apropiado.
- ***Quality Level***. Hace referencia a la calidad de renderizado del entorno. En modo Entrenamiento no es recomendable tener mucha calidad, ya que afectaría al rendimiento y el entrenamiento sería más lento. En modo Inferencia se recomienda usar el máximo nivel de calidad. Aunque esta medida también depende del juego que se está desarrollando o en qué máquina se vaya a ejecutar.

- ***Time Scale***. Valor entre 1 y 100, nos permite modificar la velocidad de ejecución del entorno. Para el modo Entrenamiento nos interesa que se ejecute lo más rápido posible para optimizar el tiempo de entrenamiento, por esta razón tomaremos el valor más alto. En el modo Inferencia la escala de tiempo tomará el valor más pequeño ya que nos interesa que tome el tiempo normal de ejecución.

- ***Target Frame Rate***. El número de fotogramas por segundo que el motor intenta mantener en la ejecución. Para la fase de entrenamiento daremos -1 para evitar perder potencia de la máquina tratando de mantener un número de fotogramas por segundo, de esta manera daremos más potencia al entrenamiento de la red neuronal. En la fase de inferencia daremos 60 fotogramas por segundo ya que se considera el estándar para que un juego vaya fluido, en este modo queremos la máxima calidad.

- ***Reset Parameters***. Son los parámetros que van a cambiar en las lecciones, para darle más o menos dificultad al aprendizaje de la red neuronal. Debemos tener los mismos parámetros que los definidos en el documento JSON para el control de las lecciones.

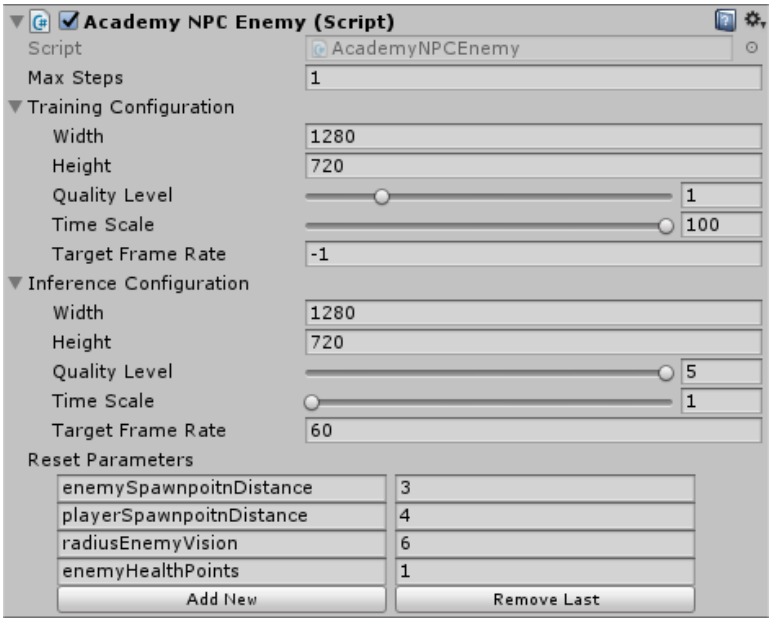

Ilustración 16. Configuración de la Academia en la fase de entrenamiento.

Ya que estamos trabajando con el algoritmo ***Proximal Policy Optimization*** (Optimización de Política Proxima o PPO), debemos hablar sobre los parámetros con los que trabaja. En el fichero ***trainer_config.yaml*** (dentro de la carpeta Python) podemos encontrar los parámetros por defecto que tendrá el algoritmos. Los cuales podemos sobrescribir para nuestro proyecto concreto. Para ello debemos escribir un apartado (como el apartado *default*) con el nombre del cerebro que estemos usando. En nuestro caso el apartado será ***DungeonBrain***.

```
default:
    trainer: ppo
    batch_size: 1024
    beta: 5.0e-3
    buffer_size: 10240
    epsilon: 0.2
    gamma: 0.99
    hidden_units: 128
    lambd: 0.95
    learning_rate: 3.0e-4
    max_steps: 5.0e4
    memory_size: 256
    normalize: false
    num_epoch: 3
    num_layers: 2
    time_horizon: 64
    sequence_length: 64
    summary_freq: 1000
    use_recurrent: false
```

A continuación, hablaremos de los parámetros para este algoritmo (PPO):

- **Gamma**. Corresponde al factor de descuento en las futuras recompensas. En las situaciones donde el agente debe estar actuando para recibir recompensas en un futuro lejano, el valor de gamma debe ser grande, en casos donde las recompensas deberían ser más inmediatas, puede ser más pequeño. El rango típico de este valor se sitúa entre 0.8 y 0.995.
- **Lambda**. Con este parámetro calculamos la Estimación de Ventaja Generalizada (GAE). Para entenderlo mejor, podemos considerarlo como cuánto depende la estimación del valor actual del Agente al calcular una estimación de valor nueva. Con un valor más bajo, confiamos más en la

estimación del valor actual, mientras que los valores altos corresponden a confiar más en las recompensas recibidas en el entorno. El rango típico de este parámetro se sitúa entre 0.9 y 0.95.

- **Buffer Size**. Determinamos con este parámetro el número de observaciones, acciones y recompensan obtenidas que se deben recopilar antes de realizar cualquier aprendizaje. Debe ser un múltiplo de Batch Size. Normalmente los valores más grandes de este parámetro corresponden a actualizaciones más estables del entrenamiento. El rango típico es entre 2048 y 409600.
- **Batch Size**. Determina el número de experiencias usadas (observaciones, acciones y recompensas) para una iteración de la actualización del método descenso de gradiente. Debe ser siempre una fracción de Buffer Size. Si estamos usando un espacio de acciones continuo este valor debe ser grande, los valores típicos se encuentran entre 512 y 5120; si estamos usando un espacio de acciones discreto este valor debería ser más pequeño, el rango de valores típico está entre 32 y 512.
- **Num Epoch**. Número de pasos a través del buffer de experiencias durante el descenso de gradiente. Disminuir este valor asegura actualizaciones más estables, pero con un aprendizaje más lento. Suele situarse entre 3 y 10.
- **Learning Rate**. Corresponde con la intensidad de cada paso en la actualización en el descenso de gradiente. Si tenemos un entrenamiento muy inestable y la recompensa no aumenta constantemente deberíamos reducir el valor. Se suele situar ente 10^{-5} y 10^{-3}.
- **Time Horizon**. Cantidad de pasos de experiencia para que el agente recopila antes de añadirlo al buffer de experiencias. Si se alcanza este límite antes del acabar un paso, se utiliza una estimación del valor para predecir la recompensa esperada. En los casos donde existan recompensas frecuentes, o si los pasos son positivamente grandes, los valores más pequeños son ideales. El rango típico se sitúa entre 32 y 2048.
- **Max Steps**. Cuantos pasos de la simulación se ejecutan durante el proceso de entrenamiento. En problemas más complejos el valor debe aumentar. El rango típico se encuentra entre 5×10^5 y 10^7.

- **Beta**. Este parámetro se corresponde con la regularización de la entropía, es decir, lo que hace que el comportamiento sea más aleatorio. Permite que los agentes exploren el espacio de acciones durante el entrenamiento. Si aumentamos el valor aseguramos que se tomen más acciones al azar. Debemos ajustarlo para que la entropía (más adelante hablaremos de ella) disminuya lentamente juntos con los aumentos de recompensa. El rango típico se encuentra entre 10^{-4} y 10^{-2}.

- **Epsilon**. Con este parámetro establecemos un umbral aceptable de divergencia entre el comportamiento antiguo y nuevo durante la actualización de la pendiente de gradiente. Si disminuimos el valor tendremos actualizaciones más estables, pero con un proceso de entrenamiento más lento. Los valores suelen estar entre 0.1 y 0.3.

- **Normalize**. Referente a las entradas del vector de observación. Determina si se aplica la normalización en las entradas. Esta normalización se basa en la media móvil y la varianza del vector de observaciones. Cuando existan problemas complejos de control continuo puede resultar útil activar la normalización, pero puede ser perjudicial con problemas simples de control discreto.

- **Number of Layers**. Número de capas ocultas en nuestra red neuronal. Para problemas simples, cuanto menos capas más rápido y más eficiente será el entrenamiento. Para problemas complejos se pueden necesitar más capas.

- **Hidden Units**. Número de unidades (neuronas) en cada capa. Al igual que en el número de capas, a mayor número de unidades, podremos resolver problemas más complejos.

Los parámetros que describo a continuación se usan sólo cuando el parámetro ***user_recurrent*** es verdadero.

- **Sequence Length**. Duración de las secuencias de experiencia pasadas a través de la red neuronal durante el entrenamiento. Debe ser lo bastante alto para capturar cualquier información que nuestro agente necesite recordar con el tiempo. Por ejemplo, si un agente necesita recordar la vida del jugador, este debe ser un valor pequeño. Pero si el agente debe recordar alguna acción que

el jugador haya realizado, entonces debe ser un valor alto. El rango de valores suele situarse entre 4 y 128.

- **Memory Size**. Corresponde al tamaño de la matriz utilizada para almacenar el estado oculto de la red neuronal recurrente. El valor debe ser múltiplo de 4, y debe ser escalado con la cantidad de información que se espera que el agente necesita recordar. El rango de valores típico está entre 64 y 512.

Los parámetros que describo a continuación se usan sólo cuando el parámetro ***user_curiosity*** verdadero.

- **Curiosity Encoding Size**. Correesponde al tamaño de la capa oculta utilizada para codificar las observaciones dentro del módulo de curiosidad intrínseca.
- **Curiosity Strength**. Corresponde a las magnitudes de la recompensa intrínseca generada por el módulo de curiosidad.

Los parámetros que hemos sobrescrito son:

```
DungeonBrain:
    gamma: 0.7
    lambd: 0.925
    buffer_size: 2048
    batch_size: 32
    num_epoch: 3
    learning_rate: 5.0e-4
    time_horizon: 32
    max_steps: 1.0e6
    beta: 1.0e-2
    epsilon: 0.1
    num_layers: 3
    hidden_units: 1024
```

Como veremos en el apartado de Resultados y Discusión, hay muchas gráficas con distintos significados. A la hora de seleccionar estos parámetros me he guiado por los consejos de los expertos que han creado esta librería, ya que cuando obtenía una función con una forma inesperada buscaba el significado de por qué e intentaba seleccionar el mejor parámetro. En el apartado de Resultados veremos su significado.

Ejecución

Una vez explicados todos los componentes, objetos, variables y parámetros del entorno, debemos ejecutarlo para proceder al entrenamiento. Para ello debemos tener activo el entorno creado en el apartado PREPARACIÓN E INTEGRACIÓN. Una vez activo el entorno debemos situarnos en la carpeta descargada **ml-agents** y allí ejecutar el siguiente comando

```
python python/learn.py unity-volume/TFG/tfg.exe --train --
    curriculum=python/curricula/tfg.json --run-id=tfg
```

El comando ejecuta el script *learn.py* que se encarga de ejecutar la API de Python, el cual requiere de más parámetros, como el ejecutable y parámetros como `--train` para establecer que vamos a realizar un entrenamiento, `--curriculum` para determinar que vamos a usar un aprendizaje curricular (donde establecemos el JSON donde están todas las lecciones) y el identificador de la sesión `--run-id` (útil para filtrar los resultados en TensorBoard y podemos comparar resultados).

Una vez se ejecutará el proyecto de Unity y en la consola de Anaconda podemos ver el progreso del entrenamiento.

4. INFERENCIA

La fase de inferencia consiste en poner al Agente en el entorno real de la simulación/juego.

Para ello el objeto que contiene el componente de la Academia debe tener la propiedad *Max Steps* igual a cero, ya que este valor significa que no hay un número máximo de pasos hasta que el agente se deba resetear.

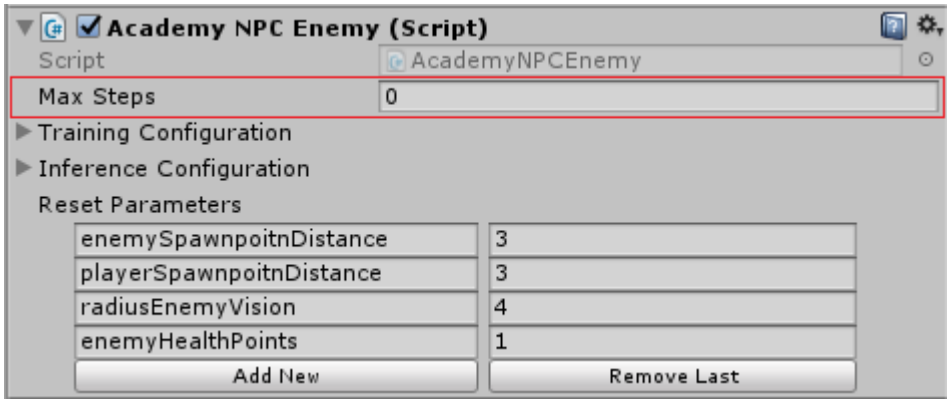

Ilustración 17. Componente academia en fase de inferencia.

En el objeto que tiene el componente Cerebro debemos realizar algunos cambios. El tipo de cerebro debemos cambiarlo a Internal (interno) y debemos seleccionar el modelo que va a usar para la toma de decisiones. El modelo que vamos a usar es el modelo entrenado en la fase de entrenamiento.

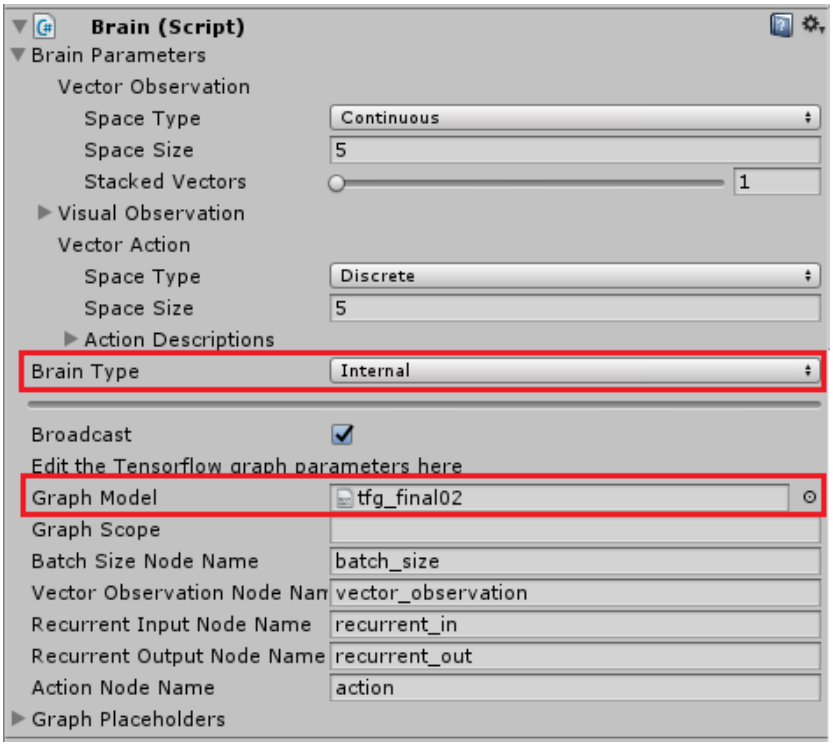

Ilustración 18. Componente cerebro en la fase de inferencia.

Otro objeto que debemos modificar es el objeto que representa al enemigo, es decir, el que tiene el componente Agente. En este caso el número máximo de pasos será 0, por el mismo motivo que en el componente Academia, y la frecuencia de decisiones valdrá 60. Este valor lo aproximamos al número de fotogramas por segundo, ya que sino el Agente a la hora de ejecutar sus acciones, ejecuta muchas acciones en un segundo (procuramos que haya 60 fotogramas por segundo) y puede llevar a resultados extraños en la fase de inferencia.

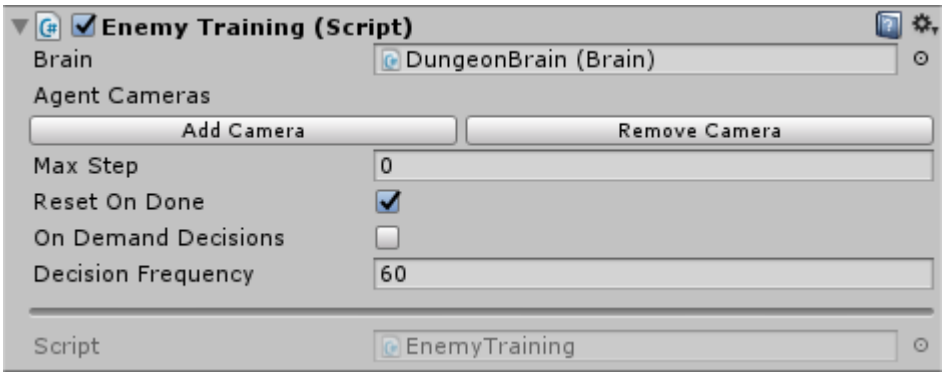

Ilustración 19. Componente Agente en la fase de inferencia.

Finalmente, el script que hemos usado para la fase de entrenamiento bastará con comentar los condicionantes dentro de la ejecución de cada acción, ya que en este caso queremos que se realice la acción pertinente en cada momento y no finalice el agente su tarea.

Una vez realizados estos cambios, la compilación del proyecto sigue los mismos pasos, pero esta vez el proyecto generado será el juego que hemos usado como herramienta.

7. RESULTADOS Y DISCUSIÓN

Durante todo el entrenamiento el Conjunto de Herramientas guarda automáticamente las estadísticas generadas durante el entrenamiento. Gracias a la librería de TensorFlow podemos usar su herramienta TensorBoard, que permite ver los resultados guardados de manera visual.

Abrimos una consola de comandos donde tengamos el plugin ML-Agents, y desde allí ejecutamos el siguiente comando:

```
tensorboard --logdir=summaries
```

de esta manera desplegamos una aplicación en la dirección **localhost:6006** o **<nombre del equipo>:6006** cuando ejecutamos el comando anterior en la consola especifica donde se despliega, que nos permite ver los resultados de manera gráfica.

Resultados del entrenamiento

Para un mejor entendimiento de resultado final, veremos el entrenamiento individual por comportamientos.

Importante destacar que en las gráficas veremos una de color más fuerte y debajo otra gráfica en tono gris. Esto se debe a que, para ver mejor la gráfica, se aplica un suavizado.

El primer comportamiento es el de **quedarse quieto** cuando el jugador está fuera de rango y el enemigo se encuentra en la posición inicial.

Empezamos comentando la entropía de la red neuronal, es decir, cuán de aleatorias son las acciones realizadas por nuestro agente. Podemos observar que para esta acción individual la gráfica tiende a cero, es decir, que cada vez que tiene que tomar la decisión de pararse la va a realizar sin ser aleatoria.

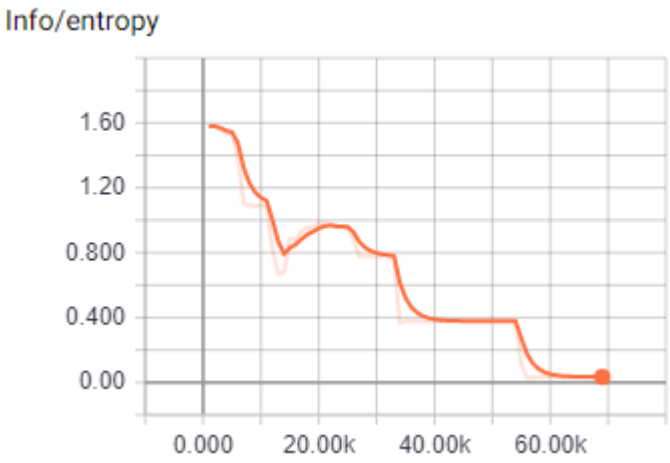

Ilustración 20. Entropia en la acción quedarse quieto.

Para hablar del aprendizaje realizado (aprendizaje según lo establecido por nosotros, hay que tener cuidado al interpretar esta gráfica, ya que podemos haber diseñado mal el entrenamiento), debemos fijarnos en la tasa de aprendizaje. Un aprendizaje exitoso debe decrementar en el tiempo, en nuestro aprendizaje podemos observar que es correcto. Aunque como ya he dicho con anterioridad, debemos asegurarnos de que las especificaciones del comportamiento que esperamos deben ser correctas.

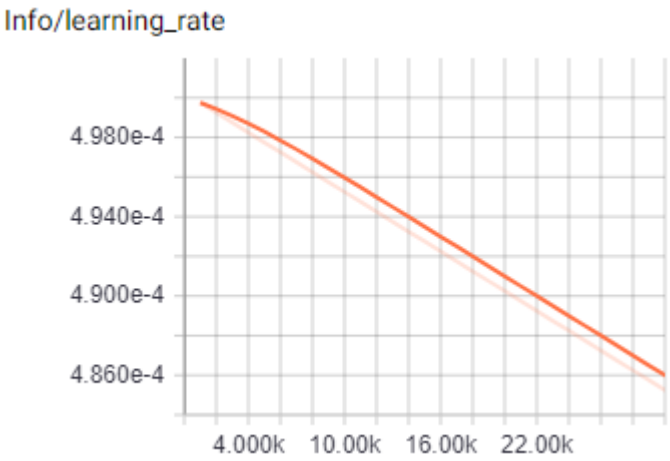

Ilustración 21. Tasa de aprendizaje en la acción quedarse quieto.

El siguiente comentario se centra en la pérdida de política (perdida de comportamiento). Representa cuánto el agente está cambiando de comportamiento. Para un entrenamiento óptimo la función debe decrecer con respecto a su valor inicial. En nuestro caso podemos ver que es menor el valor final que el inicial, aunque como veremos a continuación no es uno de los mejores resultados obtenidos, pero es válido.

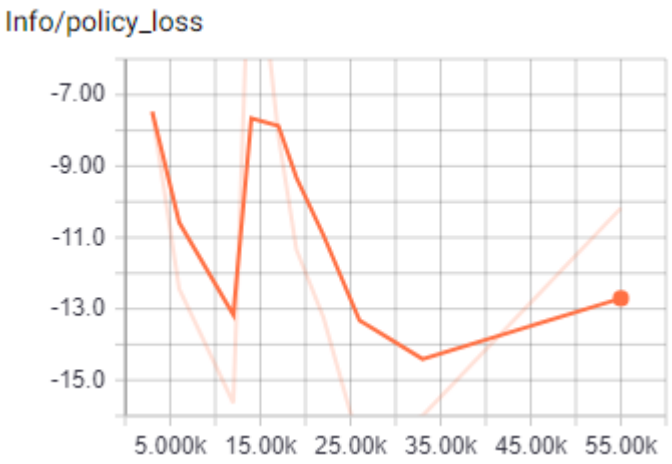

Ilustración 22. Pérdida de política en la acción quedarse quieto.

La estimación del valor medio para todos los estados visitados por nuestro agente es una buena medida para comprobar que el entrenamiento ha sido exitoso. En nuestro entrenamiento vemos que la función crece a medida que se ejecutan los pasos, esto es señal de que el entrenamiento ha ido correctamente ya que para entrenamiento exitosos la función debe crecer. Esta función para analizarla mejor debe ser comparada con la función pérdida de valor que veremos a continuación.

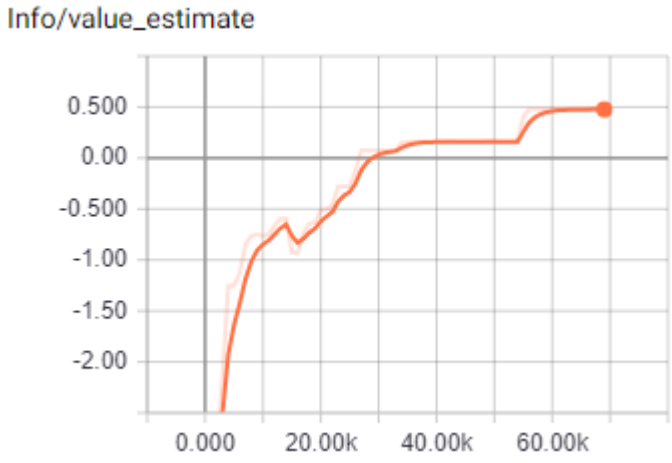

Ilustración 23. Valor estimado en la acción quedarse quieto.

Para reforzar el análisis anterior debemos tener en cuenta la función de pérdida de valor. Este valor debe incrementar durante el aprendizaje y una vez que los valores de las recompensas se estabilizan, la función debe decrementar. En nuestro caso la función decrementa desde un principio ya que no ha sido capaz de predecir el valor en cada estado.

Ilustración 24. Pérdida de valor en la acción quedarse quieto.

El siguiente comportamiento es **seguir al jugador** siempre que éste se encuentre dentro del rango para seguir del enemigo y que no estén al lado (ya que si no debería atacar).

Hablemos primero de la entropía de nuestra red neuronal. Podemos observar que en los primeros pasos ha caído la toma de decisiones aleatorias, disminuyendo progresivamente (tendiendo a cero) hasta los últimos pasos. Podemos estar seguro de que las acciones de seguir al jugador realizadas por nuestro agente serán tomadas con apenas aleatoriedad, dándonos a entender que el aprendizaje de este comportamiento es bastante bueno.

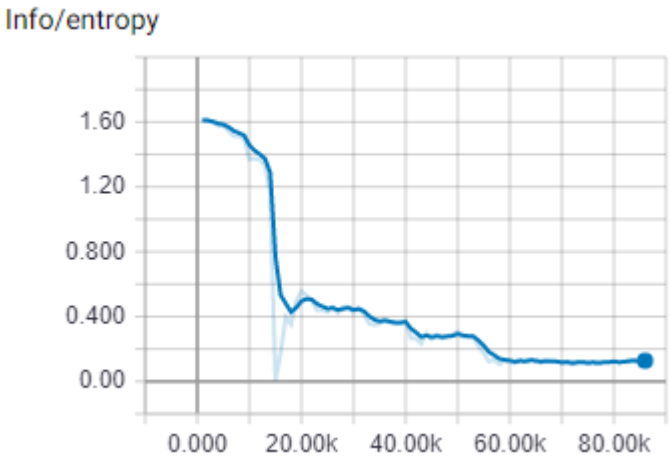

Ilustración 25. Entropía en la acción seguir al jugador.

Como hemos hablado en el anterior comportamiento, un buen índice de si el entrenamiento ha ido correcto podemos fijarnos en la tasa de aprendizaje. Esta función debe disminuir a medida que transcurre el entrenamiento. Recordar que es un índice con el que tenemos que tener cuidado, ya que un mal diseño del entrenamiento puede

tener una tasa de aprendizaje que disminuya. En nuestro entrenamiento podemos ver que disminuye, reforzando la idea de que el comportamiento que estamos entrenando lo aprende.

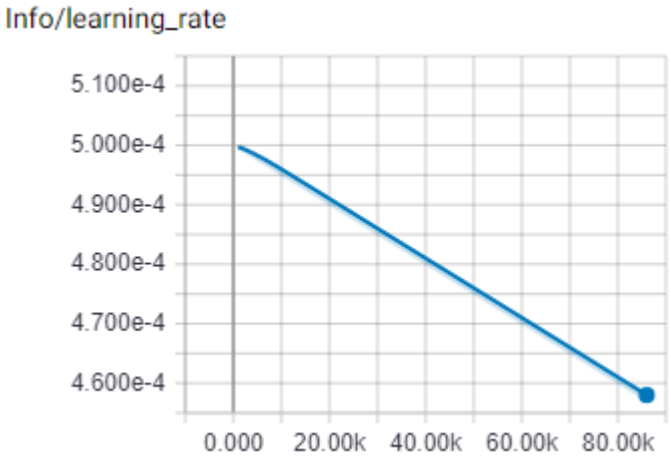

Ilustración 26. Tasa de aprendizaje en la acción seguir jugador.

Con respecto a la pérdida de política (perdida de comportamiento que tiene nuestro agente) inicialmente crece ya que está explorando todas las posibles soluciones para resolver el problema y no se ajusta al comportamiento inicial que queremos obtener, pero finalmente podemos observar que frente al valor inicial ha disminuido, siendo índice de que el entrenamiento ha sido exitoso.

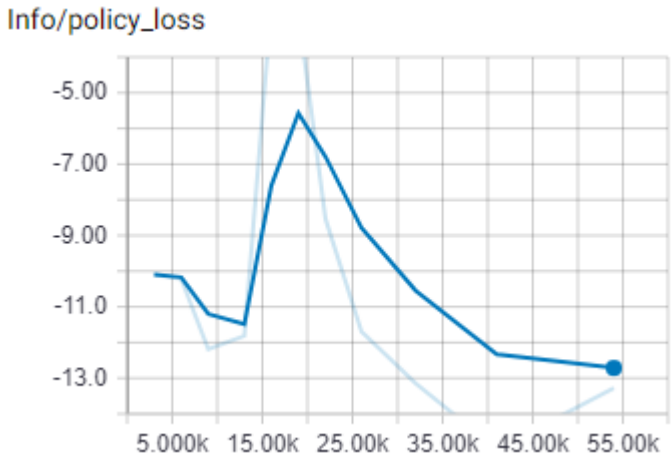

Ilustración 27. Pérdida de política en la acción seguir jugador.

Le estimación del valor medio en nuestro agente inicialmente decrece, esto no es buen signo de aprendizaje, ya que esta función debe crecer a medida que se ejecuten los pasos de aprendizaje; aunque finalmente podemos ver como la función aumenta y se estabiliza. Aunque la función no es la deseada, es bastante buena y válida como para decir que el entrenamiento ha sido exitoso.

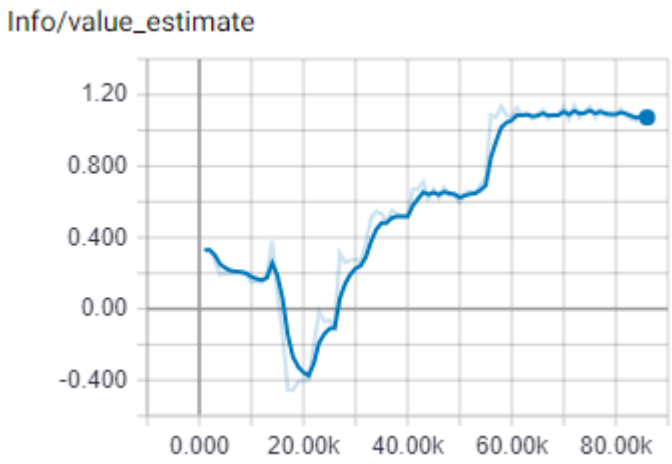

Ilustración 28. Valor estimado en la acción seguir jugador.

La función anterior debe ser analizada junto con la función de pérdida de valor, esta función debe incrementar mientras el agente aprende y crecer cuando los valores de recompensa se estabilicen. Por tanto, podemos decir que es nuestro caso, ya que con los pasos ejecutados podemos ver la fase en la que está aprendiendo y finalmente se estabilizan las recompensas.

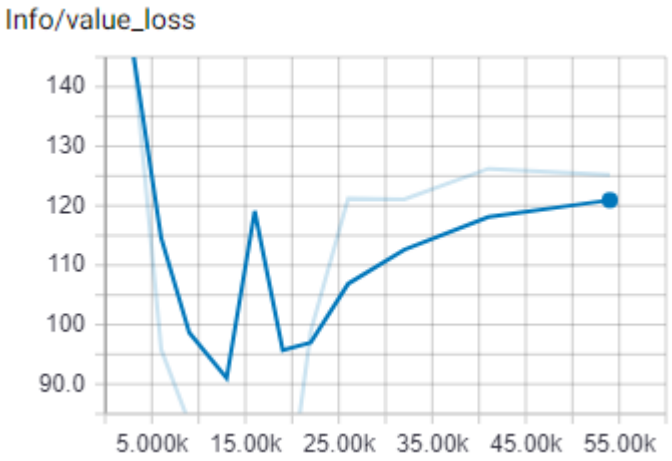

Ilustración 29. Pérdida de valor en la acción seguir al jugador.

El tercer comportamiento que analizamos será el de **volver a la posición inicial** cuando el enemigo este fuera de rango y no esté el enemigo en su posición inicial.

Observando la entropía de nuestro agente, podemos ver que a partir de los 14.000 pasos ha empezado a crecer la función, pero durante muy poco tiempo, finalmente hasta concluir el entrenamiento donde acaba con una entropía rozando el 0%. Esto es signo de que nuestro agente cuando realiza acciones las hace con la información que tiene en ese momento y no la toma de forma aleatoria. Para continuar comentado este comportamiento debemos ver las funciones restantes.

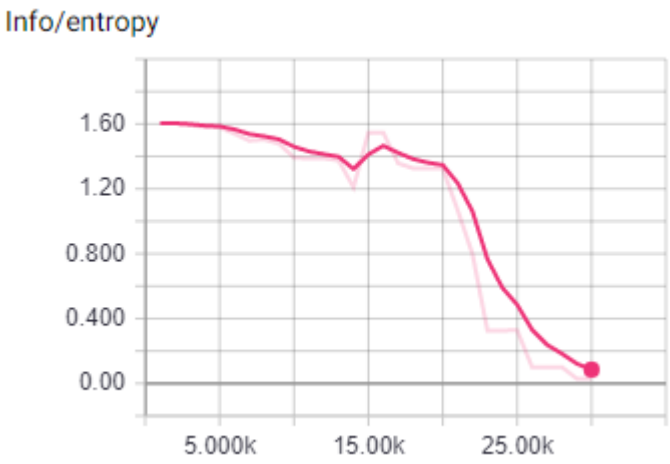

Ilustración 30. Entropía en la acción volver a la posición inicial.

Como ya hemos hablado en los otros comportamientos, una función para comprobar cómo ha sido el aprendizaje es la tasa de aprendizaje. Recordar que se basa en el aprendizaje que hemos programado, siendo esta información inútil si hemos diseñado mal el entrenamiento. En este comportamiento podemos observar que ha ido correcto, ya que dicha función disminuye con los pasos realizados por nuestro agente en el proceso de aprendizaje.

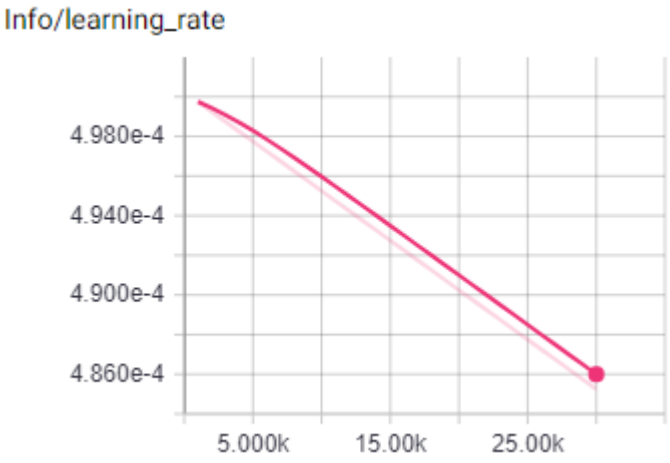

Ilustración 31. Tasa de aprendizaje en la acción volver a la posición inicial.

Con respecto a la pérdida de política (el comportamiento esperado) es bastante buena, ya que la tendencia principal de la función es decrecer. Entre los 13.000 y 17.000 la función crece, esto es signo de que nuestro agente sigue explorando otras posibilidades, pero finalmente decrece. El valor inicial es mayor que el valor final, por tanto, podemos afirmar que el entrenamiento ha sido correcto y bastante bueno.

Ilustración 32. Pérdida de política en la acción volver a la posición inicial.

Con respecto a la estimación del valor medio, podemos observar que la función crece en la mayoría de los puntos, exceptuando una pequeña bajada que no es significativa, ya que finalmente la función crece con una pendiente mayor. Esta función determina el valor estimado para todas las acciones realizadas por el agente.

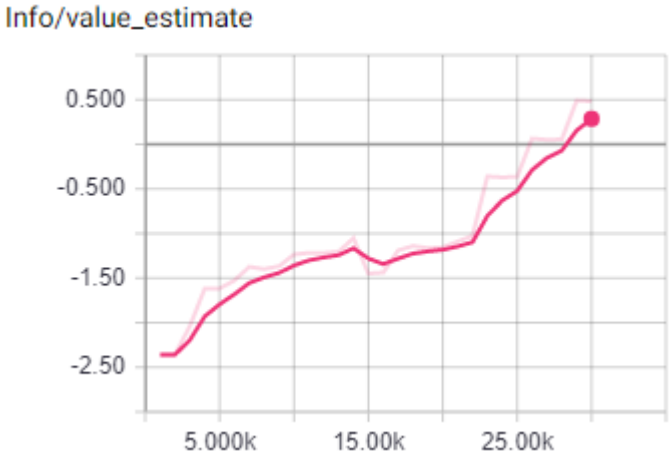

Ilustración 33. Valor estimado en la acción volver a la posición inicial.

Para dar un mayor peso a la conclusión de la función anterior debemos tener en cuenta la pérdida de valor. Esta función debe crecer al principio del entrenamiento, pero finalmente debe decrementar. Aunque para este comportamiento la función sólo decrementa, el resultado obtenido es bastante bueno, ya que una vez que el valor de las recompensas se estabilice debe decrementar. Dado que ha tenido un aprendizaje muy rápido intuimos que la función es decreciente ya que inicialmente los valores se han ajustado y estabilizado bastante rápido (aprendizaje muy rápido).

Ilustración 34. Pérdida de valor en la acción volver a la posición inicial.

El penúltimo comportamiento que analizar es el de **atacar al jugador** siempre que esté al lado.

Hablando de la entropía de nuestro agente, podemos observar que no ha habido ninguna subida durante el proceso de aprendizaje, y que la tendencia es que decrezca; por tanto, observamos que el entrenamiento ha sido correcto, ya que a medida que el tiempo ha ido pasando, las acciones tomadas eran cada vez menos aleatorias. Esta función en un entrenamiento exitoso debe disminuir, por tanto, podemos afirmar que el entrenamiento es correcto.

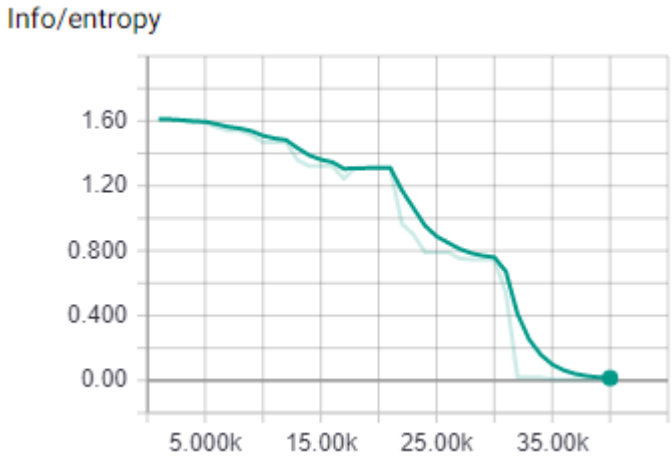

Ilustración 35. Entropía en la acción atacar al jugador.

La tasa de aprendizaje de nuestro agente en este comportamiento es muy buena, ya que como podemos observar en la función, ésta decrece. Como ya hemos hablado en los demás comportamientos, debemos comprobar a nivel de programación que el entrenamiento está bien diseñado, ya que, de lo contrario, esta información es inútil.

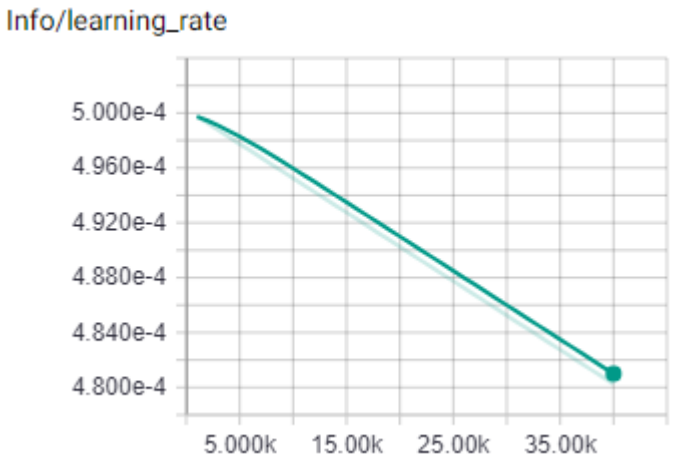

Ilustración 36. Tasa de aprendizaje en la acción atacar al jugador.

Con respecto a la pérdida de política inicialmente la función tiene un buen comportamiento, ya que decrece, aunque al final su tendencia es de crecimiento. Esto es una anomalía, ya que la pérdida de comportamiento debe incrementar cuando está explorando todas las posibles soluciones y finalmente decrecer. Aunque dicha función no tenga la tendencia esperada debemos comprobar las demás funciones para ver si el entrenamiento ha sido correcto.

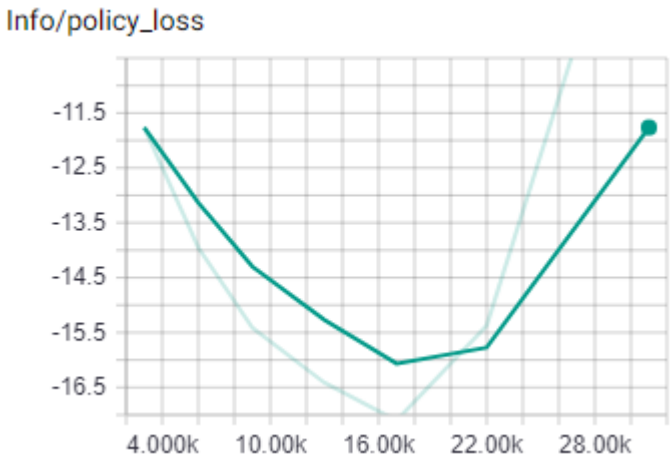

Ilustración 37. Pérdida de política en la acción atacar al jugador.

Con la función del valor estimado, podemos observar si predice el valor a obtener en cada estado. La función debe crecer con el tiempo y en nuestro caso, el comportamiento de la función es creciente. Podemos afirmar que el entrenamiento es válido, aunque para reafirmarlo debemos comprobar la función de pérdida de valor.

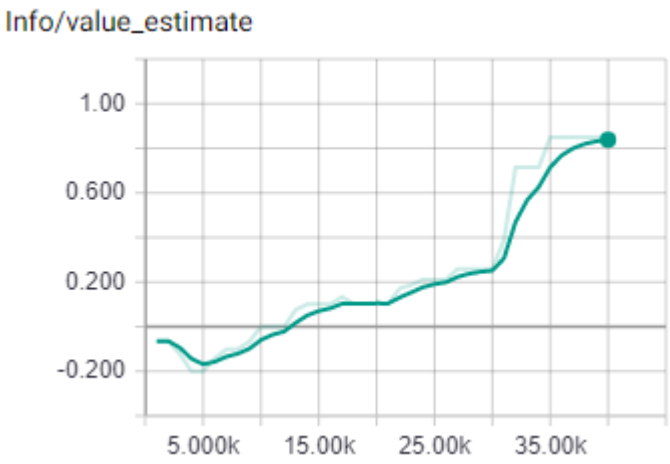

Ilustración 38. Valor estimado en la acción atacar al jugador.

La pérdida de valor debe crecer al inicio del entrenamiento y finalmente, cuando los valores de las recompensas obtenidos se estabilicen debe decrecer. En nuestro ejemplo la función solo tiene una tendencia de decrecimiento ya que el aprendizaje ha sido bastante rápido. Los valores de las recompensas obtenidas han sido bastante estables desde el principio por tanto obtenemos este comportamiento en la función.

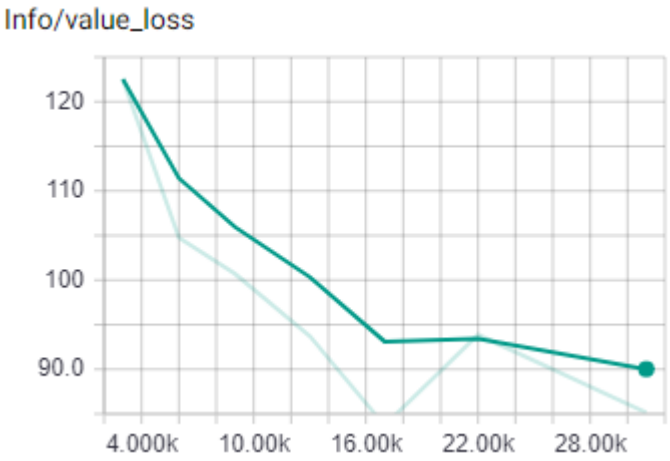

Ilustración 39. Pérdida de valor en la acción atacar al jugador.

El último comportamiento que vamos a analizar es el de **usar beneficio** para aumentar su armadura. Este se ejecutará siempre que el porcentaje de vida del enemigo sea inferior a 25%.

La entropía de nuestro agente en este comportamiento va decreciendo. Esto es bueno ya que indica que la toma de decisiones a medida que va aprendiendo es menos aleatoria que al principio. Para un entrenamiento exitoso dicha función debe decrecer.

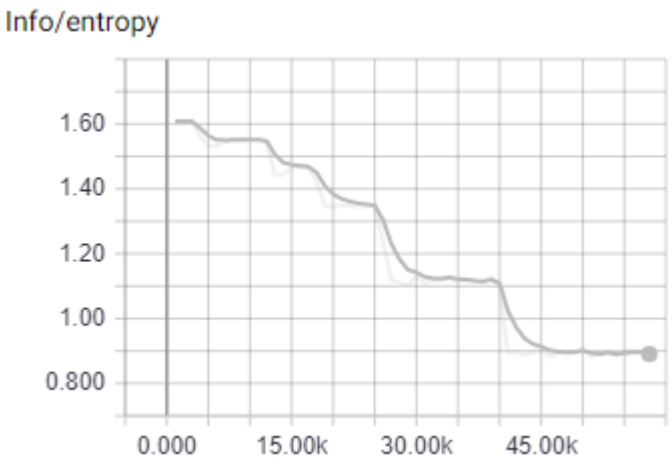

Ilustración 40. Entropía en la acción usar beneficio.

Con respecto a la tasa de aprendizaje de nuestro agente, debemos decir que es bastante buena, ya que dicha función con el tiempo debe decrecer y este es el comportamiento que tiene. Aclarar que dicha función carece de sentido si tenemos un mal diseño del entorno de aprendizaje.

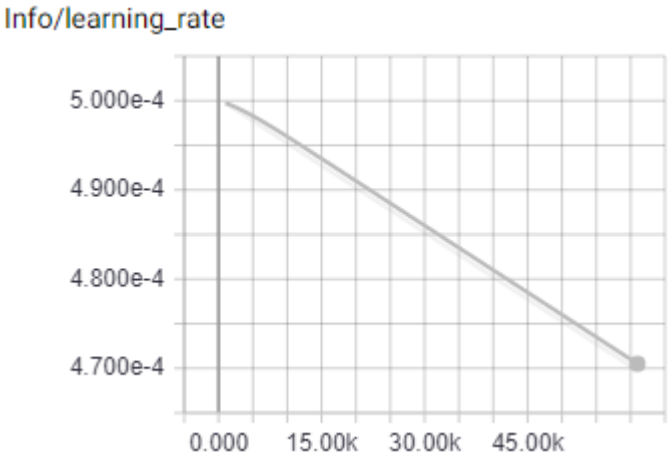

Ilustración 41. Tasa de aprendizaje en la acción usar beneficio.

La función que mide la pérdida de política no es la esperada. Esta función inicialmente debe crecer, como podemos ver en la función obtenida, pero finalmente tiene que tener un comportamiento de decrecimiento, el cual la función obtenida no tiene. Aunque este resultado no sea válido debemos comprobar las demás funciones.

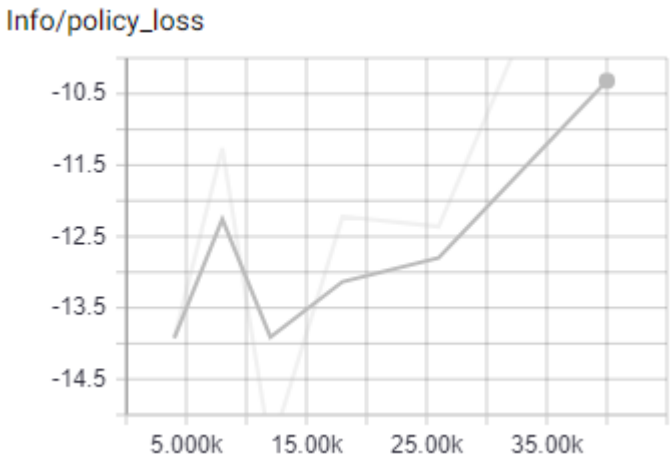

Ilustración 42. Pérdida de política en la acción usar beneficio.

La función del valor estimado determina, si el agente es capaz de determinar el valor para todos los estados visitados. Por tanto, esta función con el tiempo debe crecer, lo cual ocurre en nuestra función, hasta llegar a estabilizarse, momento en el que damos por concluido el entrenamiento. Para reforzar que el entrenamiento ha sido exitoso, debemos analizar la función de pérdida de valor.

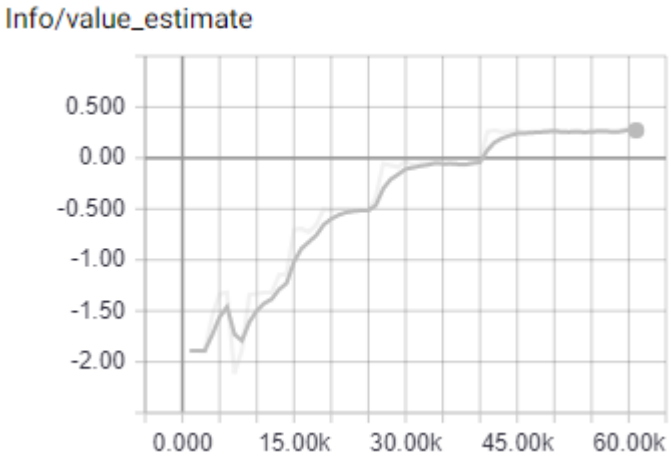

Ilustración 43. Valor estimado en la acción usar beneficio.

La función de pérdida de valor debe crecer en las fases iniciales del entrenamiento y decrecer en cuando los valores de las recompensas se estabilicen. En nuestro caso el agente aprende muy rápido, por tanto, el valor de las recompensas se estabiliza enseguida dando lugar a que la función sea sólo decreciente. Aun así, el resultado obtenido es válido, ya que la función debe terminar en decrecimiento.

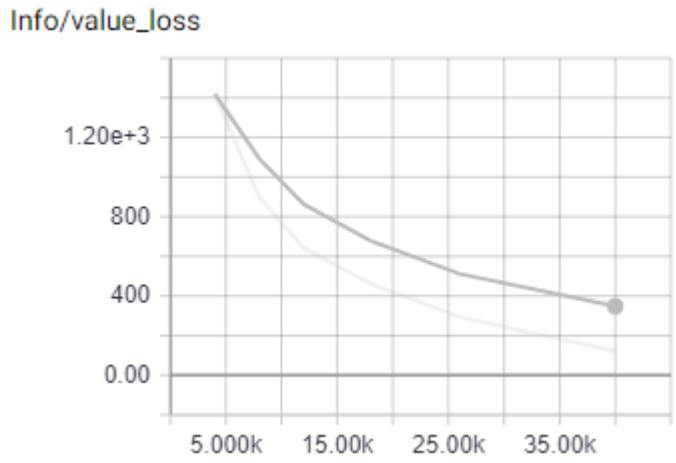

Ilustración 44. Pérdida de valor en la acción usar beneficio.

Una vez explicados los cinco comportamientos de manera individual, y visto su correcto funcionamiento, tanto de las funciones que nos describen el entrenamiento como integrándolo en nuestro juego RPG, debemos analizar el entrenamiento de nuestro agente cuando entran en juego todas las acciones posibles.

Cumulative Reward. Con esta gráfica podemos medir la recompensa media acumulada de todos los agentes en una sesión de entrenamiento. Debe aumentar en una sesión de entrenamiento exitosa.

En nuestro entrenamiento tiene una tendencia de crecimiento y decrecimiento (forma de sierra) ya que inicialmente en entrenamiento empieza para una acción y cuando cambian a la siguiente acción, la función decrece. Podemos observar que el entrenamiento ha sido exitoso, porque antes de pasar a un nuevo comportamiento la función crece.

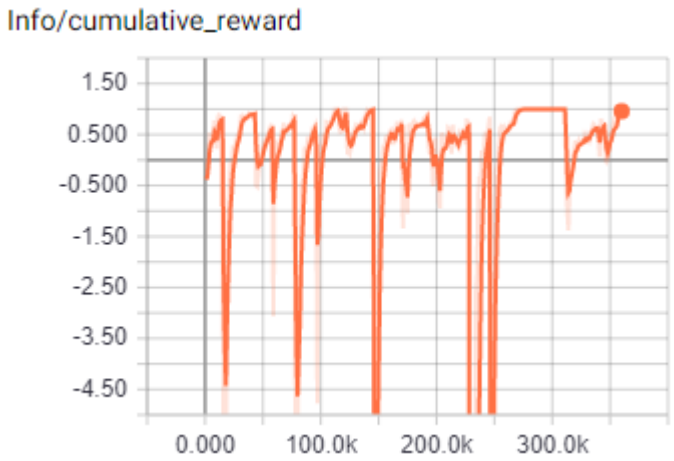

Ilustración 45. Recompensa acumulada en el entrenamiento global.

Entropy. Determina cuán de aleatorias son las decisiones de nuestro modelo. En un proceso de entrenamiento exitoso debe disminuir lentamente. Si dicha gráfica disminuye muy rápido debemos aumentar el valor ***beta*** de los parámetros de configuración.

Dado que el comportamiento de nuestra red neuronal se basa en cinco acciones, nos encontramos con un Agente bastante complicado de entrenar. La entropía va a variar durante todo el entrenamiento, ya que como aprende de manera individual teniendo en cuenta los aprendizajes pasados, habrá picos donde el comportamiento sea aleatorio y otros en los que no. Lo importante es observar la tendencia de la función, que finalmente tiende a cero.

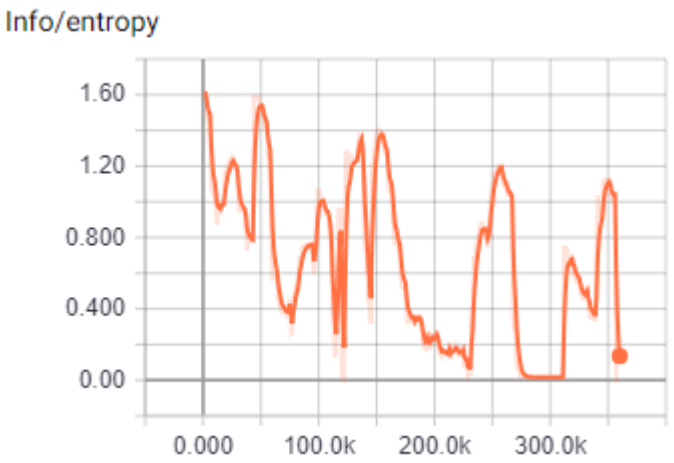

Ilustración 46. Entropía en el entrenamiento global.

Learning Rate. Determina cuánto aprende nuestro modelo mientras sigue buscando el comportamiento óptimo. Con el tiempo debe disminuir.

En nuestro caso, dado que, en los entrenamientos separados, individualmente tenían una forma muy similar, en el entrenamiento conjunto tendrá la forma deseada.

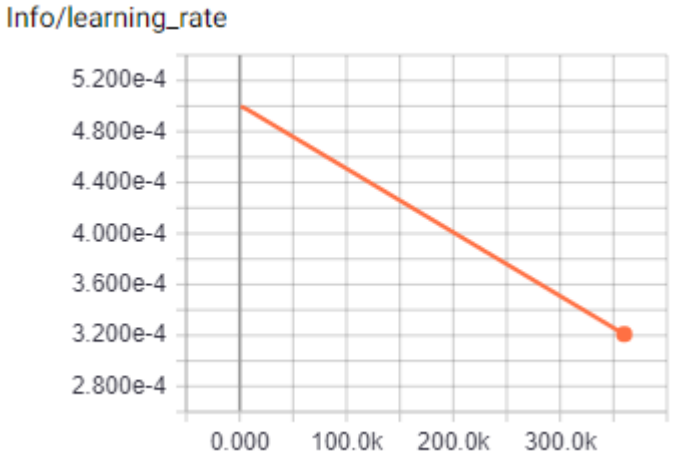

Ilustración 47. Tasa de aprendizaje en el entrenamiento global.

Policy Loss. Esta función determina cuanto pierde el modelo el comportamiento. La magnitud de esta medida debe disminuir durante una sesión de entrenamiento exitosa.

En nuestro entrenamiento a la función de pérdida de comportamiento fluctúa mucho durante el entrenamiento, pero lo importante donde debemos fijarnos es que el valor inicial es mayor que el final, por tanto, podemos decir que ha sido exitoso el entrenamiento.

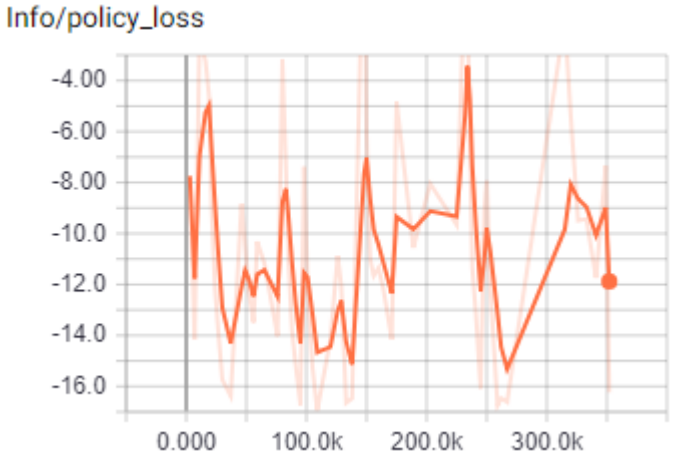

Ilustración 48. Pérdida de política en el entrenamiento global.

Value Estimate. La estimación del valor medio para todos los estados que ha probado el agente. Debe aumentar en una sesión de entrenamiento exitosa.

Inicialmente decrece ya que está explorando todas las posibles soluciones, pero con el tiempo crece. Esto se puede repetir varias veces, ya que estamos juntando varios

comportamientos a la vez, por tanto, pueden interferir unos con otros. Lo importante de dicha función es que con el tiempo tiene una tendencia de crecimiento, que es lo que determina que el entrenamiento ha sido exitoso.

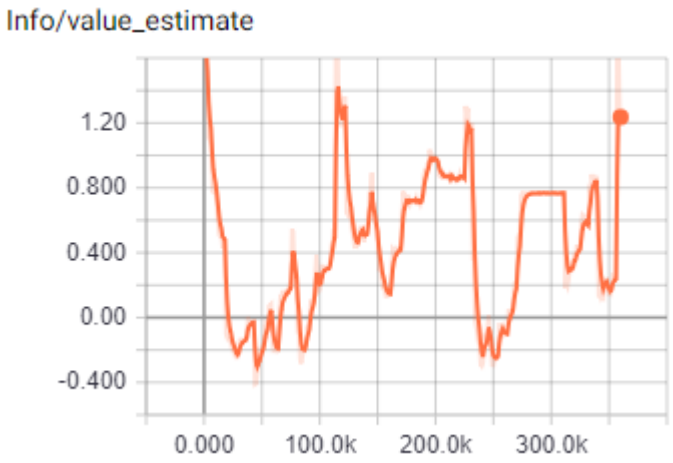

Ilustración 49. Valor estimado en el entrenamiento global.

Value Loss. La pérdida media de la actualización de la función de valor (la función anterior). Determina qué tan bien el modelo puede predecir el valor óptimo para cada estado. Debería aumentar mientras el agente está aprendiendo, y finalmente disminuir cuando la recompensa sea estable.

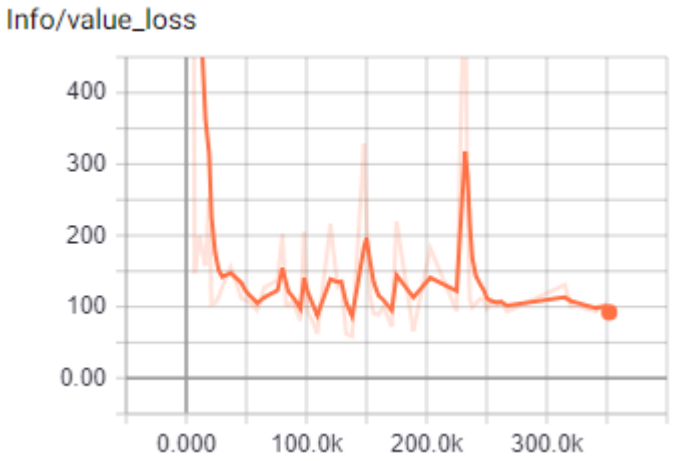

Ilustración 50. Pérdida de valor en el entrenamiento global.

DISCUSIÓN

Hemos podido observar que los resultados son mucho más óptimos cuando estamos entrenando a la red neuronal para una única acción, ya que los pesos y "bias" de la red se ajustan muy rápido. El problema viene cuando estamos trabajando con un espacio de acciones más amplio, ya que el entrenamiento se ralentiza y a veces, las redes

mientras aprenden una nueva acción puede perder el ajuste para alguna acción que aprendió con anterioridad.

Una manera de abordar este problema es contar con equipos expertos en la materia, ya que la experiencia a la hora de diseñar entornos de aprendizaje (con todo lo que conlleva) resulta una tarea muy compleja.

A pesar de este problema, podemos comprobar que las curvas de aprendizaje a la hora de entrenar la red neuronal se ajustan bastante bien a los previsto; esto nos da a entender que las redes neuronales aprenden con rapidez y eso es un punto positivo.

8. CONCLUSIONES

La inteligencia artificial hoy en día la podemos encontrar en numerosas aplicaciones, resolviendo problemas de la vida real, cada vez se usa más y en concreto el subcampo que se centra en las redes neuronales, un campo que lleva desde los años 40 y 50 y, ha vuelto a coger fuerza, ya que muchos de los problemas que surgen hoy en día presentan muy buenos resultados a la hora de abordarlos y resolverlos.

La parte del entrenamiento ha sido la más complicada, un buen diseño de un entorno de entrenamiento es bastante importante y costoso, por ejemplo, determinar el valor de las recompensas positivas o negativas que puede recibir la red neuronal al ejecutar una acción. Como hemos podido ver los algoritmos que permiten trabajar con las redes neuronales trabajan con muchos parámetros que son difíciles de determinar, ya que no hay escritos sobre cuáles son los más óptimos para cada problema. Por estos dos motivos anteriores puede resultar difícil trabajar con ellas y **sólo la experiencia en este campo nos permitirá determinar los mejores valores** para recompensas o para las entradas de los algoritmos.

Aunque debemos destacar que las redes neuronales no son la mejor solución para todo ya que hay otro tipo de problemas donde no se adaptarían correctamente o el proceso de su implementación podría ser más caro o llevar más tiempo.

Aunque en este documento hemos hablado del comportamiento de los personajes no jugadores, futuras ampliaciones que podrían realizarse en el proyecto serían, por ejemplo: generación de mapas (en el proyecto usamos algoritmos procedurales simples), generación de texturas o generación de música.

REFERENCIAS BIBLIOGRÁFICAS

LIBROS

- D. Kincaid y W. Cheney. *Abálisis numérico*. Addison-Wesley Iberoamericana
- Nilo C. Bobillo y Carlos Dehesa. *Introducción al Calculo Tensorial*. Universidad de Oviedo.

LIBROS DIGITALES

- Michael A. Nielsen. *Neural Networks and Deep Learning* [en línea]. Determination Press, 2015. Formato en html. Disponible en: http://neuralnetworksanddeeplearning.com

ARTÍCULOS

- Risi, Sebastian, and Julian Togelius. *Neuroevolution in games: State of the art and open challenges.* IEEE Transactions on Computational Intelligence and AI in Games9.1 (2017): 25-41. Disponible en: https://arxiv.org/pdf/1410.7326.pdf

WEB

- *Procedural Generation* [consulta: 15 agosto 2018]. Disponible en: https://en.wikipedia.org/wiki/Procedural_generation
- *Red neuronal artificial* [consulta: 15 agosto 2018]. Disponible en: https://es.wikipedia.org/wiki/Red_neuronal_artificial
- *The neural networks model* [consulta: 15 agosto 2018]. Disponible en: https://www.ibm.com/support/knowledgecenter/es/SS3RA7_sub/modeler_mainhelp_client_ddita/components/neuralnet/neuralnet_model.html
- *Unreal Engine* [consulta: 15 agosto 2018]. Disponible en: https://es.wikipedia.org/wiki/Unreal_Engine
- *Unity* [consulta: 15 agosto 2018]. Disponible en: https://en.wikipedia.org/wiki/Unity_(game_engine)
- *Component-Based Software Engineering* [consulta: 15 agosto 2018]. Disponible en: https://en.wikipedia.org/wiki/Component-based_software_engineering

OTRAS REFERENCIAS

- *Conjunto de herramientas para agentes de aprendizaje automático en Unity* [consulta: 15 agosto 2018]. Disponible en: https://github.com/Unity-Technologies/ml-agents/blob/master/docs
- *Diablo 3 (juego)* [consulta: 15 agosto 2018]. Disponible en: https://us.battle.net/d3/en
- *Dungeon Hunter 5 (juego)* [consulta: 15 agosto 2018]. Disponible en: http://www.dungeonhunter5.com
- *Unity3D (GameEngine)* [consulta: 15 agosto 2018]. Disponible en: https://unity3d.com/learn
- *Unreal Engine (GameEngine)* [consulta: 15 agosto 2018]. Disponible en: https://docs.unrealengine.com/en-us/
- *Mixamo (Modelos 3D)* [consulta: 15 agosto 2018]. Disponible en: https://www.mixamo.com/#/
- *Gamasutra* [consulta: 15 agosto 2018]. Disponible en: http://www.gamasutra.com

ANEXOS

Juego RPG Desarrollado

En este apartado podemos ver los aspectos más importantes a la hora de desarrollar un videojuego RPG, proporcionando una ligera idea de cómo podemos abordar el problema y en qué sitios podemos buscar información relevante acerca de la característica.

A parte de la programación dentro del desarrollo de videojuegos entran más disciplinas en el desarrollo de los productos. Una de las más importantes son los artistas (dentro de esta rama se divide en muchas más, por ejemplo, modeladores 3D, artistas de concepto, músicos, etc.), en mi caso al no tener conocimientos/habilidades de estas disciplinas he tenido que ir recogiendo modelos o imágenes de internet (los cuales son de uso libre), como por ejemplo Mixamo (ver referencias bibliográficas), que pertenece a Adobe. De modo que, para empezar en el mundo del desarrollo de videojuegos, existen páginas web donde podemos encontrar estos elementos.

A pesar de los siguientes apartados que se basan en mi experiencia, el lector puede encontrar más información u opiniones de gente que lleva tiempo en el desarrollo de videojuegos en Gamasutra o en los propios foros de los motores gráficos (ver referencias bibliográficas).

Personaje

En cualquier juego existe un personaje o personajes principales. Normalmente se suele intentar hacer de forma genérica la implementación de personajes, pero hay otra gran parte de los desarrolladores que prefieren implementar el personaje desde cero en cada juego, ya que dependiendo del juego existirán unas mecánicas u otras.

En un RPG por regla general el personaje tendrá unas estadísticas específicas, como fuerza, armadura, daño de ataque… para ello lo mejor es crear una clase que encapsule dichos atributos ya que es interesante tener separados en componentes y no todo en el mismo script del personaje. Simplemente es una clase que almacena información y no tiene ninguna función en particular, pero será de ayuda para ajustar mecánicas.

Debemos tener claro cuáles son las mecánicas del juego y como éste se va a integrar en ellas. En nuestro juego RPG el movimiento se produce al hacer clic en un sitio, por tanto, debemos tener en cuenta que existen herramientas que la mayoría de los motores gráficos implementan, llamados *Rayscast* (ver referencias bibliográficas). Estos permiten "lanzar" un rayo y determinar información del punto donde ha golpeado, por ejemplo, en nuestro caso la posición a donde debe ir.

Los *Raycast* no sólo será útil para el movimiento, sino que también para cuando este interaccionando con el entorno, ya que como dije con anterioridad, dan información del punto donde ha golpeado, como por ejemplo para saber si el NPC seleccionado es Enemigo o Amistoso.

Debemos crear un buen sistema de combate que explicaremos más adelante, ya que la mecánica es de lo más importante en un juego.

Inventario

Muchos juegos, sobre todo en los RPG, tienen de un sistema de inventario, el cual permite almacenar objetos que vamos consiguiendo a medida que avanzamos en el juego.

El sistema de inventario tiene una parte lógica y una parte de interfaz gráfica, por ello a la hora de realizar este tipo de sistemas, independiente del motor gráfico que usemos, debemos conocer qué funciones soporta, por ejemplo, en Unity existen interfaces de programación (ver referencias bibliográficas) para manejar el comportamiento de los elementos de la interfaz de usuario cuando seleccionas, arrastras y sueltas un elemento, siendo muy importantes para hacer un sistema de inventario que permite colocar objetos a gusto del jugador. Para almacenar los objetos de forma lógica siempre no apoyaremos en las estructuras de datos. En mi caso como se de antemano cuanto será el tamaño del inventario hare uso de un vector normal.

Objetos

Junto con el sistema de Inventario, el uso de objetos con los que interactuar son muy importantes en un juego RPG ya que muchos jugadores pasarán horas jugando para obtener el objeto deseado.

Para abordar este problema, he creado un GameObject con la lógica que debe tener un objeto, como por ejemplo el id, nombre, descripción, etc. De esta manera cada vez que queramos crear un Objeto debemos copiar el GameObject genérico y adaptar las propiedades al objeto en cuestión. Podríamos haber usado un ScriptableObject (ver referencias bibliográficas) para almacenar información sobre el objeto y luego cargarlo en el momento de usarlo.

Serialización

Muy importante a la hora de poder guardar el estado del juego para futuras ocasiones es el uso de la serialización (ver referencias bibliográficas). Los motores gráficos implementan sus sistemas de serialización, por tanto, es importante conoce como funcionan estos.

Permiten hacer un mapeo de los objetos, para guardar propiedades en ficheros. Lo ideal es trabajar con ficheros que estén encriptados ya que si un jugador localiza donde se almacenan las propiedades del juego puede hacer un mal uso de este, por ello aparte de serializar Objetos debemos hacerlo con algoritmos de encriptado. En la herramienta creada no hago uso de estos algoritmos, ya que sale del ámbito de estudio.

Enemigos

Junto con el combate podemos meter dentro de los aspectos más importantes este apartado, ya que en parte el sistema de combate debe estar conectado con los enemigos (pero lo óptimo es implementar las cosas genéricas para que sean dependientes).

Para diseñar a los enemigos podemos tener en cuenta muchas cosas, su tipo de movimiento, tipos de ataque, etc. Para simplificar en el juego, los enemigos son del mismo tipo y sus ataques son a corta distancia.

Misiones

En un juego RPG un motor de misiones es bastante importante, ya que el avance del juego estará basado no sólo en los escenarios o los enemigos, sino en las misiones que se irán realizando, por tanto, tener un motor de misiones permitirá la creación de estas de manera rápida sin costos adicionales en tiempo.

Dado que en este documento no nos centramos en el desarrollo de un videojuego, la manera más rápida de abordar este problema es haciendo uso de un GameObject genérico que representa una misión, con propiedades como el id, a qué cadena de misiones pertenece, nombre, descripción, requisitos para ser completada y recompensas.

De este modo cada vez que queramos crear una misión nueva, copiamos el GameObject y sobrescribimos dichos valores para la misión.

Otra manera de hacerlo sería haciendo uso de los ScriptableObject, en Unity una propiedad muy interesante ya que no existen los Data Tables (como en Unreal) que permite tener toda esa información recogida en un solo documento. Estos objetos sirven para almacenar información, y pueden tratarse como un "tipo" de plugin, ya que se integran con el editor, de esta manera podemos crear misiones sin copiar y pegar el GameObject, ya que desde el propio editor te da la opción.

Combate

En un juego RPG el combate es una de las mecánicas más importantes. En el recae bastante de experiencia de usuario. Para crear un buen sistema de combate tenemos que tener en cuenta varios aspectos como por ejemplo, los enfriamientos (*cooldowns*) de las habilidades, ya que deben tener un tiempo de reutilización, sino podríamos estar atacando todo el tiempo con la misma habilidad; distancia mínima o máxima para usar una habilidad, recordad que hay tres tipos de habilidades (por regla general), a corta distancia, a larga distancia o en área; habilidades que sean beneficios, en mi ejemplo existen gritos para aumentar el daño infligido o para aumentan la armadura y recibir menos daño.

En cuanto a las habilidades hay que tener en cuenta que el personaje tiene unas estadísticas que deben ser utilizadas como multiplicadores para el daño, salud, energía,

etc. Por tanto, una habilidad tendrá un daño/beneficio base que debe ser aumentando en base a las estadísticas, de esta manera proporcionamos la experiencia de progresión al jugador.

Mapas Procedurales

En cuanto a los mapas, en mi ejemplo son de tipo procedural, ya que ofrecen una experiencia diferente cada vez que se juega una mazmorra. De esta manera el jugador deberá elegir diferentes caminos en situaciones similares, dando una sensación de no monotonía. Juegos similares, como por ejemplo Diablo, tiene partes de mapa generados de manera automática.

La programación procedimental (ver referencias bibliográficas) es el método de creación de datos (en el ámbito de este documento: texturas, objetos, modelos, mapas, etc.) con el uso de algoritmos en lugar de forma manual. Este tipo de algoritmos reportan ciertos beneficios a la hora de desarrollar un videojuego, como por ejemplo archivos de menor tamaño, mayor cantidad de contenido o la inclusión de aleatoriedad para experiencias de juego menos predecibles, aunque no sólo se limita a esto.

Los algoritmos que usan este tipo de procedimiento se agrupan dentro de los algoritmos recursivos. Cuanto mayor sea la complejidad del algoritmo es más fácil que se escapen detalles a los desarrolladores incluso llegando a perder el control del juego (elementos con superposición, valores muy alejados de la realidad, etc.). De ahí que, junto a este tipo de programación, sea muy útil el uso de algoritmos de ramificación y poda, ya que permiten explorar todas las soluciones posibles y elegir aquellas que pueden llevarse a la realidad.

Matemáticas Aplicadas

Antes de lanzarnos al desarrollo de un videojuego o tratar con algoritmos de inteligencia artificial debemos tener claro algunos aspectos matemáticos y físicos que muchas veces como estudiantes de ingeniería informática no somos conscientes de su importancia, pero muchos de los problemas en el desarrollo de un videojuego o desarrollo de algoritmos recaen en problemas matemáticos.

Aunque Unity en sus librerías de matemáticas tiene mucha funcionalidad de las que nos podemos abstraer de cómo funcionan debemos tener unos conocimientos básicos,

ya que en la industria del videojuego no solo es programar el *gameplay*, muchas veces tendremos que desarrollar herramientas para facilitar el desarrollo al equipo o programando nuevas características del motor. Unity no ofrece el código del motor, pero permite "modificar" el entorno de desarrollo mediante *plugins*.

Del mismo modo a la hora de crear modelos de aprendizaje en Inteligencia Artificial se han de tener unos conocimientos básicos.

Interpolación

La interpolación se encuadra dentro del área de la matemática llamada análisis numérico y tiene por objeto el uso de funciones sencillas, habitualmente polinomios, para aproximar una función f(x), bien porque la expresión explícita sea muy compleja, o bien porque desconocemos la función y sólo tenemos de ella algunos valores. Nos centraremos en la aproximación mediante polinomios y más concretamente en los llamados polinomios de interpolación. El problema que se nos presenta es el siguiente:

PROBLEMA. - Dada una función f(x), de la que se conocen los n+1 valores $f(x_0)$, $f(x_1)$, $f(x_2)$, …, $f(x_n)$, encontrar un polinomio P(x), de grado $\leq n$, cuya gráfica pase por los puntos $(x_0,f(x_0))$, $(x_1,f(x_1))$, $(x_2,f(x_2))$, …, $(x_n,f(x_n))$.

La primera pregunta que nos plantea este problema es la existencia y unicidad del polinomio de interpolación, y la respuesta que nos da el análisis numérico son los siguientes teoremas que damos sin demostración:

Teorema de unicidad. - Si P(x) y Q(x) son dos polinomios de grado $\leq n$ tales que

$$P(x_k) = Q(x_k) \text{ con } k=0, 1, 2, \ldots, n$$

entonces P(x)=Q(x) para todo x.

Teorema de existencia. - Si x_0, x_1, x_2, …, x_n, son n+1 puntos distintos, el polinomio P(x) de grado $\leq n$ tal que $P(x_k)=Q(x_k)$ con k=0, 1, 2, …, n, viene dado por

P(x)=k=0nf(xk)·AK(x)AK(xk)

siendo

$$A_k(x)=(x-x_0)(x-x_1)\ldots(x-x_{k-1})(x-x_{k+1})\ldots(x-x_n)$$

$$A_k(x_k)=(x_k-x_0)(x_k-x_1)\ldots(x_k-x_{k-1})(x_k-x_{k+1})\ldots(x_k-x_n)$$

Vamos a ver un par de casos sencillos: la interpolación lineal y la interpolación cuadrática.

Interpolación lineal. - Supongamos que de una función conocemos los valores $f(x_0)$ y $f(x_1)$. Según lo visto anteriormente, el polinomio de interpolación P(x) será de grado ≤ 1, y por tanto su gráfica será la recta (polinomio de grado 1) que pasa por los puntos $(x_0,f(x_0))$ y $(x_1,f(x_1))$. Esto es:

$$P(x)=f(x0)+f(x1)-f(x0)x1-x0\cdot(x-x0)$$

Veamos un sencillo ejemplo de aplicación: sabemos que log 31=1,4914 y log 32=1,5051. Determinar log 31,7.

Aquí es f(x)= log x por tanto, de la fórmula anterior, tenemos

$$P(x)=1,4914+1,5051-1,491432-31(x-31)$$

$$P(x)=1,4914+0,0137(x-31)$$

Haciendo x=31,7, tenemos:

$$P(31,7)=1,4914+0,0137\cdot(31,7-31)=1,5010$$

Es decir, log 31,7=1,5010.

Interpolación cuadrática. - Supongamos que de una función conocemos los valores $f(x_0)$, $f(x_1)$ y $f(x_2)$. Según lo visto anteriormente, el polinomio de interpolación P(x) será de grado ≤ 2. En este caso tendremos un polinomio cuadrático (una parábola) que pasa por los puntos $(x_0,f(x_0))$, $(x_1,f(x_1))$ y $(x_2,f(x_2))$. Tenemos

$$P(x) = f(x0)\cdot(x-x1)\cdot(x-x2)(x0-x1)\cdot(x0-x2)+f(x1)\cdot(x-x0)\cdot(x-x2)(x1-x0)\cdot(x1-x2)$$

$$+f(x2)\cdot(x-x0)\cdot(x-x1)(x2-x0)\cdot(x2-x1)$$

Veamos un ejemplo:

La siguiente tabla muestra medidas de alturas en metros alcanzada por un movimiento vertical y sus correspondientes tiempos en dichos instantes.

1	2	3
68	88	94

Se halla el polinomio P(x):

$$P(x)=68\cdot(x-2)\cdot(x-25)(1-2)\cdot(1-25)+88\cdot(x-1)\cdot(x-2,5)(2-1)\cdot(2-2,5)+94\cdot(x-1)\cdot(x-2)(2,5-1)\cdot(2,5-2)$$

$$P(x)=45,33\cdot(x-2)\cdot(x-2,5)-176\cdot(x-1)\cdot(x-2,5)+125,33\cdot(x-1)\cdot(x-2)$$

De donde, se obtiene P(1, 5) = 79,33 y P(4) = 95,97

El método usado hasta ahora aparece en la literatura matemática como método de interpolación de Lagrange. Cuando el número de puntos crece, este método se hace algo fatigoso, siendo necesario el uso de otros métodos como el método de interpolación de Newton o de diferencias finitas, el método de interpolación de Hermite o el método de interpolación por splines. Remitimos al lector interesado al libro señalado en la bibliografía *Análisis numérico* de D. Kincaid y W. Cheney, Addison-Wesley Iberoamericana.

Tensores

El concepto de tensor es un concepto matemático complejo, pues engloba muchos objetos matemáticos (vectores, covectores, operadores lineales, funciones bilineales, funciones multi lineales), así como nuevos objetos creados a la medida (en nuestro caso, como veremos, relacionados con las redes neuronales) y que pretende dar un tratamiento unificado de todos estos objetos. Es precisamente esta generalización lo que hace que el concepto de tensor sea de difícil asimilación. Para tratar de entender que es, nos fijaremos en un concepto matemático que estudiamos en el bachillerato: el producto escalar. No olvidemos que el producto escalar es la herramienta matemática que nos sirve para medir distancias en geometría. Más tarde veremos qué relación tienen los tensores con las redes neuronales.

Suponemos conocido el concepto de espacio vectorial. Definimos una función bilineal (que es un tensor), G: **V**x**V** → **R**, como aquella que asocia a cada pareja de vectores (**x,y**) ϵ **V**x**V,** siendo además lineal en ambos argumentos:

$$G(\lambda \cdot \boldsymbol{x_1} + \mu \cdot \boldsymbol{x_2}, \boldsymbol{y}) = \lambda \cdot G(\boldsymbol{x_1}, \boldsymbol{y}) + \mu \cdot G(\boldsymbol{x_2}, \boldsymbol{y})$$

$$G(\boldsymbol{x}, \lambda \cdot \boldsymbol{y_1} + \mu \cdot \boldsymbol{y_1}) = \lambda \cdot G(\boldsymbol{x}, \boldsymbol{y_1}) + \mu \cdot G(\boldsymbol{x}, \boldsymbol{y_2})$$

El producto escalar es un ejemplo de función bilineal (o tensor). Dados dos vectores

$$\boldsymbol{x} = x^1 \cdot \boldsymbol{e_1} + x^2 \cdot \boldsymbol{e_2}$$

$$\boldsymbol{y} = y^1 \cdot \boldsymbol{e_1} + y^2 \cdot \boldsymbol{e_2}$$

En una base B(**e**$_1$**, e**$_2$) se define el producto escalar como:

$\boldsymbol{x} \cdot \boldsymbol{y} = g(\boldsymbol{x}, \boldsymbol{y}) = (x^1 \cdot \boldsymbol{e_1} + x^2 \cdot \boldsymbol{e_2}) \cdot (y^1 \cdot \boldsymbol{e_1} + y^2 \cdot \boldsymbol{e_2}) = x^1 \cdot y^1(\boldsymbol{e_1} \cdot \boldsymbol{e_1}) + x^1 \cdot y^2(\boldsymbol{e_1} \cdot \boldsymbol{e_2}) + x^2 \cdot y^1(\boldsymbol{e_2} \cdot \boldsymbol{e_1}) + x^2 \cdot y^2(\boldsymbol{e_2} \cdot \boldsymbol{e_2}) = x^1 \cdot y^1 \cdot g_{11} + x^1 \cdot y^2 \cdot g_{12} + x^2 \cdot y^1 \cdot g_{21} + x^2 \cdot y^2 \cdot g_{22}$

o en forma matricial

$$\boldsymbol{x} \cdot \boldsymbol{y} = \begin{pmatrix} x^1 & x^2 \end{pmatrix} \cdot \begin{pmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{pmatrix} \cdot \begin{pmatrix} y^1 \\ y^2 \end{pmatrix}$$

La función $g(\boldsymbol{x}, \boldsymbol{y})$ recibe el nombre de tensor métrico y queda determinado por la matriz

$$\begin{pmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{pmatrix}$$

En unos ejes con base ortonormal (perpendiculares y normalizados a la unidad) tiene como valor

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Y el producto escalar toma la forma que conocemos del bachillerato

$$\boldsymbol{x} \cdot \boldsymbol{y} = g(\boldsymbol{x}, \boldsymbol{y}) = = x^1 \cdot y^1 + x^2 \cdot y^2$$

Si hacemos $\boldsymbol{x} = \boldsymbol{y}$ en el producto escalar y hallamos su raíz cuadrada, estaríamos calculando la distancia desde el origen al punto de coordenadas $(x^1 \quad x^2)$**;** es decir, asociamos a cada punto del espacio, mediante su tensor métrico, la distancia al origen de coordenadas. Igualmente, en Relatividad General, se asociado a cada punto un tensor, llamado tensor de curvatura, que indica cómo se "curva" el espacio en ese punto. Y así, podemos asociar al "espacio" cualquier característica, definiendo un cierto tensor. Hemos escrito espacio entre comillas, pues éste no tiene que ser el espacio geométrico.

¿Qué relación tienen entonces los tensores con las redes neuronales? Las redes neuronales artificiales se basan en las redes neuronales biológicas mediante un modelo matemático que usa grafos

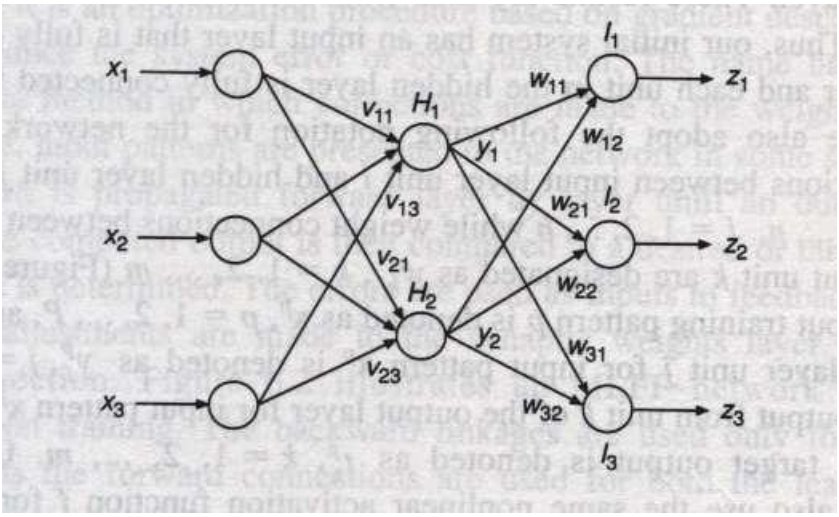

Ilustración 51. Relación entre tensores y redes neuronales.

Tomado de D. Balbontín Noval, F. J. Martın Mateos, J. L. Ruiz Reina, Dpto. Ciencias de la Computación e Inteligencia Artificial. Universidad de Sevilla

El funcionamiento básico es el siguiente: cada nodo (unidad, neurona) se conecta mediante arcos dirigidos (i→j) a los que se asigna un peso w_{ij} (un número). La red (y por tanto cada nodo) recibe una entrada y devuelve una salida que calculamos mediante una función $a_i = g(\sum_0^n w_{ij}\, a_j)$ que podemos manejar como un tensor. No profundizaremos más, pues como comentamos al principio el concepto de tensor es un concepto complejo y aquí sólo hemos pretendido dar unas pinceladas sobre el mismo.

Gradiente. Máximos y mínimos de funciones

A veces nos enfrentamos a problemas en los que estamos interesados en la dirección de crecimiento o decrecimiento más rápido de una función o en determinar que valores hacen máxima o mínima a dicha función. Por ejemplo, calentamos una placa metálica y estamos interesados en cómo es la temperatura en ciertos puntos de la misma o en qué puntos la temperatura es máxima o mínima. La herramienta matemática que usamos en este caso es el **gradiente** y que está relacionado con la derivada (variación de una función). Vamos, en primer lugar, a dar la definición de gradiente para posteriormente tratar de entender su significado.

Para ser más ilustrativo, vamos a usar el caso de una función bidimensional; es decir $f: R^2 \rightarrow R$. La función $f = f(x, y)$ puede ser la temperatura de una placa metálica. La derivada direccional de f, que representamos como $\boldsymbol{f'_u}$, en la dirección del vector unitario $\boldsymbol{u} = (\boldsymbol{e_x}, \mathbf{e_y})$ es

$$f'_u = \frac{\partial f}{\partial x}\, e_x + \frac{\partial f}{\partial y}\, e_y$$

Al vector $\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$ se le denomina gradiente de f y la derivada direccional la podemos escribir como

$$f'_u = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) . (e_x, \mathbf{e_y}) = \nabla f . \boldsymbol{u}$$

Si observamos esta ecuación vemos que el gradiente de una función nos da la dirección de crecimiento más rápido de una función (recordemos el concepto de derivada y su significado) y que el decrecimiento está en la dirección opuesta. Es decir, en cualquier punto del plano (recordemos que trabajos en R^2) podemos calcular el gradiente de una función (**campo de gradientes**) y si multiplicamos este por un vector que nos da una dirección (la que sea), obtenemos su derivada direccional; es decir, como varía la función en esa dirección y el gradiente, por sí solo, nos da la dirección de crecimiento más rápido.

¿Y cómo calculamos los máximos y mínimos de la función? Si para una función real de variable real calculamos su derivada e igualamos a cero, para una función de dos o más variables calculamos su gradiente y lo igualamos a cero; es decir $\nabla f = 0$. Los puntos donde se anula el gradiente se llaman **puntos estacionarios** o **críticos**. Para saber si un punto $(\boldsymbol{x_0}, \mathbf{y_0})$ es máximo o mínimo construimos el determinante de la llamada matriz hessiana, |H|

$$\begin{vmatrix} f_{xx} & f_{xy} \\ f_{xy} & f_{yy} \end{vmatrix} = \begin{vmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{vmatrix}$$

Si |H|>0 y $\frac{\partial^2}{\partial x^2} f > 0$ se dice que tenemos un máximo.

Si |H|>0 y $\frac{\partial^2}{\partial x^2} f < 0$ se dice que tenemos un mínimo.