Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Deep Learning en Videojuegos: Aprendizaje por Refuerzo en el Entorno Unity

Deep Learning in Video Games: Reinforcement Learning in the Unity Environment

Germán Andrés Pescador Barreto

La Laguna, a 6 de septiembre de 2019

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

D. **Jose 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:

"Deep Learning en Videojuegos: Aprendizaje por refuerzo con el entorno Unity"

ha sido realizada bajo su dirección por D. **Germán Andrés Pescador Barreto**, con N.I.F. 54.606.919-V.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 10 de septiembre de 2019

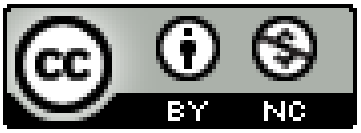
Agradecimientos

Este Trabajo de Fin de Grado está dedicado a mi familia, que ha hecho posible que llegue hasta aquí con su esfuerzo, y a mis amigos, que me han aguantado todo este tiempo y me han insistido y ayudado para que siga adelante en todo momento.

Agradezco también especialmente el apoyo de mis tutores, que se han empantanado conmigo en esta tecnología aún en desarrollo.

A todos aquellos investigadores que creen en el potencial de los juegos para el aprendizaje, ya sea de una máquina o de un ser vivo.

Licencia



Resumen

Este trabajo de fin de grado se encuentra dentro de la línea de trabajos "*Deep Learning* en Videojuegos", cuyo objetivo general es plantear soluciones a diversos problemas combinando herramientas para el desarrollo de videojuegos con técnicas de aprendizaje automático basadas en redes neuronales, más conocidas últimamente como Deep Learning. En esta ocasión, se pretende explorar el aprendizaje por refuerzo en un entorno realista provisto por el motor de videojuegos Unity, con ayuda del kit de herramientas de *Machine Learning* ML-Agents, publicado por el equipo de desarrollo de Unity en GitHub.

Antes de entrar en profundidad en el trabajo, se analiza la historia de esta tecnología, términos y conceptos indispensables para comprender su funcionamiento, y su lugar en un mundo donde actualmente la Inteligencia Artificial tiene mucha fuerza. Una vez expuesto esto, se procede a los aspectos técnicos del proyecto, así como su planificación inicial y su avance a lo largo del desarrollo, mostrando las simulaciones desarrolladas y estudiando su funcionamiento. Como cierre a esta memoria, se discuten los resultados obtenidos a lo largo del proyecto y las posibles mejoras que podrían aplicarse en caso de continuar con esta línea de trabajo.

Palabras clave: Deep Learning, redes neuronales, aprendizaje automático, aprendizaje por refuerzo, Unity, Machine Learning, Inteligencia Artificial.

Abstract

This project is one in the line of projects “Deep Learning in Video Games”, whose general goal is to present solutions to diverse problems combining tools for game development with automated learning techniques based on neural networks, known more lately as Deep Learning. In this occasion, the objective is to explore reinforcement learning in a realist environment provided by the Unity video game engine, with the help of the Machine Learning toolkit ML-Agents, published by Unity’s development team in GitHub.

Before going deep into the project, the history of this technology is analysed, along with terms and concepts that are essential for the understanding of how it works, and its place in a world where nowadays Artificial Intelligence has a lot of strength. Once this is exposed, we proceed to the technical aspects of the project, as well as its initial planning and its progress throughout development, showing the developed simulations and studying how they work. As closure to this document, the results obtained throughout the project are discussed, along with the possible enhancements that could be applied in case of continuing with this line of work.

Keywords: Deep Learning, neural networks, automated learning, reinforcement learning, Unity, Machine Learning, Artificial Intelligence.

Índice general

Índice de figuras

Capítulo 1
Introducción

El objetivo de este trabajo es la exploración del aprendizaje por refuerzo en un entorno realista provisto por el motor de videojuegos Unity con ayuda de su kit de herramientas de Machine Learning. Este kit, bautizado ML-Agents y actualmente en desarrollo, incluye todo tipo de herramientas útiles para la creación, configuración, y ejecución de cualquier escenario donde se quiera aplicar aprendizaje por refuerzo, además de una selección de algoritmos para el entrenamiento de agentes inteligentes en estos entornos mediante una conexión con la librería de python Tensorflow. En este proyecto se utiliza el algoritmo *Proximal Policy Optimization* (PPO), introducido más adelante.

Antes de entrar en materia, es necesaria una explicación de qué es y cómo funciona el aprendizaje automático, repasando algunos de los tipos que existen, para después continuar con una sencilla introducción a los entornos de desarrollo y simulación para el entrenamiento de inteligencias artificiales.

1.1 Aprendizaje automático

El aprendizaje automático, comúnmente conocido como Machine Learning, es una rama del campo de la Inteligencia Artificial que realiza un estudio científico de los algoritmos y modelos estadísticos que utiliza un ordenador para realizar una tarea específica sin utilizar instrucciones explícitas. Estos algoritmos construyen un modelo matemático basándose en datos de ejemplo, o “datos de entrenamiento”, con el objetivo de poder realizar predicciones o decisiones ante datos completamente nuevos sin haber sido programados explícitamente para realizar esas tareas

1.1.1 Historia

Los intentos por computerizar procesos de aprendizaje comienzan a surgir alrededor de la década de los 50 en la búsqueda de una inteligencia artificial general equiparable a una inteligencia humana, pero no es hasta que se reduce el objetivo del aprendizaje a tareas especializadas que no se hacen avances reales, siendo especialmente significativo el estudio [1] publicado en 1959 por Arthur L. Samuel utilizando el clásico juego de las damas para probar algoritmos desarrollados por él mismo. Así crea dos algoritmos capaces de jugar a las damas sin ser explícitamente programados para ello: un algoritmo de aprendizaje por memoria, que almacena todos los movimientos realizados en sus partidas analizadas junto al valor de la jugada para compararlos con el estado de la partida a resolver, y un algoritmo de aprendizaje por generalizaciones, el cual descarta el almacenamiento de todos los movimientos y almacena únicamente una valoración general de los movimientos que haya visto previamente.

Aparte del almacenamiento necesario para utilizarlos, la diferencia más notoria entre estos algoritmos se encuentra en su toma de decisiones ante distintas situaciones una vez entrenados. El aprendizaje por memoria presenta una gran habilidad para tomar las decisiones adecuadas en situaciones de recompensa a largo plazo, o cuando se requieren técnicas altamente especializadas. Por su parte, el aprendizaje por generalizaciones muestra ser de gran ayuda en situaciones en las que el abanico de posibilidades es muy amplio, así como cuando las recompensas se encuentran a corto plazo. Ante los resultados obtenidos, Samuel se mostró a favor de continuar sus investigaciones para desarrollar nuevos procedimientos que combinasen ambos procedimientos. "*As a result of these experiments one can say with some certainty that it is now possible to devise learning schemes which will greatly outperform an average person and that such learning schemes may eventually be economically feasible as applied to real-life problems*" (Samuel, Arthur L., 1959, p.220). Así nace el aprendizaje automático.

1.1.2 Situación actual

Desde los primeros avances en el campo, mencionados al comienzo de este capítulo, el estudio del aprendizaje automático ha avanzado notoriamente. Numerosos algoritmos surgen constantemente, pero todos ellos pueden dividirse en distintos tipos según diversos factores. Basando la división en el resultado que se desea obtener del algoritmo, se encuentran seis tipos distintos de aprendizaje automático: aprendizaje supervisado, aprendizaje no supervisado, aprendizaje semi-supervisado, aprendizaje por refuerzo, transducción, y aprendizaje para aprender o meta-aprendizaje. [2]

Los más básicos de estos aprendizajes son el supervisado y el no supervisado, ambos caras de una misma moneda. Los dos buscan resolver problemas de clasificación, aunque poseen una diferencia fundamental. Mediante el aprendizaje supervisado el algoritmo entrenará con ayuda de datos cuya clasificación es conocida, de manera que sus características puedan relacionarse con las características de nuevos datos sin una clasificación clara. Así pueden llegar a clasificarse todo tipo de conjuntos de datos, pero en aquellas ocasiones en que no se dispone de unos datos de entrenamiento con una clasificación clara, entra en juego el uso del aprendizaje no supervisado. Esta aproximación al problema pretende relacionar cúmulos de datos buscando patrones en sus propiedades estructurales, creando de esta manera sus propias clasificaciones.

Avanzando al aprendizaje por refuerzo, clave en este trabajo, puede considerarse que tiene sus datos de entrenamiento a medio camino entre estos dos paradigmas. No se conoce la salida adecuada, correspondiente a las acciones a realizar, para una determinada entrada, determinada por las observaciones que se tienen sobre el entorno además del estado propio, pero puede saberse si una acción es o no correcta, debiendo continuar la búsqueda de la acción adecuada en caso negativo. Este proceso no aplica a acciones individuales sino a cada secuencia de acciones realizadas en un mismo intento aislado de cumplir la tarea a dominar, y es que el algoritmo no suele ser recompensado por estas acciones individuales (a este tipo de algoritmos se les llama algoritmos bandido, y se alejan de la inteligencia general). Así, se busca el aprendizaje de estrategias de control (políticas) que permitan la actuación en un entorno dinámico desconocido gracias a la generalización de las experiencias obtenidas en los entornos de entrenamiento. [3]

Cuando se habla en el apartado anterior del estudio de A. L. Samuel en el juego de las damas puede verse un claro ejemplo de cómo los juegos pueden ser un magnífico entorno de prueba para algoritmos de aprendizaje automático. Esto es gracias a que los juegos suelen tener una serie de reglas definidas las cuales, junto a un entorno controlado y simple, dejan un abanico de acciones limitado para el jugador sin dejar de ser un reto intelectual. Así encontramos que muchos de los logros en el campo del aprendizaje automático, y sobre todo dentro del aprendizaje por refuerzo, son mostrados en juegos, al principio tan simples como el tres en raya (Michie, Donald, 1963, p.232)[4], complicándose hasta el nivel del backgammon (Tesauro, Gerald, 1994, p.215) [5], y llegando en la actualidad a hacerse demostraciones de sistemas que llegan a jugar Go con la habilidad suficiente para ganar al campeón del mundo en una partida. [6] Pero cuando se vence al campeón del mundo aún hay espacio para mejora, y es así como se comienza a perseguir un rendimiento superhumano.

AlphaGo, el algoritmo desarrollado por Google que venciera al campeón del mundo en 2016, utiliza una combinación de aprendizaje supervisado, utilizando partidas entre jugadores profesionales, y aprendizaje por refuerzo, jugando contra sí mismo una vez ha terminado el aprendizaje inicial. Para lograr un desarrollo superhumano, era necesario descartar el condicionamiento inicial que genera el aprendizaje mediante partidas profesionales, y por ello se opta por refinar un algoritmo que únicamente entrene mediante aprendizaje por refuerzo contra sí mismo. De esta manera aparece AlphaGo Zero(Silver, David, et al., 2017, p.354) [7], que tras entrenar durante 72 horas (frente a los varios meses de entrenamiento de AlphaGo) conociendo únicamente las reglas del juego, es capaz de vencer a su antecesor con una aplastante racha de cien victorias.

1.2 Entornos de desarrollo

Tradicionalmente los entornos de desarrollo en los que se prueban los algoritmos son propios de cada equipo de desarrolladores, pero por cuestiones de recursos o simple conveniencia actualmente muchos algoritmos son entrenados utilizando videojuegos comerciales o los motores gráficos de los mismos. A continuación se abordan dos entornos diseñados para este fin.

1.2.1 Arcade Learning Environment

El Arcade Learning Environment (Bellemare, Marc G., et al., 2013, p.253) [8], o ALE, es un entorno de aprendizaje desarrollado con el objetivo de entrenar algoritmos que alcancen una especie de inteligencia general. Para ello hace uso de un gran número de videojuegos de la consola Atari 2600, permitiendo observar en distintos entornos el desempeño de los algoritmos desarrollados de manera que no se evalúe a un agente en los mismos entornos en los que ha sido entrenado, dado que esto puede dar una sobreestimación de las capacidades del algoritmo utilizado.

Al tratarse de juegos para una consola lanzada en 1977, la complejidad de estos es relativamente baja para un ser humano, puesto que en su mayoría son juegos en dos dimensiones y de no muy larga duración, pero resultan ideales para el estudio de algoritmos de aprendizaje por refuerzo que generen agentes capaces de alejarse de la especialización en un sólo juego y se aproximen a una inteligencia general que domine todos los juegos. Esto es posible gracias a que los juegos son relativamente simples y los controles de la consola son limitados, dándole al agente un vector de acción con solo cinco opciones (arriba, abajo, izquierda, derecha, y disparo) que comparten todos los juegos, y es que la tecnología aún no ha avanzado lo suficiente como para dominar distintas tareas de manera general en entornos mucho más complicados.

El principal problema de este entorno surge del hecho de que su propósito sea apoyar las investigaciones que buscan alcanzar una inteligencia general, pues esto limita el público que puede utilizarlo para desarrollar nuevas tecnologías que aprovechen el aprendizaje por refuerzo. Cuando el objetivo es generar agentes especializados, se necesitan entornos configurables que se acerquen más al mundo real.

1.2.2 OpenAI Gym

Por su parte, OpenAI Gym (Brockman, Greg, et al., 2016) [9] ofrece un abanico de posibilidades mucho más amplio en lo que a la naturaleza de los entornos de entrenamiento se refiere. Los entornos disponibles crecen constantemente, a diferencia de la librería estática de videojuegos del ALE, y plantean una gran variedad de problemas útiles para el entrenamiento de agentes especializados.

La oferta de entornos del OpenAI Gym va desde videojuegos de la Atari 2600 hasta el complejo simulador de físicas MuJoCo (Todorov, Emanuel, Tom Erez, and Yuval Tassa, 2012) [10], llegando a contar incluso con un entorno para que los algoritmos entrenados puedan aprender a comportarse como otros algoritmos, imitando computaciones sencillas. El problema es que todos estos entornos son controlados por los desarrolladores de OpenAI Gym, que son quienes deciden qué entornos son añadidos a la aplicación. Esto limita a los usuarios pues, aunque abstraer el entorno para que los escenarios sirvan como un estándar en el que comparar distintos algoritmos pueda ser de ayuda para muchos desarrolladores, dificulta el trabajo de usuarios más alejados de la investigación científica que únicamente buscan una manera de probar e implementar el aprendizaje por refuerzo con algoritmos ya existentes en soluciones propias.

Capítulo 2
Unity y el aprendizaje por refuerzo

En lo que a motores de videojuegos se refiere, Unity es una de las aplicaciones más cercanas a sus usuarios, con herramientas disponibles de forma completamente gratuita y una gran comunidad de desarrolladores. Es utilizado tanto para su objetivo original, la creación de videojuegos, como para investigaciones que hacen uso del motor para simular entornos con fines tan diversos como ayudar en el tratamiento de trastornos mentales [14]. Con incontables productos en el mercado, es una tecnología con mejoras constantes.

2.1 Aprendizaje sin herramientas

Unity se ha usado como entorno de aprendizaje con anterioridad, siendo utilizadas adaptaciones de algoritmos simples para el entrenamiento de agentes inteligentes que pueden ser utilizados como personajes no jugables, o como personajes principales cuando la diversión proviene de la evolución del agente durante su entrenamiento.

El problema de esta aproximación radica ya no solo en la dificultad que implica programar un algoritmo de aprendizaje automático en un entorno que no está diseñado para ello, sino en la configuración del agente que va a hacer uso del algoritmo para aprender, y es que aunque Unity permite el uso de programas escritos en C# durante la ejecución de un juego, la cantidad de ajustes que puede llegar a necesitar la configuración de un agente a lo largo de los entrenamientos hace bastante poco accesible este entorno para usuarios interesados en el aprendizaje automático. Es por ello que, aunque pueden encontrarse proyectos que hacen uso de este entorno para experimentar con aprendizaje por refuerzo desde cero [11], no comienza a ser utilizado por la comunidad científica hasta el lanzamiento en 2017 de la versión 0.1 del “ML-Agents toolkit” [12], un kit de herramientas de aprendizaje automático publicado por los desarrolladores de Unity.

2.2 ML-Agents toolkit

Con la llegada de este kit de herramientas, Unity pasa a ser un entorno de gran accesibilidad tanto para investigadores como para usuarios más casuales, y notándose su impacto en la comunidad de Unity con la aparición de proyectos muy interesantes, como adaptaciones de entornos de OpenAI Gym [13]. Este lanzamiento forma parte de la iniciativa de Unity por hacer que su plataforma se convierta en el entorno de referencia para la investigación en Inteligencia Artificial, y junto a la potencia creativa del editor queda un entorno de desarrollo muy potente, con posibilidad de editar los escenarios a voluntad con distintas opciones físicas sin dejar de lado la configuración de los agentes y la posibilidad de cargar algoritmos de diseño propio.

2.2.1 Herramientas

La principal mejora que trae este kit a la plataforma es la conexión de Unity con una API de Python, y es que con esto se permite el uso, sin librerías de terceros, de la que posiblemente sea la librería más importante en este campo, TensorFlow, una versión abierta del código utilizado por Google para su investigación con Google Brain. A través de Tensorflow se programan y ejecutan la mayoría de algoritmos de aprendizaje automático, y Unity los trae a su plataforma haciendo uso de una conexión mediante protocolos gRPC para controlar desde Python la simulación que tiene lugar en el motor Unity.

Esta conexión viene acompañada de un kit de desarrollo de software con el que poder convertir cualquier escena construida en el editor en un entorno de aprendizaje, existiendo tres componentes esenciales: Agente, Cerebro y Academia. [12]

2.2.2 Funcionamiento

Los componentes de un entorno de aprendizaje se encuentran en una retroalimentación constante durante el proceso de entrenamiento. Los Agentes se encargan de obtener información del entorno y transmitirla a sus respectivos Cerebros, de quienes reciben la siguiente acción a ejecutar. Cada Cerebro puede estar conectado a cualquier número de Agentes, con la única condición de que sus espacios de acción y observación coincidan, y son responsables de la toma de decisiones siguiendo cuatro modelos posibles según el proceso que sigan. Existen las opciones de controlar manualmente a los agentes con un cerebro Player (lo cual es interesante para casos de aprendizaje por imitación o simplemente para probar el correcto funcionamiento de un Agente), tomar decisiones mediante heurísticas, ejecutar un modelo de aprendizaje interno que realice todos los cálculos desde C#, o hacer uso de un cerebro externo que se beneficie de las bondades de TensorFlow. En este último caso el flujo de información es constante entre el Cerebro y la API de Python a través de la Academia, que es la encargada de controlar tanto el propio entorno de aprendizaje como las condiciones en las que se ejecuta la simulación. En la figura 2.1 puede apreciarse el funcionamiento de estos componentes de una manera más visual.

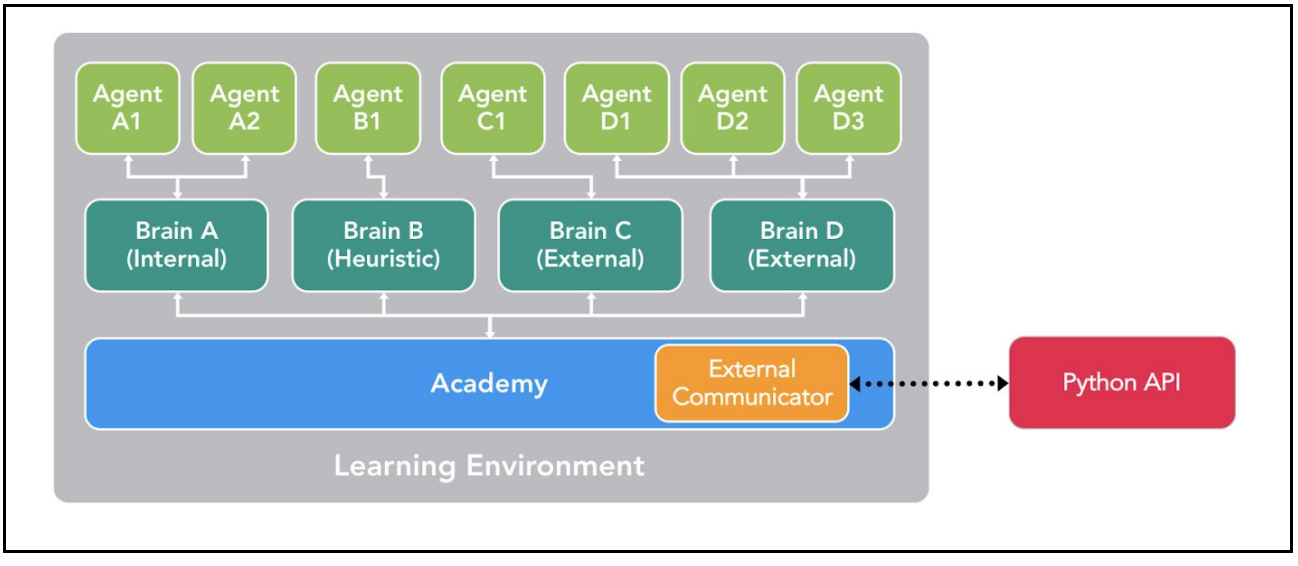


Figura 2.1: Esquema de un entorno de aprendizaje [12]

2.2.3 Proximal Policy Optimization Algorithm

Si bien se ofrecen una serie de algoritmos base con los que experimentar la conexión con TensorFlow en este entorno, el más utilizado a la hora de mostrar las bondades de este kit de herramientas es una implementación de un algoritmo PPO, o “Proximal Policy Optimization”(Schulman, John, et al., 2017) [15], el cual es un tipo de algoritmo de aprendizaje profundo por refuerzo (es decir, aprendizaje por refuerzo con la ayuda de redes neuronales para el aprendizaje de políticas con funciones no lineales) relativamente nuevo que ha arrojado muy buenos resultados a la par o incluso mejor que algoritmos referentes en la investigación del aprendizaje por refuerzo pero con una mayor facilidad de configuración e implementación.

Este algoritmo surge del análisis de los algoritmos ya existentes, observando dónde fallan y analizando el por qué. Muchas veces al entrenar un algoritmo clásico surge el problema de que se intenta mejorar en la dirección incorrecta y esto deja al agente en un mal lugar para continuar su entrenamiento, lo cual se puede arreglar implementando regiones de confianza en las que probar nuevas soluciones antes de renovar las políticas. Para evitar grandes cambios y aportar estabilidad al entrenamiento, se plantea la limitación de cómo de diferente puede ser una nueva política respecto a la anterior mediante una serie de cálculos de gran complejidad. Esta complejidad es un problema dado que estos cálculos necesitan ejecutarse millones de veces, así que con el PPO se opta por hacer una adaptación de los algoritmos clásicos para que sus cálculos se acerquen a los ideales mediante la adición de pequeñas limitaciones con una nueva función bautizada “Clipped surrogate objective function”, que constantemente limita los cambios en las políticas.

Capítulo 3
El proceso de aprendizaje

Con un conocimiento básico de la tecnología, puede comenzarse a experimentar con ella. Para ello es necesario, teniendo desde el principio una visión de las acciones que se quiere enseñar al agente, desarrollar el entorno de entrenamiento, programar los agentes y ejecutar y supervisar el entrenamiento.

3.1 Desarrollo del entorno

Si bien pueden tenerse en mente muchas ideas alimentadas por los grandes proyectos que desarrollan empresas como Google u OpenAI, muchas veces la tecnología puede quedarse corta en cuanto a su capacidad para aprender a realizar ciertas tareas, o simplemente puede pecarse de un mal diseño que afecte negativamente al aprendizaje. Es por ello que se deben desarrollar entornos simples e ir avanzando hacia la situación deseada tras comprobar que un aprendizaje correcto es posible.

A la hora de integrar los modelos de obstáculos u otros objetos con los que deban interactuar los agentes debe tenerse en cuenta que Unity no es un entorno de diseño 3D y por tanto no posee las herramientas para crear diseños realistas en el propio editor. Para cubrir este problema, si no se tienen conocimientos y acceso a programas de edición 3D, se recomienda el uso de utilidades como la Asset Store de Unity, que permite obtener toda clase de recursos para su uso en el editor, muchos de ellos de forma gratuita.

El modelo que vaya a usarse de Agente ha de colocarse en la posición en donde se quiera que comience el entrenamiento, teniendo en cuenta su postura en caso de que se tenga en mente un diseño que haga uso directo de las articulaciones del modelo.

3.2 Programación de los agentes

Una vez se conocen el reto a afrontar y las condiciones del entorno en que entrenarán los agentes, puede comenzarse la programación de éstos, la cual se realiza en C# heredando de una clase abstracta Agent preparada para realizar todas las comunicaciones pertinentes con el resto de elementos. Esta clase permite el control de muchas de las variables del agente desde el editor de Unity, y en ella destacan cuatro métodos utilizados para la inicialización del agente, la recolección de observaciones, la ejecución de acciones, y el reinicio del agente. Todos estos métodos se han de personalizar en el código del agente mediante un override.

3.2.1 Inicialización del agente

Para la inicialización del agente, que sucede al comenzar la ejecución del entrenamiento, se utiliza el método “InitializeAgent”. Es necesario hacer uso del mismo en el código siempre que el agente desarrollado haga uso de recursos más allá de su modelo base, ya sean extremidades físicas, código de scripts externos, o variables controladas por la academia, puesto que estos recursos necesitan ser conectados a las variables del código principal del agente para poder manejar y utilizar su información.

3.2.2 Recolección de observaciones

Para que el agente pueda decidir qué hacer a continuación, este necesita saber el estado de su entorno. Para ello se le aporta un vector de observaciones al cerebro del agente, el cual será la entrada que se le de a la red neuronal utilizada por el algoritmo de aprendizaje. Este vector se construye desde el método “CollectObservations” presente en la clase abstracta Agent al igual que el método de inicialización. Para construirlo, se envían las variables a controlar como argumentos del método “AddVectorObs”, teniendo en cuenta que cada variable añadida a este vector deberá verse reflejada en la definición del espacio del mismo en la configuración del Cerebro en el editor, o no podrá ejecutarse entrenamiento alguno.

A la hora de decidir las observaciones que se le deben aportar a un agente, se puede plantear la cuestión de qué necesitaría un ser humano para realizar las acciones que se le piden al agente y reflejarlas en el editor. En el caso de una mano que deba manipular objetos [13], por ejemplo, se necesita conocer por lo menos la situación del objeto manipulado, así como la posición relativa de cada parte móvil del agente, y aunque pueda pensarse que también sería beneficiosa una sensación táctil del objeto, en el experimento citado se demuestra que no es necesaria, mostrando cómo no siempre más información tiene por qué ser mejor para el aprendizaje del agente. Es por ello que, de nuevo, se recomienda el desarrollo de situaciones de baja complejidad antes de afrontar grandes retos.

3.2.3 Ejecución de acciones

Tras el análisis de los valores del vector de observación por parte del Cerebro, se establecen una serie de valores en el vector de acción del agente. Cada valor del vector de acción es una variable de tipo float si el espacio del vector es continuo, y de tipo int si el vector es discreto. La diferencia entre estos dos tipos radica en la mayor precisión de las acciones que pueden realizarse con las variables float en un vector de espacio continuo, siendo por ello más valioso para aquellos agentes que deban realizar tareas complejas como simulación de movimientos humanos en tiempo real, pero esa precisión adicional supone un mayor coste computacional, aunque el algoritmo PPO ha demostrado superar con creces a otros algoritmos en entornos con control continuo.

El vector de acción es recibido por el agente como un argumento del método “AgentAction”, donde se actúa en consecuencia de los valores del vector. Así, se le ordenan los pasos a seguir al modelo de la simulación, ya sean un movimiento o cualquier otro tipo de acción que tenga consecuencias en el entorno de aprendizaje y/o la situación del agente en este. Es tras la ejecución de acciones que, en la misma función, puede comprobarse el estado de la tarea a realizar. Si el agente ha cumplido la tarea a realizar, conseguido un logro, o simplemente ha realizado una acción considerada positiva, se le puede recompensar a través del método “AddReward”, al que se llamará con un número como argumento que deberá reflejar el peso de la acción realizada, y en el caso primero se puede llamar tras la recompensa al método “Done” para comenzar un nuevo paso de entrenamiento. También se puede utilizar el método “AddReward” a modo de castigo dándole como argumento un número negativo, pero en la mayoría de los casos se pretende que el agente se de cuenta por sí solo de qué acciones no le son beneficiosas.

3.2.4 Reinicio del agente

El reinicio del agente tiene lugar cuando en alguna parte del código se llama al método “Done”, que indica el fin de un paso de entrenamiento ya sea porque algún sensor detecta una situación negativa o porque la última acción ha permitido completar la tarea, siempre y cuando la opción de reiniciar al agente al llegar a este estado esté activada en el editor. Cuando esto sucede, el método “AgentReset” tiene como tarea dejar al agente y su entorno en unas condiciones que permitan comenzar un nuevo paso de entrenamiento. Esto puede implicar devolver al agente a su posición y postura originales y/o realizar cambios en el entorno, especialmente si el entrenamiento incluye aprendizaje por currículum.

3.3 Entrenamiento

Con un entorno y un agente funcionales, se puede proceder al entrenamiento. En el caso de entrenar usando el algoritmo PPO, la ejecución se realiza con ayuda de una terminal Anaconda de comandos que facilita el uso de un entorno Python. En el momento de ejecutar el entrenamiento desde la raíz del proyecto, se deben especificar una serie de parámetros que cumplen el siguiente formato:

“mlagents-learn <dirección del archivo de configuración> (opciones)”

El archivo de configuración es aquel que contiene los hiperparámetros del entrenamiento, los cuales son utilizados por el algoritmo de aprendizaje para su funcionamiento dentro de las variables establecidas, que controlan desde la longitud del entrenamiento hasta la profundidad de la red neuronal utilizada. En muchas ocasiones la obtención de resultados negativos en los entrenamientos se debe a un mal uso de estas variables, y es por ello que debe tenerse extremo cuidado al realizar cambios en las mismas, probando pequeños ajustes cada vez que sea necesario modificarlas.

Las opciones por su parte dependen del entrenamiento a realizar, y todas ellas deben ir precedidas por dos guiones. La ejecución más básica requiere de las opciones “run-id” y “train”, definiendo la primera un identificador con el que hacer referencia al entrenamiento en los registros, mientras que la segunda ejecuta el algoritmo en modo de entrenamiento. Ejecuciones posteriores utilizando los resultados de ese entrenamiento como base deben hacer referencia al mismo identificador y además incluir la opción “load”. En el momento de la ejecución, la terminal queda a espera del editor de Unity, en donde debe ejecutarse el entorno a través del botón de “play”, a menos que se utilice una entorno compilado, situación en la que se indaga en el siguiente capítulo.

Los resultados del entrenamiento pueden ser consultados con ayuda de la herramienta TensorBoard mediante la ejecución en otra terminal del siguiente comando:

“tensorboard --logdir=<dirección del fichero de registros>”

Según los resultados obtenidos, se puede iterar a una nueva fase de desarrollo con una configuración diferente para los distintos elementos del entorno de aprendizaje, ya sea ajustando el reto a afrontar, las características del agente, o los hiperparámetros del entrenamiento.

Capítulo 4
Simulaciones

El objetivo Utilizando las herramientas del ML-Agents toolkit, se ha experimentado modificando el entorno prefabricado “Walker”, creado como un entorno de prueba por el equipo de desarrollo de Unity, y un entorno creado desde cero que ha sido bautizado como “Cubo”.

4.1 Walker

El entorno Walker guarda estrechas similitudes con entornos del OpenAI Gym en los que se pretende enseñar a caminar a agentes bípedos. En este entorno se presenta a once agentes, todos controlados por un mismo cerebro externo modelado por el algoritmo PPO, que han de aprender a utilizar las dieciséis partes de su cuerpo, las cuales pueden apreciarse en la figura 4.1, para alcanzar cada uno un objetivo propio al final de un escenario plano.

4.1.1 Configuración

Al comienzo de una simulación, el agente está programado para configurar, preparar y conectar todas las partes de su cuerpo mediante el uso de un controlador de articulaciones compartido entre varios de los entornos de prueba de ML-Agents.

En lo que a las observaciones del agente respecta, el Walker recibe por cada una de las partes de su cuerpo valores que le indican si la parte en cuestión está tocando el suelo, su velocidad relativa al resto del cuerpo, su velocidad angular, su posición respecto a las caderas, y si no es el final de una extremidad o las caderas, la rotación que sufre en cada uno de los tres ejes además de la fuerza que está ejerciendo la articulación. Aparte de estas observaciones también recibe la dirección al objetivo, la posición de la cadera como indicador de la posición general del agente, y los vectores de las direcciones superior y frontal de la cadera para conocer una aproximación de su postura. En total se cuentan 212 valores distintos en el vector de observación del Walker.

El vector de acción es mucho más simple, afectando a 39 variables del agente. Con él se le asignan valores a la rotación objetivo de cada parte del cuerpo, así como a la fuerza que cada articulación debe aplicar. Estos valores deben utilizarse correctamente y con precisión para que el agente pueda realizar una locomoción efectiva, por lo que puede esperarse un periodo de entrenamiento largo antes de que comience a dar sus primeros pasos hacia el objetivo.

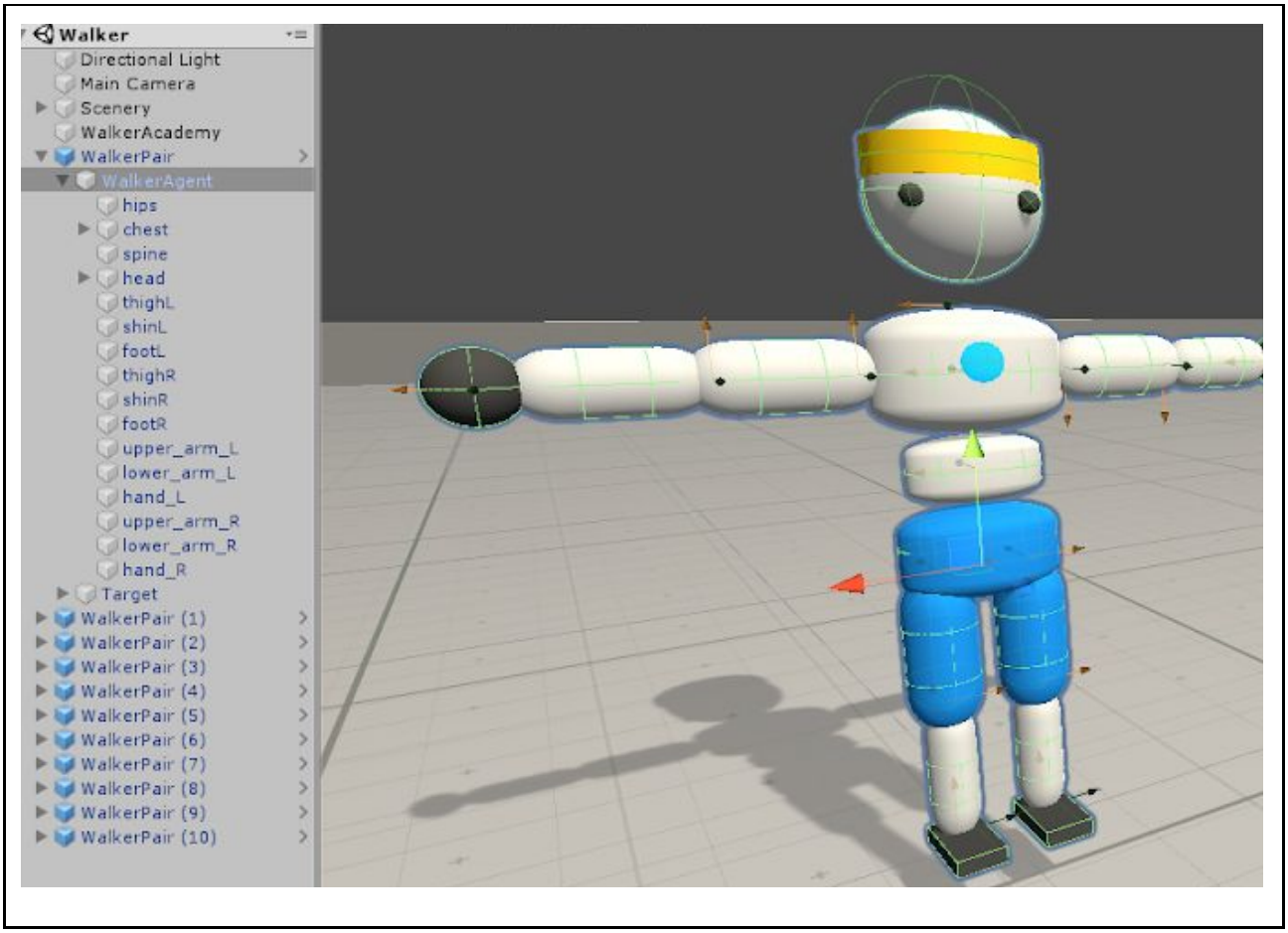


Figura 4.1: Modelo del agente Walker junto a un despliegue de todas las partes del mismo

Cada agente es recompensado a medida que avanza la simulación según la alineación de su velocidad y la rotación del cuerpo con respecto a la dirección hacia su objetivo, además de la altura de la cabeza del modelo. Con estas recompensas se busca un comportamiento en el cual el agente intente avanzar recto hacia el objetivo, mirando hacia el mismo mientras mantiene la cabeza alta para una correcta postura. También se le otorga un pequeño castigo proporcional al movimiento de la cabeza, con miras a desalentar un movimiento excesivo de la misma, y un castigo estándar por cada parte del cuerpo distinta a los pies que establezca contacto con el suelo en algún momento, situación que también desencadena un reinicio del agente.

4.1.2 Entrenamiento básico

El primer entrenamiento del Walker, llamado WalkerOne, se realiza con la versión 0.7 de ML-Agents desde el editor y con la configuración base del mismo, que establece una duración de entrenamiento de dos millones (2M) de pasos. Debido a la complejidad del entrenamiento del Walker, éste tarda cerca de 42 horas en completarse, aunque arroja buenos resultados con valores de recompensa alrededor de los 1500 puntos de manera estable. Se decide entonces aumentar el número de pasos a realizar en el entrenamiento a los cinco millones (5M), con esperanzas de encontrar aún mejores resultados, pero nada más comenzar la ejecución esto no tiene lugar, sino que las recompensas acumuladas sufren una caída en picado, llegando a revelar un valor de -5.826 en el paso 2.162M.

43 horas después de los primeros resultados, el aprendizaje resulta ser positivo con una media de recompensas aproximada de 1600, pero no se considera que el entrenamiento haya sido correcto, dadas las grandes pérdidas que se han sufrido en esta segunda parte del mismo, por lo que se detiene la ejecución en el paso 3.5M y se realizan otros dos entrenamientos, WalkerTwo y WalkerFour, con la misma configuración, en los que se sufren pérdidas catastróficas tras las que los agentes pierden la capacidad de mover algunas extremidades.

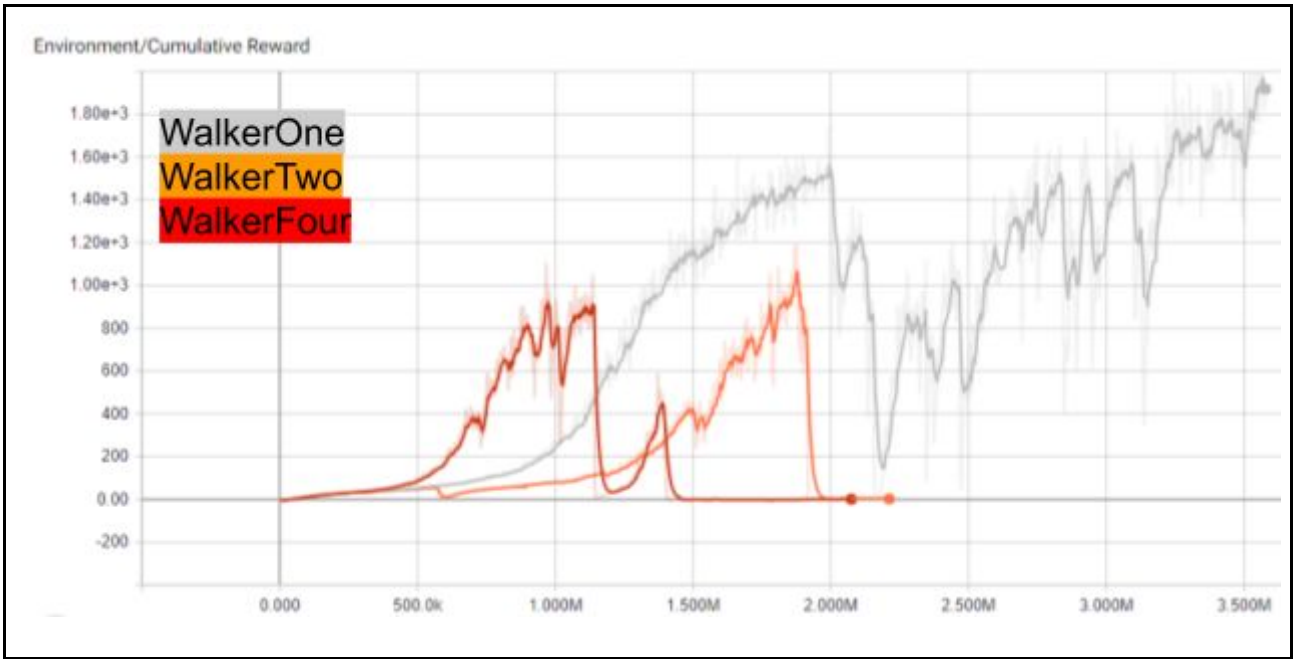


Figura 4.2: Primeros entrenamientos en el entorno Walker.

Ante estos resultados, el camino a seguir es incierto. Existen referencias a entrenamientos fallidos con características similares entre la comunidad de la herramienta, pero no se le puede dar una razón exacta, hasta que se descubre que, tal y como funciona el algoritmo PPO, el ratio de aprendizaje, responsable de establecer el tamaño de los pasos del descenso de gradiente que da el algoritmo en busca de mejoras en las políticas, comienza el entrenamiento con un valor de 3e-4 y va tendiendo a 0 a medida que se acerca al final del entrenamiento. Es por ello que el aumento del tamaño del entrenamiento del primer Walker una vez este ya había sido entrenado pudo ser la razón de que los cambios en sus políticas fuesen tan erráticos como para causar un efecto tal en las recompensas obtenidas, pero esto sigue sin explicar los problemas en los otros dos intentos. En el momento de redacción de esta memoria se sospecha que puede deberse a hiperparámetros que no se tuvieron en cuenta o a un simple “bug” del entorno, aunque reiniciar el mismo no solucionaba el problema.

Tras el lanzamiento de la versión 0.8.2 de ML-Agents se reanudan las sesiones de entrenamiento, siendo la primera un nuevo intento de ejecución básica del Walker, BasicWalker, con las mismas condiciones iniciales que WalkerOne. Esta iteración supera a su predecesora tanto en el dominio de la tarea como en el tiempo de ejecución, llegando tras 37 horas de entrenamiento a una recompensa media de 1800, y no solo eso, sino que comienza a mejorar en la tarea mucho más rápido. Todas estas mejoras se atribuyen al uso de una versión más avanzada de ML-Agents, y al no observar problemas aparentes se decide comenzar a realizar cambios en el entorno.

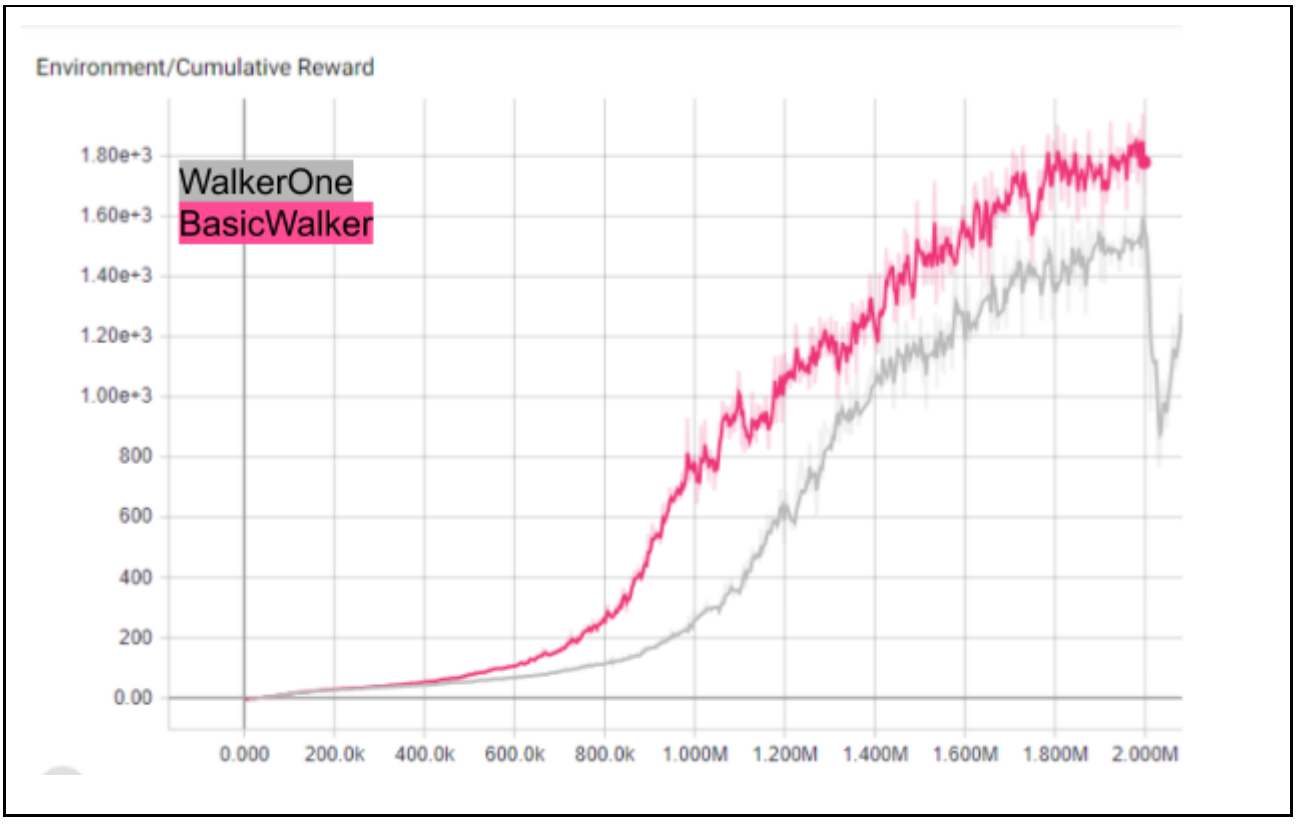


Figura 4.3: Comparación de los entrenamientos WalkerOne y BasicWalker.

4.1.3 Entrenamiento con percepción por rayos

Observando correr a los agentes surge la idea de añadir obstáculos al circuito, por lo que se decide realizar una mejora en sus observaciones y se les aporta visión a través de "raycasting" haciendo referencia al ya existente código "RayPerception3D", lo cual les otorga la posibilidad de recibir como información si un objeto detectable está en el camino de un rayo lanzado por el agente. Tras añadir el script a los agentes como un componente adicional en el editor se modifican las observaciones, otorgando a cada agente una matriz de 21 rayos, distribuidos en 3 filas, con un alcance de 50 unidades, capaces de detectar, por el momento, el suelo.

Los primeros intentos no son satisfactorios dado que en cuanto el agente comienza a avanzar la matriz de rayos queda estática en la posición de salida, y es que tal y como está implementado el agente en el escenario de prueba el mismo no es más que un contenedor de todas sus partes, las cuales se alejan de él al comenzar a avanzar y vuelven a su lado al reiniciarse. Comprendida esta característica del Walker, se traslada el componente del script de raycasting del agente a la cabeza de éste, y se realizan modificaciones en el código para la correcta comunicación con el mismo en su nueva ubicación.

Ya con todo en orden, se comienza el entrenamiento WalkerRay, el cual termina mostrando un desempeño muy similar al visto anteriormente en WalkerOne. Con 50 horas de entrenamiento para alcanzar los dos millones de pasos y una recompensa media aproximada de 1450, al compararlo con BasicWalker este entrenamiento muestra el efecto a nivel computacional de un aumento en el vector de observación. Con un mayor número de entradas para la red neuronal, el aprendizaje se ralentiza y el valor medio de recompensa sufre.

Con esperanzas en el potencial de haberle dado al agente una forma de visión, se procede a aumentar la duración del aprendizaje, primero a cuatro millones de pasos y posteriormente a cinco, donde se encuentra el valor de recompensa más alto encontrado en un Walker en este proyecto, 3319. Tras haber realizado estos pequeños incrementos con éxito, se decide probar un aumento hasta los diez millones de pasos. Esto resulta ser fatídico para el entrenamiento, que sufre pérdidas constantes, y tras una caída a un valor de 27.64 cerca del paso 6.65M, de la que el agente se recupera parcialmente, se decide abandonar este entrenamiento, con 236 horas acumuladas. Se cree que el algoritmo pudo soportar la progresión a cuatro millones de pasos puesto que el ratio de aprendizaje se redujo lo suficientemente rápido como para que las políticas no sufrieran cambios excesivos, pero el cambio de cinco a diez millones de pasos, aunque establece el ratio de aprendizaje en el mismo punto que el cambio de dos a cuatro millones al considerarse que el entrenamiento está exactamente a la mitad, observa un descenso del ratio de aprendizaje mucho más lento y deja a la política, ya refinada, demasiado expuesta. El cambio de cuatro a cinco millones apenas trae efectos negativos al ser bastante pequeño.

En la figura 4.4 puede verse la gran similitud de la inestabilidad presente en WalkerOne y WalkerRay como consecuencia del aumento indiscriminado de la duración del entrenamiento. Una solución a estos problemas puede ser la modificación del hiperparámetro correspondiente al ratio de aprendizaje en el archivo de configuración, y es que en teoría, con valores menores al original, el algoritmo a la hora de hacer el cálculo de qué valor debe tener este ratio lo reducirá proporcionalmente a la modificación hecha en la configuración. Se requiere mayor experimentación en este apartado para confirmar el funcionamiento de esta solución.

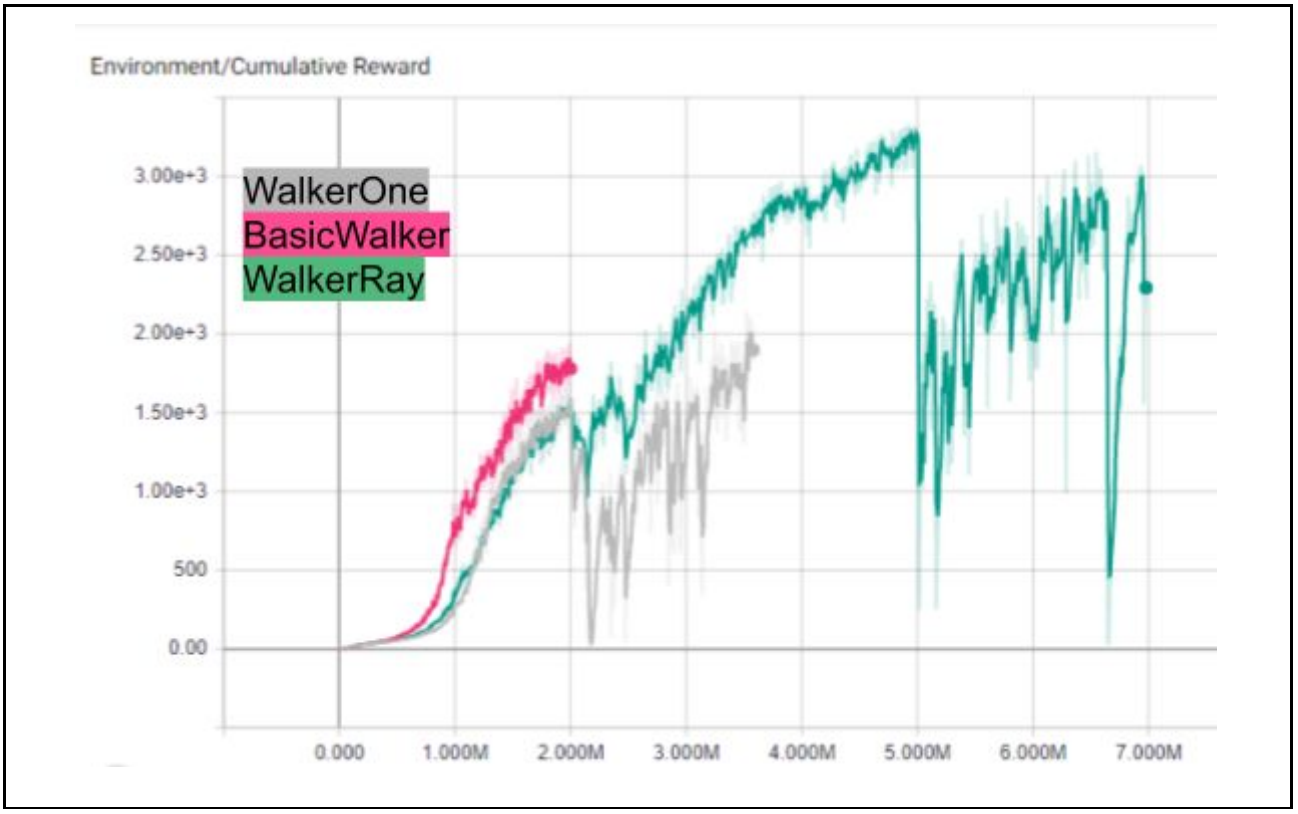


Figura 4.4: Comparación de los entrenamientos WalkerOne, BasicWalker y WalkerRay.

4.1.4 Entrenamiento con obstáculos

Tras los buenos resultados obtenidos en la iteración anterior, se procede con la idea de añadir obstáculos al escenario. Este reto se plantea de manera que los agentes aprendan a afrontarlos a medida que sus habilidades motrices aumentan, de forma que la dificultad sea acorde al estado del aprendizaje del agente. Para ello la solución ideal es añadir aprendizaje por currículum al entrenamiento, el cual permite darle a la Academia una serie de parámetros para que puedan darse varias lecciones, las cuales traen valores asociados que los agentes deben revisar cada vez que se reinician para implementar los cambios necesarios en sus entornos individuales, de los cuales son responsables.

El aprendizaje por currículum se implementa a través de un nuevo archivo de configuración localizado en una subcarpeta “curricula” dentro de la carpeta ya existente para configuración del proyecto. Este archivo, que debe tener el mismo nombre que el cerebro para el que ha sido diseñado, comienza definiendo si se quiere avanzar en las lecciones según progreso del entrenamiento o recompensa alcanzada, para luego definir las fronteras a superar para avanzar de lección, expresadas en progreso como decimales entre 0 y 1 y en recompensa como la recompensa a obtener antes de avanzar. Por cada una de estas fronteras, cada parámetro especificado en el currículum debe tener un valor, además de uno adicional utilizado al principio del entrenamiento, momento que corresponde con la lección 0.

Tras la configuración de un currículum con distintas alturas máximas y mínimas definidas como parámetros, se escriben unas pocas líneas de código con las que generar vallas en el escenario durante la inicialización del agente mediante la instanciación de un objeto cubo cuya escala es modificada para cumplir como valla. La altura y posición de las vallas son alteradas mediante un método de actualización que es llamado cada vez que el agente se reinicia, de manera que el entorno al que se enfrentan los agentes sea dinámico y no aprendan únicamente a resolver una distribución estática de obstáculos.

Con el vector de observación modificado para poder detectar los obstáculos con la matriz de rayos y una división de los agentes en dos plataformas distintas para que tengan mayor libertad de movimiento sin interponerse unos en los entrenamientos de otros, se procede al entrenamiento WalkerFence0 con la configuración de entrenamiento en diez millones de pasos utilizada en la última ejecución de WalkerRay, puesto que se espera la necesidad de un entrenamiento mucho más largo para la obtención de resultados. A partir de este punto, a la ejecución del entrenamiento a través de consola se le añade la opción “curriculum” en la cual se indica la dirección de la carpeta donde se almacena el currículum a utilizar. Esta iteración procede con grandes dificultades, hasta que tras una caída a valores negativos de recompensa alrededor del paso 1.4M, las recompensas aumentan de manera sostenida. Es a partir de este punto que los agentes descubren la posibilidad de ir hacia los recorridos de otros agentes para evitar sus propios obstáculos, lo cual se achaca a un fallo en el diseño del entorno, puesto que aunque sea una solución que demuestra inteligencia no es válida en el contexto del reto planteado, que en este momento es superar las vallas. Ante este comportamiento se decide detener el entrenamiento tras 32 horas completadas y hacer uso de muros en el escenario que dividan los entornos individuales.

Tras la colocación de estos muros, tal y como se puede apreciar en la figura 4.5, se da paso a los entrenamientos WalkerFence1, WalkerFence2, y WalkerFence3. WalkerFence1 es descartado por un error humano en el comienzo de la ejecución. WalkerFence2 realiza con éxito trescientos mil (300k) pasos de entrenamiento en poco más de 6 horas, pero después queda en un estado similar al que se vio previamente en WalkerTwo y WalkerThree, con el agente incapaz de aprender. Lo mismo se encuentra con el intento siguiente, WalkerFence3, que “muere” esta vez después de casi 6 horas tras realizar doscientos mil (200k) pasos.

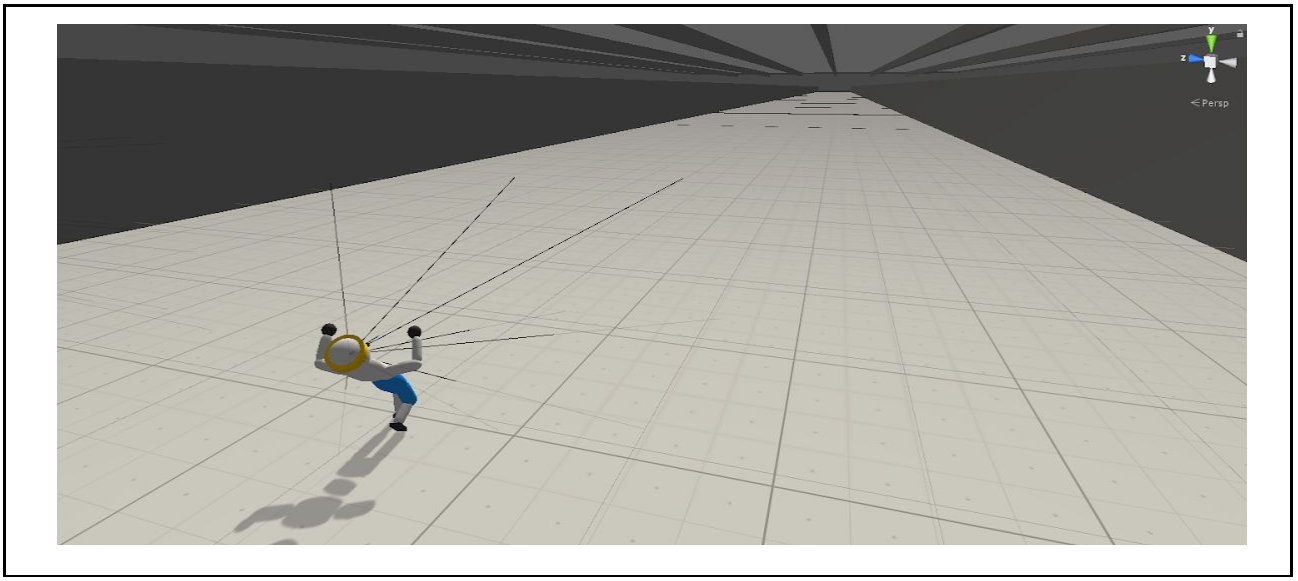


Figura 4.5: Captura del comienzo del entrenamiento WalkerFence4.

Para el entrenamiento WalkerFence4, en vista de los últimos datos recibidos, se opta por simplificar la matriz de rayos del agente, dejándola en tres filas con tres rayos cada una. Con esto se pretende reducir el coste computacional del entrenamiento, en espera de obtener mejores resultados. El entrenamiento se ejecuta de forma satisfactoria, con resultados mejores que sus predecesores de la misma fase, pero “muere” tras 38 horas en el paso 1.7M del entrenamiento. Con tres entrenamientos fallidos de esa manera, se plantea una simplificación del problema. Así se crean dos modificaciones, WalkerFarFence y WalkerFenceShort.

WalkerFarFence es un intento de solucionar el problema desde la perspectiva de que éste surja porque el Agente no es capaz de afrontar los obstáculos sin un buen entrenamiento previo en liso. Para ello podría adaptarse el currículum de manera que la altura de los obstáculos fuese igual a cero, pero el objeto seguiría existiendo en un plano bidimensional y el agente podría detectarlo, por lo que se elimina la generación de los primeros obstáculos, dándole más espacio en el que no recibe información referente a ellos que pueda afectar a su entrenamiento.

WalkerFenceShort en cambio parte de la base de que el entrenamiento pueda verse afectado por su longitud, establecida en diez millones de pasos, y las consecuencias que esta genera. Por ello, el único cambio que incluye es una modificación de la configuración para realizar un entrenamiento de cinco millones de pasos.

En este punto, dado que es necesaria la ejecución de varios entornos a la vez con distintas configuraciones, se compilan los entornos a través de la opción de compilación de Unity y se cambia el comando de ejecución para añadir las opciones de entorno y puerto. Con la opción de entorno “env” se indica la dirección de la compilación del entorno que se quiera utilizar, y con la opción de puerto “base-port” se le da a la consola un puerto en el que establecer la conexión con el entorno Unity ejecutado. Esto es necesario debido a que una ejecución en el editor no es compatible con entrenamientos paralelos, y el puerto en el que se realizan las conexiones es normalmente el 5005, por lo que distintas ejecuciones sin una opción base-port debidamente configurada hacen uso del mismo puerto y fallan automáticamente.

Los primeros dos intentos de WalkerFarFence (WalkerFarFence0 y WalkerFarFence1) fallan de la misma manera que los entrenamientos anteriores tras 700k pasos y 17 horas, y 850k pasos y 20 horas, respectivamente. WalkerFarFence2 por su parte aguanta hasta los 1.8M de pasos con 41 horas de entrenamiento, pero sufre lo mismo que el resto de intentos. WalkerFenceShort0 cae de la misma manera tras 1.5M de pasos en 35 horas.

4.1.5 Conclusiones

Por motivos de tiempo para la entrega de este proyecto, no pudo encontrarse una solución a los problemas encontrados en las últimas etapas de entrenamiento del Walker. Aún habiendo buenos resultados inicialmente, en el momento en el que los obstáculos entran en juego el proyecto deja de avanzar como se espera, y entra en un intento constante por encontrar una solución en el que se puede perder mucho tiempo.

Tras la redacción de todo el proceso de desarrollo y estudio de este entorno surgen nuevas ideas como mejoras en el aprendizaje por currículum de manera que los agentes, preparados para la detección de obstáculos, entrenen en el entorno básico con la configuración básica y, una vez hayan aprendido a correr, se introduzcan los obstáculos y muros en el entorno.

Queda claramente un gran vacío en este trabajo al no conocer la causa real de las pérdidas catastróficas que sufren muchos de los entrenamientos, comparadas con los exitosos entrenamientos BasicWalker y WalkerRay en la figura 4.6, pero no puede hacerse más que teorizar al respecto, y es que aunque ocurran grandes pérdidas en el entrenamiento, sea por el ratio de aprendizaje o por otro motivo, en teoría el algoritmo de entrenamiento debería ser capaz de salir de ahí, aunque en el proceso reflejase la inestabilidad vista en la figura 4.4. Queda la posibilidad de que estos problemas tengan raíz en el hecho de que ML-Agents es aún una tecnología en desarrollo, pero no puede saberse con seguridad sin reproducir los mismos entrenamientos en una versión final de la herramienta. También puede deberse a la configuración propia del Walker como modelo de prueba, y es que con las nuevas versiones de la herramienta se han ido sucediendo varios arreglos en muchos de los escenarios de prueba, pero es algo que no puede saberse con seguridad hasta que no sea mencionado oficialmente por el equipo de desarrollo.

Por lo pronto, lo opción más atractiva para intentar abordar estos problemas en proyectos futuros es realizar un estudio más profundo centrado en los hiperparámetros de los entrenamientos.

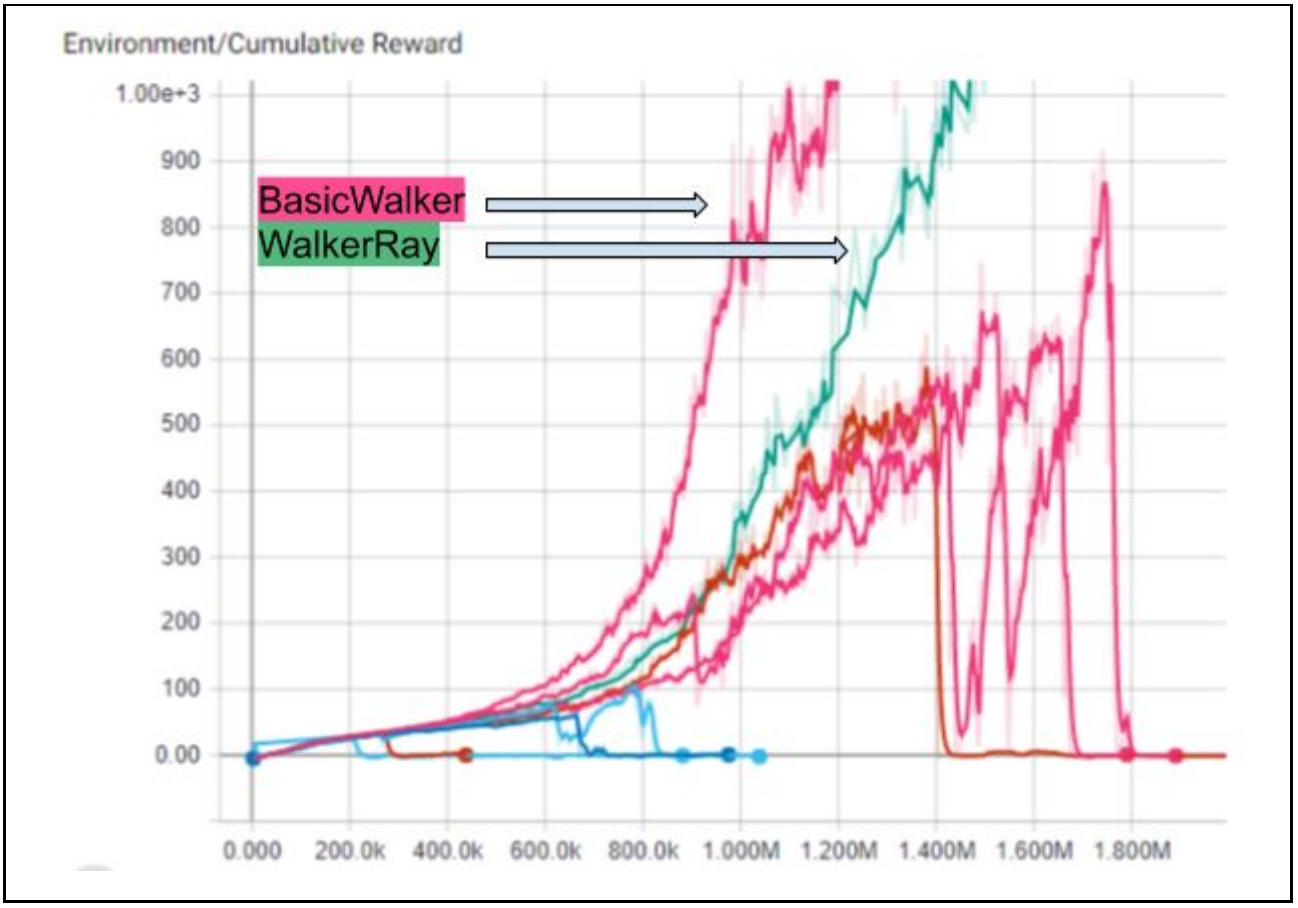


Figura 4.6: “Muertes” de distintos entrenamientos comparadas con los comienzos de BasicWalker y WalkerRay.

4.2 Cubo

El entorno Cubo surge de la necesidad de experimentar en un entorno más simple y ágil que Walker, el cual tarda varios días en completar algunos de sus entrenamientos. Cubo es creado a partir del tutorial de creación de nuevos entornos de aprendizaje presente en la documentación de ML-Agents. [16]

4.2.1 Etapas tempranas

Las primeras etapas del entorno Cubo representan situaciones de extremada simpleza. Cubo comienza con un agente cuyo modelo es una bola de un metro de diámetro que para ser recompensado ha de alcanzar un objetivo, el cual cambia de posición cada vez que es alcanzado. Este objetivo es representado con un cubo de 1x1x1m.

Inicialmente el agente recibe como observaciones su posición en el entorno, la posición del objetivo, y su velocidad en los ejes X y Z. Sus acciones se limitan a modificar su velocidad en estos mismos ejes, lo cual se realiza aplicando una fuerza externa sobre el modelo del agente. El agente únicamente será reiniciado cuando se caiga de la plataforma, cuyas medidas son 10x10 metros.

El primer entrenamiento, Roller, muestra buenos resultados, con un desempeño casi perfecto tras realizar 50k pasos en cinco minutos. Demuestra capacidad para no salir de la plataforma y es capaz de desplazarse hacia el objetivo independientemente de la posición de ambos. Ante esto se plantea la posibilidad de que la tarea pueda ser demasiado sencilla, además de no extrapolable, con el agente conociendo la posición exacta del objetivo en todo momento, por lo que las observaciones se modifican para eliminar la percepción de posiciones y se añade una visión por rayos muy similar a la del entorno Walker, con dos filas de rayos con los que el agente pueda detectar los elementos frente a él.

El entrenamiento realizado a continuación, RollerRay, falla por completo, manteniéndose la recompensa media obtenida en valores equivalentes a los del comienzo de la ejecución en todo momento. Esto es debido a que no se ha tenido en cuenta la naturaleza del modelo del agente al añadir la visión de rayos, puesto que al recibir fuerza la bola rueda y los rayos rotan junto al modelo, lo cual hace que la solución aportada sea completamente inefectiva.

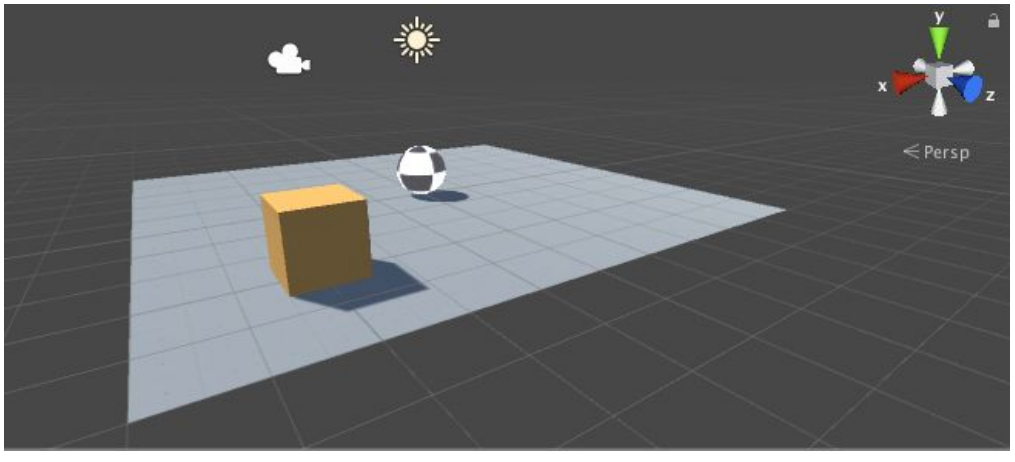


Figura 4.7: Entorno Cubo en su etapa más básica. [16]

4.2.2 Cambio de modelo y optimización de la visión

Ante los resultados de la primera etapa, se reemplaza el modelo del agente por un cubo, pero sigue existiendo el mismo problema, y es que la fuerza aplicada hace volcar al modelo. Se intenta solucionar mediante el bloqueo de la rotación respecto a los ejes X y Z, pero aunque se resuelve el problema inicial, el agente sigue sin aprender durante los entrenamientos. La duración de éstos se extiende a 500k pasos, dado el aumento en la complejidad de la tarea, y esto extiende el tiempo de entrenamiento a una media de 55 minutos.

La imposibilidad de aprender ocurre en este caso porque el agente no obtiene suficiente información del entorno como para generar una estrategia de control viable. Se realizan varios intentos dándole al agente la capacidad de rotar en el eje Y para que pueda hacer un mejor uso de los rayos, pero no resultan fructíferos, muy posiblemente porque se pasa por alto darle información al agente acerca de su rotación. Mayor experimentación es requerida en este punto.

Descartando la rotación, se procede a aumentar el rango de visión del agente, haciendo que tenga una visión de 360 grados. Esto se consigue mediante la adición de nuevos rayos junto a una redistribución de los ya existentes, quedando en cada fila 8 rayos con espacios de 45 grados entre ellos. Con estos cambios se logran por fin buenos resultados en un entrenamiento con percepción de rayos, pero la comparación de este entrenamiento, CubeRay5, con Roller muestra que hay bastante espacio de mejora. Como la visión de 360 grados ha resultado ser de gran utilidad, se duplican los rayos existentes, dejándolos en una distribución de un rayo cada 22.5 grados, para un total de 16 rayos.

Esta nueva distribución aporta grandes beneficios al entrenamiento CubeRay8, que mejora con mayor rapidez y alcanza valores casi tan altos como los de Roller, permitiendo avanzar en el desarrollo del entorno hacia un reto más complejo.

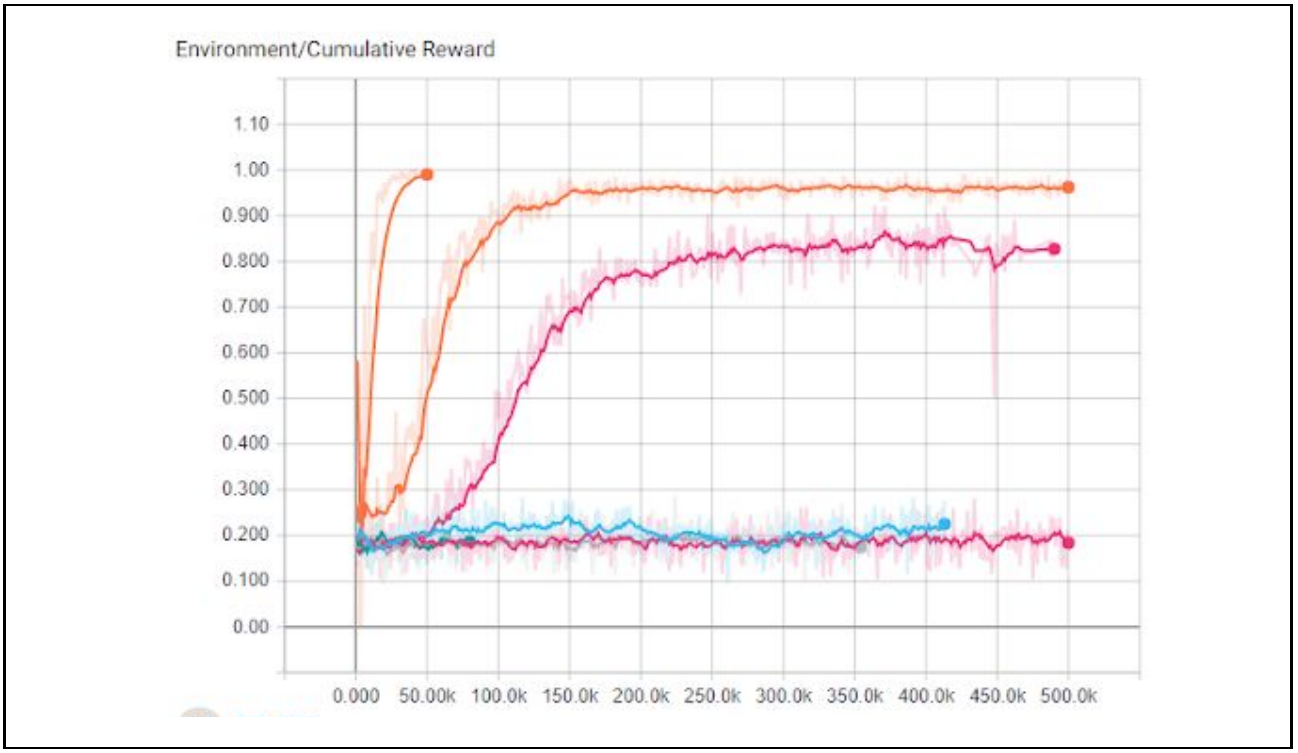


Figura 4.8: En orden de izquierda a derecha: Roller, CubeRay8 y CubeRay5, frente a los intentos fallidos

4.2.3 Entrenamiento en Pasarela

Habiendo conseguido una implementación exitosa de la visión por rayos en el entorno Cubo, se decide implementar aprendizaje por currículum para que la posición del objetivo sea cada vez más lejana. Para poder hacer esto se aumenta el tamaño de la plataforma, que pasa a medir 110 metros de largo por 10 de ancho.

Antes de avanzar más en el desarrollo, se realiza un entrenamiento de control CubePlatform1 con el objetivo en distancias cercanas. Con la idea de implementar obstáculos en un futuro, se cambia el valor de la recompensa otorgada por alcanzar el objetivo, y es por ello que este entrenamiento adicional se considera necesario. Así, los entrenamientos posteriores pueden tener un rendimiento de referencia.

El currículum se configura de manera que hay cinco lecciones. En la primera el objetivo tiene como centro de aparición, pues sigue siendo un objetivo dinámico, un punto a 10 metros de distancia. Tras completar el 20% del entrenamiento, es decir 100k pasos, el objetivo se aleja hasta los 25 metros, y lo mismo ocurre para el 40, 60 y 80%, con distancias de 50, 75 y 100 metros respectivamente. Entrenando con esta configuración, CubePlatform3, se aprecia inicialmente un rendimiento parejo al de CubePlatform1, pero en el paso 100k, coincidiendo con el primer cambio de lección, la recompensa obtenida disminuye. Se recupera valor de recompensa, pero no se llega a igualar la puntuación obtenida en el entrenamiento de control y pronto se realiza un nuevo cambio de lección, tras el que los valores de recompensa no hacen sino tender a la baja durante el resto del entrenamiento, especialmente tras completarse el 60% del entrenamiento, momento en el que el objetivo comienza a aparecer a 75 metros del agente, fuera del alcance de su percepción. Igualmente, se considera el entrenamiento como exitoso y se decide integrar una generación de obstáculos similar a la vista en el entorno Walker.

Esta vez los obstáculos se plantean como cubos iguales a los modelos del agente y el objetivo, con la diferencia de que poseen la etiqueta “wall”. Inicialmente se programa su generación de manera que sean generados en cualquier posición dentro de la plataforma y vaya apareciendo un mayor número de obstáculos a medida que el entrenamiento progresa, pero pronto se demuestra que esta aproximación causa graves problemas. Esta generación de obstáculos satura partes de la plataforma, cortándola en algunas zonas y generando escenarios que son imposibles de resolver para el agente. La única forma de solucionar este problema, sin tener que asegurar mediante algoritmos como el A* que existe un camino viable, es regular la generación de los obstáculos.

Como arreglo a la generación de obstáculos se hace que éstos hayan de aparecer de manera ordenada, colocándose uno cada X metros hasta llegar a la altura del objetivo, siendo X una variable controlada desde currículum. La distancia de generación es, al igual que con el objetivo, un punto desde el cual pueden generarse con un pequeño rango de aleatoriedad. Inicialmente se entrena a los agentes con generación cada 0 metros, es decir desactivada, en la primera lección, y 5 metros a partir de la segunda.

Los primeros entrenamientos con este modelo de generación acaban mal debido a que los agentes no tienen tiempo de aprender a afrontar los obstáculos, por lo que se cambia la duración del entrenamiento a un millón (1M) de pasos. Con el objetivo de que el agente tenga tiempo suficiente de aprender sin que los cambios de lección arruinen el entrenamiento, se prueba un avance de lección mediante recompensa en el currículum, según el cual las lecciones únicamente avanzan cuando el agente logra obtener una recompensa con un valor de 2000 o más de forma estable.

Se ejecuta el entrenamiento CubeObstacles7 con esta configuración, que termina el millón de pasos en dos horas. El primer cambio de lección sucede cerca del paso 100k, y a partir de ese momento va aumentando progresivamente el valor de recompensa, aunque nunca llega a la recompensa suficiente como para cambiar de lección de nuevo. En este momento surge la idea de realizar un cambio en los entrenamientos e integrar los obstáculos desde la primera lección, con la hipótesis de que el encuentro temprano del agente con los obstáculos supondría una mejor adaptación para las lecciones más avanzadas. El currículum vuelve a avanzar por progreso.

Con esta nueva configuración se realiza el entrenamiento CubeObstacles10, que efectivamente demuestra una adaptación mucho mejor a los obstáculos. Esto puede verse en la caída de las recompensas cerca del paso 100k con el primer cambio de lección, y es que ésta es más suave que la de CubeObstacles7, y no sólo eso, sino que el aprendizaje en la nueva lección es más rápido.

Observando una tendencia a mejorar en la tercera lección, se decide aplazar la cuarta lección hasta el final del entrenamiento, eliminando la última lección temporalmente. Los resultados obtenidos en este entrenamiento, CubeObstacles12, muestran en la zona extendida una mejora lineal normal en cualquier aprendizaje por refuerzo cerca de terminar, pero, sin importar esta mejora, en el momento de avanzar de lección los resultados caen a valores similares a los de CubeObstacles10.

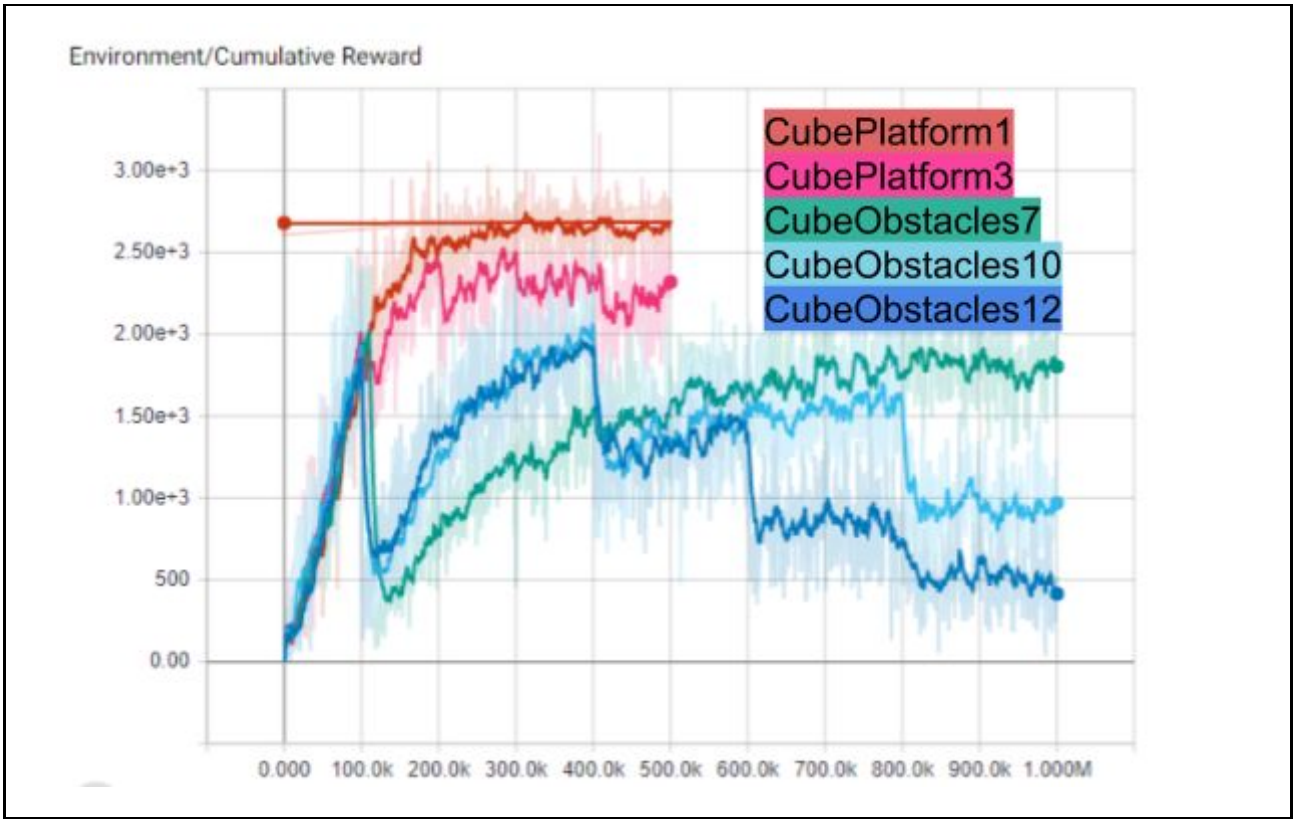


Figura 4.9: Comparación de CubePlatform 1 y 3, y CubeObstacles 7, 10 y 12
La línea sobre CubePlatform1 se debe a un error en los registros.

Dada la mejora en los resultados observada previamente al aumentar los pasos máximos del entrenamiento, se decide hacer una última ronda de entrenamientos que alcancen los dos millones (2M) de pasos. El primero de estos entrenamientos es CubeObstacles13, que obtiene con el entrenamiento adicional una clara ventaja respecto a CubeObstacles10, acorde con la evolución seguida por CubeObstacles12 en la tercera lección, pero al igual que el resto de entrenamientos, deja de mejorar en cuanto el objetivo sale de su campo de percepción. Para intentar mejorar los resultados se efectúa un cambio en el currículum que aumenta el número de lecciones a 10. Estas 10 lecciones se suceden de manera que, a partir del paso 100k, la distancia de generación del objetivo aumenta en 10 metros cada 200k pasos. Con un aumento más paulatino de la distancia entre el objetivo y el agente, se espera un aprendizaje más sólido.

Desde que se implementan los obstáculos, éstos cambian de posición en cada paso con el objetivo de entrenar agentes que puedan valerse en cualquier entorno de características similares, pero se desea comprobar el desempeño de un agente a través de un entorno estático, de manera que se desactiva la función de actualización de obstáculos y se ejecuta el entrenamiento CubeStaticObs. Resulta interesante comparar sus resultados con los de otros entrenamientos, y es que aunque alcanza picos de recompensa récord entre estos entrenamientos, sufre enormemente ante un cambio de entorno, cayendo muy por debajo del resto de entrenamientos. Esto demuestra la importancia de los entornos dinámicos para asegurar el aprendizaje de una tarea, y no de un único caso, pues sin ellos no se aprende a generalizar.

Volviendo al desarrollo principal, se incrementa el número de capas ocultas de la red neuronal del cerebro de 2 a 3 capas, y se le dobla el número de nodos, pasando de 128 a 256 nodos. De esta manera se espera una mejor adaptación al entorno dinámico en el que se está entrenando.

El primer entrenamiento con esta configuración, CubeExtraLayer, rompe todos los resultados anteriores, superando incluso algunos de los picos comentados en CubeStaticObs. En un intento por empujar un poco más hacia resultados mejores, se implementa en el agente un script de contacto con suelo de Unity, modificado para ser usado como controlador del contacto con obstáculos. Los resultados de entrenar un agente que detecta contacto con los cubos y puede continuar, CubeDetectObs, y uno que al detectarlos debe reiniciarse, CubeStrictObs, son prácticamente idénticos a CubeExtraLayer, como puede verse en la figura 4.11.

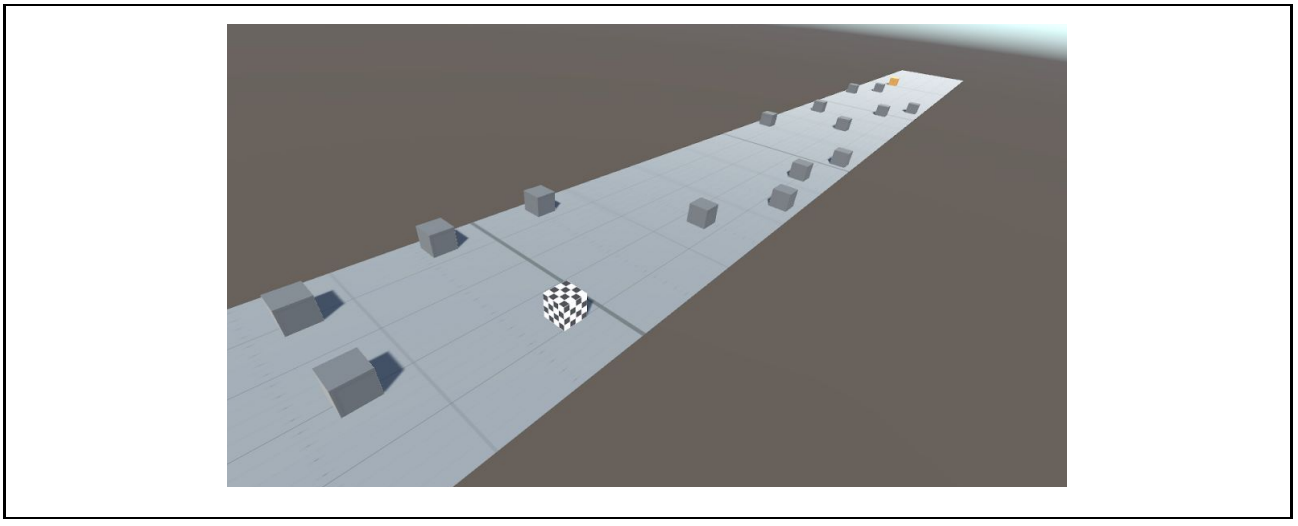


Figura 4.10: Ejecución de CubeDetectObs

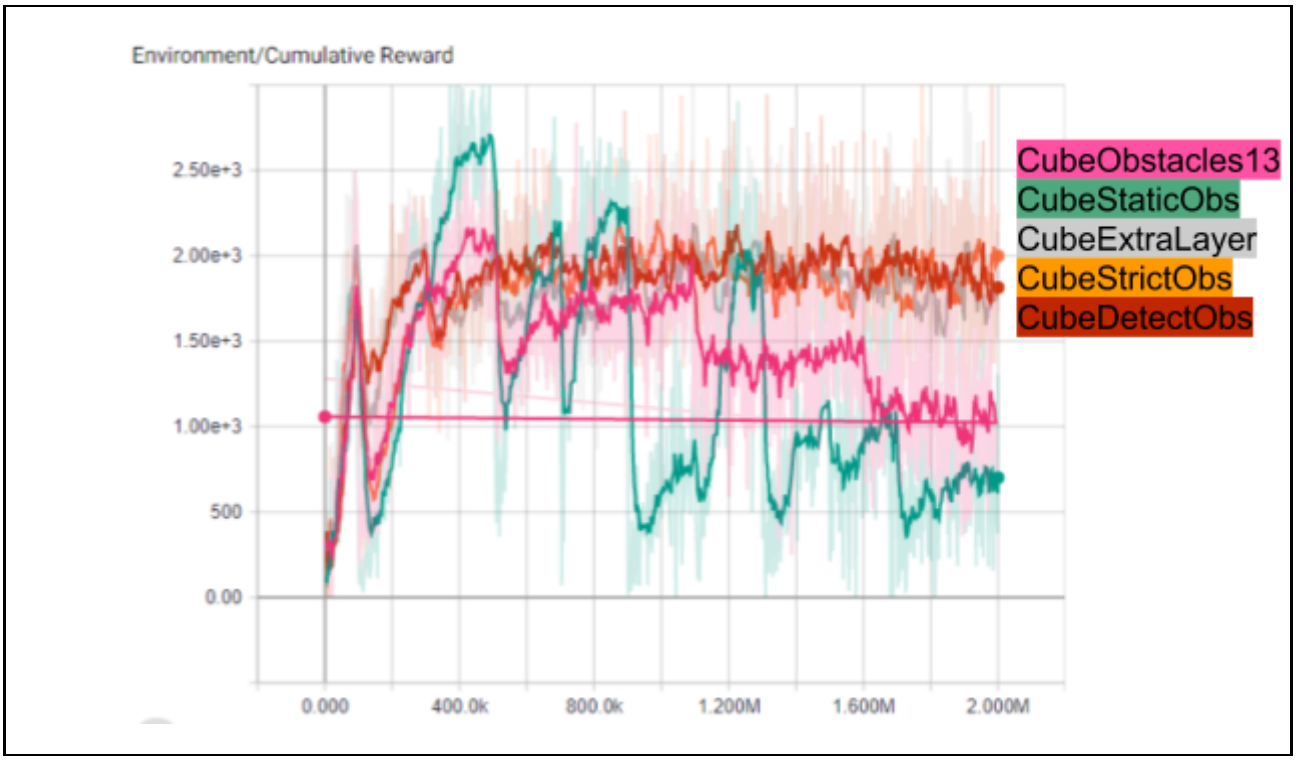


Figura 4.11: Comparación de CubeObstacles13, CubeStaticObs, CubeExtraLayer, CubeStrictObs y CubeDetectObs.
La línea bajo CubeObstacles13 se debe a un error en los registros.

4.2.4 Conclusiones

Si bien los resultados del primer entrenamiento demostraron perfección en la ejecución de las acciones del agente, en cuanto se acercan sus observaciones a una situación más real, en este caso mediante la retirada de su “omnisciencia” para conocer posiciones absolutas y la adición de un sistema de visión, los entrenamientos se convierten en retos mucho más grandes.

El mayor problema está en el hecho de que durante gran parte del desarrollo se subestiman las necesidades de observación del agente. Así es que éste va prácticamente a ciegas buscando el cubo una vez entra en las lecciones que lo sitúan fuera de su campo de percepción. Además, se intentó desarrollar un entorno dinámico, pero bajo el objetivo y los obstáculos en constante cambio hay un problema que no se ha tenido en cuenta: la plataforma. Tal y como está diseñado el entorno, el cubo siempre deberá buscar el cubo yendo en la misma dirección. Esto lleva a reforzar políticas de control que favorecen demasiado el desplazamiento hacia delante, aunque CubeDetectObs logra realizar pequeñas maniobras para continuar buscando el objetivo tras chocar con un cubo.

Posibles mejoras de este parte del proyecto incluirían la adaptación del entorno para la búsqueda del objetivo en un plano cuadrado que se extienda alrededor del agente, como el escenario inicial, pero a una escala mayor, aunque el algoritmo PPO no arroja muy buenos resultados sobre este tipo de problemas, como puede verse en los resultados del entorno de prueba Pyramid presentado en el trabajo de investigación [12] publicado por Unity acerca de ML-Agents. También se plantea la implementación de aprendizaje por imitación para que el agente aprenda de un jugador humano.

Capítulo 5
Conclusiones y líneas futuras

Esta es claramente una tecnología en desarrollo, aunque los avances se suceden de una manera increíblemente rápida, y es muy probable que Unity sea el entorno en el que se realicen los próximos grandes logros de este campo de la inteligencia artificial. Con las mejoras continuas en el kit de herramientas ML-Agents, que aún sin salir de su fase beta ya da muy buenos resultados, los pronósticos sólo pueden ser positivos.

Tras las conclusiones vistas en las dos simulaciones realizadas, aunque no se ha logrado dominar al completo la herramienta, que como ha podido verse ofrece un abanico de posibilidades inmenso, el proyecto ha servido como introducción al mundo de la investigación del aprendizaje automático y se ha podido experimentar el impacto que pueden tener en el aprendizaje profundo por refuerzo determinados cambios en las incontables variables presentes en el entrenamiento de agentes inteligentes.

Muy posiblemente se continúe esta línea de trabajo en el Trabajo Final de Máster del nuevo Máster de Desarrollo de Videojuegos de la Universidad de La Laguna con la implementación de entornos más complejos y modelos más realistas utilizando los conocimientos obtenidos a lo largo del máster, además de la experimentación con otros métodos de aprendizaje como el aprendizaje por imitación, indagando más en el funcionamiento de los hiperparámetros.

Capítulo 6
Summary and Conclusions

This is clearly a technology in development, although advances are taking place at an incredibly fast pace, and it's quite probable that Unity will become the environment in which the next big achievements in this area of research will happen. With the continuous enhancements in the ML-Agents toolkit, which even without having left its beta stage is giving very good results, forecasts can only be positive.

After the conclusions seen in the two simulations developed, although complete mastery of the toolkit hasn't been achieved, toolkit which as it has been seen offers an immense range of possibilities, the project has been useful as an introduction to the world of machine learning research and it's been possible to experiment the impact that certain changes in the countless variables present in the training of intelligent agents can have in deep reinforcement learning.

This line of work will quite possibly continue in the Final Master's Project of the new master's degree in Video Game Development of the University of La Laguna, with the implementation of more complex environments and more realist models using the knowledge acquired throughout the degree, in addition to experimentation with other learning methods such as imitation learning, going deeper into the theory behind hyperparameters.

Capítulo 7
Presupuesto

Para el presupuesto del proyecto se toma el sueldo de un investigador junior de 816 euros al mes y una aproximación de 200 horas de trabajo, que son el equivalente a 25 jornadas laborales. Distribuidas en semanas con 5 jornadas laborales cada una, completan 5 semanas, que sería el equivalente a poco más de un mes de investigación y desarrollo, aproximadamente 1000 euros. Las tecnologías utilizadas son todas gratuitas y en su mayoría de código abierto. Sumando el coste de un equipo con potencia suficiente como para trabajar ágilmente tanto en el desarrollo como en los entrenamientos, lo cual ronda los 1600 euros según los componentes escogidos, aproxima los costes del proyecto a 2600 euros.

Bibliografía

[1] Samuel, Arthur L. "Some studies in machine learning using the game of checkers." *IBM Journal of research and development* 44.1.2 (1959): 206-226.

http://www.cs.virginia.edu/~evans/greatworks/samuel1959.pdf

[2] Ayodele, Taiwo Oladipupo. "Types of machine learning algorithms." *New advances in machine learning*. IntechOpen, 2010.

http://cdn.intechweb.org/pdfs/10694.pdf

[3] Jordan, Michael I., and Tom M. Mitchell. "Machine learning: Trends, perspectives, and prospects." *Science* 349.6245 (2015): 255-260.

http://www.cs.cmu.edu/~tom/pubs/Science-ML-2015.pdf

[4] Michie, Donald. "Experiments on the mechanization of game-learning Part I. Characterization of the model and its parameters." The Computer Journal 6.3 (1963): 232-236.

https://academic.oup.com/comjnl/article-pdf/6/3/232/1030939/6-3-232.pdf

[5] Tesauro, Gerald. "TD-Gammon, a self-teaching backgammon program, achieves master-level play." Neural computation 6.2 (1994): 215-219.

https://www.aaai.org/Papers/Symposia/Fall/1993/FS-93-02/FS93-02-003.pdf

[6] Chouard, Tanguy. "The Go Files: AI computer wraps up 4-1 victory against human champion." Nature News (2016).

https://www.nature.com/articles/550336a

[7] Silver, David, et al. "Mastering the game of go without human knowledge." Nature 550.7676 (2017): 354.

https://www.nature.com/articles/nature24270?sf123103138=1

[8] Bellemare, Marc G., et al. "The arcade learning environment: An evaluation platform for general agents." Journal of Artificial Intelligence Research 47 (2013): 253-279.

https://arxiv.org/pdf/1312.5602.pdf)

[9] Brockman, Greg, et al. "Openai gym." arXiv preprint arXiv:1606.01540 (2016).

https://arxiv.org/pdf/1606.01540.pdf

[10] Todorov, Emanuel, Tom Erez, and Yuval Tassa. "Mujoco: A physics engine for model-based control." 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 2012.

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.296.6848&rep=rep1&type=pdf

[11] Jabrils. “WRITING MY FIRST MACHINE LEARNING GAME! (1/4)” Youtube (2017).

https://www.youtube.com/watch?v=ZX2Hyu5WoFg

[12] Juliani, Arthur, et al. "Unity: A general platform for intelligent agents." *arXiv preprint arXiv:1809.02627* (2018).

https://arxiv.org/pdf/1809.02627

[13] Booth, Joe, and Jackson Booth. "Marathon environments: Multi-agent continuous control benchmarks in a modern video game engine." arXiv preprint arXiv:1902.09097 (2019). https://arxiv.org/ftp/arxiv/papers/1902/1902.09097.pdf

[14] Macía Varela, Francisco M. “VR Exposure Therapy for environmental stressors in autism“ University of Glasgow, Manuscript in preparation (2019).

[15] Schulman, John, et al. "Proximal policy optimization algorithms." arXiv preprint arXiv:1707.06347 (2017).
https://arxiv.org/pdf/1707.06347

[16] Unity development team “Making a New Learning Environment” ML-Agents documentation (2019)
https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Create-New.md