Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

IA en Videojuegos: Deep Reinforcement Learning

AI in Videogames: Deep Reinforcement Learning

Manuel Armillas Hernández

La Laguna, 24 de mayo de 2023

D. **Jesús Miguel Torres Jorge**, con N.I.F. 43.826.207-Y profesor Contratado Doctor adscrito al Departamento de **Ingeniería Informática y de Sistemas** de la Universidad de La Laguna, como tutor

D. **José Demetrio Piñeiro Vera**, con N.I.F. 43.774.048-B profesor Titular de Universidad adscrito al Departamento de **Ingeniería Informática y de Sistemas** de la Universidad de La Laguna, como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"IA en Videojuegos: Deep Reinforcement Learning"

ha sido realizada bajo su dirección por D. **Manuel Armillas Hernández**,

con N.I.F. 43837925V.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 24 de mayo de 2023

Agradecimientos

En primer lugar quería agradecer a mi familia por estar ahí siempre, en las buenas y en las malas, ayudándome a seguir adelante y no rendirme.

También quería agradecer a mis amigos por ayudarme a distraerme, por sacarme siempre una sonrisa y por escucharme siempre.

Quiero agradecer a mis tutores, Jesús y Demetrio, que siempre han estado a mi disposición cuando he necesitado ayuda.

Por último pero no menos importante, quería agradecer a mi padre por enseñarme tanto sobre la vida, estoy seguro que le hubiera hecho muy feliz ver la persona en la que me he convertido.

Licencia

Resumen

El objetivo de este trabajo ha sido el de desarrollar un videojuego en Unity3D que funcionara con Deep Reinforcement Learning (subrama del aprendizaje automático que combina el aprendizaje por refuerzo y el aprendizaje profundo), haciendo uso de la librería ML Agents de Unity para el entrenamiento de agentes.

A lo largo del proyecto comentaremos cada uno de los juegos desarrollados, explicando su funcionamiento y objetivo y haciendo énfasis en las herramientas utilizadas y la codificación del agente

Palabras clave: DRL, Unity3D, agente, ML Agents

Abstract

The objective of this project has been to develop a Unity3D video game that works with Deep Reinforcement Learning (a subbranch of machine learning that combines reinforcement learning and deep learning), making use of Unity's ML Agents library for training agents.

Throughout the project we will comment on each of the games developed, explaining its operation and objective and emphasizing the tools used and the agent's coding.

Keywords: DRL, agent, Unity3D, ML Agents

Índice general

Índice de figuras

Capítulo 1 Introducción

La Inteligencia Artificial en un videojuego es todas aquellas técnicas que permiten recrear una ilusión de inteligencia o un cierto comportamiento que tiene que realizar un NPC (Non playable character) para interactuar con nosotros los usuarios. Mejorar la inteligencia artificial de un videojuego siempre ha sido una prioridad a la hora de desarrollarlos dada la importancia de sentir que el mundo está vivo o de agregarle dificultad al juego.

El objetivo de este proyecto es el de entrenar a un agente que sea capaz de jugar por sí mismo en un videojuego, todo ello empleando técnicas de *Deep Reinforcement Learning* (DRL).

1.1 Deep Reinforcement Learning

Para el desarrollo del agente utilizaremos técnicas de DRL, técnica que juega un papel muy importante en la IA de los videojuegos.

El Aprendizaje por Refuerzo Profundo (DRL) es la combinación del Aprendizaje por Refuerzo y el Aprendizaje Profundo. Puede resolver una amplia gama de tareas complejas de toma de decisiones que anteriormente estaban fuera del alcance de una máquina para resolver problemas del mundo real con inteligencia similar a la humana.

El aprendizaje por refuerzo es un tipo de machine learning en el que los agentes aprenden la política óptima por ensayo y error. Al interactuar con el entorno, DRL puede ser exitoso aplicado con éxito a tareas secuenciales de toma de decisiones.

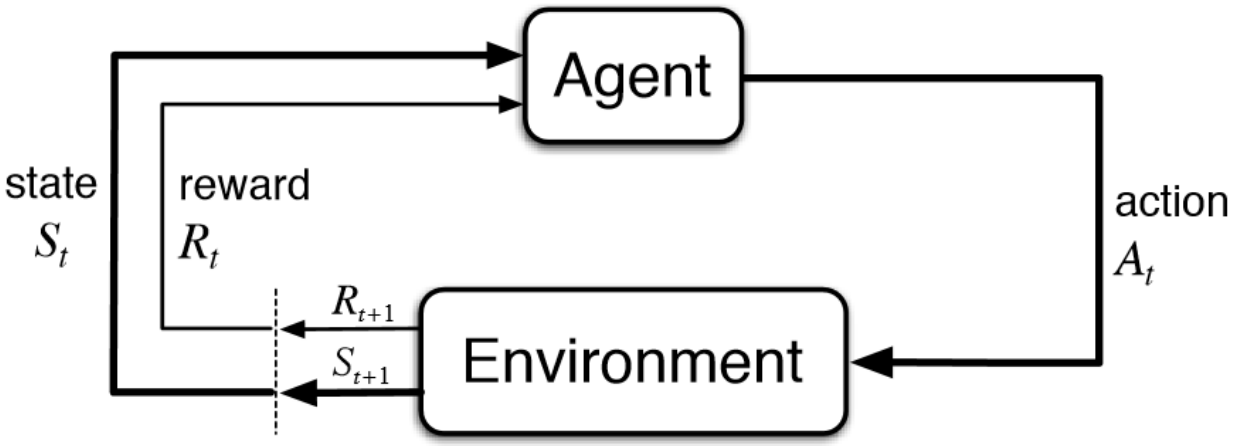

Figura 1.1.1: Bucle aprendizaje por refuerzo

Este es el bucle del aprendizaje profundo. El agente ejecuta una acción sobre el entorno y cambia de estado dependiendo del estímulo que reciba del mismo, pudiendo ser o no una recompensa.

1.2 ML Agents

ML Agents es una librería descargable para Unity3D (también disponible para Python) que nos permite realizar el entrenamiento de un agente, diseñando su comportamiento y acciones mediante las funciones que esta incluye. Hablaremos más en profundidad de ella en el siguiente capítulo.

Desarrollar el escenario de entrenamiento con un motor de videojuegos nos da ciertas ventajas. Una de ellas es que nos facilita mucho el diseño del entorno, pudiéndose hacer de forma gráfica y ahorrándonos horas de codificación. Otra ventaja es que puedes realizar cosas mucho más complejas jugando con físicas, eventos o desarrollando un juego mucho más complejo.

1.3 Estado del arte

Hemos tratado ML-Agents como un framework de Unity3D donde se nos permite entrenar distintos tipos de agentes para distintos juegos, pero esta herramienta va más allá de esto, veamos algunas de sus aplicaciones:

- Videojuegos: como ya sabemos, se puede emplear para el entrenamiento de agentes.

- Simulaciones y entrenamiento: la herramienta puede ser utilizada para entrenar agentes virtuales en simulaciones que imiten el comportamiento esperado en el juego con el fin de testear el funcionamiento del mismo.
- Robótica: ML-Agents se puede utilizar para entrenar y mejorar el comportamiento de robots autónomos. Estos robots pueden aprender a realizar tareas complejas y adaptarse a entornos cambiantes utilizando técnicas de aprendizaje por refuerzo.
- Optimización de procesos industriales: La herramienta puede utilizarse para optimizar procesos industriales al entrenar agentes virtuales para tomar decisiones eficientes. Por ejemplo, se pueden entrenar agentes para controlar la producción en una fábrica, maximizar la eficiencia energética o reducir los tiempos de espera en una cadena de suministro.

Estas son algunas aplicaciones reales donde se usa ML Agents, pero profundizaremos más en la parte de videojuegos y robótica.

Videojuegos

En el ámbito real en videojuegos se emplea ML-Agents sobre todo en el entrenamiento de NPC, ya sean enemigos o aliados. Podemos encontrar muchos proyectos que emplean esta herramienta para suplir esta necesidad.

Por otro lado, se puede utilizar ML-Agents para generar contenidos en los videojuegos de forma automática. Se genera contenido procedimental en el juego, como niveles, terrenos o misiones. Los agentes de aprendizaje por refuerzo pueden explorar y aprender de diferentes configuraciones de contenido para generar experiencias de juego únicas y variadas.

Estas aplicaciones hacen que ML-Agents sea un toolkit a tener en cuenta si estás trabajando con Unity3D, te abre un gran abanico de posibilidades, desde NPCs a ahorrarte horas de trabajo mediante la generación automática de contenido.

Robótica

Ya hemos comentado antes que el toolkit de ML-Agents es capaz de entrenar robots. Algunos proyectos que se pueden realizar con esta tecnología es, por ejemplo, un brazo robótico que permita clasificar y agarrar objetos. ¿Cómo se hace esto?

Primero, se recopilan imágenes y datos de entrenamiento que representaban diferentes objetos con diversas formas y colores. Estos datos se utilizan para entrenar el modelo de aprendizaje por refuerzo.

Luego, se crea un entorno de simulación en Unity donde se colocan los objetos y se simula el brazo robótico. El entorno proporciona información visual y de estado al agente de aprendizaje.

Se proporciona información visual a través de la cámara del robot así como la posición del y orientación del brazo. Ya hemos hablado de esta forma de recopilar observaciones antes (Camera Sensor).

Finalmente, se entrena al agente en el entorno diseñado. El agente aprende a mover el brazo robótico, así como a clasificar los objetos.

Uno de los principales beneficios que da mezclar la robótica con ML-Agents es que se puede enseñar a los robots a realizar tareas específicas, como la clasificación de objetos. Esto tiene aplicaciones prácticas en la industria, donde los robots pueden ser entrenados para realizar tareas de selección o clasificación en almacenes, líneas de producción, etc.

1.4 Algoritmos aprendizaje por refuerzo profundo

Existe una gran variedad de algoritmos para el problema del DRL, algunos más usados que otros depende de la finalidad o del comportamiento esperado. Algunos ejemplos de algoritmos más utilizados son Deep Q-network (DQN), Deep Deterministic Policy Gradient (DDPG), Soft Actor Critical (SAC) y Proximal Policy Optimization (PPO). ML Agents tiene la capacidad de usar PPO como principal algoritmo de aprendizaje.

PPO es un algoritmo de DRL desarrollado por OpenAI y se considera lo último en cuanto a algoritmos de aprendizaje reforzado y presenta un equilibrio entre rendimiento y comprensión.

El objetivo de PPO es mejorar la política de acción de un agente para maximizar su recompensa acumulada a lo largo del tiempo. En lugar de optimizar la política directamente, PPO utiliza una estrategia de optimización basada en muestras. Esto implica recolectar muestras de experiencia a través de interacciones del agente con el entorno y luego usar esas muestras para mejorar gradualmente la política.

PPO tiene 2 etapas principales, la recolección de datos y la etapa de actualización de políticas. En la de recolección de datos el agente recopila datos de experiencia mediante la interacción con el entorno, teniendo en cuenta la recompensa obtenida, las acciones tomadas y otros valores que puedan ser relevantes.

En la etapa de actualización de políticas se utilizan los datos recolectados para mejorar la política actual. PPO también introduce un término de regularización que controla la magnitud de los cambios en la política en cada iteración, utilizando una medida de proximidad entre políticas antiguas y nuevas para evitar cambios drásticos y asegurar una

optimización gradual y estable.

1.5 Planificación del proyecto

El objetivo inicial del proyecto fue el de entrenar un agente para varios niveles de un laberinto o un *graybox* utilizando la tecnología ML-Agents disponible en Unity haciendo uso de algoritmos de DRL. Finalmente, se optó por hacer un laberinto con varios niveles.

Las tareas previstas fueron las siguientes:

- Investigación de la herramienta
- Diseño de los distintos niveles del juego
- Desarrollo del juego en Unity
- Hacer una lista de las posibles observaciones que puede tener el agente
- Testeo del juego
- Entrenamiento
- Testeo del agente

Esta fue la planificación inicial para el proyecto, sin embargo, mientras se realizaba la primera tarea de investigación me di cuenta de las posibilidades que esta daba y decidí cambiar ciertas cosas. Lo primero es que quería enseñar algunas de las herramientas de ML-Agents que me parecían interesantes. Por otro lado, me resultaba más ameno diseñar varios juegos distintos con comportamientos diferentes a centrarme en un único juego donde el agente realiza acciones similares en cada nivel.

Opté por redirigir el proyecto a hacer juegos totalmente diferentes, donde la fuente de recompensa del agente fuera distinta en cada uno y poder plasmar las distintas tecnologías donde más convenía.

Las tareas previstas para esta nueva fase fueron las mismas que las mencionadas anteriormente, pero haciendo las tareas de forma cíclica para cada juego.

1.6 Experimentos realizados

Como ya hemos mencionado en las secciones anteriores, hemos hecho uso tanto de Unity3D como de la librería ML Agents para el diseño de los agentes y el entorno.

Los agentes han sido diseñados usando una amplia variedad de herramientas que la librería nos facilita y usando acciones diferentes con las que interactúan con el entorno para actualizar su política.

Los experimentos realizados son:

- Agente que busca una bola usando observaciones y aprendizaje por imitación.
- 2 agentes que compiten por encontrar comida usando visión por cámara (reconocimiento de imágenes).
- 2 niveles de un dungeon escape usando Ray Perception Sensor.
- Un agente que tiene que sobrevivir contra zombies que vienen a por él usando disparos y su movimiento.
- Volleyball 1 vs 1
- Football 3 vs 3

En futuros capítulos hablaremos más en profundidad de cada uno de los experimentos realizados (si han sido exitosos o no), comentando cada uno de los métodos empleados, el entorno, tendencias del agente, ficheros de configuración…

Capítulo 2 Background general de la herramienta

En el capítulo anterior se ha introducido la herramienta utilizada para llevar a cabo el proyecto, indicando para qué sirve y mencionando algunas de las funcionalidades que esta incluye pero sin entrar en detalles. A lo largo de este capítulo hablaremos de la librería más en profundidad en pos de entender a la perfección cómo esta funciona y ver su gran abanico de posibilidades.

2.1 Introducción

Como ya hemos mencionado anteriormente, ML-Agents es una librería descargable para Unity3D (también disponible para Python) que nos permite realizar el entrenamiento de un agente, diseñando su comportamiento y acciones mediante las funciones que esta incluye; permitiendo además un tuneo muy sencillo y amplio mediante un fichero de configuración.

Para ejecutar la librería y poner en marcha el entrenamiento simplemente debemos escribir el comando:

mlagents-learn [ruta fichero de configuración] --run-id=[nombre del entrenamiento] *--initialize-from=[nombre de un entrenamiento desde el que querramos arrancar] --tensorboard* ,

siendo las variables en cursiva aquellas que son opcionales. El orden al poner cada una de estas variables es indiferente. Una vez pulsado *enter* y puesto en ejecución nos aparecerá el símbolo de Unity en la consola y solo tendremos que pulsar el botón *Play* en Unity para que el entrenamiento comience.

Si hemos utilizado la opción *--tensorboard* se nos abrirá una ventana en el navegador con los detalles de la evolución temporal del entrenamiento, lo cual nos será útil a la hora de ver gráficamente con datos la variación de la bondad del comportamiento de nuestro agente.

La herramienta es capaz de entrenar diversos tipos de agentes, con distintas formas de recopilar observaciones (como veremos más adelante). Además, tiene una gran capacidad de tuning gracias a todas las herramientas que incluye y al gran número de parámetros modificables en su fichero de configuración.

2.2 Ficheros de configuración

Los ficheros de configuración es una de las funcionalidades que permiten tunear el comportamiento del agente, la velocidad de entrenamiento, entre otras. Las variables que estas incluyen hacen que podamos modificar el tipo de entrenamiento (algoritmo que se va a usar), el número de neuronas, número de capas, la forma en la que percibe la recompensa, entre otras. A lo largo de esta sección estaremos hablando de cada una de ellas, indicando cómo influyen en el comportamiento del agente. El formato de estos ficheros es *.yaml*.

Veamos un ejemplo de fichero de configuración:

```yaml
behaviors:
    MoveAgent:
        trainer_type: ppo
        hyperparameters:
            batch_size: 128
            buffer_size: 2048
            learning_rate: 0.0003
            beta: 0.01
            epsilon: 0.2
            lambd: 0.95
            num_epoch: 3
            learning_rate_schedule: linear
        network_settings:
            normalize: false
            hidden_units: 512
            num_layers: 4
            vis_encode_type: simple
        reward_signals:
            extrinsic:
                gamma: 0.99
                strength: 1.0
        keep_checkpoints: 5
        max_steps: 1000000
        time_horizon: 128
        summary_freq: 5000
        threaded: true
```

Figura 2.2.1: Fichero de configuración

Este es el fichero de configuración usado para el agente que busca una bola (hablaremos en otro capítulo de dicho juego). Se pueden apreciar varias cosas, la primera, en la línea 2, donde se indica el nombre del Behaviour, el cual debe coincidir con el campo que aparece en el inspector para el *Behaviour name* (si no salta error), lo cual no es más que un *id* para identificar el entrenamiento.

En la siguiente línea se especifica el tipo de entrenamiento, lo cual no es más que el algoritmo de Reinforcement Learning a usar que, como ya mencionamos en la introducción (Capítulo 1), se trata de un PPO (Proximal Policy Optimization), algoritmo de DRL que es el que usaremos a lo largo del proyecto.

A continuación se encuentra el bloque de *hyperparameters*, que junto al bloque de *network_settings* y *reward_signals* nos va a permitir el tuning de nuestro agente. Hablemos de cada una de las variables de este bloque, tanto de su significado como de su valor:

- **learning_rate**: Tasa de aprendizaje inicial para descenso de gradiente. Corresponde a la fuerza de cada paso de actualización de descenso de gradiente. Por lo general, esto debe reducirse si el entrenamiento es inestable y la recompensa no aumenta constantemente. Rango estandar: `1e-5` - `1e-3`

- **batch_size**: Número de experiencias en cada iteración de descenso de gradiente. Esto siempre debe ser varias veces más pequeño que buffer_size. Si usa acciones continuas, este valor debe ser grande (del orden de 1000). Si está utilizando solo acciones discretas, este valor debe ser más pequeño (del orden de 10). Rango estándar: (Continuous - PPO): `512` - `5120`; (Continuous - SAC): `128` - `1024`; (Discrete, PPO & SAC): `32` - `512`.

- **buffer_size**: Número de experiencias a recopilar antes de actualizar el modelo de política. Corresponde a cuántas experiencias se deben recopilar antes de realizar cualquier aprendizaje o actualización del modelo. Esto debería ser varias veces más grande que batch_size. Normalmente, un buffer_size más grande corresponde a actualizaciones de entrenamiento más estables. Rango estándar: `2048` - `409600`

- **learning_rate_schedule**: Determina como el learning rate cambia a lo largo del tiempo. Para PPO, recomendamos disminuir la tasa de aprendizaje hasta max_steps para que el aprendizaje converja de manera más estable. Sin embargo, en algunos casos (p. ej., entrenamiento durante un período de tiempo desconocido), esta función se puede desactivar. linear decae el learning_rate linealmente, llegando a 0 en max_steps, mientras que constant mantiene constante el ritmo de aprendizaje durante toda la ejecución de entrenamiento.

- **beta**: Fuerza de la regularización de la entropía, que hace que la política sea "más aleatoria". Esto asegura que los agentes exploren adecuadamente el espacio de acción durante el entrenamiento. Aumentar esto asegurará que se tomen más acciones aleatorias. Esto debe ajustarse de manera que la entropía (medible desde TensorBoard) disminuya lentamente junto con los aumentos en la recompensa. Si la entropía cae demasiado rápido, se debería aumentar la beta, si la entropía cae muy lentamente, disminuya beta.

- **bete_schedule**: Determina cómo cambia beta con el tiempo. linear disminuye beta linealmente, llegando a 0 en max_steps, mientras que constant mantiene constante a beta durante toda la ejecución de entrenamiento. Si no se establece explícitamente, el cronograma beta predeterminado se establecerá en hiperparámetros -> learning_rate_schedule.

- epsilon:Influye en la rapidez con la que la política puede evolucionar durante el entrenamiento. Corresponde al umbral aceptable de divergencia entre la política antigua y la nueva durante la actualización de descenso de gradiente. Establecer este valor en un valor pequeño dará como resultado actualizaciones más estables, pero también ralentizará el proceso de capacitación. Rango estándar: 0.1 - 0.3

- **epsilon_schedule**: Determina cómo cambia épsilon con el tiempo (solo PPO). linear disminuye épsilon linealmente, llegando a 0 en max_steps, mientras que constant mantiene el épsilon constante durante toda la ejecución de entrenamiento. Si no se establece explícitamente, el programa épsilon predeterminado se establecerá en hiperparámetros -> learning_rate_schedule.

- **lambd**: Parámetro de regularización (lambda) utilizado al calcular la Estimación de Ventaja Generalizada (GAE). Esto se puede considerar como cuánto confía el agente en su estimación de valor actual al calcular una estimación de valor actualizada. Los valores bajos corresponden a confiar más en la estimación del valor actual (que puede tener un alto sesgo), y los valores altos corresponden a confiar más en las recompensas reales recibidas en el entorno (que pueden tener una variación alta). El parámetro proporciona una compensación entre los dos, y el valor correcto puede conducir a un proceso de entrenamiento más estable. Rango estándar: 0.9 - 0.95

- **num_epoch**: Número de pases para realizar a través del búfer de experiencia al realizar la optimización de descenso de gradiente. Cuanto mayor sea el batch_size, mayor será aceptable para hacer esto. Disminuir esto asegurará actualizaciones más estables, a costa de un aprendizaje más lento. Rango estándar: 3 - 10

- **shared_critic**: Si las redes de función de política y valor comparten o no una columna vertebral. Puede ser útil usar una red troncal compartida cuando se aprende a partir de observaciones de imágenes. Posibles valores: true o false

Pasemos a hablar delas network_settings. Su cometido es el de configurar la red neuronal, número de capas de la red, número de neuronas por capa…

- **hidden_units**: Número de unidades en las capas ocultas de la red neuronal. Corresponde a cuántas unidades hay en cada capa completamente conectada de la red neuronal. Para problemas simples donde la acción correcta es una combinación directa de las entradas de observación, este valor debería ser pequeño. Para problemas donde la acción es una interacción muy compleja entre las variables de observación, debería ser mayor. Rango estándar: 32 - 512
- **num_layers**: El número de capas ocultas en la red neuronal. Corresponde a cuántas capas ocultas están presentes después de la entrada de observación.

Para problemas simples, es probable que menos capas entrenen más rápido y de manera más eficiente. Pueden ser necesarias más capas para problemas de control más complejos. Rango estándar: 1 - 3

- **normalize**: la normalización se aplica a las entradas de observación de vectores. Esta normalización se basa en el promedio móvil y la varianza de la observación del vector. La normalización puede ser útil en casos con problemas de control continuo complejos, pero puede ser perjudicial con problemas de control discreto más simples. Su valor puede ser true o false.
- **vis_encode_type**: Tipo de codificador para codificar observaciones visuales (hablaremos más en profundidad de este tipo de observaciones en futuros capítulos).

  ….

- **conditioning_type**: Tipo de condicionamiento para la política utilizando observaciones de objetivos. none trata las observaciones de objetivos como observaciones regulares, hyper (predeterminado) utiliza una HyperNetwork con observaciones de objetivos como entrada para generar algunos de los pesos de la política. Hay que tener en cuenta que cuando se usa hiper, la cantidad de parámetros de la red aumenta considerablemente. Por lo tanto, se recomienda reducir el número de hidden_units cuando se utiliza este conditioning_type.

Por último hablemos de las reward_signals. Estas se encargan de controlar cómo percibe el agente las recompensas que recibe, veamos qué posibilidades tenemos:

Extrinsic reward

Este tipo de reward se corresponde con el valor de recompensa recibido por el entorno, es decir, por las acciones que le otorgan recompensa. Es el tipo de reward_signal es la más común y usada. Veamos sus posibles parámetros:

- **strength**: Factor por el que multiplicar la recompensa otorgada por el entorno. Los rangos típicos variarán según la señal de recompensa. Su valor suele ser 1.0
- **extrinsic**: Factor de descuento para futuras recompensas provenientes del entorno. Esto se puede considerar como qué tan lejos en el futuro el agente debería preocuparse por las posibles recompensas. En situaciones en las que el agente debería estar actuando en el presente para prepararse para las recompensas en un futuro lejano, este valor debería ser grande. En los casos en que las recompensas son más inmediatas, puede ser menor. Debe ser estrictamente menor que 1.

Curiosity intrinsic reward

Le otorga al agente una recompensa por explorar el entorno. Puede ser útil para agentes que están en un entorno muy grande con pocas recompensas.

Veamos sus parámetros:

- **strength**: Magnitud de la recompensa de curiosidad generada. Esto debe escalarse para garantizar que sea lo suficientemente grande como para no verse abrumado por señales de recompensa extrínsecas en el entorno. Asimismo, no debe ser demasiado grande para abrumar la señal de recompensa extrínseca. Rango estándar: `0.001` - `0.1`
- **gamma**: Factor de descuento para futuras recompensas.Rango estándar: 0.8 - 0.995
- **learning_rate**: Tasa de aprendizaje utilizada para actualizar el módulo de curiosidad intrínseca. Por lo general, esto debería reducirse si el entrenamiento es inestable y la pérdida de curiosidad es inestable. Rango estándar: `1e-5` - `1e-3`

GAIL intrinsic reward

Para habilitar GAIL (suponiendo que haya grabado demostraciones), se han de proporcionar estos ajustes:

- **strength**: Factor por el cual multiplicar la recompensa bruta. Tener en cuenta que al usar GAIL con una señal extrínseca, este valor debe establecerse más bajo si sus demostraciones no son óptimas (por ejemplo, de un humano), de modo que un agente capacitado se centre en recibir recompensas extrínsecas en lugar de copiar exactamente las demostraciones. Se debe mantener la fuerza por debajo de 0,1 en esos casos. Rango estándar: 0.01 - 1.0
- **gamma**: Factor de descuento para recompensas futuras. Rango estándar: 0,8 - 0,9.
- **demo_path**: (Obligatorio, no predeterminado) La ruta al archivo .demo o directorio de archivos .demo donde están grabadas las demostraciones.
- **network_settings**: Las especificaciones de red para el GAIL discriminator. El valor de hidden_units debe ser lo suficientemente pequeño para alentar al discriminador a comprimir la observación original, pero tampoco demasiado pequeño para evitar que aprenda a diferenciar entre el comportamiento demostrado y el real. Aumentar drásticamente este tamaño también afectará negativamente los tiempos de entrenamiento. Rango estándar: 64 - 256
- **learning_rate**: Tasa de aprendizaje utilizada para actualizar el discriminador.

Normalmente, esto debería reducirse si el entrenamiento es inestable y la pérdida de GAIL es inestable. Rango estándar: 1e-5 - 1e-3

- **use_actions**: Determina si el discriminador debe discriminar basándose tanto en observaciones como en acciones, o solo en observaciones. Establecer en True si desea que el agente imite las acciones de las demostraciones, y en False si prefiere que el agente visite los mismos estados que en las demostraciones pero posiblemente con acciones diferentes. Al establecer en False es más probable que sea estable, especialmente con demostraciones imperfectas, pero puede aprender más lentamente.
- **use_vail**: Habilita un cuello de botella variacional dentro del discriminador GAIL. Esto obliga al discriminador a aprender una representación más general y reduce su tendencia a ser "demasiado bueno" discriminando, lo que hace que el aprendizaje sea más estable. Sin embargo, aumenta el tiempo de entrenamiento. Se puede habilitar esto si nota que el aprendizaje por imitación es inestable o no puede aprender la tarea en cuestión.

Con esto concluimos la parte de reward_signals del fichero de configuración, pero aún quedan ciertos parámetros que comentar que no están incluídos en ningún bloque de los antes mencionados sino que son independientes:

- **summary_freq**: Número de experiencias que deben recopilarse antes de generar y mostrar estadísticas del entrenamiento. Esto determina la granularidad de los gráficos en Tensorboard.
- **time_horizon**: Cuántos pasos de experiencia recopilar por agente antes de agregarlo al búfer de experiencia. Cuando se alcanza este límite antes del final de un episodio, se usa una estimación de valor para predecir la recompensa general esperada del estado actual del agente. Como tal, este parámetro compensa entre una estimación menos sesgada, pero con mayor varianza (long time horizon) y una estimación más sesgada, pero menos variada (short time horizon). En los casos en los que hay recompensas frecuentes dentro de un episodio, o los episodios son prohibitivamente grandes, un número más pequeño puede ser más ideal. Este número debe ser lo suficientemente grande para capturar todo el comportamiento importante dentro de una secuencia de acciones de un agente. Rango estándar: 32 - 2048
- **max_steps**: Número de pasos del entrenamiento. Una vez se supera este número de pasos el entrenamiento finaliza.
- **keep_checkpoints**: El número máximo de puntos de control del modelo para conservar. Los puntos de control se guardan después de la cantidad de pasos especificados por la opción checkpoint_interval. Una vez que se alcanza el número máximo de puntos de control, el punto de control más antiguo se elimina al guardar

un nuevo punto de control. Su valor por defecto es 5.
- **checkpoint_interval**: El número de experiencias recopiladas entre cada punto de control por el entrenador. Se guarda un máximo de puntos de control keep_checkpoints antes de que se eliminen los antiguos. Cada punto de control guarda los archivos .onnx en el directorio results/.
- **init_path**: Inicializa el modelo desde un modelo ya entrenado que se encuentra en la ruta especificada en este parámetro.
- **threaded**: Permite que los entornos avancen mientras actualiza el modelo. Esto podría resultar en una aceleración del entrenamiento. Su valor es booleano (true o false).

2.3 ML Agents en Unity

Durante este apartado estaremos viendo la herramienta desde dentro de Unity, hablando de todas las funcionalidades que nos ofrece y cómo funcionan, además de las funciones de la librería para programar el comportamiento del agente.

Empecemos comentando cada uno de los componentes asociados al agente que lo ayudan a recabar observaciones o a tunear el comportamiento y algunos de sus parámetros, algunos de ellos ya las hemos mencionado, como el CameraSensor, RayPerceptionSensor…

Decision Requester

Este componente no se usa para recabar observaciones, sino para indicarle al agente cada cuántos pasos ha de tomar una decisión, el número por defecto es 5 pasos académicos por decisión. Es un componente obligatorio para cualquier tipo de agente en ML Agents puesto que es el componente ligado a las decisiones que toma el agente.

Behavior Parameters

Todo Agente debe tener un comportamiento. El comportamiento determina cómo un Agente toma decisiones. Veamos la forma de este componente y comentemos sus campos, aunque entraremos en detalles de cómo asignar su valor en el siguiente capítulo:

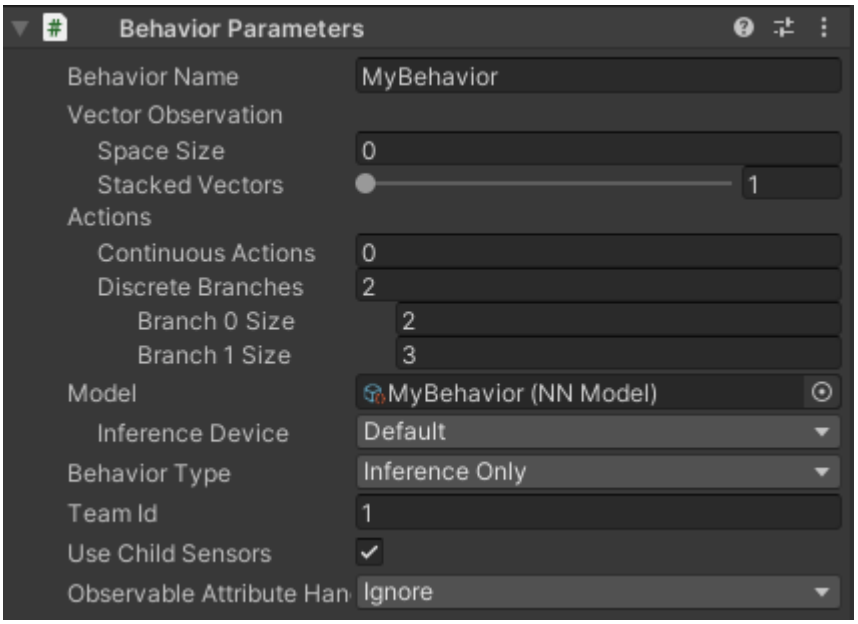

Figura 2.3.1: Behavior Parameters de un agente en el inspector

- **Vector Observation Space Size:** Antes de tomar una decisión, un agente recoge sus observaciones sobre su estado en el entorno. El vector de observación es un vector de números en punto flotante que contienen información relevante para que el agente tome decisiones. El número asignado a este campo (tamaño del vector) depende del número de observaciones que se recojan.
- **Actions**:Un agente recibe instrucciones en forma de acciones. ML-Agents clasifica las acciones en dos tipos: continuas y discretas.
- **Model**: Modelo entrenado que usará el agente en caso de querer poner en funcionamiento.
- **Behavior Type**: Este campo define el tipo de comportamiento del agente, para entrenar debe estar en *Heuristic Only*, en caso de querer usar el modelo entrenado debe estar en *Inference Only*.
- **Team Id**: Define el ID del equipo del agente en caso de usar el módulo Self-Play para competición entre agentes.

Demonstration Recorder

Este es el componente que tenemos que usar en caso de querer hacer Imitation Learning, donde el agente aprende a partir de una demo que grabamos y le pasamos como parámetro. Veámoslo:

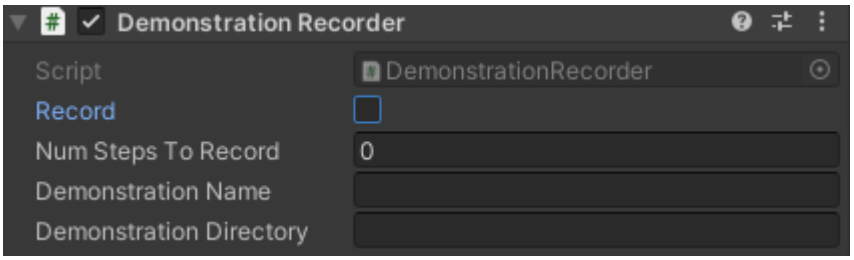

Figura 2.3.2: Demonstration Recorder

- **Record**: Marcar esta casilla en caso de querer hacer una demo para el entrenamiento, en caso de que así sea le damos al Play y jugamos como esperamos que el agente se comporte.
- **Nums Steps To Record**: Número de pasos a grabar, mantener en 0 en caso de querer hacerlo de forma indefinida.
- **Demonstration Name**: ID de la demo.
- **Demonstration Directory**: Directorio donde se guarda la demo grabada.

Ray Perception Sensor 3D

Este componente es uno de los que nos permite recabar observaciones del entorno como la posición de los GameObjects usando rayos que hacen colisión con estos. Si estos objetos tienen un tag reconocible por el Ray Perception, el agente interactuará de una forma u otra con el mismo.

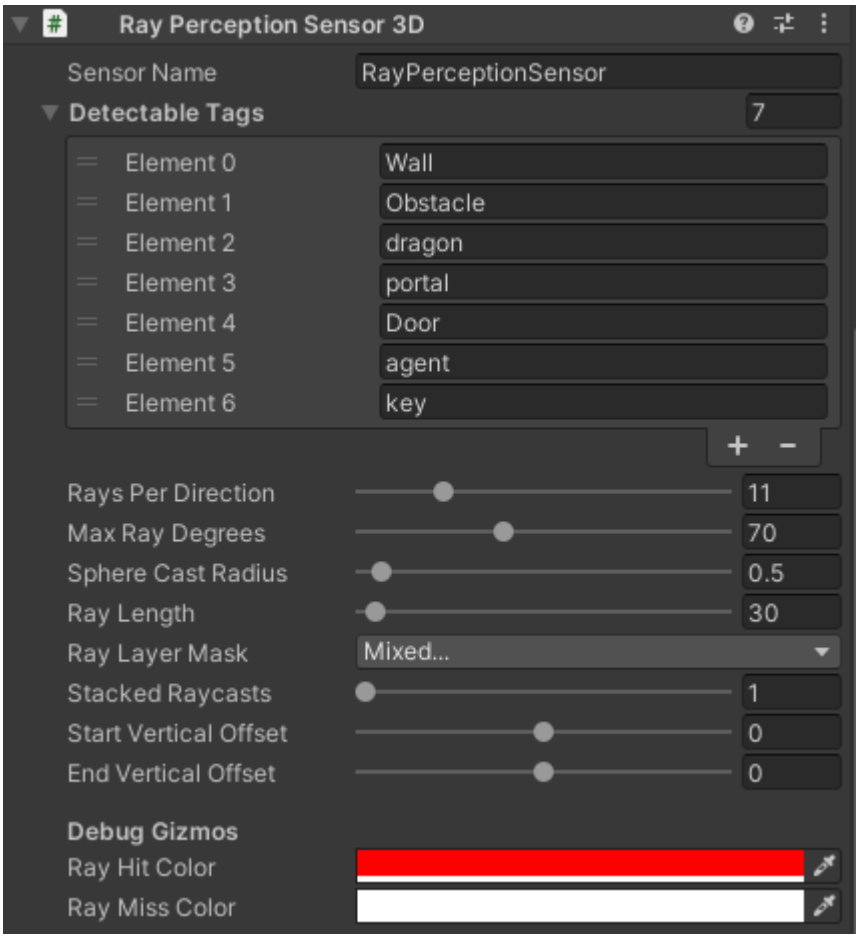

Figura 2.3.4: Componente RayPerceptionSensor3D

- **Detectable Tags**: Aquí se registran los tags de los objetos que queremos que el agente reconozca para interactuar con ellos.
- **Start Vertical Offset**: Punto desde el que salen los rayos del agente (0 indica el centro).
- **End Vertical Offset**: Punto en el que acaban los rayos en comparación al agente (0 indica el centro).

Si el valor es el mismo para estos 2 últimos parámetros, querrá decir que los rayos van en línea recta. Por tanto, si se quiere conseguir inclinación, habrá que jugar con el End Vertical Offset para hacer que los rayos obtengan inclinación.

Camera Sensor

Este componente permite al agente tener una visión del entorno mediante el uso de una

cámara, por lo tanto, en cada paso el agente estará haciendo un aprendizaje de los elementos del entorno para poder interactuar con ellos.

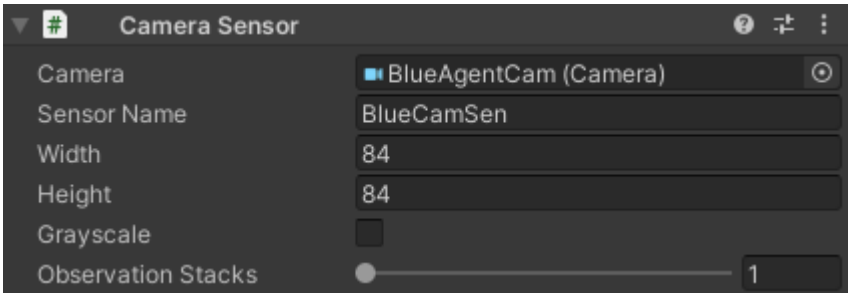

Figura 2.3.5: Camera Sensor

- **Camera**: Este campo indica el objeto cámara, el cuál debe ser un objeto hijo del objeto Agente.
- **Sensor Name**: ID con el que se identifica al sensor
- **Width:** El ancho de píxeles de la cámara.
- **Height**: Altura de píxeles de la cámara.

Estos 2 últimos campos determinan la resolución de la cámara con la que verá el agente.

- **Grayscale**: Marcar la casilla si se quiere que la visión de la cámara sea en escala de grises en vez de en color, cosa que ayuda en la velocidad de entrenamiento pero, a veces, no en la calidad.

A continuación hablemos de las diferentes funciones que vienen incluídas en la librería ML Agents de Unity, las cuales nos permiten programar el funcionamiento del agente. Veamos cada una de estas funciones:

void Initialize()

Se asemeja a la función Start() de Unity, y se utiliza para inicializar ciertos valores del agente, tales como el objeto Rigidbody, la posición inicial…

void OnEpisodeBegin()

Es una función que se llama al inicio del episodio del agente y de la simulación. Un episodio es un ciclo completo de interacción entre el agente y el entorno. Durante un episodio, el agente realiza una serie de acciones en respuesta a las observaciones que recibe del entorno, y el entorno responde con nuevas observaciones y una recompensa que indica qué tan bien está desempeñando el agente.

Por tanto, se suele usar esta función para reiniciar ciertos valores de cara al siguiente episodio, como la posición del agente o de ciertos objetos, alguna variable que nos ayude en la lógica de una función…

void EndEpisode()

Se utiliza para finalizar un episodio.

void AddReward()

Añade una recompensa al agente, puede ser positiva o negativa. La recompensa también se puede añadir para grupos de agentes (o equipos), en cuyo caso se usa la función **AddGroudReward()**.

void CollectObservations(VectorSensor sensor)

Llamado en cada paso que el Agente solicita una decisión. Esta es una forma posible de recopilar las observaciones del entorno del Agente además de las antes mencionadas (Camera Sensor, RayPerceptionSensor…).

```
public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(canShoot);
    sensor.AddObservation(transform.forward);
    sensor.AddObservation(killingSpree);
}
```

Figura 2.3.6: CollectObservations

void OnActionReceived()

Llamado cada vez que el Agente recibe una acción a realizar. Recibe la acción elegida por el Agente. También es común asignar una recompensa en este método. Veamos un ejemplo:

```
public override void OnActionReceived(ActionBuffers actions)
{
    movementAction = (int)actions.DiscreteActions[0];
    rotationAction = (int)actions.DiscreteActions[1];

    // Move the agent based on the input values
    switch (movementAction)
    {
        case 0:
            transform.Translate(Vector3.forward * moveSpeed * Time.deltaTime);
            break;
        case 1:
            transform.Translate(-Vector3.forward * moveSpeed * Time.deltaTime);
            break;
        case 2:
            transform.Translate(Vector3.right * moveSpeed * Time.deltaTime);
            break;
        case 3:
            transform.Translate(-Vector3.right * moveSpeed * Time.deltaTime);
            break;
        default:
            break;
    }

    switch (rotationAction)
    {
        case 0:
            transform.Rotate(Vector3.up, -rotationSpeed * Time.deltaTime);
            break;
        case 1:
            transform.Rotate(Vector3.up, rotationSpeed * Time.deltaTime);
            break;
        default:
            break;
    }


    if (actions.ContinuousActions[0] > 0.5f)
    {
        Shoot();
    }

    AddReward(-1 / MaxStep);
}
```

Figura 2.3.7: OnActionReceived

void Heuristic()

Transforma la entrada del teclado en acciones para que podamos controlar al agente de forma manual y así comprobar su funcionamiento antes de empezar a entrenarlo.

Capítulo 3 Experimentos realizados

Durante este capítulo hablaremos de los diferentes juegos realizados; comentaremos la lógica detrás de sus observaciones del entorno, sus ficheros de configuración, las acciones que toma en agente en sus decisiones y veremos el *tensorboard* para ver los detalles del entrenamiento.

3.1 BallFinder

BallFinder fué el primer juego realizado como toma de contacto con la herramienta. Es un juego simple, pero que sirve para introducir y explicar el funcionamiento de ML-Agents.

El juego consiste en un agente en un entorno pequeño delimitado con muros en los que tiene que encontrar una de las 2 bolas. Las colisiones con muros matan al agente.

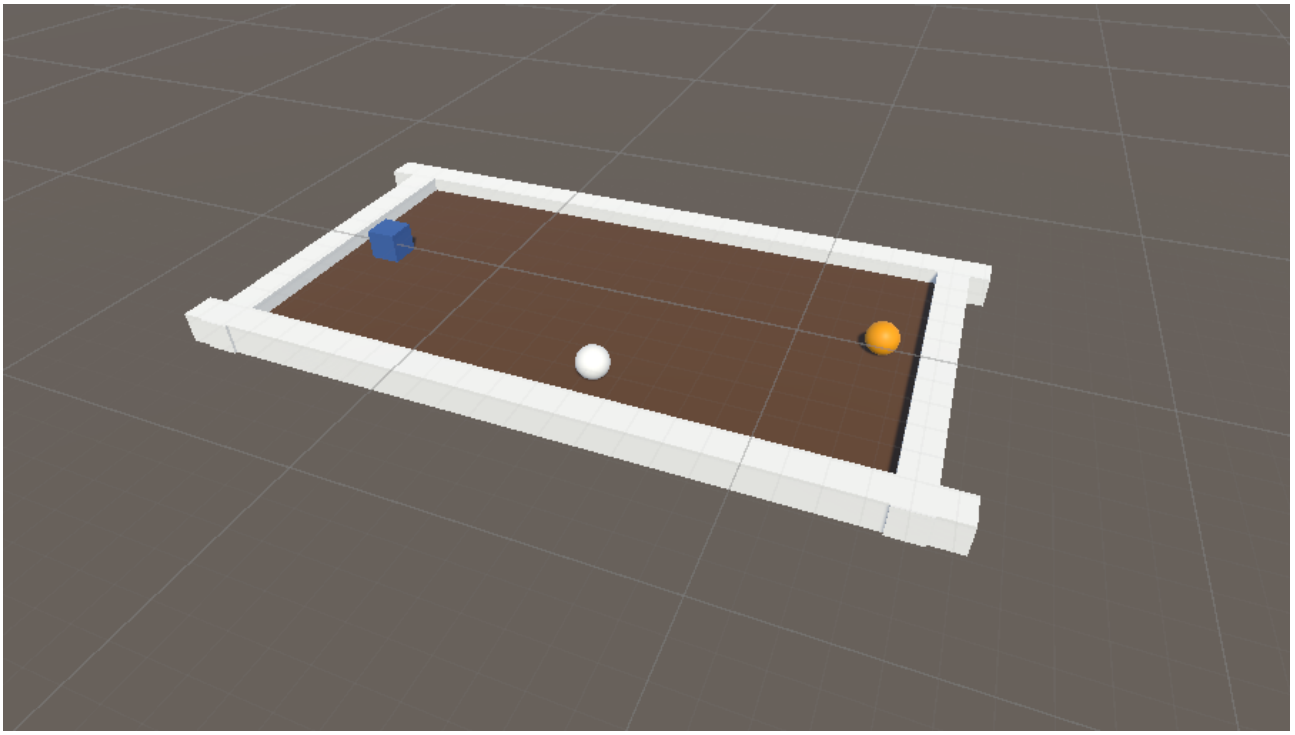

Figura 3.1.1: BallFinder

Acciones y observaciones del agente

Las observaciones para este ejemplo son bastante obvias. El agente necesita saber su posición actual y la posición de ambas bolas, por lo que el espacio de observaciones es 9 (3 inputs con posiciones *xyz* cada uno). La posición de los muros no es necesario saberla puesto que el agente conoce su posición actual por lo que aprende los límites del entorno mediante ensayo y error.

```
public override void CollectObservations(VectorSensor sensor) {
    sensor.AddObservation(transform.localPosition);
    sensor.AddObservation(targetTransform.localPosition);
    sensor.AddObservation(target2Transform.localPosition);
}
```

Figura 3.1.2: CollectObservations BallFinder

En cuanto a las acciones tomadas por el agente lo que queremos es que este se mueva en el plano *xz* ya que no queremos que tenga ninguna acción de salto.

```
public override void OnActionReceived(ActionBuffers actions) {
    // 2 posibles acciones, que se mueva en el eje X o en el Z
    float moveX = actions.ContinuousActions[0];
    float moveZ = actions.ContinuousActions[1];

    float moveSpeed = 2f;
    transform.localPosition += new Vector3(moveX, 0, moveZ) * Time.deltaTime * moveSpeed;
}
```

Figura 3.1.3: OnActionReceived BallFinder

Fases del entrenamiento

El entrenamiento ha sido dividido en varias partes para que el agente se adaptara mejor a los cambios en la política de entrenamiento. Veamos cada una de estas fases:

Una bola - Imitation Learning

Para el primer escenario de entrenamiento decidí usar una única bola para que el agente interiorizara que al hacer contacto con la bola recibía recompensa y cuando lo hacía con los muros la perdía. Al ser un caso muy sencillo vi la posibilidad de hacer uso de una de las tecnologías que quería utilizar, Imitation Learning. Grabé una demo como he comentado en el capítulo anterior y se la pasé al fichero de configuración (línea 23 de la imágen).

```yaml
behaviors:
  MoveAgent:
    trainer_type: ppo
    hyperparameters:
      batch_size: 256
      buffer_size: 1024
      learning_rate: 3.0e-4
      beta: 5.0e-4
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: false
      hidden_units: 128
      num_layers: 2
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
      gail:
        strength: 0.5
        demo_path: Demos/Demo2Stage1_21.demo
    behavioral_cloning:
      strength: 0.5
      demo_path: Demos/Demo2Stage1_21.demo
    max_steps: 5000000
    time_horizon: 64
    summary_freq: 20000
```

Figura 3.1.4: Config file imitation learning

Algo que puede llamar la atención del fichero de configuración es que hay 2 *reward_signals* y hay una de ellas (extrinsic) cuya *strength* es mayor que la otra (gail). Esto tiene una explicación muy lógica y es que cuando hacemos entrenamiento por imitación lo que queremos es que nuestro agente supere la demo que hemos hecho. Es por eso que las recompensas del entorno (extrinsic) tienen la máxima fuerza, mientras que las de imitación (gail) tienen una fuerza inferior para que el agente no haga exactamente las mismas acciones que en nuestra demo.

El entrenamiento de esta fase fue muy sencillo y tomó algo menos de 500 mil pasos académicos.

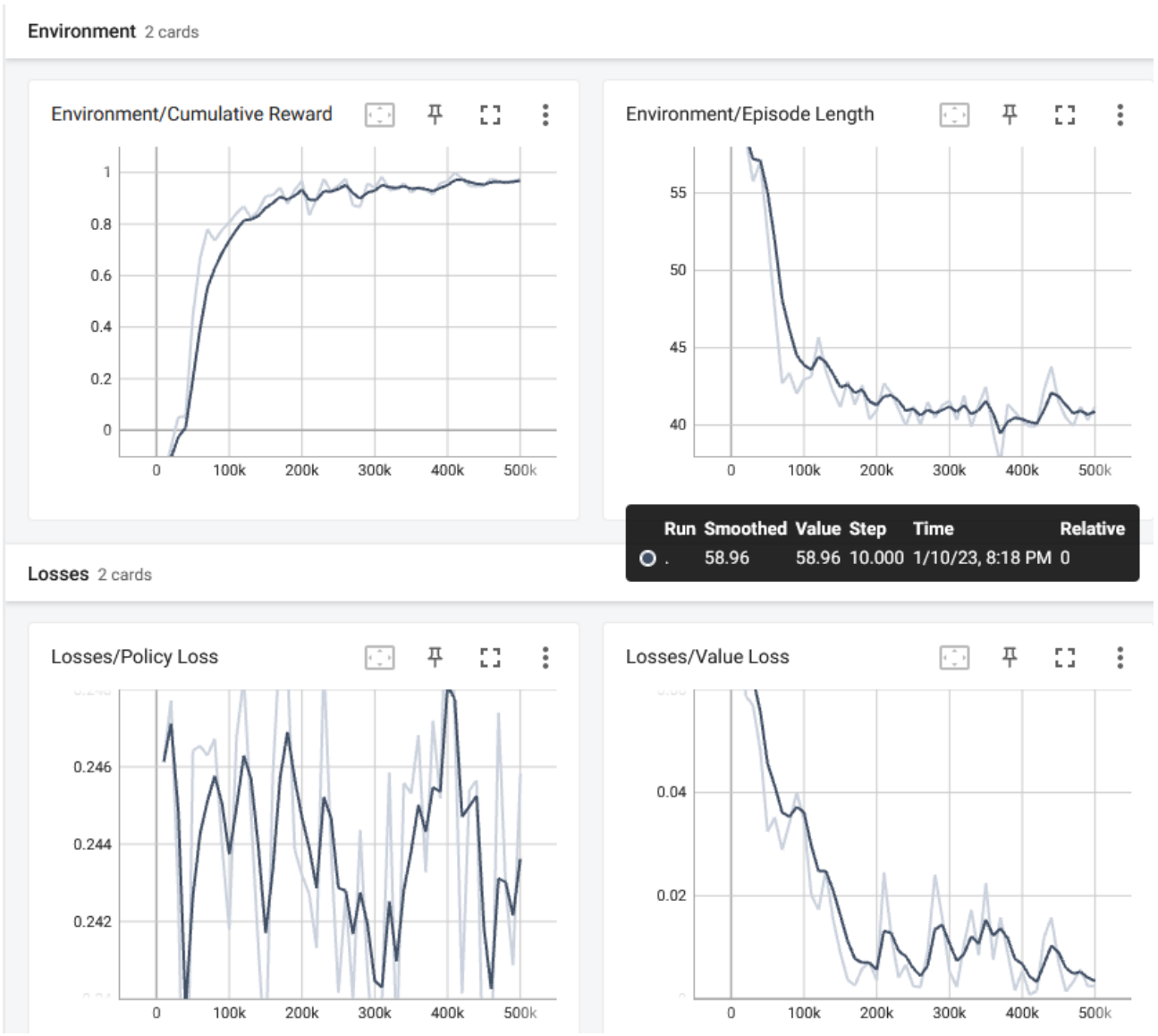

Figura 3.1.5: TensorBoard BallFinder

Este es el TensorBoard de esta fase del entrenamiento del BallFinder. Como se puede ver en el gráfico *Cumulative Reward* la recompensa del agente empieza a aumentar hasta el 1 a partir de los 50k de pasos que coincide con el número de pasos en los que se empieza a acortar la duración de los episodios (*Episode Lenght*) y del valor de pérdida (*Value Loss*). Cuando la recompensa recibida está tan cerca del 1 (máxima recompensa ganable en esta fase de entrenamiento) podemos asumir que el entrenamiento ha sido exitoso.

2 bolas - Extrinsic only

Para esta fase del entrenamiento se decidó agregar una bola más (Figura x) para comprobar cómo se comportaba el agente cuando tenía 2 posibles recompensas y una de ellas tenía un valor mayor que la otra. La bola blanca da una recompensa de +4 mientras

que la naranja de +1.

Se inicializa el entrenamiento desde el modelo generado en la fase anterior, por tanto no usé Imitation Learning para esta fase sino únicamente las recompensas del entorno. Por consiguiente usamos el mismo fichero de configuración pero quitando el *gail* y el *behavioral_cloning* (parámetros asociados al entrenamiento por imitación).

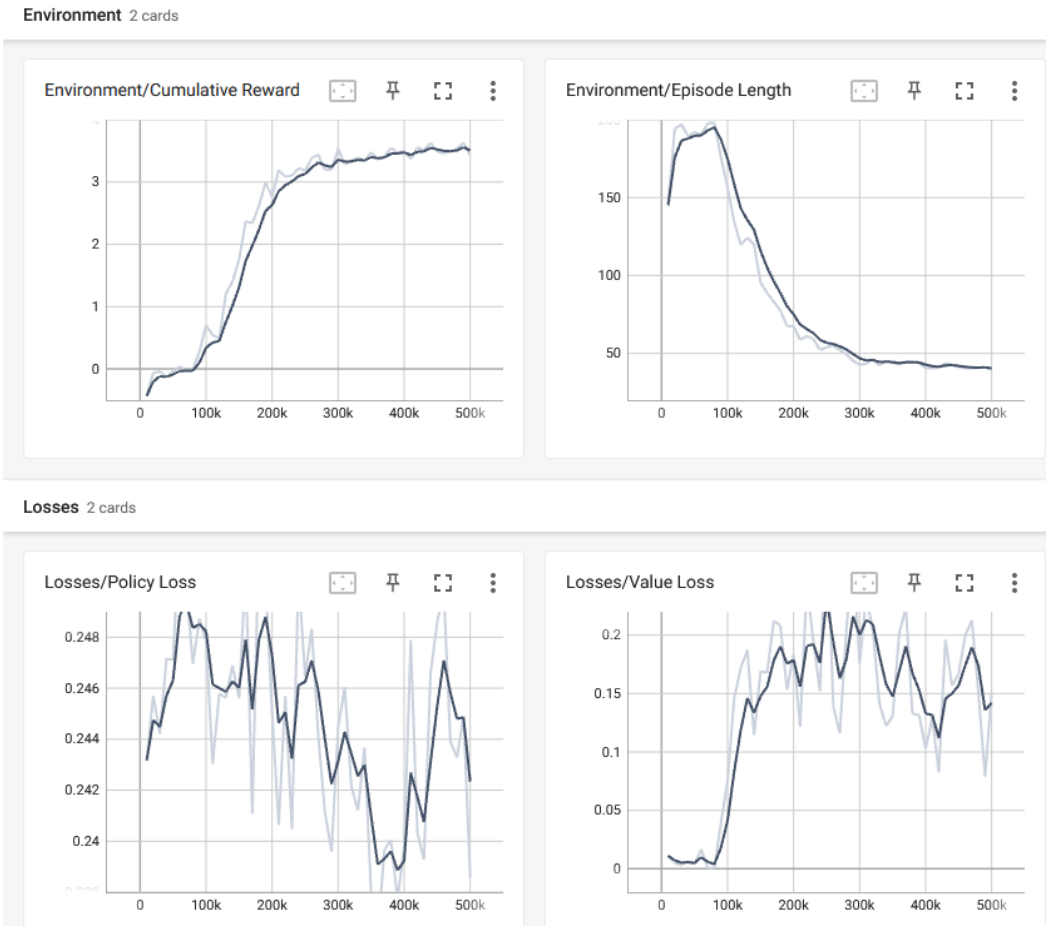

Figura 3.1.6: TensorBoard BallFinder

Para estas métricas podemos ver como el *Episode Length* empieza siendo más corto para ir aumentando hasta los 100k de pasos. Esto puede ser debido a que el modelo se comporte de forma similar al primero es sus primeros episodios por la nueva observación añadida de la segunda bola.

Podemos hacer una analogía entre las gráficas de *Cumulative Reward* y *Episode Length* ya que ambos empiezan a mejorar llegados al punto de los 100k, donde la primera empieza a aumentar sobrepasando los +3 de recompensa y esta última a decrecer hasta tener menos de 50 pasos por episodio.

Se puede observar que en la gráfica de *Value Loss* aumenta a partir del mismo punto de inflexión que hemos mencionado anteriormente. No se ha encontrado una explicación clara acerca del porqué pasa esto pero si tengo una hipótesis: el agente escoge la bola blanca (+4) la gran mayoría de veces y el único momento en el que no lo hace es cuando la bola naranja aparece justo delante de la blanca, haciéndole ganar +1 en vez de +4 y es ahí cuando sucede la pérdida.

3.2 VersusGame

VersusGame es un juego que consiste en 2 agentes compitiendo entre sí por ver quién de los 2 encuentra el mayor número de bolas verdes (comida) intentando evitar las bolas rosas (veneno).

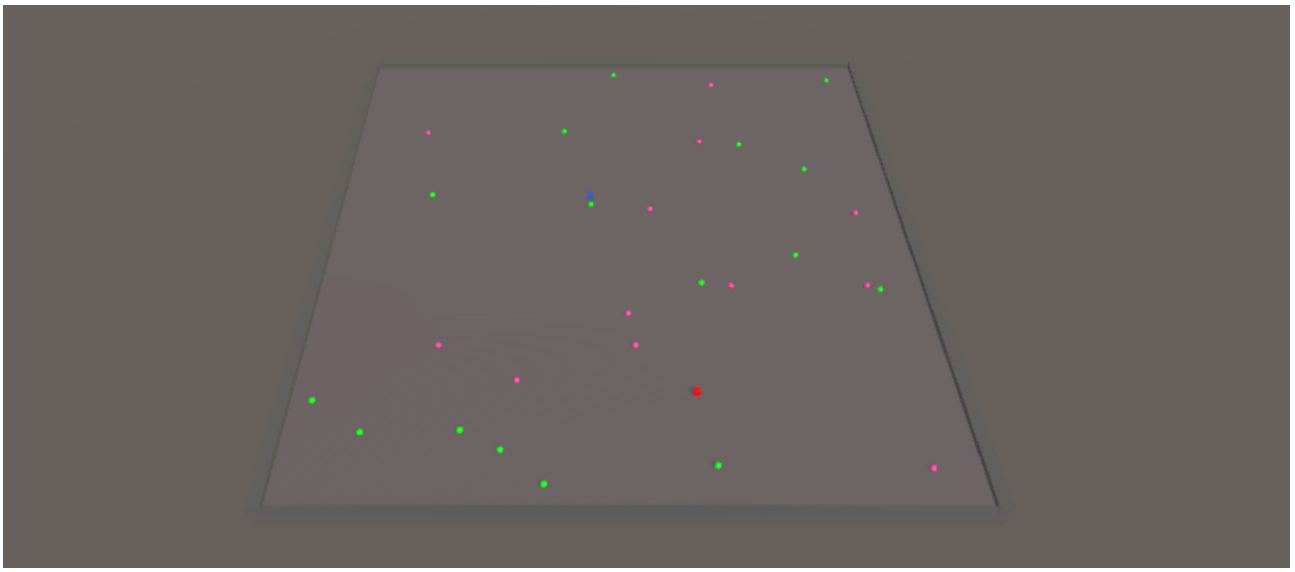

Figura 3.2.1: VersusGame

Los agentes están representados en la imágen con el color rojo y el azul.

Acciones y observaciones del agente

Cada uno de los agentes tiene un CameraSensor que recoge las observaciones del entorno. Estas cámaras que les sirven de ojos tienen una resolución de 84x84 píxeles, por lo que la calidad de imágen no es muy nítida.

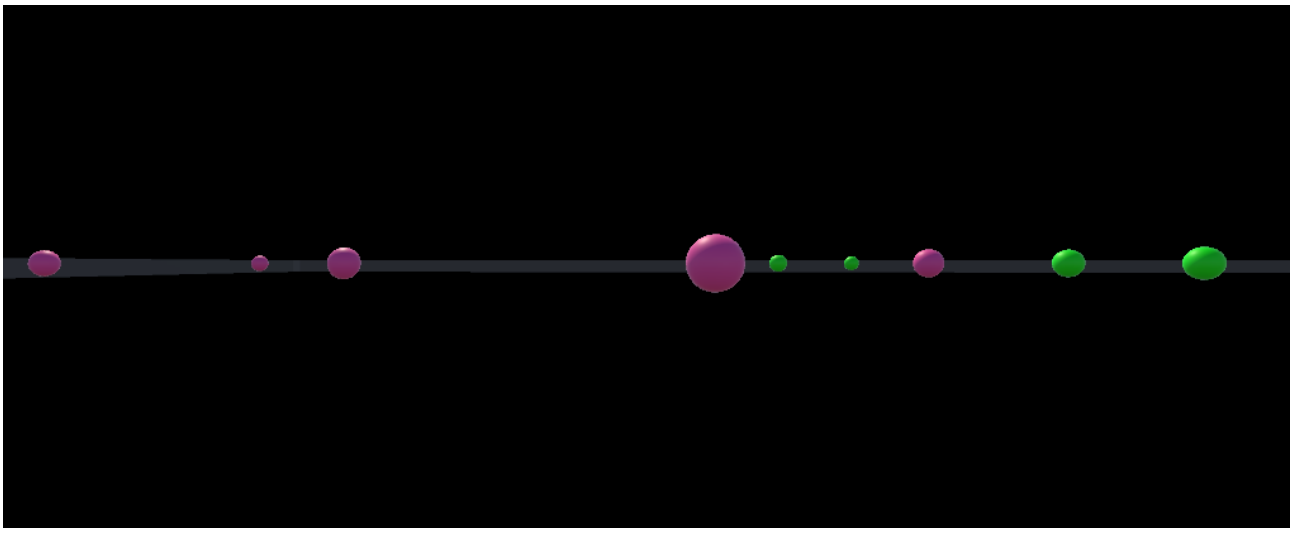

Figura 3.2.2: Cámara en HD

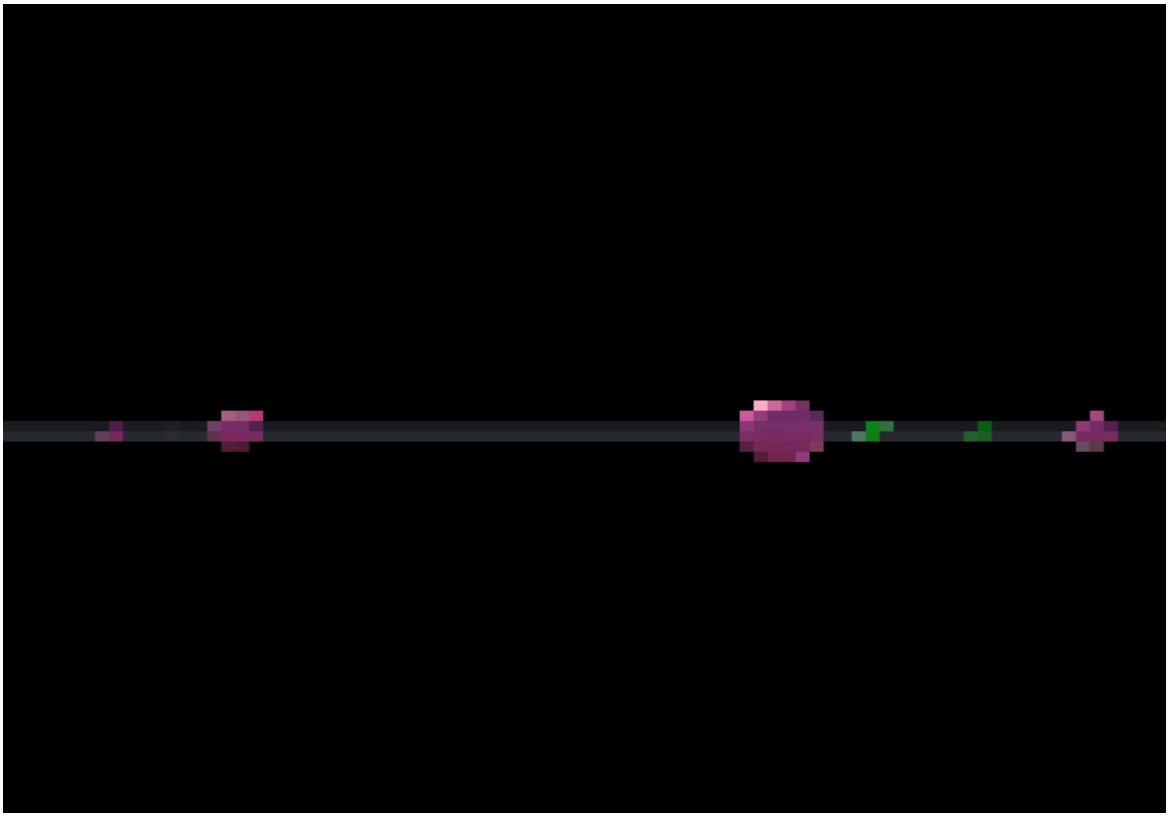

Figura 3.2.3: Cámara en 84x84

Estas 2 imágenes reflejan cómo se ve en resolución nativa vs cómo ve el agente y podemos observar que los objetos más lejanos son los más difíciles de distinguir, por tanto el agente presentará problemas para reconocer objetos que se encuentran a una cierta distancia.

En cuanto a las acciones que puede tomar el agente a la hora de tomar sus decisiones

tenemos un cambio con respecto al BallFinder, el agente puede rotar. Al tener una cámara esto es lo más conveniente ya que si este movimiento no estuviera tener un CameraSensor sería inútil.

```
public override void OnActionReceived(ActionBuffers actions)
{
    float vertical = actions.DiscreteActions[0] <= 1 ? actions.DiscreteActions[0] : -1;
    float horizontal = actions.DiscreteActions[1] <= 1 ? actions.DiscreteActions[1] : -1;
    characterController.ForwardInput = vertical;
    characterController.TurnInput = horizontal;

    AddReward(-1f / 10000f); // penalizar el n?mero de pasos que da
}
```

Figura 3.2.4: OnActionReceived

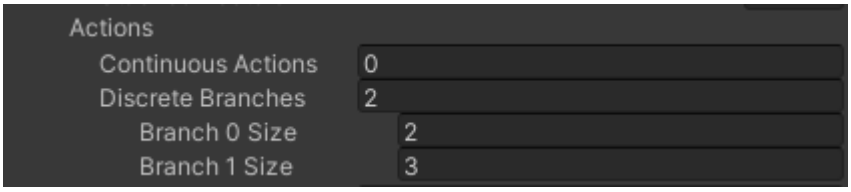

Figura 3.2.5: Actions en el inspector

Estamos usando acciones discretas para poder usar 2 ramas, una para la rotación y otra para el movimiento. La primera rama va asociada al movimiento de rotación (size=2) y la segunda para el movimiento hacia adelante, izquierda y derecha. Hay una penalización en el número de pasos que da el agente para incitarle a hacer el menor número de pasos posibles.

Fases del entrenamiento

El entrenamiento de este agente ha sido dividido en tres fases debido a que el funcionamiento no estaba siendo el esperado y se tuvo que ir cambiando ciertas cosas en el código del agente y del juego. Para este juego el fichero de configuración utilizado tiene una recompensa por curiosidad para que el agente explore el entorno, ya que su superficie es grande.

Fase 1

Durante esta fase el agente comenzó a aprender sobre cómo gestionar las observaciones recibidas por la cámara y por tanto no hubo un progreso notorio.

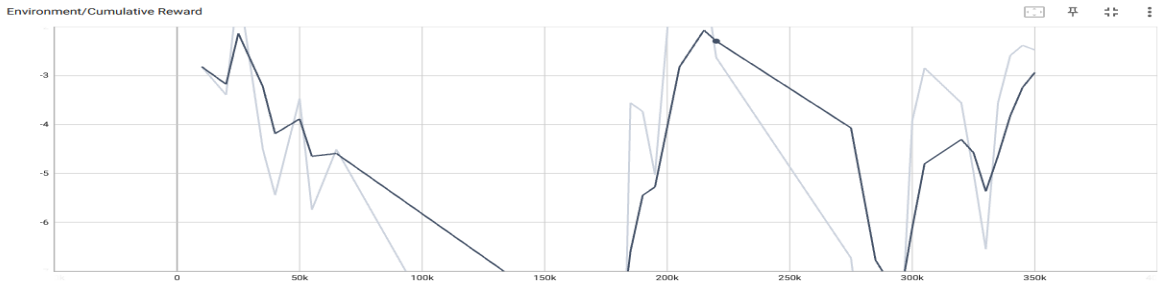

Figura 3.2.6: CumulativeReward

Fase 2

Durante esta fase se arregló un error en el que las bolas no reaparecían de forma correcta y un bug en el que la cámara no estaba viendo los objetos que tenía que tocar para cobrar la recompensa, todo ello por un error en el que los objetos no estaban en el mismo *layer* en el que observaba la cámara. Esta puede ser una de las razones por el desastroso resultado de la fase 1. Veamos el *tensorboard* para ver los resultados del entrenamiento:

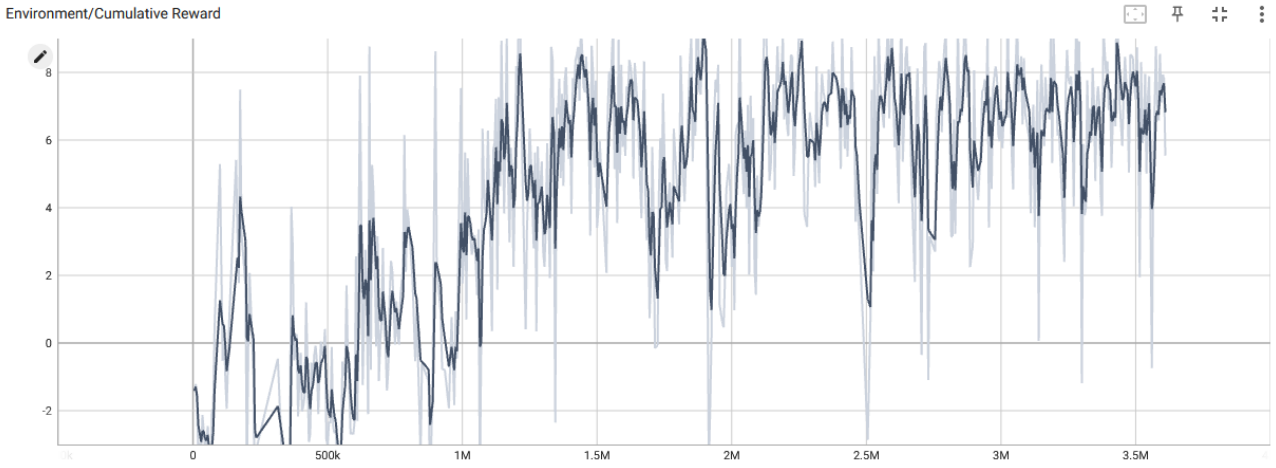

Figura 3.2.7: CumulativeReward

Como se puede ver en la gráfica el agente ha sido entrenado de manera exitosa puesto que desde el paso 1 millón hasta que finaliza no ha dejado de crecer obteniendo una recompensa positiva.

Figura 3.2.8: ELO

En esta gráfica se muestra el ELO de los agentes. El ELO es una métrica del módulo Self-Play de ML-Agents para medir el nivel de habilidad del jugador y está presente en todos los juegos multijugador competitivos de hoy en día. En la fase actual el ELO se encuentra al final del entrenamiento en 1110 (aprox) que compararemos con el ELO de futuras fases en pos de ver si los agentes han mejorado.

Fase 3

Durante esta fase lo único que se quería conseguir era seguir entrenando al agente para mejorar su funcionamiento. La *Cumulative Reward* de esta fase sigue siendo igual que la anterior puesto que el modelo fue inicializado desde el modelo de la Fase 2. Sin embargo el ELO sí que ha cambiado. Veamos el *tensorboard*:

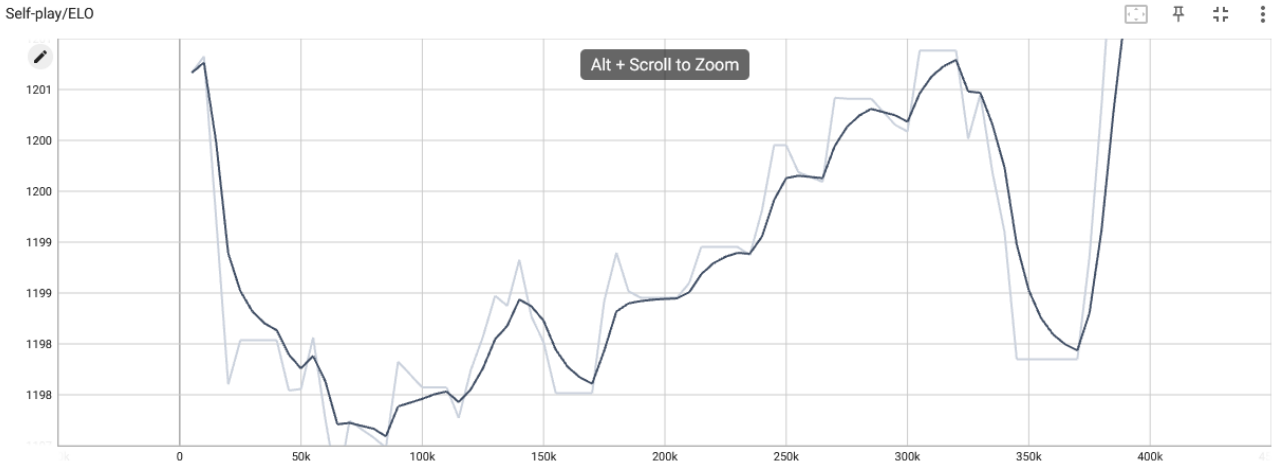

Figura 3.2.9: ELO

El ELO en esta fase ha aumentado notoriamente hasta los 1200 puntos, por lo que

podemos asumir que el agente ha mejorado su nivel de habilidad con respecto a la fase anterior.

3.3 Football 3vs3

Como su propio nombre indica, este juego consiste en un Football 3 contra 3, donde cada uno de los integrantes del equipo tiene un rol específico, portero, defensa y delantero.

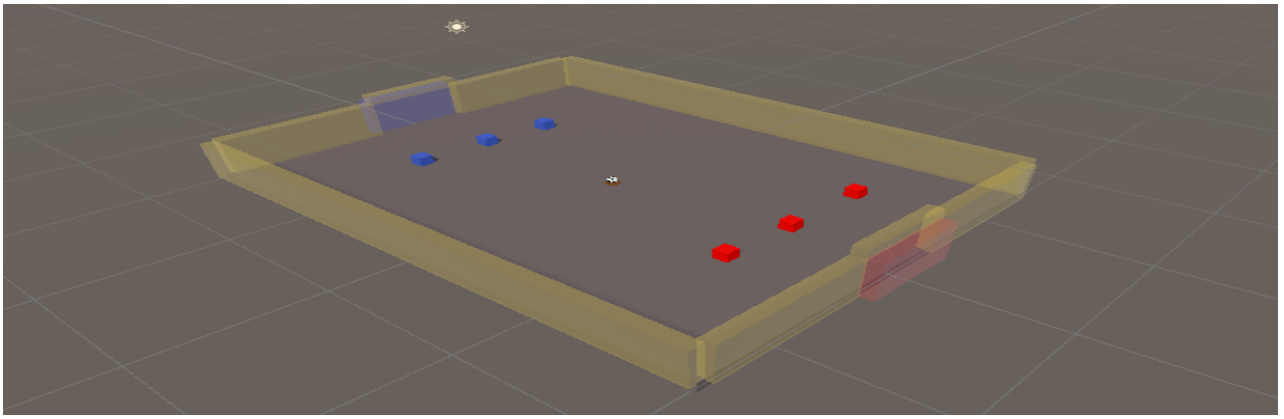

Figura 3.3.1: Football 3vs3

En este experimento se quiso hacer uso de la tecnología RayPerceptioSensor3D para captar las observaciones.

Este fue uno de los experimentos fallidos del proyecto puesto que no conseguí hacerlo funcionar de la forma esperada y, además, sentía que el modelo entrenado no funcionaba de una forma muy inteligente.

Acciones y observaciones del agente

Como ya se ha comentado, las observaciones del agente son captadas mediante un RayPerceptionSensor3D con la siguiente configuración: 11 rayos por dirección a lo largo de 120 grados

Las acciones usadas son las mismas que en el juego VersusGame, movimiento hacia adelante, izquierda y derecha y rotación.

Fases del entrenamiento

Este entrenamiento ha sido dividido en muchísimas fases, 12 para ser exactos, y se han realizado más de 100 millones de iteraciones para intentar hacer funcionar al modelo. En cada fase se arreglaron diferentes errores y se seguía entrenando al agente inicializando desde el modelo anterior. Comentaremos la primera y última fase ya que si no se extendería demasiado para un juego que no ha sido entrenado con éxito.

El fichero de configuración utilizado es el siguiente:

```yaml
behaviors:
  MyBehavior:
    trainer_type: ppo
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: constant
    network_settings:
      normalize: false
      hidden_units: 512
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 50000000
    time_horizon: 1000
    summary_freq: 10000
    self_play:
      save_steps: 50000
      team_change: 200000
      swap_steps: 2000
      window: 10
      play_against_latest_model_ratio: 0.5
      initial_elo: 1200.0
```

Figura 3.3.2: Config file Football

Fase 1

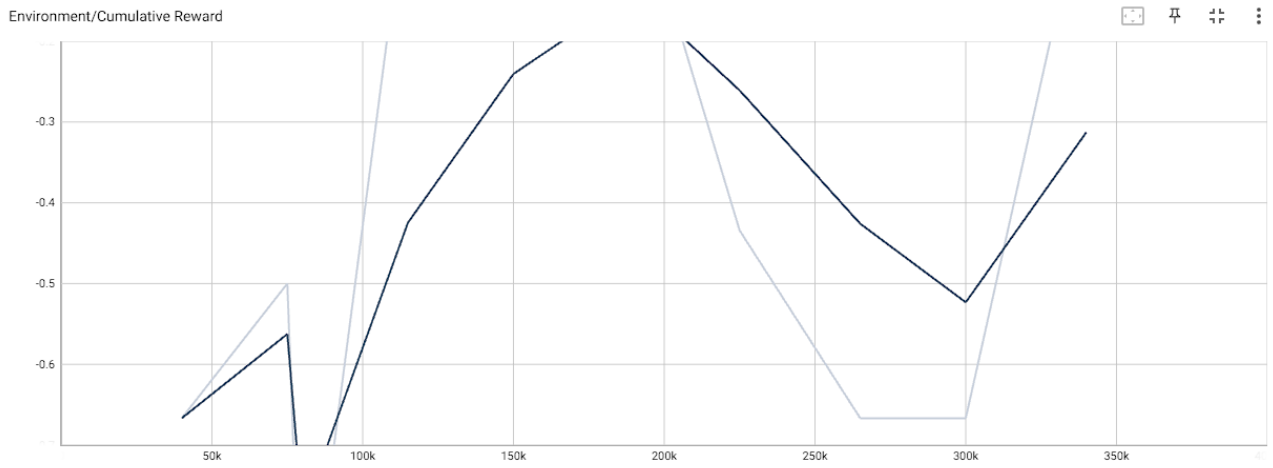

Figura 3.3.3: TensorBoard Football

Al ser la primera fase de entrenamiento el agente no obtiene aún una recompensa positiva de manera prolongada.

Fase 12

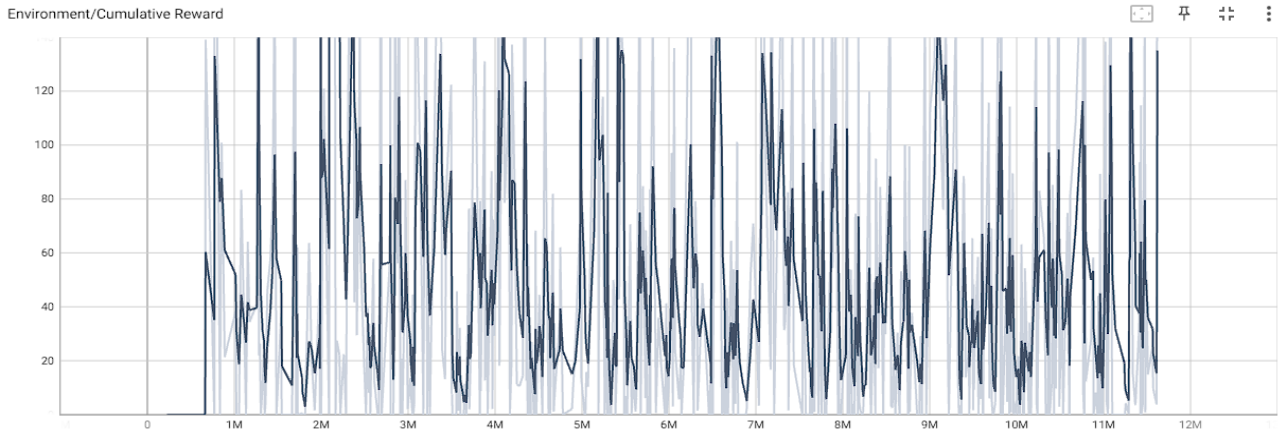

Figura 3.3.4: *TensorBoard Football*

Aquí pasa algo sorprendente, la recompensa media del agente ronda entre los 20 y 120 de recompensa, siendo la más grande de ningún experimento de los que he hecho. Con una recompensa así la pregunta es: ¿por qué es un experimento fallido?

La recompensa es muy alta porque el agente que hace de delantero mete goles (principal fuente de recompensa), pero el portero y el defensa no saben hacer su función de forma correcta. Es por esto también que la recompensa es tan alta, no hay nadie que defienda la portería.

Figura 3.3.5: TensorBoard Football

Si vemos la gráfica del ELO también podemos ver que es más alto que en el VersusGame, juego que fue entrenado de forma exitosa. Podemos concluir que no solo la recompensa conseguida es lo único que importa a la hora de que un agente esté entrenado o no, sino que el comportamiento que quieres que tenga importa aún más.

3.4 RoomEscape

RoomEscape es un juego que consta de 2 niveles, uno de ellos donde un grupo de agentes se enfrenta a un dragón que tiene que derrotar para conseguir una llave que les servirá para abrir la puerta y escapar. Uno de los agentes se sacrifica para matar al dragón y así obtener la llave y que el resto escape. una vez han obtenido la llave solo tendrán que hacer colisión con la puerta para escapar.

El segundo nivel es muy parecido, pero esta vez tienen que perseguir a un ladrón que les ha robado la llave y matarle para conseguirla y escapar.

Chocar con muros u obstáculos resta recompensa al agente.

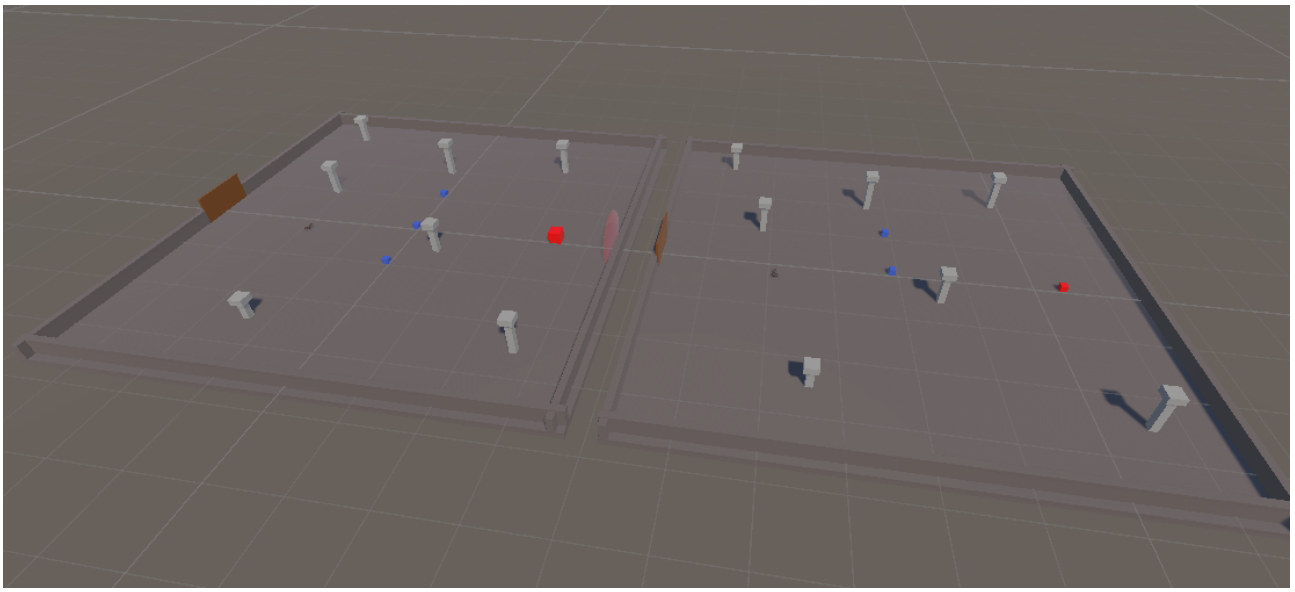

Figura 3.4.1: RoomEscape

Acciones y observaciones del agente

Las acciones del agente son las mismas que hemos comentado en los 2 últimos juegos, ya que quería que tuvieran el mismo movimiento.

Para las observaciones se ha utilizado un RayPerceptionSensor3D, al igual que en el juego de Football pero con una configuración para los rayos distinta:

Además hay que añadir una observación que es muy importante en el CollectObservations(), una para que el agente sepa si tiene o no la llave para poder abrir la puerta. Esto se va a representar mediante una variable booleana que estará en true cuando el agente tenga la llave.

```
public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(IHaveAKey);
}
```

Figura 3.4.2: Observación RoomEscape

Fases del entrenamiento

Se ha utilizado el mismo fichero de configuración para ambos niveles:

```yaml
behaviors:
  DungeonEscape:
    trainer_type: ppo
    hyperparameters:
      batch_size: 1024
      buffer_size: 10240
      learning_rate: 0.0003
      beta: 0.01
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: constant
    network_settings:
      normalize: false
      hidden_units: 256
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 50000000
    time_horizon: 64
    summary_freq: 60000
```

Figura 3.4.3: Config file RoomEscape

Empecemos hablando del nivel 1 (derrotar al dragón):

Fase 1

En esta fase el agente empieza a gestionar los estímulos de las observaciones que recibe del entorno, por tanto no se espera que la recompensa del agente sea buena, sino que empiece a aprender.

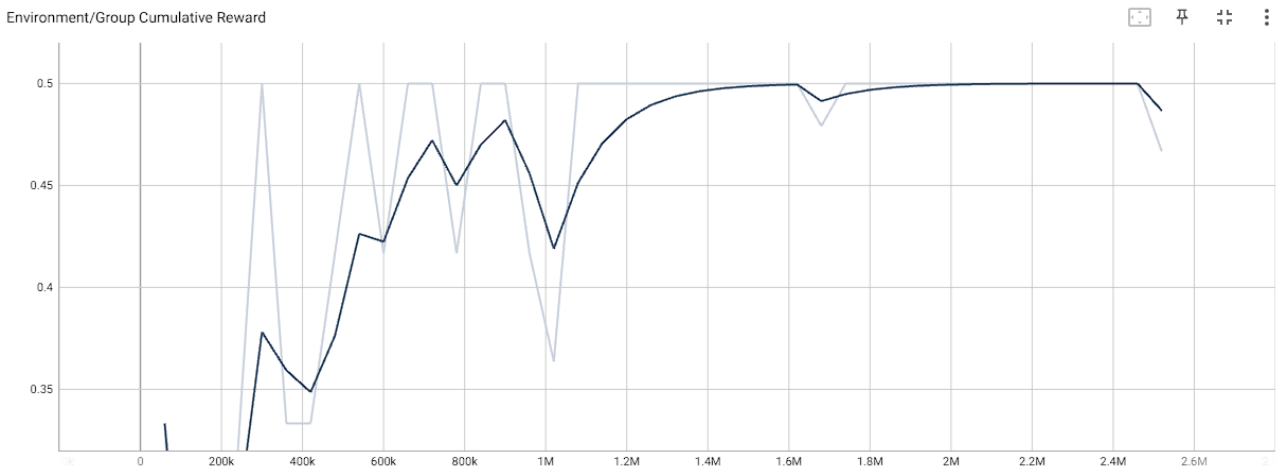

Figura 3.4.4: TensorBoard RoomEscape

La recompensa en este caso es de grupo (Group Cumulative Reward) puesto que tienen que trabajar de manera conjunta para obtenerla, todos para uno y uno para todos. Es por esto que uno de los agentes se sacrifica para conseguir la llave y salir exitosamente de la sala.

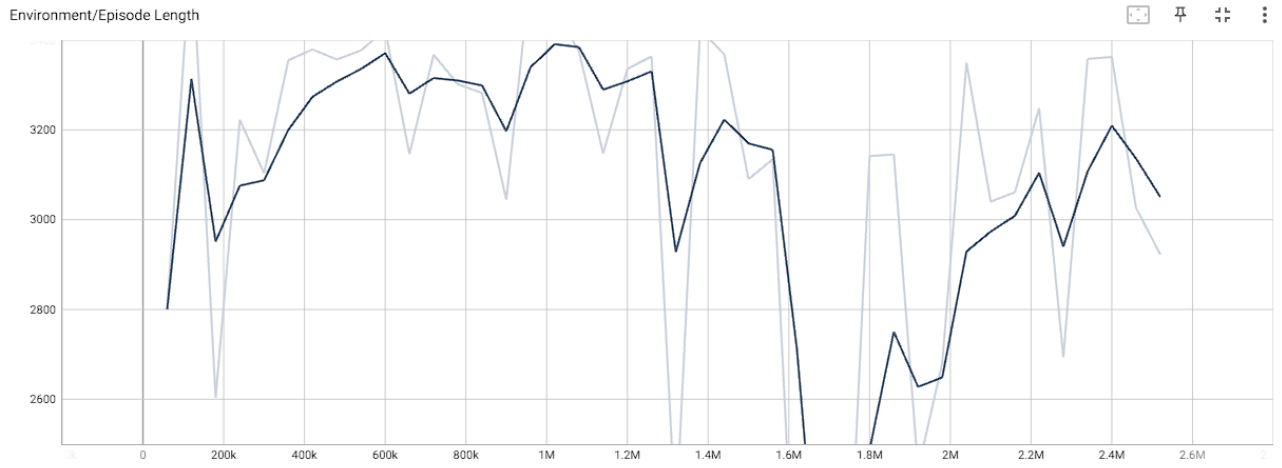

Figura 3.4.5: TensorBoard RoomEscape

Podemos ver que a partir de los 400k de pasos los agentes empiezan a asimilar el objetivo del juego y, por tanto, aumentar su recompensa, o eso podemos pensar, pero si miramos la gráfica de arriba no se ve tan así, puesto que la duración de los episodios sigue siendo muy larga. Así que veamos la fase 2.

Fase 2

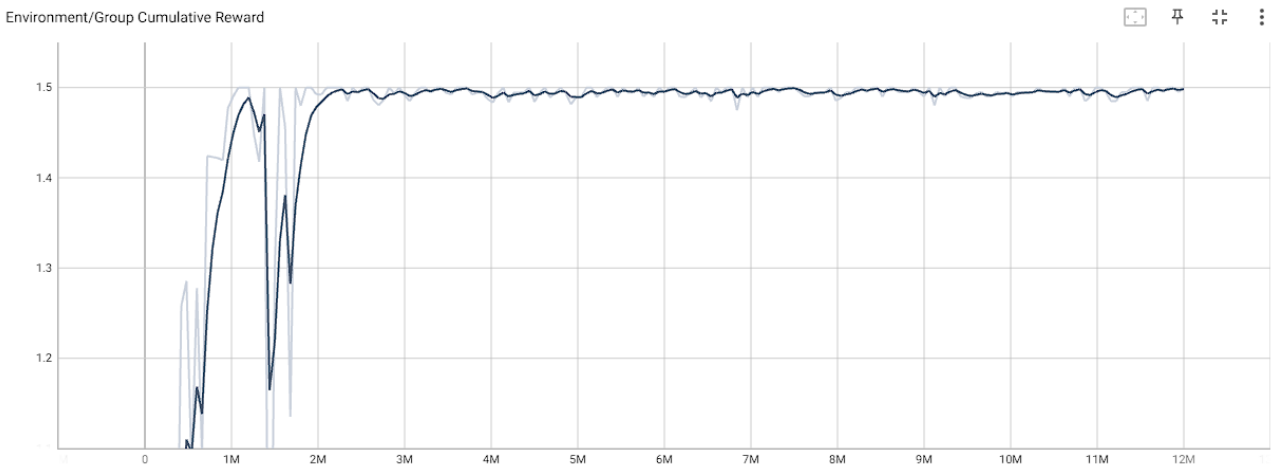

Figura 3.4.6: TensorBoard RoomEscape

Aquí ya podemos ver el progreso de los agentes y cómo han llegado a su máximo. A partir de las 2.4M de pasos el agente empieza a conseguir una recompensa de forma constante, asimilando que los muros, los obstáculos y la puerta si no tienes llave quitan recompensa. Veamos ahora el tamaño de los episodios:

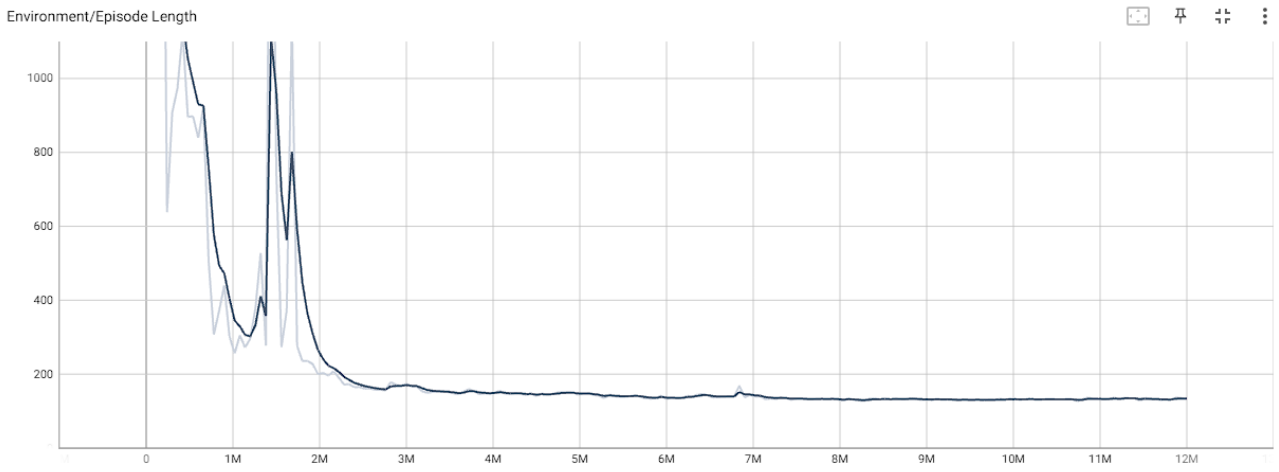

Figura 3.4.7: TensorBoard RoomEscape

La gráfica nos confirma que es a partir de los 2M de pasos que el agente se estabiliza y alcanza su punto máximo de funcionamiento.

Pasemos a hablar del nivel 2:

El entrenamiento también se ha dividido en 2 fases:

Fase 1

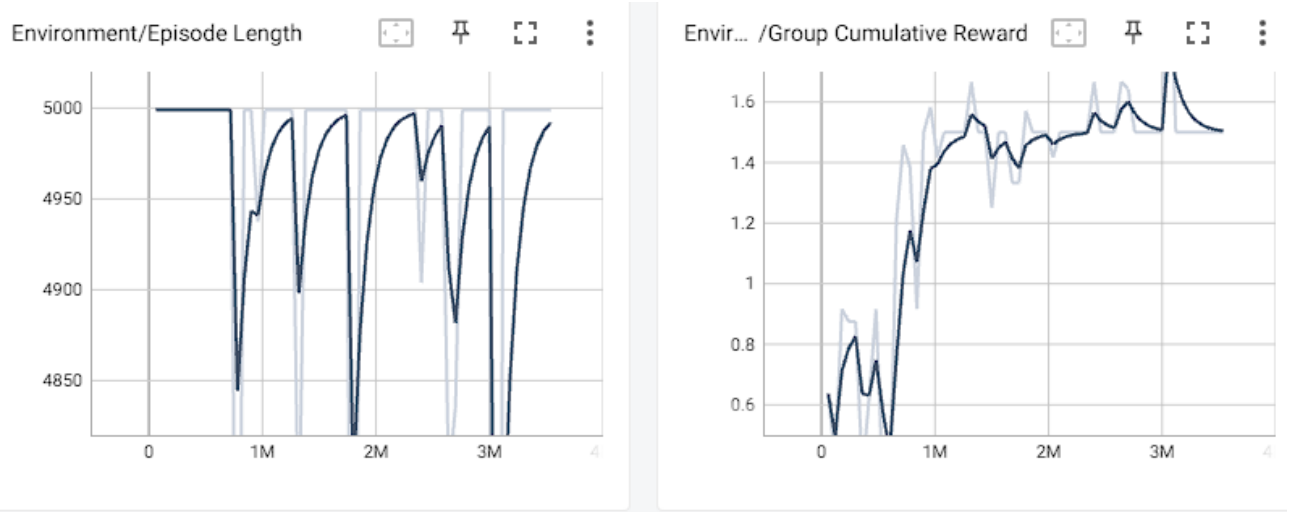

Figura 3.4.8: TensorBoard RoomEscape

En esta primera fase se logra un resultado bastante satisfactorio y es que la recompensa de grupo va escalando desde los 500k de pasos en adelante hasta alcanzar 1.6 de recompensa, lo cual no está nada mal para este juego en concreto por la cantidad de recompensa que se da al agente por cumplir su objetivo.

Aunque si vemos el tamaño de episodio vemos que aún es muy grande como para decir que lo hemos entrenado con éxito. Por lo que debemos entrenarlo una vez más para intentar disminuir esto lo máximo posible.

Fase 2

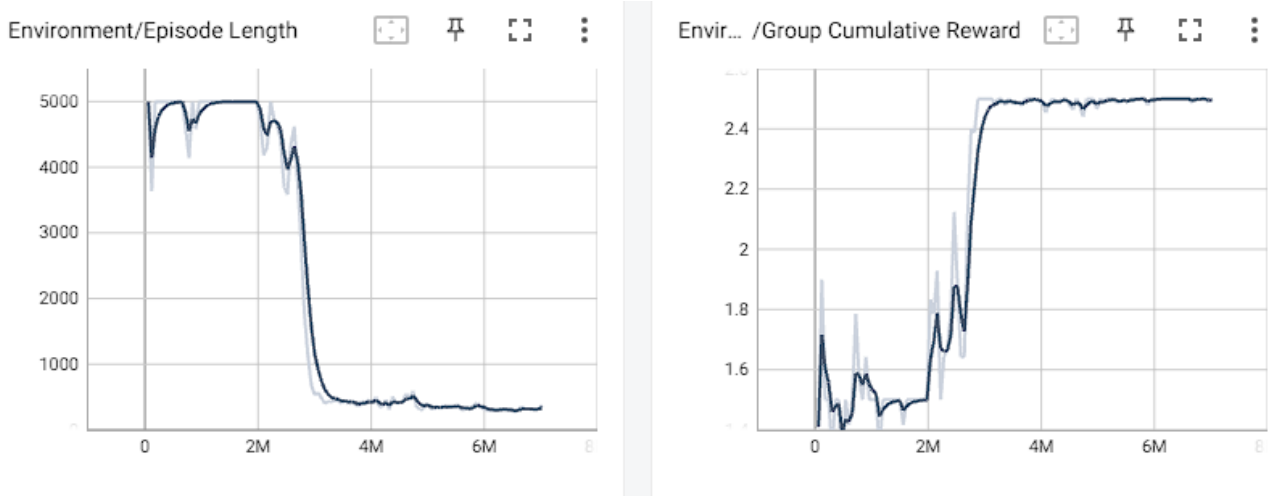

Figura 3.4.9: TensorBoard RoomEscape

En esta fase podemos ver la mejora del agente. El tamaño de episodio baja hasta menos de 500 pasos por episodio y la recompensa sube de 1.6 hasta más de 2.4. En ambas gráficas un mismo valor medio se mantiene desde los 3M de iteraciones durante varios millones más, sin cambios, por lo que asumimos que el agente alcanza su punto máximo.

3.5 AI vs Zombies

Este juego consiste en un agente que se puede mover y rotar para disparar los zombies que se le aproximan. El agente tiene un delay para disparar, por lo que tiene que combinar bien el movimiento con el disparo.

Matar zombies da una recompensa positiva, mientras que si un zombie llega a alcanzar al agente recibe una recompensa negativa.

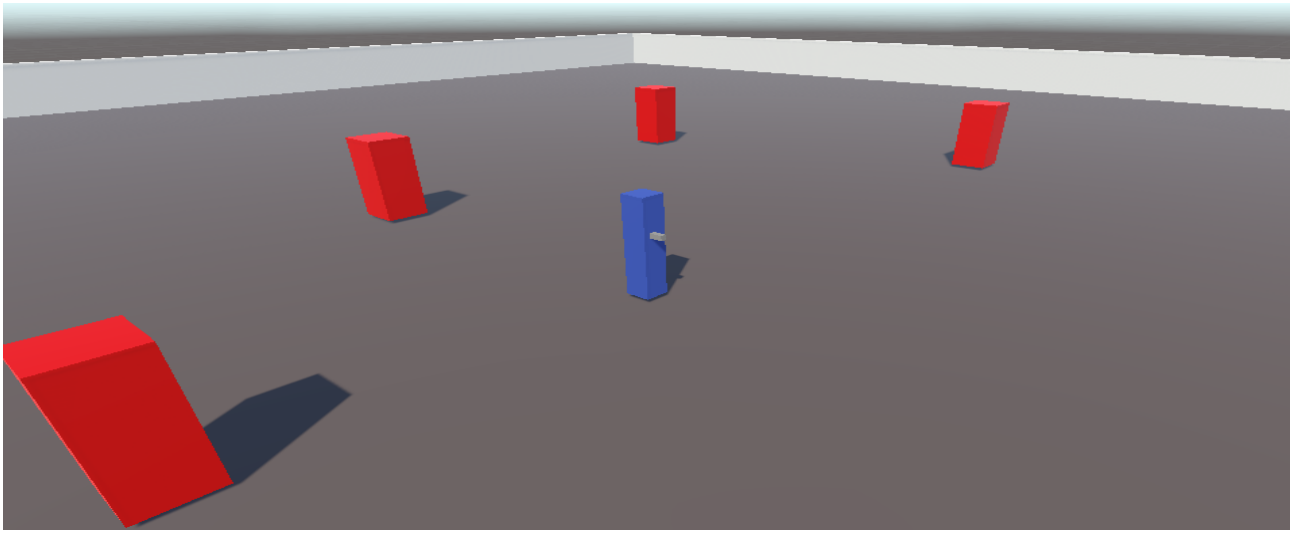

Figura 3.5.1: AIvsZombies

Acciones y observaciones del agente

Para las acciones llevadas a cabo por el agente hemos hecho lo mismo que en los juegos VersusGame o Football 3vs3, donde el agente puede rotar sobre el eje *y* y además pueda moverse hacia delante y los lados. Sin embargo, el agente tiene una acción extra, la acción de disparo:

```
if (actions.ContinuousActions[0] > 0.5f)
{
    Shoot();
}
```

Figura 3.5.2: Acción de disparo

Este condicional va dentro de la función OnActionReceived(). Como se puede ver es una acción continua, mientras que las del movimiento eran discretas, por tanto, en este agente, tendremos las acciones discretas para el movimiento del agente y las continuas para la acción de disparo:

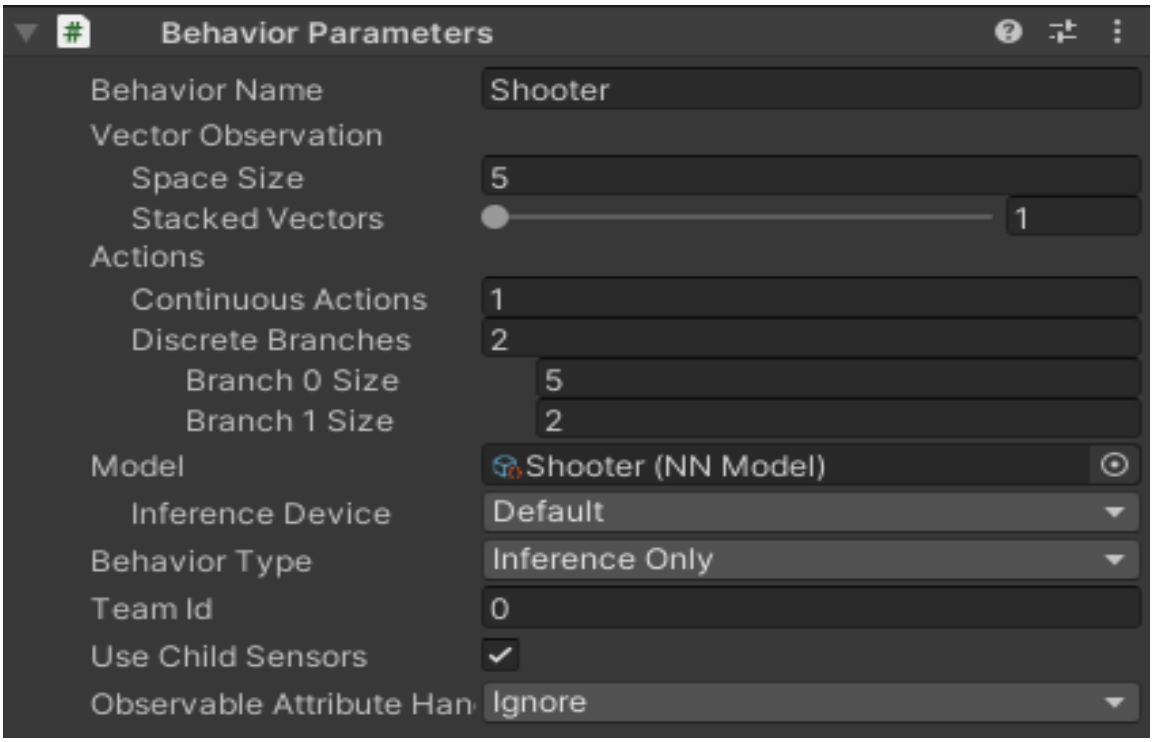

Figura 3.5.3: Behavior Parameter AIvsZombies

En cuanto a las observaciones que recibe el agente he optado por usar un RayPerceptionSensor3D puesto que es lo más conveniente para detectar los enemigos que tiene alrededor.

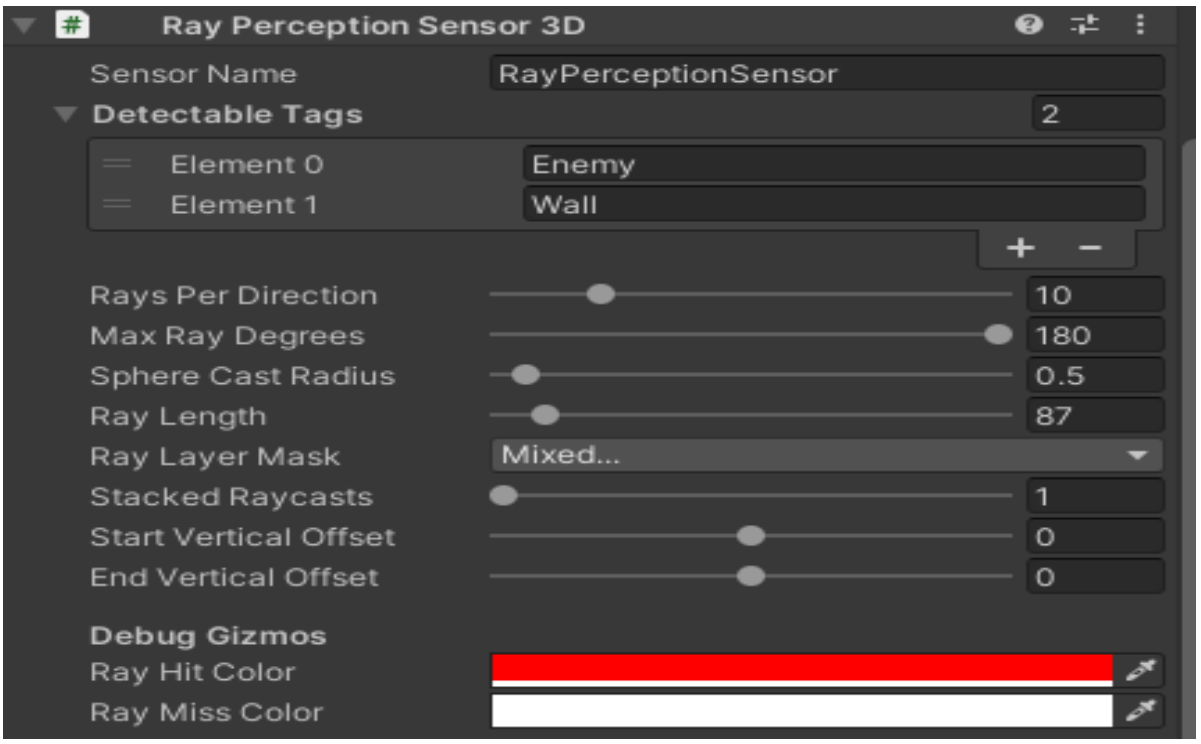

Figura 3.5.4: RayPerceptionSensor AIvsZombies

Además tenemos que tener unas observaciones más en cuenta: el lugar al que está apuntando y si puede disparar o no (recordemos el delay de disparo). Además también tiene una observación que detecta cuantos enemigos ha matado, si este número es múltiplo de 3 da una recompensa extra al agente.

```
public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(canShoot);
    sensor.AddObservation(transform.forward);
    sensor.AddObservation(killingSpree);
}
```

Figura 3.5.5: CollectObservations AIvsZombies

Fases del entrenamiento

He dividido el entrenamiento en 3 fases, la primera en la que el agente está completamente quieto delante de un enemigo también quito para asimilar que la acción de disparar a ese objeto es lo que le da la recompensa. En la segunda se habilita el movimiento de rotación y en la tercera fase el agente puede tanto rotar como moverse o disparar. Veamos las fases de entrenamiento:

Fase 1

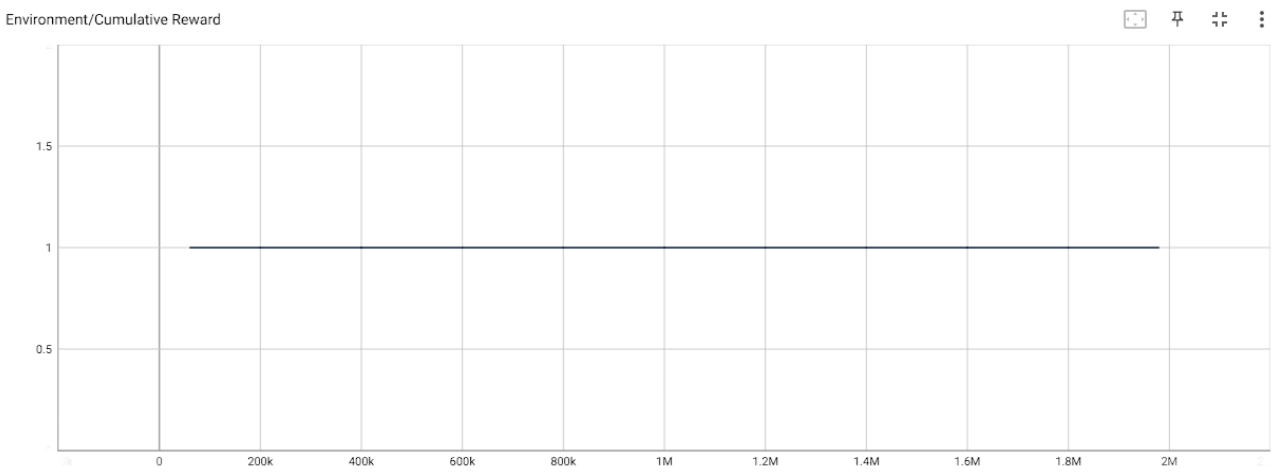

Figura 3.5.6: TensorBoard AIvsZombies

En esta fase, obviamente, la recompensa del agente va a ser 1, puesto que está frente de un objetivo que no se mueve y lo único que tiene que hacer es disparar.

Fase 2

En esta fase el agente solo puede rotar y se enfrenta a objetivos quietos, por lo que tiene que sincronizar su movimiento con el disparo para matar enemigos.

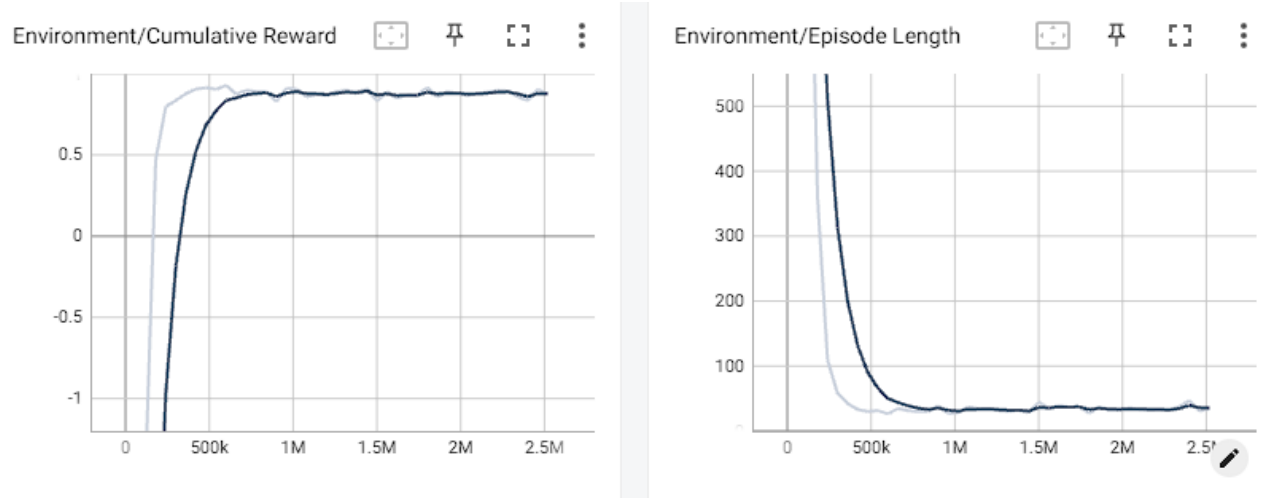

Figura 3.5.7: TensorBoard AIvsZombies

Vemos que el agente ha aprendido a girar y disparar a enemigos

Fase 3

En esta fase se quería que el agente aprendiera también a moverse en el eje xz.

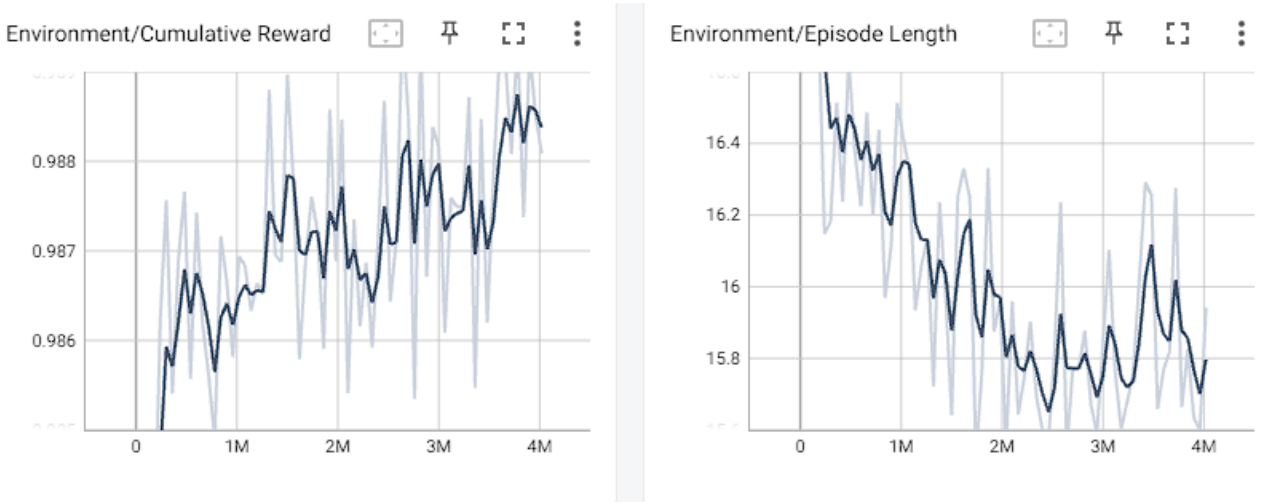

Figura 3.5.8: TensorBoard AIvsZombies

Vemos como no solo se ha implementado el movimiento en el eje xz sino que además se ha conseguido mejorar al agente.

En este punto empecé a entrenar al agente pero con los zombies moviéndose hacia él para matarle.

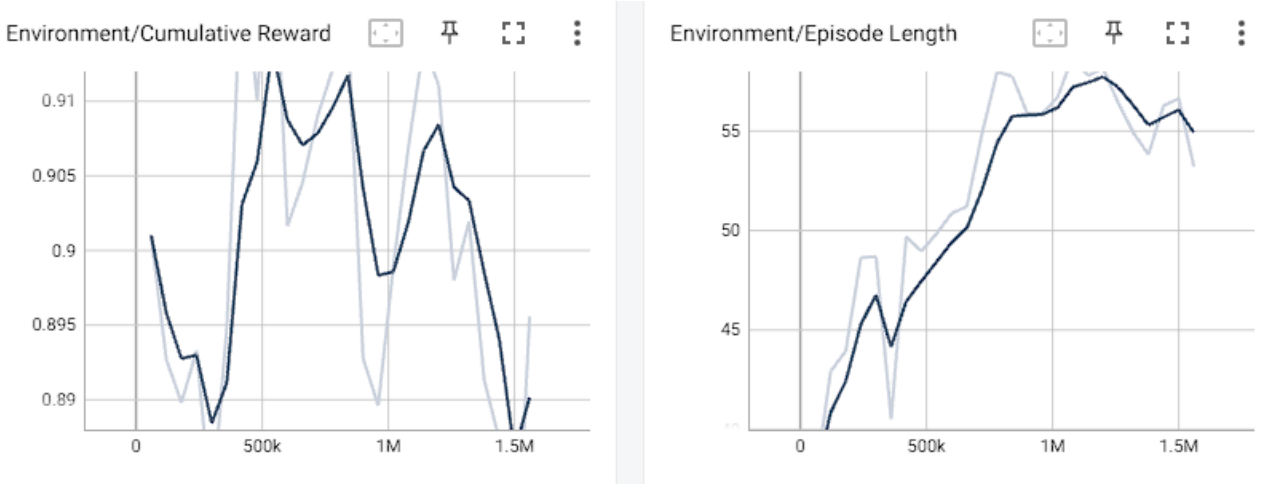

Figura 3.5.9: TensorBoard AIvsZombies

Finalmente queda así, bajando un poco la recompensa media y subiendo el número de pasos que hace el agente por episodio, cosa que es totalmente normal teniendo en cuenta que se dispara a objetivos en movimiento. La precisión de disparo del agente se sitúa en un 70%.

Capítulo 4 Conclusiones y líneas futuras

Hemos visto muchísimas herramientas de ML-Agents para el entrenamiento de agentes inteligentes, todas ellas totalmente diferentes y que cubren distintos tipos de observación, dando a este framework una capacidad enorme a la hora de entrenar agentes en diferentes ámbitos como ya hemos visto (videojuegos y robótica principalmente).

Como todo, tiene puntos malos, y puede no funcionar muy bien en cierto tipo de entornos, como por ejemplo un laberinto en el que los muros limiten al agente a la hora de tomar decisiones. Otra cosa es que solo se puede entrenar un modelo a la vez a pesar de tener 2 agentes en el entorno.

Pensando a futuro se pueden cambiar diferentes cosas de los modelos actualmente desarrollados, como por ejemplo en el AI vs Zombies, donde se podría entrenar a los zombies en otro entorno y poner a funcionar ambos modelos para ver su comportamiento. Otra cosa que se me ocurre y que me hubiera gustado desarrollar es un walker, que se le recompensara por andar de forma erguida y entrenarlo para que pudiera llevar a cabo diversas tareas. Las posibilidades son tantas como nos imaginemos.

En conclusión, ML-Agents es un framework útil, potente y fácil de usar para cualquier persona que entienda de programación. Obviamente, tiene sus limitaciones y puede no ser muy útil si no se tienen conocimientos adecuados de ciertas cosas como vectores, robótica, física…

Capítulo 5 Summary and Conclusions

ML-Agents is a framework which works with Unity that allows the user to train an agent using different types of technologies such as camera sensors, ray perceptors or simple observations.

In this project we have used ML-Agents to train various agents to show how each of this technologies works, from an agent that have to find a ball using simple observations, to 2 agents that compete to find the bigger amount of green balls using a camera sensor, or one that have to shoot zombies for his survival using a ray perceptor. We also commented on each of the training phases the graphs for the cumulative reward of the agent and the episode length making conclusions of the results.

We have seen many ML-Agents tools for training intelligent agents, all of them totally different and covering different types of observation, giving this framework enormous capacity when it comes to training agents in different fields as we have already seen (video games and mainly robotics).

Like everything, it has bad points, and it may not work very well in certain types of environments, such as a maze in which the walls limit the agent when making decisions. Another thing is that you can only train one model at a time despite having 2 agents in the environment.

In conclusion, ML-Agents is a useful, powerful and easy-to-use framework for anyone who understands programming. Obviously, it has its limitations and may not be very useful if you do not have adequate knowledge of certain things such as vectors, robotics, physics...

Capítulo 6 Presupuesto

El proyecto ha llevado alrededor de 350 horas de trabajo y otras 400 horas de cómputo donde en ordenador ha estado funcionando entrenando agentes. En total el tiempo de cómputo del PC ha sido de 750 horas sumando ambos, por tanto, teniendo en cuenta que el kW/h está en Canarias a un precio medio aproximado de 0.15 euros el kWh, el precio aproximado de cómputo es de unos 80 euros.

Por otro lado, un programador junior en España cobra 10.77 euros la hora, por lo que el precio por el servicio es de 10.77 euros x 350 horas de trabajo, que da un resultado de 3769.5 euros.

Sintetizando el coste del proyecto sería de 3849.5 euros aproximadamente.

Bibliografía

Introducción a Deep Reinforcement Learning

Diseño de NPC con machine learning

Generación automática de contenidos en videojuegos

Ejemplo de robótica en videojuegos

Documentación ML-Agents

Lista de reproducción con tutoriales de ML-Agents para principiantes