



viu

**Universidad
Internacional
de Valencia**

Aprendizaje por refuerzo en sistemas mono y multiagente

Titulación:
Máster U. en
Inteligencia Artificial

Curso académico
2021-2022

Alumno/a:
Valderico Carratala Rizzo
D.N.I: 48792178R

Director/a de TFM:
Gabriel Enrique Muñoz Ríos

Convocatoria:
Segunda

Tipo de TFM:
Académico/investigación



Universidad
Internacional
de Valencia

Aprendizaje por refuerzo en sistemas mono y multiagente:

Análisis e implementación de diferentes sistemas de Deep RL sobre el framework de Unity

Autor

Valderico Carratalá Rizzo

Tutor/es

Gabriel Enrique Muñoz Ríos

Lead Data Scientist



Master en Inteligencia Artificial

ALICANTE, Septiembre 2022

"Nuestras virtudes y nuestros defectos son inseparables, como la fuerza y la materia. Cuando se separan, el hombre ya no existe".

Nikola Tesla.

Agradecimientos

Este trabajo no se habría podido llevar a cabo de no haber sido por la entera disposición de mi tutor Gabriel Enrique Muñoz Ríos, bajo cuya supervisión escogí el tema de este trabajo de fin de máster, y gracias al cual pude ir avanzando a lo largo de las diferentes etapas de este proyecto.

También me gustaría agradecer la ayuda y hospitalidad de mis compañeros de clase, los cuales a lo largo del curso han servido de apoyo constante y me han motivado a seguir aprendiendo.

Estoy agradecido también a mis profesores de instituto, de la universidad y del máster, los cuales me motivaron a seguir progresando constantemente, y a llegar donde me encuentro ahora mismo.

Por último, no puedo terminar sin agradecer a mis amigos, familia y a mi pareja, los cuales me han apoyado en todo momento y cuyo estímulo constante y amor me ha servido como motor para superar los obstáculos que han ido surgiendo a lo largo de todo este trayecto.

Es a ellos a quien dedico este trabajo.

*A mis abuelos Valderico Rizzo Aguirre y M^a del Carmen de Pazos Romero,
apoyos constantes sin los cuales no habría llegado hasta aquí.*



Índice general

Índice de figuras	III
Resumen	1
1. Introducción	3
1.1. Motivación	4
1.2. Estructura de la memoria	4
2. Objetivos	5
3. Estado del arte	6
3.1. Aprendizaje por refuerzo	6
3.1.1. Clasificación de problemas de aprendizaje por refuerzo	8
3.2. Deep Reinforcement Algorithms	9
3.2.1. PPO	9
3.2.2. MA-POCA (MultiAgent POsthumous Credit Assignment)	10
3.2.3. Convolutional Neural Networks (CNN)	11
3.2.4. Imitation Learning	12
3.2.5. Curriculum Learning	14
3.2.6. Self Play	15
3.3. Proyectos recientes	16
4. Metodología y entorno de trabajo	17
4.1. Configuración del comportamiento	17
4.1.1. Fichero de configuración YAML	18
4.2. Diseño de los entornos en Unity	21
4.2.1. Configuración del proyecto de Unity	21
4.2.2. Creación de un entorno de ejemplo	21
4.2.3. Implementación y diseño del agente	24
4.2.4. Configuración de las propiedades del Agente	27
4.2.5. Configuración del script del Agente	28



4.2.6. Configuración final del agente en el editor	29
4.2.7. Definición de escenarios multiagente	31
5. Experimentación	33
5.1. Diseño del agente final	33
5.1.1. Sistema de Checkpoints	37
5.2. Entornos finales	38
5.2.1. Entornos monoagentes	39
5.2.2. Entornos multiagentes	51
6. Resultados y Conclusiones	65
7. Limitaciones y Perspectivas de Futuro	67
A. Apéndize	71
A.1. Repositorio Github	71
A.2. Vídeo de demostración	72
A.3. Instalación y requisitos	73
A.3.1. Creación de un environment de conda	73
A.3.2. Instalación de Unity y ml-agents	73
A.3.3. Fichero de configuración YAML en el que se incluyen todas las configura- ciones posibles usando un entrenador PPO	75
A.4. Ejecución entrenamiento	77
A.4.1. Fichero de configuración YAML en el entorno de ejemplo mencionado en la metodología	77
A.5. Análisis del entrenamiento	79
A.6. Implementación del entrenamiento	80
A.7. Experimentación: 'Ficheros YAML'	82
A.7.1. Fichero de configuración YAML utilizado para resolver los entornos mono- agente y multiagentes	82
A.8. Imágenes de los códigos utilizados en el entorno de ejemplo mencionado en la metodología	84
A.9. Código del sistema de checkpoints utilizado	86
Bibliografía	89

Índice de figuras

3.1. Esquema básico del funcionamiento de RL.	7
3.2. Definición recursiva de Q mediante la ecuación de Bellman.	7
3.3. Taxonomía de algoritmos en RL.	8
3.4. Función de coste basada en Clipped surrogate objective.	10
3.5. Pseudocódigo del algoritmo PPO	10
3.6. Pseudocódigo del algoritmo MA-POCA	11
3.7. Comparativa del proceso de entrenamiento utilizando técnicas de Imitation learning.	13
3.8. Ejemplo de Curriculum Learning.	15
3.9. Proyecto multiagente realizado por OpenAI	16
4.1. Ventana Proyecto de Unity.	21
4.2. Creación del entorno en Unity: Suelo.	23
4.3. Creación del entorno en Unity: Target.	23
4.4. Creación del entorno en Unity: Agente.	23
4.5. Creación del entorno en Unity: Entorno de ejemplo.	24
4.6. Creación del entorno en Unity: Agrupar los componentes.	24
4.7. Configuración de las propiedades del Agente desde el editor de Unity.	27
4.8. Configuración final del agente.	30
4.9. Configuración del Id para entornos adversariales.	31
5.1. Modelo de conducción del robot Turtlebot3.	34
5.2. Modelo de conducción del Agente Final.	34
5.3. RayPerceptionSensorComponent3D nº1.	35
5.4. RayPerceptionSensorComponent3D nº2	35
5.5. CameraSensor (21X21 px).	35
5.6. Configuración final de los parámetros del agente.	36
5.7. Sistema de Checkpoints: Imagen.	37
5.8. Sistema de Checkpoints: Mensajes durante la ejecución.	37
5.9. Sistema de Checkpoints: Agrupación de checkpoints en el editor de Unity.	38
5.10. Sistema de Checkpoints: Configuración desde el editor de Unity.	38
5.11. Entorno monoagente 1: Climb.	39

5.12. Curriculum Learning en Entorno monoagente 1: Climb.	40
5.13. Resultados Entorno monoagente 1: Climb.	43
5.14. Resultados del Curriculum learning en el Entorno monoagente 1: Climb.	43
5.15. Entorno monoagente 2: Push.	44
5.16. Curriculum Learning en Entorno monoagente 2: Push.	45
5.17. Resultados Entorno monoagente 2: Push.	47
5.18. Resultados Entorno monoagente 2: "Push".	47
5.19. Entorno monoagente 3: Bridge.	48
5.20. Curriculum Learning en Entorno monoagente 3: Bridge.	49
5.21. Resultados Entorno Bridge 3: Bridge.	51
5.22. Resultados Entorno Bridge 3: Bridge.	51
5.23. Entorno multiagentes 1: Climb.	52
5.24. Gráficas del entrenamiento de los entornos multiagente 1: Climb.	54
5.25. Comparación del resultado entre el entorno multiagente y los entorno monoagente Climb	54
5.26. Entornos multiagente 2: Push.	55
5.27. Gráficas del entrenamiento de los entornos multiagentes 2: "Push".	57
5.28. Resultado del entrenamiento de los entornos multiagentes 2: "Push".	57
5.29. Comparativa del entrenamiento de los entornos multiagente 2: Push, iniciando el entrenamiento desde 0 y partiendo de un entrenamiento anterior.	58
5.30. Resultado del entrenamiento de los entornos multiagente 2: Push, iniciando el en- trenamiento desde 0 y partiendo de un entrenamiento anterior.	58
5.31. Entorno multiagente 3: Bridge.	59
5.32. Resultado de las pruebas sobre el entorno multiagente 3: Bridge.	62
5.33. Resultado de las pruebas sobre el entorno multiagente 3: Bridge.	62
5.34. Resultado de las pruebas finales sobre el entorno multiagente 3: Bridge.	63
5.35. Mejor recompensa y tiempo medio por episodio en el entorno multiagente 3: Bridge. .	63
A.1. Ventana del administrador de paquetes.	74
A.2. Selección el archivo package.json.	74
A.3. Ejecutar el entrenamiento en la terminal.	78
A.4. Ejemplo de Tensorboard.	80
A.5. Implementación del modelo ya entrenado.	81
A.6. Entorno de ejemplo en modo de inferencia.	81
A.7. Fichero YAML para entornos monoagente (PPO).	82
A.8. Fichero YAML para entornos multiagentes (MA-POCA).	83
A.9. Parte del fichero YAML con Curriculum Learning para el Entorno monoagente 1: Climb.	83
A.10. Código para obtener la altura al utilizar Curriculum learning.	84
A.11. Fichero YAML para entornos multiagentes con Self-Play.	84
A.12. Código del agente: OnEpisodeBegin().	85

ÍNDICE DE FIGURAS

A.13. Código del agente: CollectObservations().	85
A.14. Código del agente: OnActionReceived().	86
A.15. Sistema de Checkpoints: Función CheckpointsMulti().	86
A.16. Sistema de Checkpoints: Función TrackCheckpoints().	87

Resumen

La implementación de sistemas inteligentes en entornos físicos o donde se requiere que un objeto o agente interactúe con el entorno para llevar a cabo una tarea específica, es un problema que recientemente ha cobrado mucha importancia en los campos de la inteligencia artificial y la robótica. Este tipo de sistemas requieren de la implementación de una serie de técnicas y herramientas, las cuales permitan y faciliten la interacción entre el agente y el entorno. El creciente desarrollo de la rama del aprendizaje por refuerzo ha permitido que cada vez sea más viable el desarrollo de soluciones de este tipo, y la implementación de las mismas en entornos físicos reales. Aún así, para obtener un resultado óptimo mediante el desarrollo de este tipo de soluciones, todavía se requieren de grandes esfuerzos, tanto en términos de costes de implementación como de tiempo de ejecución, debido a que requieren de grandes cantidades de datos y/o a la implementación de mecanismos de simulación y ampliación de la experiencia. Además, generalmente también requieren de sistemas con gran capacidad de cómputo y almacenamiento, debido a la gran cantidad de cálculos y datos que se necesitan durante el entrenamiento de los agentes. Estos requisitos hacen que el desarrollo de soluciones de aprendizaje por refuerzo sea costoso y difícil de desarrollar, y solo sea factible para empresas o instituciones con grandes recursos y presupuesto.

El objetivo de este Trabajo Fin de Máster es implementar una serie de sistemas que me permita analizar la viabilidad del desarrollo de agentes inteligentes de forma fácil, rápida y a bajo coste. Para ello, se implementarán una serie de sistemas en los cuales, uno o varios agentes aprendan a realizar una tarea simple mediante la interacción directa con el entorno gracias al aprendizaje por refuerzo. Posteriormente a la implementación de estos sistemas, se procederá a analizar el comportamiento de este tipo de agentes y a comparar las diferencias en los sistemas con uno o más agentes con el fin de analizar las diferencias en la eficacia y eficiencia de los comportamientos. Para llevar a cabo este proyecto será necesario tanto el diseño de los entornos personalizados, así como el de los agentes, y la interacción entorno-agente. Finalmente, se plantea la implementación de este tipo de algoritmos en sistemas físicos reales mediante robots móviles, abarcando las posibles limitaciones que se puedan presentar.

Palabras clave: Aprendizaje por refuerzo, sistemas monoagente, sistemas multiagente, inteligencia artificial, robots móviles.

Introducción

1

Los recientes avances conseguidos en el campo del aprendizaje por refuerzo (RL) han demostrado la capacidad de este tipo de algoritmos para encontrar soluciones a problemas cada vez más complejos. Estos avances comenzaron a popularizarse en 2015, donde gracias a la combinación de algoritmos de RL con técnicas de aprendizaje profundo, la IA de Google AlphaGo ([Alp \(a\)](#)) conseguía la victoria total sobre un jugador profesional de Go, y posteriormente, contra el 18 veces campeón del mundo de ese mismo juego. Dos años más tarde, presentaron AlphaZero ([Alp \(c\)](#)), capaz de jugar tanto a Shogi como al Ajedrez, y de vencer a los mejores jugadores del mundo, y en sus próximas versiones será capaz además de resolver multitud de juegos de Atari ([MuZ](#)). Por otra parte, la empresa OpenAI también presentó en 2019 AlphaStar ([Alp \(b\)](#)), una bot capaz de jugar a nivel profesional al videojuego StarCraft 2. También es interesante hablar de la librería de OpenAI [Gym](#), la cual permite entrenar algoritmos para aprender a jugar a juegos de la Atari 2600, incluso desde equipos modestos.

El aprendizaje por refuerzo ([Sutton y Barto \(2018\)](#)) es una técnica de aprendizaje automático que se basa en la manera que tenemos los seres humanos para aprender, de modo que los agentes reciben recompensas por cada una de sus acciones, y estas serán positivas o negativas dependiendo de si cumplen o no algún objetivo. De esta manera, los agentes pasarán de comportarse de manera aleatoria, a seguir una política de acción que maximice la recompensa, y que les permita cumplir su objetivo. Esta técnica puede llegar a ser muy interesante y potente, ya que no necesita de una gran cantidad de datos para el entrenamiento, como si que ocurre en otras ramas de la IA, sino que estos datos se generan directamente de la interacción agente-entorno.

Aun así, mientras que el comportamiento del aprendizaje por refuerzo en sistemas con un solo agente está bastante avanzado y se pueden conseguir generalmente buenos resultados, en el caso de entornos con varios agentes el desarrollo de estos algoritmos está mucho más atrás. Esto se debe a que los algoritmos actuales no están pensados para entornos donde los agentes requieren de la cooperación o competición entre ellos para lograr el objetivo, y hace que el desarrollo de estos algoritmos sea mucho más complejo.

Por estas razones, el objetivo de este proyecto es estudiar los algoritmos de aprendizaje por refuerzo y ser capaz de aplicar este tipo de aprendizaje sobre diversos entornos simulados. En dichos entornos, uno o varios agentes deben ser capaces de interactuar entre sí además de con el entorno para ser capaces de aprender a resolver tareas específicas, permitiéndonos así poder estudiar estos algoritmos y analizar su posible aplicación en entornos más complejos.

1.1. Motivación

La razón por la que se ha llevado a cabo este proyecto, además del interés personal hacia la robótica, la IA y a las nuevas tecnologías en general, viene dada principalmente por el interés hacia el campo del multiagent learning en entornos dinámicos y cooperativos. Este campo presenta todavía varios problemas en los algoritmos de aprendizaje, por lo que se pretende experimentar y analizar el comportamiento de este tipo de técnicas frente a los algoritmos más clásicos utilizados para entornos monoagente, de manera que se puede entender mejor cómo funcionan, y los problemas que pueden afectar al proceso de aprendizaje.

Otra de las motivaciones es el uso de este tipo de algoritmos en el área de los videojuegos. Por ello, para el desarrollo de este proyecto utilizaremos el entorno de simulación de Unity, y su librería de aprendizaje por refuerzo 'ML-Agents' ([ML- \(a\)](#)), gracias a los cuales seremos capaces de diseñar y entrenar entornos tanto monoagentes como multiagentes, sin necesidad de una enorme capacidad computacional.

Por otro lado, parte de la motivación de este proyecto surge de un deseo personal de expandir mis conocimientos en el área de la IA, y quería aprovechar para hacer un proyecto en el que pudiera poner en práctica los conocimientos adquiridos a lo largo de estos años.

1.2. Estructura de la memoria

Este trabajo se compone de un total de 7 secciones. En esta primera sección se presenta la introducción y la motivación personal. A continuación, en la sección [2](#) se proponen todos y cada uno de los objetivos que se pretenden alcanzar en la realización de este trabajo. En la sección [3](#) se encuentra el estado del arte, donde se dará una breve explicación sobre los fundamentos del aprendizaje por refuerzo, y se expondrá su estado actual con algunos ejemplos recientes cuyos objetivos eran similares al de este proyecto. En la sección [4](#), hablaré sobre la metodología empleada para llevar a cabo este trabajo. Después en la sección [5](#), se implementan los algoritmos de aprendizaje por refuerzo utilizando diversos entornos con un solo agente y mediante sistemas multiagentes. Una vez explicado todo, en la sección [6](#) se analizarán las soluciones obtenidas, así como el efecto de la colaboración de varios agentes para mejorar el rendimiento de los algoritmos, y se establecerán las conclusiones para este trabajo. Para terminar, en la sección [7](#), expondremos las ideas de futuro para este proyecto y sus limitaciones, y se plantea la posibilidad de implementar este tipo de soluciones en sistemas físicos mediante robots móviles y la utilización de la plataforma de robótica ROS.

Objetivos

2

El objetivo general para este Trabajo Fin de Máster se basa en el deseo por profundizar en las técnicas de aprendizaje por refuerzo, y la aplicación de las mismas en el ámbito de la robótica.

Como objetivos principales de este trabajo, se plantean los siguientes:

- 1. Aplicar técnicas de aprendizaje por refuerzo tanto en entornos monoagentes como multiagentes.**
- 2. Evaluar el rendimiento de los agentes entrenados.**
- 3. Determinar las limitaciones y posibles mejoras del sistema implementado.**
- 4. Analizar el comportamiento de los agentes en los entornos simulados, y el posible uso de este tipo de técnicas en entornos reales.**
- 5. Dar mayor peso a la creación del entorno, y analizar los efectos del mismo durante el proceso de aprendizaje.**

Como motivación y objetivos secundarios para el desarrollo de este trabajo, podemos plantear los siguientes:

- 1. Profundizar en las técnicas de aprendizaje por refuerzo.**
- 2. Analizar la factibilidad del desarrollo de soluciones de aprendizaje por refuerzo sin equipos con altas capacidades computacionales.**
- 3. Mejorar el nivel del lenguaje de programación C# y en el uso de Unity, así como aprender sobre el uso de su interfaz de RL 'ML-Agents Toolkit'.**
- 4. Poner en práctica los conceptos de Aprendizaje por Refuerzo vistos en el máster así como los conocimientos previos de robótica, tales como:**
 - Importancia del entorno y las observaciones obtenidas del mismo.
 - Importancia del diseño de los agentes.
 - Importancia del diseño de una función de recompensa adecuada a la tarea.

Estado del arte

3

En este capítulo se establecerán y explicarán los conceptos necesarios para el correcto entendimiento de este proyecto. Así pues, se hablará de qué es y cómo funciona el aprendizaje por refuerzo, y se expondrán los diferentes métodos de aprendizaje que se utilizarán a lo largo del proyecto.

3.1. Aprendizaje por refuerzo

El aprendizaje por refuerzo (RL), como hemos introducido en el primer capítulo ([Sutton y Barto \(2018\)](#)) es una técnica de aprendizaje automático que se basa en el refuerzo positivo o negativo para modificar el comportamiento de un agente, lo cual proviene de la psicología. El refuerzo positivo es una técnica de enseñanza que se centra en el premio de una conducta deseada, con el fin de aumentar la probabilidad de que se repita. En contraste, el refuerzo negativo se centra en proporcionar un estímulo desagradable para disminuir la probabilidad de que se repita una conducta no deseada. Se podría decir que el objetivo del aprendizaje por refuerzo es entrenar un agente inteligente que es capaz de interactuar con un entorno de manera inteligente.

Antes de proseguir, se procederá a definir los siguientes términos:

- **Agente:** Un agente es una entidad que interactúa con elementos del entorno (que estén disponibles en el momento de la interacción) para llegar a un objetivo, el cual generalmente es maximizar una recompensa
- **Entorno:** El entorno es el mundo donde el agente vive e interactúa con los elementos de su alrededor. El entorno no sólo se compone de los elementos visuales de la simulación, si no también de la lógica que subyace y que define cómo se puntuá, tiempos disponibles, etc. Puede ser simulado, o real.
- **Señal de recompensa:** La recompensa es el feedback que nos devuelve el entorno para evaluar cómo lo está haciendo el agente.

En contraposición a otras áreas de la IA que se entrena utilizando colecciones de muestras etiquetadas, en el aprendizaje por refuerzo los agentes son capaces de aprender mediante la interacción directa con el entorno y tomando decisiones en tiempo real, generando sus propias muestras durante la interacción.

Esta interacción se repite continuamente, de manera que las interacciones se realizarán en una secuencia de pasos discretos ($t = 1, 2, 3, \dots$), y en cada paso ' t ' el agente interactúa con su entorno, observando el estado del entorno ' S_t ' y utilizando esta representación del estado para ejecutar acciones acorde a ello ' a '. Cada acción produce un refuerzo positivo o negativo ' R_t ', que se utiliza para modificar la política de acción del agente de manera que se maximice el refuerzo positivo en el tiempo. En la figura 3.1 se puede observar una representación del proceso explicado.

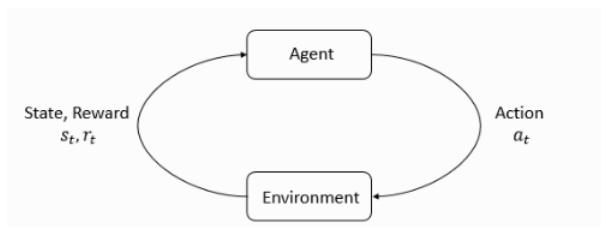


Figura 3.1: Esquema básico del funcionamiento de RL. Obtenida de [Spi \(a\)](#)

El objetivo del agente es por lo tanto aprender una política de acción ' π ' que maximice la recompensa positiva a largo plazo, lo cual en algunos casos se traduce en una función de valor de estado-acción $Q(s, a)$, que define el valor del par estado ' s ' acción ' a ' siguiendo la política ' π '. Se puede definir como el valor esperado de la suma de las recompensas futuras desde el estado actual ' s ', como las estimaciones de los siguientes estados hasta el fin del episodio, ponderándolos con la probabilidad de llegar a dicho estado desde ' s ' tomando una acción ' a '. Ya que trabajaremos con la recompensa esperada a futuro, utilizaremos la ecuación de Bellman para modelar este comportamiento, dado que proporciona una definición recursiva con la que podremos encontrar la función óptima Q .

En la figura 3.2 se puede ver la fórmula $Q^*(s, a)$, la cual es igual a la suma de la recompensa inmediata, después de ejecutar la función A en el estado S , y la recompensa futura esperada después de la transición al siguiente estado S' :

$$Q^{*}(s, a) = r + \gamma \max_{a'} Q(s', a')$$

Figura 3.2: Definición recursiva de Q mediante la ecuación de Bellman. Obtenida de [Spi \(a\)](#)

3.1.1. Clasificación de problemas de aprendizaje por refuerzo

Otra anotación que es interesante realizar, es la división de los algoritmos de RL en Model-Free y Model-Based ([Spi \(b\)](#)), la cual se puede ver en la figura 3.3. En Model-Free la simulación no conoce un modelo para predecir siguientes estados en el entorno y por lo tanto la estrategia que busca el agente es dependiente sólo del estado actual en el que se encuentra. En cambio, en Model-Based sí que conocemos un modelo de cómo se comporta el entorno en relación con las posibles acciones que el agente tome durante la ejecución. Al tener el conocimiento de este modelo, es típico combinar los algoritmos de aprendizaje con refuerzo con otras soluciones de inteligencia artificial como técnicas de planificación y búsquedas óptimas, como es el caso de AlphaGo y AlphaZero, que utilizan por ejemplo Árboles de búsqueda de Monte Carlo (MCTS).

Dentro del apartado Model-Free (que es en lo que se basa nuestro proyecto) encontramos otra división entre algoritmos basados en Policy Optimization (On-policy) y los basados en Q-Learning (Off-policy) ([Spi \(b\)](#)). Los métodos de la familia de Policy Optimization optimizan los parámetros ' θ ' ya sea directamente mediante el ascenso del gradiente, o indirectamente, maximizando las aproximaciones locales. Esta optimización casi siempre se realiza en On-policy, lo cual significa que durante el proceso de aprendizaje la estrategia que nuestros agentes siguen puede ir cambiando en el tiempo, y el agente sólo puede usar la experiencia pasada de una política específica.

En cambio, los algoritmos de Q-Learning se basan en un aproximador $Q\theta(s,a)$ para la función de valor de acción óptima, $Q^*(s,a)$. Por lo general, utilizan una función objetivo basada en la ecuación de Bellman como hemos comentado anteriormente. Esta optimización casi siempre se realiza fuera de la política, lo que significa que cada actualización puede usar datos recopilados en cualquier momento durante el entrenamiento, independientemente de cómo el agente decidiera explorar el entorno cuando se obtuvieron los datos, y por lo tanto, que se han ido generando con diferentes versiones del modelo del agente.

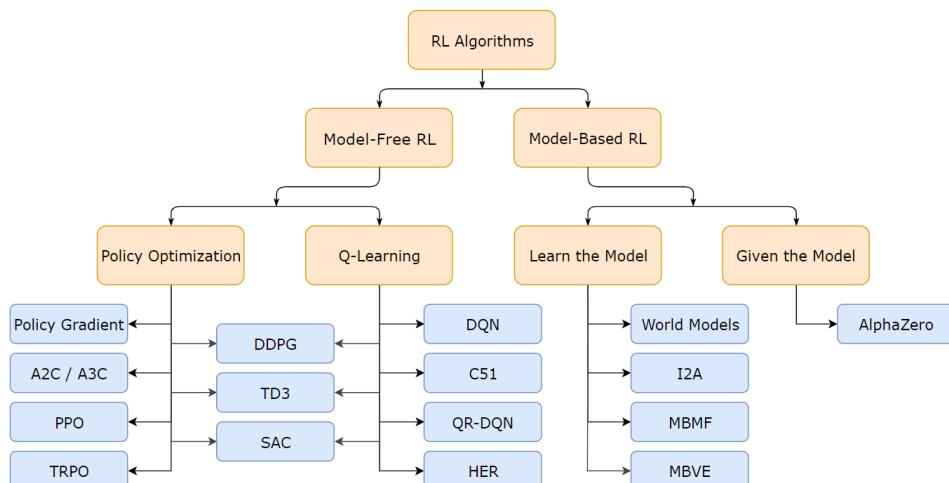


Figura 3.3: Taxonomía de algoritmos en RL. Obtenida de [Spi \(b\)](#)

Así pues, la principal fortaleza de los métodos de optimización de políticas es que se basan en principios, en el sentido de que se optimizan directamente para lo que se desea. Esto tiende a hacerlos estables y confiables, y es la principal razón por la cual hemos elegido el algoritmo de PPO (perteneciente a policy optimization) para el desarrollo de este proyecto.

3.2. Deep Reinforcement Algorithms

ML-Agents proporciona una implementación de dos algoritmos de aprendizaje por refuerzo: PPO y SAC ([Joseph y Ali \(2017b\)](#), [Tuomas Haarnoja \(2018\)](#), [ML- \(b\)](#)). Además también proporciona varios métodos para implementar técnicas de imitation learning, y presenta métodos adicionales que pueden ayudar en el entrenamiento de comportamientos para tipos específicos de entornos, como MA-POCA ([Joseph y Ali \(2017a\)](#), [MA-](#)).

A continuación vamos a explicar el funcionamiento general de cada uno de los métodos que se han probado en este proyecto.

3.2.1. PPO

Los métodos de policy gradients son fundamentales para los avances recientes en el uso de redes neuronales profundas para el control, desde juegos de mesa y videojuegos hasta locomoción 3D. Pero obtener buenos resultados a través de estos métodos es un desafío, ya que son sensibles a la elección del tamaño del batch y suelen requerir millones (o miles de millones) de pasos de tiempo para aprender tareas simples.

Estos algoritmos tienen muchas partes 'móviles' que son difíciles de depurar y requieren un esfuerzo considerable de ajuste para obtener buenos resultados. Es por ello que la idea clave en la que se centra el algoritmo de PPO es en mejorar lo máximo posible la política actual, usando la trayectoria recolectada, y controlando que el aprendizaje no colapse, calculando para ello una actualización en cada paso que minimice la función de coste mientras asegura que la desviación de la política anterior sea relativamente pequeña. Gracias a esto, PPO logra un equilibrio entre la facilidad de implementación, la complejidad de las muestras y la facilidad de ajuste.

La función que se utiliza para controlar el cambio de la política en cada iteración (Fig. 3.4) es una función de coste basada en Kullback-Leibler divergence, o basada en Clipped surrogate objective. Estas dos funciones controlan que la actualización de la política esté dentro de estos rangos controlados. Si se sobrepasan estos rangos, se tomarían los valores límites para llevar a cabo la actualización. Esta estrategia nos asegura que no actualizamos la política en exceso, evitando afectar negativamente a la convergencia del proceso de optimización.

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1 - \varepsilon, 1 + \varepsilon)\hat{A}_t)]$$

Figura 3.4: Función de coste basada en Clipped surrogate objective. Obtenida de Joseph y Ali (2017b)

Los resultados son tan prometedores, que PPO se ha convertido en el algoritmo de aprendizaje por refuerzo predeterminado de OpenAI debido a su facilidad de uso y buen rendimiento. La figura 3.5 muestra el pseudocódigo de PPO.

Algorithm 1 PPO-Clip

- 1: Input: initial policy parameters θ_0 , initial value function parameters ϕ_0
- 2: **for** $k = 0, 1, 2, \dots$ **do**
- 3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
- 4: Compute rewards-to-go \hat{R}_t .
- 5: Compute advantage estimates, \hat{A}_t (using any method of advantage estimation) based on the current value function V_{ϕ_k} .
- 6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg \max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \min \left(\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_k}(a_t | s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.

- 7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg \min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \left(V_{\phi}(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

- 8: **end for**
-

Figura 3.5: Pseudocódigo del algoritmo PPO. Obtenida de Joseph y Ali (2017b)

3.2.2. MA-POCA (MultiAgent POsthumous Credit Assignment)

Para poder llevar a cabo el entrenamiento de sistemas multiagente, ML-Agents nos proporciona MA-POCA (MultiAgent POsthumous Credit Assignment), un novedoso sistema de entrenamiento que entrena a un crítico centralizado y a una red neuronal, los cuales actúan como un 'entrenador' para todo un grupo de agentes. Este nuevo sistema permite otorgar recompensas al equipo en conjunto para que los agentes aprendan la mejor manera de contribuir a lograr esa recompensa. Aún así, los agentes también pueden recibir recompensas individuales, de manera que el equipo trabajará en conjunto para ayudar a cada individuo a lograr esos objetivos.

Hay que tener en cuenta, que durante un episodio, los agentes pueden agregarse o eliminarse del grupo, como cuando los agentes aparecen o mueren en un juego. Si los agentes se eliminan

a la mitad del episodio (por ejemplo, si los compañeros de equipo mueren o son eliminados del juego), estos aún sabrán si sus acciones contribuyeron a que el equipo ganara más tarde, lo que permite que los agentes tomen medidas beneficiosas para el grupo incluso si resultan en que uno de los individuos deba ser eliminado del juego (es decir, auto sacrificio). Además, MA-POCA también se puede combinar con el juego automático (Self-Play) para entrenar equipos de agentes para que jueguen entre sí.

La figura 3.6 muestra el pseudocódigo del entrenador MA-POCA.

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

```
for episode = 1 to  $M$  do
    Initialize a random process  $\mathcal{N}$  for action exploration
    Receive initial state  $\mathbf{x}$ 
    for  $t = 1$  to max-episode-length do
        for each agent  $i$ , select action  $a_i = \mu_{\theta_i}(o_i) + \mathcal{N}_t$  w.r.t. the current policy and exploration
        Execute actions  $a = (a_1, \dots, a_N)$  and observe reward  $r$  and new state  $\mathbf{x}'$ 
        Store  $(\mathbf{x}, a, r, \mathbf{x}')$  in replay buffer  $\mathcal{D}$ 
         $\mathbf{x} \leftarrow \mathbf{x}'$ 
    for agent  $i = 1$  to  $N$  do
        Sample a random minibatch of  $S$  samples  $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$  from  $\mathcal{D}$ 
        Set  $y^j = r_i^j + \gamma Q_i^{\mu'}(\mathbf{x}'^j, a'_1, \dots, a'_N)|_{a'_k=\mu'_k(o_k^j)}$ 
        Update critic by minimizing the loss  $\mathcal{L}(\theta_i) = \frac{1}{S} \sum_j (y^j - Q_i^{\mu}(\mathbf{x}^j, a_1^j, \dots, a_N^j))^2$ 
        Update actor using the sampled policy gradient:
        
$$\nabla_{\theta_i} J \approx \frac{1}{S} \sum_j \nabla_{\theta_i} \mu_i(o_i^j) \nabla_{a_i} Q_i^{\mu}(\mathbf{x}^j, a_1^j, \dots, a_i^j, \dots, a_N^j)|_{a_i=\mu_i(o_i^j)}$$

    end for
    Update target network parameters for each agent  $i$ :
    
$$\theta'_i \leftarrow \tau \theta_i + (1 - \tau) \theta'_i$$

end for
end for
```

Figura 3.6: *Pseudocódigo del algoritmo MA-POCA. Obtenida de Joseph y Ali (2017a)*

3.2.3. Convolutional Neural Networks (CNN)

Como bien se ha comentado en la introducción, los últimos avances que se han conseguido lograr en el campo de el aprendizaje por refuerzo y los cuales han revolucionado el estado del arte, se deben principalmente a la combinación del aprendizaje profundo con las técnicas de aprendizaje por refuerzo, conocido como Aprendizaje por Refuerzo Profundo, o Deep Reinforcement Learning (deep RL) ([BSC y UPC, Volodymyr Mnih](#)).

Al añadir una red neuronal, se consigue aprender con éxito políticas de control directamente a partir de una entrada sensorial de alta dimensión (p. ej. una imagen obtenida a partir de una cámara), utilizando además el aprendizaje por refuerzo. El modelo que se utiliza para ello es una

red neuronal convolucional (CNN), cuya entrada son píxeles sin procesar y cuya salida es una función de valor que estima futuras recompensas.

Gracias a esta funcionalidad, se nos abre la posibilidad de que un agente sea capaz de aprender a partir de la información obtenida de una imagen. Así pues, el toolkit de ML-Agents implementa esta función, e incluso nos permite usar varias cámaras para integrar dicha información como observaciones del agente. Esto puede ser útil cuando se tiene un agente el cual requiere de uno o de varios puntos de vista para funcionar, ya sean por ejemplo imágenes en 1a o 3a persona, o imágenes aéreas.

Para implementarlo, el toolkit de ML-Agents ofrece tres arquitecturas de red, las cuales debido al tamaño del kernel de convolución, nos presentan una limitación respecto al tamaño mínimo de las observación que cada tipo de codificador puede manejar:

- Un codificador simple que consta de dos capas convolucionales (20x20).
- Una implementación de la red propuesta por [Volodymyr Mnih \(2015\)](#), la cual consta de tres capas convolucionales (36x36).
- La red IMPALA Resnet, la cual consta de tres capas apiladas, cada una con dos bloques residuales, formando así la red más grande de las tres (15 x 15).

La elección de que arquitectura usar depende de la complejidad visual de la escena, así como de los recursos computacionales disponibles.

Esta función se aprovechará para dotar al agente de una cámara de 21x21px, la cual se espera que ayude al agente a interactuar con los diferentes entornos.

3.2.4. Imitation Learning

A menudo a la hora de desarrollar soluciones de aprendizaje por refuerzo, es más intuitivo simplemente enseñar al agente el comportamiento que queremos que realice, en lugar de intentar que aprenda a través de métodos de prueba y error. Por ejemplo, en lugar de intentar capacitar indirectamente a un conductor con la ayuda de una función de recompensa, podríamos darle ejemplos de carreras y recorridos realizados en el mundo real para guiar su comportamiento. El aprendizaje por imitación o imitation learning es por lo tanto, un proceso que utiliza pares de observación-acción obtenidos de una demostración grabada previamente, con el objetivo de aprender una política.

El imitation learning es una técnica que se puede utilizar sola o en conjunto de otros algoritmos de aprendizaje por refuerzo. Si se usa sola, puede proporcionar un mecanismo para aprender un tipo específico de comportamiento (es decir, un modo específico de resolver la tarea). Si se usa junto al aprendizaje por refuerzo, se puede por ejemplo reducir drásticamente el tiempo que tarda el agente en resolver una tarea, además de permitir que el agente pueda encontrar estrategias más óptimas para resolverla. Esto puede ser especialmente útil en entornos con escasa recompensa, o donde la complejidad del entorno es tal que el agente tardaría demasiado en aprender mediante únicamente prueba-error.

La figura 3.7 muestra una comparativa del proceso de entrenamiento entre varios entrenamientos realizados con y sin métodos de imitation learning. Esta comparativa se obtuvo del repositorio oficial de ML-Agents ([ML- \(c\)](#)), la cual se realizó sobre uno de los entornos de prueba de la librería de MI-Agents, más concretamente el entorno 'Pyramids':

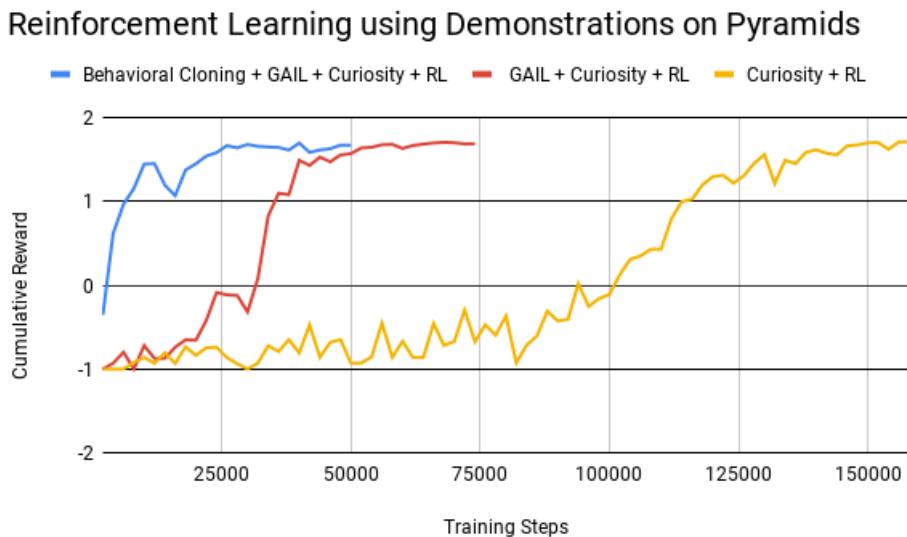


Figura 3.7: Comparativa del proceso de entrenamiento utilizando técnicas de Imitation learning. Obtenida de [ML- \(c\)](#)

Como se puede ver en la imagen anterior, mediante la combinación de Behavioral Cloning, GAIL, Curiosity y RL, se puede obtener un sistema capaz de aprender a realizar la tarea con la misma o mayor precisión que únicamente utilizando técnicas de RL, e inclusive con un tiempo de entrenamiento mucho menor.

El kit de herramientas de ML-Agents nos proporciona una manera de aprender directamente a partir de demostraciones, así como de usarlas para ayudar a acelerar el entrenamiento basado en recompensas (RL). Para ello se incluyen dos algoritmos: **Behavioral Cloning (BC)** y **Generative Adversarial Imitation Learning (GAIL)**, los cuales se pueden combinar en la mayoría de escenarios, y son útiles especialmente en entornos con escasas recompensas.

Además de habilitar tanto GAIL como Behavioral Cloning, se puede continuar activando el sistema de recompensas extrínsecas, de manera que con la correcta combinación de los 3 se obtengan comportamientos sobrehumanos, y con un tiempo mucho menor.

3.2.4.1. GAIL

La técnica de GAIL ([Ho y Ermon \(2016\)](#), [GAI](#)), o aprendizaje por imitación adversario generativo, utiliza un enfoque adversarial mediante el cual se recompensa al agente por comportarse de manera similar a un conjunto de demostraciones. GAIL se puede usar con o sin recompensas extrínsecas y puede funcionar bien cuando se tiene un número limitado de demostraciones. Para ello, se enseña a una segunda red neuronal, el discriminador, a distinguir si una observación/acción proviene de una demostración o es producida por el agente. Este discriminador puede luego examinar una nueva observación/acción y proporcionarle una recompensa basada en qué tan cerca cree que está esta nueva observación/acción de las demostraciones proporcionadas.

Así pues, en cada paso de entrenamiento el agente intenta aprender a cómo maximizar esa recompensa. Luego, se entrena al discriminador para distinguir mejor entre demostraciones y estados/acciones del agente. De esta manera, mientras el agente imita cada vez mejor las demostraciones, el discriminador se vuelve cada vez más estricto y el agente debe esforzarse más para engañarlo.

Este enfoque permite que el agente aprenda una política que produce estados y acciones similares a las demostraciones, permitiendo además que funcione con menos demostraciones que la clonación directa de las acciones. Además de aprender únicamente a partir de demostraciones, la señal de recompensa GAIL se puede combinar con una señal de recompensa extrínseca para guiar el proceso de aprendizaje.

3.2.4.2. Behavioral Cloning (BC)

En contraposición a GAIL que enseña al agente a comportarse de manera similar a las demostraciones, Behavioral Cloning (BC) ([Vinicius G. Goecks \(2019\)](#), [Beh](#)) entrena al Agente para imitar exactamente las acciones que se muestran en un conjunto de demostraciones. Esto hace que BC no pueda generalizar más allá de los ejemplos que se muestran en las demostraciones. BC por lo tanto tiende a funcionar mejor cuando existen demostraciones para casi todos los estados que el agente pueda experimentar, o cuando se utiliza en conjunto de GAIL y/o una recompensa extrínseca.

3.2.5. Curriculum Learning

Otro de los métodos que nos ofrece ML-Agents para facilitar y acelerar el proceso de entrenamiento, es el del curriculum learning ([VPetru Soviany \(2021\)](#), [Cur](#)). Esta técnica es una forma de entrenar un modelo de aprendizaje automático, de modo que se introducen gradualmente los aspectos más difíciles de un problema, de tal manera que el modelo siempre aprende de manera óptima. Esta idea se basa en la forma en que los humanos normalmente aprendemos. Si pensamos en cualquier educación infantil, siempre hay una ordenación de clases y temas. La aritmética se enseña antes que el álgebra, por ejemplo, y asimismo, se enseña álgebra antes que cálculo. Las habilidades y conocimientos que se han ido aprendiendo en las materias previas proporcionan una base para lecciones posteriores. Este mismo principio es el que se puede aplicar al aprendizaje

automático, donde el entrenamiento en tareas más sencillas puede proporcionar una base para tareas más difíciles en el futuro.

Esta técnica es muy útil para entornos donde se desee realizar una tarea compleja. Así pues, como punto de partida comenzaremos a entrenar con una tarea más simple, de manera que el agente pueda aprender fácilmente a realizar la tarea. A partir de ahí, podemos aumentar paulatinamente la dificultad de la tarea hasta que se consiga obtener el comportamiento deseado, y finalmente completar la tarea que en un principio era casi imposible de realizar.

Una manera fácil de entenderlo es mediante un ejemplo que proporciona ML-Agents llamado 'WallJump' (Fig.3.8), donde un agente aprende a sortear un muro el cual cada vez es más alto, con el objetivo de llegar a la meta.

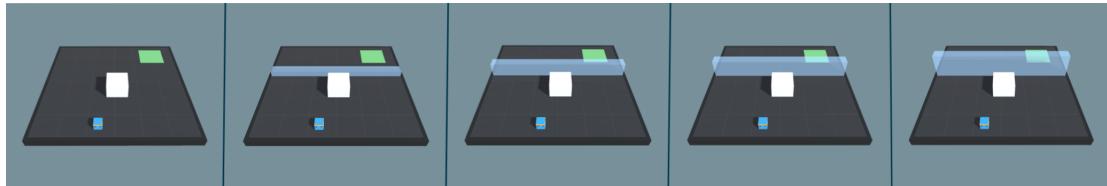


Figura 3.8: Ejemplo de Curriculum Learning. Obtenida de [Wal](#)

3.2.6. Self Play

Finalmente, ML-Agents también proporciona un mecanismo de Self-Play para entrenar entornos de confrontación tanto simétricos como asimétricos. Un juego simétrico sería aquel en el que los agentes opuestos son iguales tanto en forma, como en el diseño de su función y objetivo (p. ej. Tenis). Esto significa que todos los agentes tienen las mismas observaciones y acciones, además de que aprenden de la misma función de recompensa, por lo que pueden compartir la misma política. En los juegos asimétricos no es el caso, ya que los agentes en este tipo de juegos no siempre tienen las mismas observaciones o acciones (p. ej. Hide and Seek o fútbol). Gracias al Self-Play los agentes aprenden compitiendo contra versiones fijas y pasadas de su oponente, de manera que se proporciona un entorno de aprendizaje más estable.

Esta técnica se puede aplicar tanto en PPO como en SAC, sin embargo, desde la perspectiva de un agente individual, estos escenarios parecen tener una dinámica no estacionaria debido a que el oponente a menudo cambia. Esto puede causar problemas significativos en el mecanismo de reproducción de experiencias que utiliza SAC, y por lo tanto ML-Agents recomiendan utilizarlo en PPO en su lugar.

3.3. Proyectos recientes

Recientemente se ha desarrollado en OpenAI un sistema de interacción de herramientas para sistemas multiagente ([Hid](#)), el cual mediante Self-Play permite ir observando cómo los agentes son capaces de descubrir el uso de herramientas progresivamente más complejas mientras juegan un simple juego de escondite (Fig.3.9). En este entorno, los agentes juegan al escondite en equipo mientras hay objetos dispersos por todo el entorno que los agentes pueden agarrar y bloquear en su lugar, así como habitaciones y paredes inamovibles generadas aleatoriamente por las que los agentes deben aprender a navegar.

Este trabajo es muy interesante, porque pone a prueba sistemas multiagentes y demuestra cómo mediante la interacción de los agentes estos van desarrollando estrategias cambiantes con el tiempo, siendo esto el resultado del auto currículum inducido por la competencia de múltiples agentes y la simple dinámica del escondite.

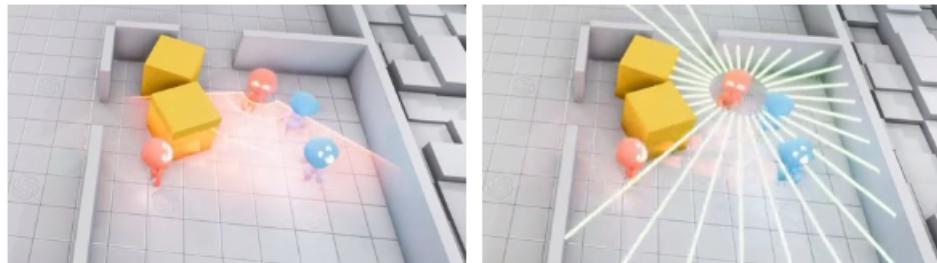


Figura 3.9: *Proyecto multiagente realizado por OpenAI. Obtenida de [Hid](#)*

Metodología y entorno de trabajo

4

En esta sección hablaremos de cada uno de los componentes y herramientas que hacen funcionar nuestro sistema. Así pues, explicaremos por orden los diferentes pasos que hay que llevar a cabo hasta conseguir entrenar a un agente, a excepción del apartado previo de 'Instalación y requisitos' y de un apartado donde se explica cómo se realiza el proceso de entrenamiento y la implementación del modelo final del agente, los cuales se pueden encontrar en el Apéndice A.

Además, se pueden encontrar también imágenes del código básico utilizado para hacer funcionar un agente, así como un enlace a un repositorio de Github donde se pueden encontrar todos los códigos y elementos de Unity utilizados para el desarrollo de este proyecto, además de una carpeta con demos de cada uno de los entornos realizados en el siguiente apartado.

4.1. Configuración del comportamiento

El kit de herramientas de Unity ML-Agents proporciona una amplia variedad de escenarios, métodos y opciones de entrenamiento. Como tal, las ejecuciones de los entrenamientos pueden requerir diferentes configuraciones de entrenamiento y pueden generar diferentes artefactos y estadísticas. Para realizar todos estos entrenamientos, es necesario un fichero de configuración donde queden plasmados los valores de los diferentes hiperparámetros, el método de aprendizaje por reforzamiento a utilizar, así como las configuraciones para los parámetros ambientales (como Curriculum Learning).

Es importante resaltar que para entrenar con éxito un comportamiento con ML-Agents, es necesario ajustar la configuración y los hiperparámetros del entrenamiento, de manera que el agente diseñado pueda aprender correctamente. A continuación se presentan algunas de las preguntas más esenciales a la hora de ajustar el proceso de entrenamiento.

- ¿Utilizar PPO o SAC ?
- ¿Usar redes neuronales recurrentes para agregar memoria a los agentes ?
- ¿Usar el módulo de curiosidad intrínseca ?
- ¿Pre-entrenar usando behavioral cloning ?
- ¿Incluir las señales de recompensa intrínsecas de GAIL ?
- ¿Usar self-play (Suponiendo que el entorno incluya varios agentes) ?

4.1.1. Fichero de configuración YAML

Para la configuración del comportamiento y de los diferentes hiperparámetros comentados anteriormente, es necesario utilizar un fichero YAML. La sección principal del archivo de configuración es un conjunto de configuraciones básicas para cada comportamiento en la escena. Algunas de las configuraciones son obligatorias, mientras que otras son opcionales.

Para ayudar a entenderlo, se muestra en el Apéndice A el contenido de un archivo de configuración de muestra en el que se incluyen todas las configuraciones posibles usando un entrenador PPO (memoria (LSTM), clonación de comportamiento, curiosidad, GAIL y self-play).

4.1.1.1. Configuraciones comunes del entrenador

Algunos de los parámetros del fichero de configuración más importantes y los cuales merece la pena mencionar, son los siguientes:

- **trainer_type**: El tipo de entrenador a utilizar (PPO, SAC o MA-POCA).
- **summary_freq**: Número de experiencias que deben recopilarse antes de generar y mostrar estadísticas de entrenamiento. Esto determina la granularidad de los gráficos en Tensorboard.
- **time_horizon**: Cuántos pasos de experiencia recopilar por agente antes de agregarlo al buffer de experiencia. Como tal, este parámetro compensa entre una estimación menos sesgada, pero con mayor varianza (horizonte de tiempo largo) y una estimación más sesgada, pero menos variada (horizonte de tiempo corto). 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.
- **max_steps**: Número total de pasos (es decir, observación recopilada y acción tomada) que se deben realizar en el entorno (o en todos los entornos si se usan varios en paralelo) antes de finalizar el proceso de entrenamiento. Si se tienen varios agentes con el mismo nombre de comportamiento dentro de un mismo entorno, todos los pasos realizados por estos agentes contribuirán al mismo recuento max_steps.
- **hyperparameters ->learning_rate**: Tasa de aprendizaje inicial para descenso de gradiente. Por lo general, debe reducirse si el entrenamiento es inestable y la recompensa no aumenta constantemente. Rango típico: 1e-5 - 1e-3
- **hyperparameters ->batch_size**: Número de experiencias en cada iteración de descenso de gradiente. Se aconseja que si se usan acciones continuas, este valor debe ser grande (del orden de 1000), mientras que si está utilizando sólo acciones discretas, este valor debe ser más pequeño (del orden de 10). Además, siempre debe ser varias veces más pequeño que

buffer_size. Rango típico: (Continuous - PPO): 512 - 5120; (Continuous - SAC): 128 - 1024; (Discrete, PPO & SAC): 32 - 512.

- **hyperparameters ->buffer_size:** Número de experiencias a recopilar antes de actualizar la política del modelo. Esto debería ser varias veces más grande que buffer_size . Normalmente, un buffer_size más grande corresponde a actualizaciones de entrenamiento más estables. Rango típico: PPO: 2048 - 409600; SAC: 50000 - 1000000
- **hyperparameters ->learning_rate_schedule:** Determina cómo cambia la tasa de aprendizaje con el tiempo. Para PPO se recomienda disminuir la tasa de aprendizaje hasta max_steps para que el aprendizaje converja de manera más estable. Sin embargo, en algunos casos (p. ej., entornos complejos con entrenamientos durante un período de tiempo desconocido), esta función se puede desactivar. Para SAC, se recomienda mantener constante la tasa de aprendizaje para que el agente pueda seguir aprendiendo hasta que su función Q converja naturalmente.
- **network_settings ->hidden_units:** Número de unidades en las capas ocultas de la red neuronal. Para problemas simples donde la acción correcta es una combinación directa de las entradas de observación, esto debería ser pequeño. Para problemas donde la acción es una interacción muy compleja entre las variables de observación, esto debería ser mayor. Rango típico: 32 - 512
- **network_settings ->num_layers:** Número de capas ocultas en la red neuronal. Corresponde a cuántas capas ocultas están presentes después de la entrada de observación, o después de la codificación CNN de la observación visual. Para problemas simples, es probable que menos capas entrenen más rápido y de manera más eficiente, pero para problemas de control más complejos pueden ser necesarias más capas. Rango típico: 1 - 3
- **network_settings ->normalize:** Si la normalización se aplica a las observaciones del agente. 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.

4.1.1.2. Configuraciones específicas del entrenador PPO

En nuestro caso, vamos a utilizar el entrenador PPO, por lo que a continuación mostramos los diferentes parámetros específicos para configurarlo correctamente:

- **hyperparameters ->b1:** Fuerza de la regularización de la entropía. 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. Este parámetro debe ajustarse de manera que la entropía disminuya lentamente junto con los aumentos en la recompensa. Si la entropía cae demasiado rápido, aumentar la beta. Si la entropía cae muy lentamente, disminuir la beta.

- **hyperparameters ->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 entrenamiento.
- **hyperparameters ->beta_schedule**: Determina cómo cambia beta con el tiempo. Si de manera linear, llegando a 0 en max_steps, o de manera constante durante toda la ejecución del entrenamiento.
- **hyperparameters ->epsilon_schedule**: Similar al parámetro anterior, pero determinando cómo cambia épsilon con el tiempo.
- **hyperparameters ->num_epoch**: Número de pasos a través del búfer de experiencia para realizar la optimización de descenso de gradiente. Disminuir esto asegurará actualizaciones más estables, a costa de un aprendizaje más lento.

4.1.1.3. Configuraciones específicas del entrenador MA-POCA

MA-POCA usa las mismas configuraciones que PPO y no hay parámetros adicionales específicos de POCA.

4.1.1.4. Configuraciones de las señales de recompensa

La sección de señales de recompensa permite la especificación de configuraciones para señales de recompensa tanto extrínsecas (es decir, basadas en el entorno) como intrínsecas (por ejemplo, curiosidad y GAIL). Cada señal de recompensa debe tener definidos al menos dos parámetros, fuerza y gamma, además de cualquier hiperparámetro específico de la clase. El parámetro de fuerza es un factor por el que multiplicar la recompensa otorgada por el entorno, mientras que el parámetro de gamma es un 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.

A continuación se muestran los posibles sistemas de recompensas que se pueden implementar en ML-Agents. Acudir a la documentación de ML-Agents para más información.

- **Extrinsic Rewards.**
- **Curiosity Intrinsic Reward.**
- **GAIL Intrinsic Reward.**
- **Behavioral Cloning.**

- Recurrent Neural Networks.
- Self-Play.

4.2. Diseño de los entornos en Unity

Una vez se ha realizado la configuración previa e instalado correctamente tanto Unity como ML-Agents, para crear nuevos entornos lo primero necesario es crear un proyecto de Unity. Una vez hecho esto, procedemos a diseñar el entorno 3D, así como la configuración de las físicas y las interacciones entre elementos del entorno, como finalmente el diseño del agente que se moverá por el mundo. A continuación mostraré el proceso que hay que llevar a cabo para la construcción y configuración correcta del entorno.

4.2.1. Configuración del proyecto de Unity

La primera tarea a realizar es simplemente crear el proyecto de Unity e importar los assets de ML-Agents.

1. Iniciar Unity Hub y crear un nuevo proyecto de Unity, llamado en mi caso 'TFM'.
2. Agregar el paquete ML-Agents Unity al proyecto como se explica en la sección de instalación.

El paquete se debería ver de la siguiente manera (Fig.4.1):

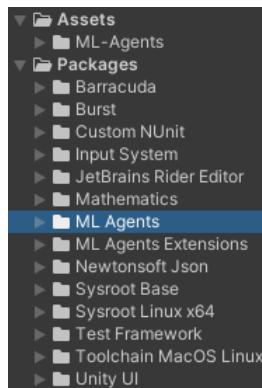


Figura 4.1: Ventana Proyecto de Unity con los assets necesarios.

4.2.2. Creación de un entorno de ejemplo

A continuación, crearemos una escena muy simple para que actúe a modo de ejemplo y pueda servir como base para entender el resto de entornos. Los componentes físicos del entorno incluyen un Plano que actúa como el piso para que el agente se mueva, una esfera que actúa como la meta o el objetivo que el agente debe buscar, y un cubo que representa al propio Agente.

4.2.2.1. Crear el plano del suelo (Fig.[4.2](#))

1. Hacer click derecho en la ventana de Jerarquía y seleccionar 3D Object >Plane.
2. Nombrar el objeto "Floor"
3. Seleccione el Plano del suelo para ver sus propiedades en la ventana del Inspector.
4. Establecer la posición y el tamaño del suelo.

4.2.2.2. Crear la esfera de destino/meta (Fig.[4.3](#))

1. Hacer click derecho en la ventana de Jerarquía y seleccionar 3D Object >Cube.
2. Nombrar el objeto "Target"
3. Seleccionar el Cubo Objetivo para ver sus propiedades en la ventana del Inspector.
4. Establecer la posición y el tamaño del Cubo.

4.2.2.3. Crear el agente

Para la creación del agente, usaremos un preset de la librería de ML-Agents para obtener el modelo 3D, y una vez conseguido, lo modificaremos para adaptarlo a nuestro proyecto.

1. Moverse a la carpeta `ml-agents\Project\Assets\ML-Agents\Examples\SharedAssets` (Fig.[4.4](#))
2. Arrastrar el elemento AgentCube_Blue a nuestro proyecto. (Fig.[4.5](#))
3. Agregarle el componente Rigidbody.
4. Agregarle el componente Box Collider.

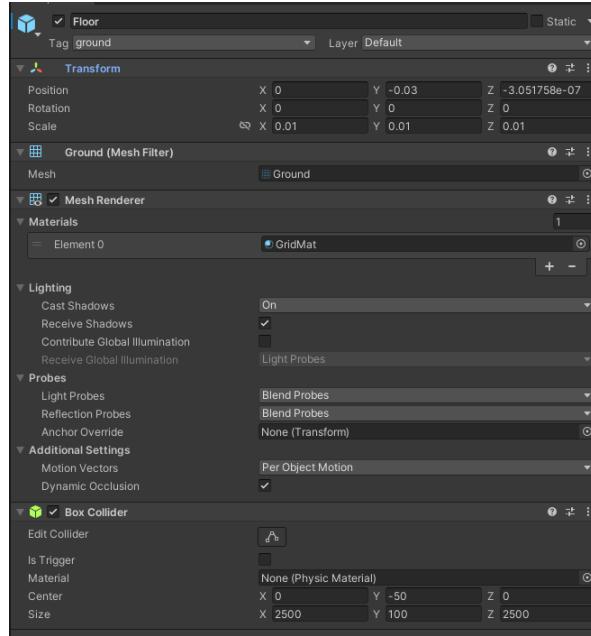


Figura 4.2: Creación del entorno en Unity: Suelo.

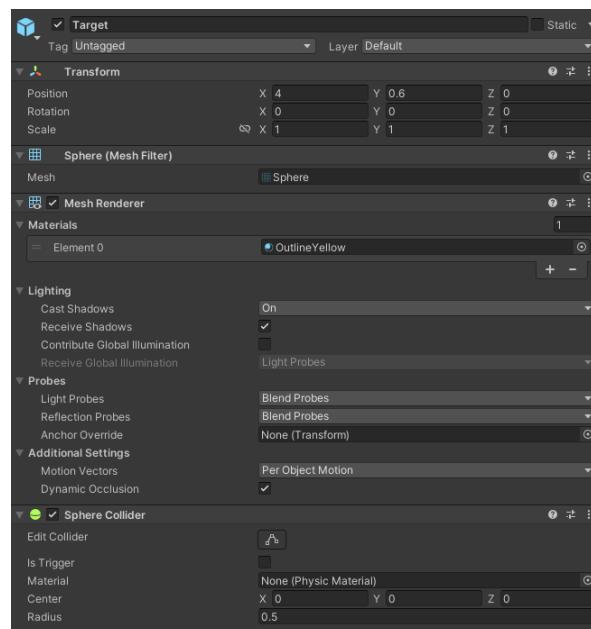


Figura 4.3: Creación del entorno en Unity: Target.



Figura 4.4: Creación del entorno en Unity: Agente.

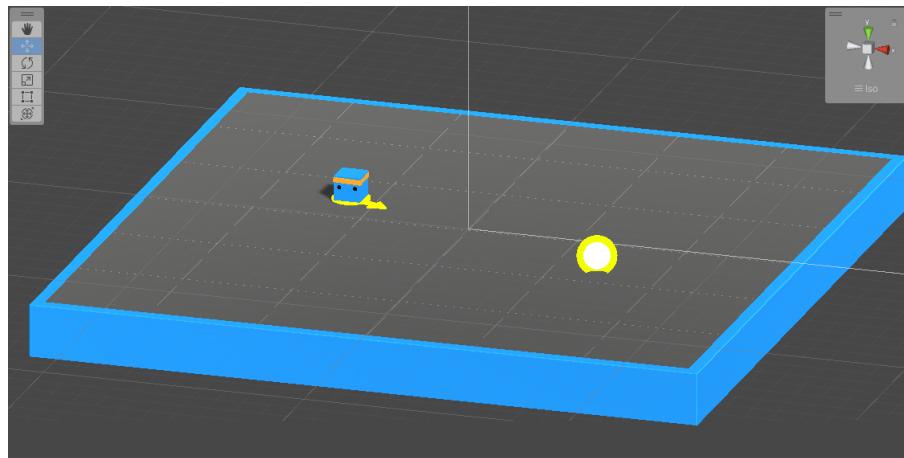


Figura 4.5: Creación del entorno en Unity: Entorno de ejemplo.



Figura 4.6: Creación del entorno en Unity: Agrupar los componentes.

4.2.2.4. Agrupar los componentes en un área de Aprendizaje (Fig.4.7)

Una vez hecho todo esto, procedemos a agrupar el suelo, el agente y el objetivo, todos en un mismo GameObject vacío, lo cual simplificará alguno de los pasos posteriores, además de permitirnos tener todos los elementos del entorno agrupados.

4.2.3. Implementación y diseño del agente

Una vez creado el agente, es necesario implementar un código para definir su comportamiento. Para ello, seguimos los siguientes pasos:

1. Seleccionar el GameObject de AgentCube_Blue para verlo en la ventana Inspector.
2. Hacer clic en Add Component.
3. Añadir un nuevo script de C# en la lista de componentes.
4. Nombra el script "MoveToGoalAgent".
5. Hacer clic en Create and Add.

Ahora bien, una vez creado el modelo del agente, necesitamos definir tanto el número de observaciones que el agente va a tener (sensores), como el tipo de acciones que tomará, como las recompensas que vendrán de tomar dichas acciones.

4.2.3.1. Acciones y Actuadores

Así pues, definiremos primero el tipo de acciones que puede tomar un agente y su diferencia para el entrenamiento. Antes que nada, aclarar que ni la política del agente ni el algoritmo de entrenamiento saben nada sobre lo que significan los valores de acción en sí mismos. El algoritmo de entrenamiento simplemente prueba diferentes valores para la lista de acciones y observa el efecto sobre las recompensas acumuladas a lo largo del tiempo y de los episodios del entrenamiento. Un Agente recibe por lo tanto instrucciones en forma de acciones, y ML-Agents las clasifica en dos tipos: continuas y discretas.

- **Acciones continuas:** Cuando la política de un agente tiene acciones continuas, las `Actions` que se pasan a la función son una matriz con una longitud igual al valor de `Continuous Action Size`. Los valores individuales en la matriz tienen cualquier significado que se les atribuya. Si se asigna por ejemplo un elemento en la matriz como la velocidad de un Agente, el proceso de entrenamiento aprenderá a controlar la velocidad del Agente a través de este parámetro.
- **Acciones discretas:** Cuando la política de un agente usa acciones discretas, las `ActionActions` que se pasan a la función son una matriz de enteros con una longitud igual al tamaño de rama discreta. Al definir las acciones discretas, cada rama es una matriz de números enteros, cuyo valor corresponde al número de posibilidades para cada rama. Por ejemplo, si quisiéramos un Agente que pueda moverse en un plano y saltar, podríamos definir dos ramas (una para movimiento y otra para saltar). Definimos la primera rama para tener 5 acciones posibles (no moverse, ir a la izquierda, ir a la derecha, ir hacia atrás, ir hacia adelante) y la segunda para tener 2 acciones posibles (no saltar, saltar), por ejemplo.
- **Recomendaciones:**
 - En general, cuantas menos acciones se usen, más se facilitará el aprendizaje.
 - Es recomendable recortar los valores de las acciones continuas a un rango apropiado. Al usar el modelo PPO proporcionado por ML-Agents, se recortan automáticamente estos valores entre -1 y 1.

4.2.3.2. Observaciones y Sensores

Antes de tomar una decisión, el agente recoge su observación sobre su estado en el mundo. El vector de observación es por lo tanto, un vector de números de coma flotante el cual contiene la información relevante para que el agente sea capaz de tomar decisiones. Aunque existen más tipos en el toolkit, a continuación explicaremos los 3 tipos de observaciones utilizados en el proyecto:

- **Vector Observations:** Observaciones que pueden ser tanto vectoriales como visuales, y que se representan en listas de flotantes (p. ej. Posiciones (x,y,z) del agente). Para obtener los mejores resultados al entrenar, se recomienda que se normalicen los componentes del vector de características al rango [-1, +1] o [0, 1], ya que la red neuronal de PPO a menudo puede converger en una solución más rápido. Las Vector Observations deben incluir todas las variables relevantes para permitir que el agente tome una decisión informada de manera óptima e, idealmente, sin información superflua. Además, las variables categóricas deben codificarse como one-hot.
- **Visual Observations:** Las observaciones visuales generalmente se proporcionan al agente a través de un CameraSensor o un RenderTextureSensor. Recopilan información de la imagen y la transforman en un tensor 3D que se puede alimentar a la red neuronal convolucional (CNN) de la política del agente. Los agentes que usan observaciones visuales son útiles cuando el estado es difícil de describir numéricamente. Sin embargo, también suelen ser menos eficientes y más lentos de entrenar y, a veces, no obtienen ningún éxito en comparación con las observaciones de vectores. Como tal, solo deberían usarse cuando no es posible definir correctamente el problema utilizando observaciones vectoriales o Raycast.
- **Raycast Observation:** Los Raycasts son otro método posible para proporcionar observaciones a un agente. Estos se pueden implementar fácilmente agregando un RayPerceptionSensorComponent3D (o RayPerceptionSensorComponent2D) al GameObject del agente y funciona de manera similar a un sensor Lidar. Durante el periodo de observación, varios rayos (o esferas, según la configuración) se proyectan en el mundo físico, y los objetos que golpean determinan el vector de observación que se produce. Ambos componentes del sensor tienen varias configuraciones, como la cantidad de etiquetas detectables, la cantidad de rayos por dirección, etc. Este tipo de observación se usa mejor cuando hay información espacial relevante para el agente, la cual no requiere una imagen totalmente renderizada para transmitirla. Es conveniente utilizar la menor cantidad de rayos y etiquetas posibles para resolver el problema, de manera que se mejore la estabilidad del aprendizaje y el rendimiento del agente.

4.2.3.3. Decisiones

El ciclo observación-decisión-acción-recompensa se repite cada vez que el Agente solicita una decisión. Los agentes solicitarán una decisión cuando se llame a **Agent.RequestDecision()**, o en el caso de agregar el componente **Decision Requester** al GameObject del agente, el agente solicitará decisiones por su cuenta a intervalos regulares. Tomar decisiones en intervalos de pasos regulares es generalmente más apropiado para simulaciones basadas en físicas. Por ejemplo, si un agente en un simulador robótico debe proporcionar un control preciso de los pares de torsión, debe tomar sus decisiones en cada paso de la simulación. Por otro lado, un agente que solo necesita tomar decisiones cuando ocurren ciertos eventos de juego o simulación, como en un juego por turnos, debería llamar a **Agent.RequestDecision()** manualmente. En nuestro caso, utilizaremos un **Decision Requester** con valor de 5, de manera que cada 5 steps se tome una acción. Para sistemas

cuya velocidad sea mayor, o que se requiera tomar decisiones en tiempo real, puede ser adecuado establecer este parámetro con un valor de 1, de manera que en cada paso se tome una acción.

4.2.3.4. Recompensas

A continuación, se introducen algunos consejos útiles a la hora de definir las funciones de recompensa:

- Usar **AddReward()** para acumular recompensas entre decisiones. Usar **SetReward()** para sobrescribir cualquier recompensa anterior acumulada entre decisiones.
- La magnitud de cualquier recompensa normalmente no debe ser superior a 1 para garantizar un proceso de aprendizaje más estable.
- Las recompensas positivas suelen ser más útiles para dar forma al comportamiento deseado de un agente que las recompensas negativas. Las recompensas negativas excesivas pueden hacer que el agente no aprenda ningún comportamiento significativo.
- Para las tareas de locomoción, normalmente se utiliza una pequeña recompensa positiva (+0,1) para premiar el avance hacia delante.
- Si se desea que el agente termine una tarea rápidamente, a menudo es útil proporcionar una pequeña penalización para cada paso en el que el agente no completa la tarea.

4.2.4. Configuración de las propiedades del Agente

Una vez definidos todos los parámetros necesarios para la configuración del agente, procederemos a modificarlos desde el editor de Unity (Fig.4.7).

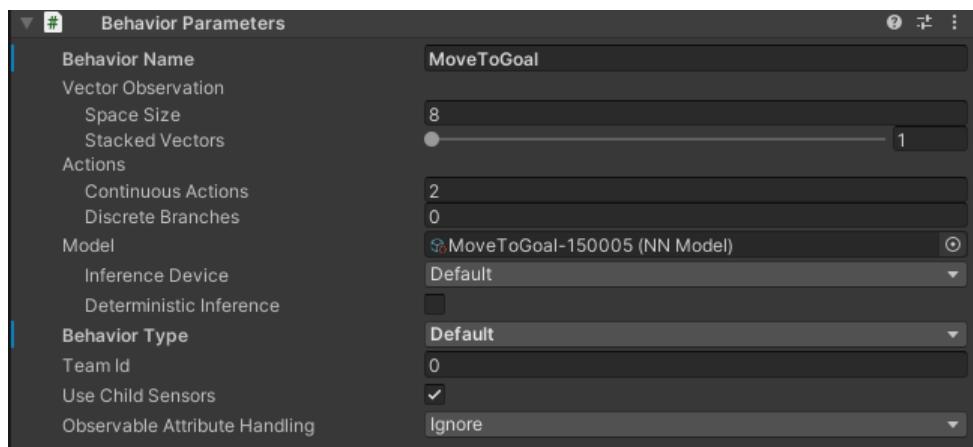


Figura 4.7: *Configuración de las propiedades del Agente desde el editor de Unity.*

Los parámetros más importantes a comentar son los siguientes:

- **Behavior Name:** Nombre identificador para el comportamiento. Los agentes con el mismo nombre de comportamiento aprenderán la misma política.
- **Vector Observation:**
 - **Space Size:** Longitud de la observación vectorial para el agente.
 - **Stacked Vectors:** Número de observaciones vectoriales anteriores que se apilan y se usarán colectivamente para tomar decisiones.
- **Actions:**
 - **Continuous Actions:** Número de acciones continuas concurrentes que puede realizar el agente.
 - **Discrete Branches:** Matriz de enteros que define múltiples acciones discretas concurrentes. Los valores en la matriz 'Discrete Branches' corresponden al número de valores discretos posibles para cada rama.
- **Model:** Modelo de red neuronal utilizado para la inferencia (obtenido después del entrenamiento)
- **Inference Device:** Usar CPU o GPU para ejecutar el modelo durante la inferencia
- **Behavior Type:** Determina si el agente realizará el entrenamiento, la inferencia o usará su método Heuristic () .
- **Team ID:** Utilizado para definir equipos al utilizar Self-Play
- **Use Child Sensors:** Si se deben usar todos los componentes de sensor adjuntos a los hijos de este objeto (Raycast, CameraSensors, etc)
- **Max Step:** Número máximo de pasos por agente. Una vez alcanzado este número, el agente se reiniciará.

4.2.5. Configuración del script del Agente

Una vez explicado todo esto, a continuación procederemos a abrir y editar el script del Agente.

1. **Importar las librerías de ML-Agent**
2. **Cambiar la clase base de MonoBehaviour a Agent.**
3. **Eliminar la función Update().**

Hasta ahora, estos son los pasos básicos para agregar ML-Agents a cualquier proyecto de Unity. A continuación, es necesario agregar la lógica, la cual permitirá que nuestro Agente aprenda a moverse hacia la meta mediante el aprendizaje por refuerzo. Más específicamente, necesitaremos extender tres métodos de la clase base 'Agent' .

Recordar que el código de este entorno de ejemplo, así como el resto de códigos y elementos de Unity utilizados para el desarrollo de este proyecto se pueden encontrar en la sección del Apéndice A y en el repositorio Github:

- **OnEpisodeBegin()**: Al comienzo de cada episodio, se llama a esta función para configurar el entorno para un nuevo episodio. Por lo general, la escena se inicia de manera aleatoria para permitir que el agente aprenda a resolver la tarea bajo una serie de condiciones pre-determinadas. En este ejemplo, cada vez que el Agente (Cubo) alcanza su objetivo (Esfera), el episodio finaliza y el objetivo (Esfera) se mueve a una nueva ubicación aleatoria, y si el Agente se cae de la plataforma, volverá a colocarse en el suelo. (Fig.[A.12](#)).
- **CollectObservations(VectorSensor sensor)**: Función que se encarga de alimentar a la red neuronal de un vector de características con el cual el agente procederá a tomar decisiones. Para que un agente aprenda con éxito una tarea, debemos proporcionar la información correcta. En este, la información que recopila nuestro Agente incluye la posición del objetivo, la posición del propio agente y la velocidad del agente. Esto ayuda al Agente a aprender a controlar su velocidad para que no sobrepase el objetivo y se caiga de la plataforma. En total, la observación vectorial del agente contiene 8 valores. (Fig.[A.13](#)).
- **OnActionReceived(ActionBuffers actionBuffers)**: La parte final del código del Agente es esta función, que recibe acciones y asigna la recompensa.
 - **Acciones**: Para moverse hacia el objetivo, el Agente (Cubo) debe poder moverse en las direcciones x y z. Como tal, el agente necesita 2 acciones: la primera determina la fuerza aplicada a lo largo del eje x; y la segunda determina la fuerza aplicada a lo largo del eje z. El agente aplica los valores de la matriz `actionBuffers[]` a su componente Rigidbody `rBody`, usando `Rigidbody.AddForce()`.
 - **Recompensas**: El aprendizaje por refuerzo requiere recompensas para señalar qué decisiones son buenas y cuáles son malas. El algoritmo recompensa al agente por completar la tarea asignada. En este caso, el agente calcula la distancia para detectar cuando alcanza el objetivo, y recibe una recompensa de 1,0 por alcanzarlo, y de -0.5 si se cae de la plataforma. Para finalizar el episodio, se llama a `EndEpisode()` (Fig.[A.14](#)).

4.2.6. Configuración final del agente en el editor

Una vez todos los componentes están en su lugar, es momento de conectarlos todo desde el Editor de Unity. Para ello, agregaremos al Agente los diferentes componentes necesarios para que sea compatible con nuestro Script:

1. Seleccionar el GameObject del Agente para mostrar sus propiedades en la ventana del Inspector.
2. Arrastrar el GameObject del Target en la jerarquía al campo de destino en el script del Agent..

3. Agregar un Decision Requester script con el botón de Add Component y establecer el periodo de decisión a 5 (Cada 5 steps, tomará una acción).
4. Agregar un Behavior Parameters script con el botón de Add Component, y configurar los siguientes parámetros:
 - **Behavior Name:** MoveToGoal
 - **Vector Observation (Space Size)** = 8
 - **Actions (Continuous Actions)** = 2

En el inspector, el Agente debería verse así (Fig.4.8):

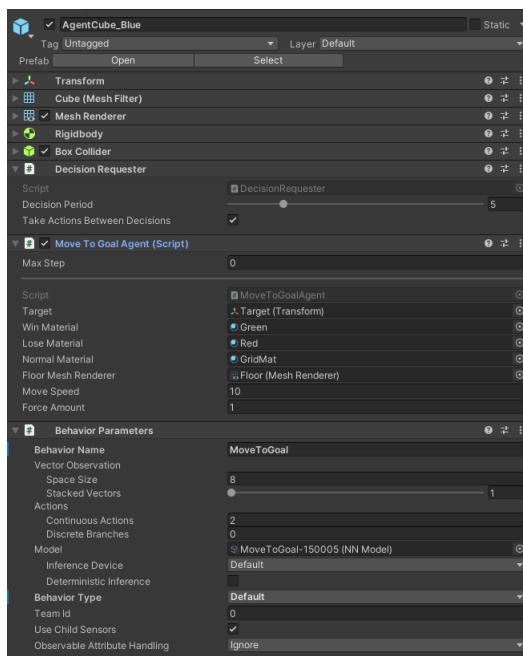


Figura 4.8: Configuración final del agente.

Ahora por fin estamos listos para el entrenamiento, cuyo proceso se define en el Apéndice A, donde se muestra tanto el fichero de configuración YAML utilizado para entrenar este entorno de ejemplo, como el resto de pasos necesarios para entrenar el agente, e implementar el modelo final una vez entrenado.

Dado que este ejemplo tiene un objetivo muy simple con solo unas pocas entradas y salidas, el uso de batches y tamaños de búfer pequeños acelera considerablemente el entrenamiento. Sin embargo, si se agrega más complejidad al entorno o se cambian las funciones de recompensa o de observación, como veremos en los entornos finales, se puede ver como el entrenamiento funciona mejor con diferentes valores de hiperparámetros.

4.2.7. Definición de escenarios multiagente

Finalmente, si se deseará hacer un escenario multiagente, las modificaciones que hay que realizar sobre el entorno, así como sobre el fichero de configuración YAML, serían las siguientes:

4.2.7.1. Grupos para escenarios cooperativos

El comportamiento cooperativo en ML-Agents se puede habilitar creando una instancia de **SimpleMultiAgentGroup**, generalmente en un controlador de entorno o script similar, y agregando los agentes necesarios mediante el método **RegisterAgent (agent agent)**. Hay que tener en cuenta que todos los agentes agregados al mismo SimpleMultiAgentGroup deben tener el mismo nombre de comportamiento y parámetros de comportamiento. El uso de SimpleMultiAgentGroup permite que los agentes dentro de un grupo aprendan a trabajar juntos para lograr un objetivo común (es decir, maximizar una recompensa otorgada por el grupo), incluso si uno o más de los miembros del grupo se eliminan antes de que finalice el episodio. Luego puede usar este grupo para agregar establecer recompensas, finalizar o interrumpir episodios a nivel de grupo usando los métodos **AddGroupReward()**, **SetGroupReward()**, **EndGroupEpisode()** y **GroupEpisodeInterrupted()**.

Los grupos de multiagentes deben usarse con el entrenador MA-POCA, que está diseñado explícitamente para entrenar entornos cooperativos. Al usar MA-POCA, los agentes que se desactivan o eliminan de la escena durante el episodio, seguirán aprendiendo a contribuir a las recompensas a largo plazo del grupo, incluso si no están activos en la escena para experimentarlas.

4.2.7.2. Equipos para escenarios adversariales

En los escenarios donde tenemos equipos que se enfrentan entre sí, es necesario activar el Self-play e incluirse en la configuración del entrenador. Para distinguir a los agentes opuestos, es necesario establecer el parámetro del ID del equipo en el script de parámetros de comportamiento del agente, como se ve en la siguiente imagen (Fig.4.9). El ID del equipo debe ser 0 o un número entero mayor que 0. Y en juegos simétricos, dado que todos los agentes (incluso en equipos opuestos) compartirán la misma política, deben tener el mismo 'Nombre de comportamiento' en su Script de parámetros de comportamiento.

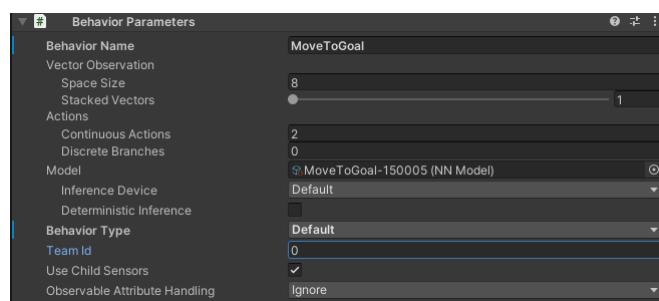


Figura 4.9: Configuración del Id para entornos adversariales.

Experimentación

5

En este capítulo vamos a hablar de la experimentación que hemos realizado durante el desarrollo del proyecto y a exponer los resultados obtenidos. En concreto, vamos a hablar de los diferentes entornos que hemos creado, tanto monoagentes como multiagentes, así como de las diferentes configuraciones y técnicas de entrenamiento que hemos utilizado en cada caso y de los resultados logrados.

Se adjunta además en el Apéndice A un enlace, el cual lleva a un vídeo de demostración donde se puede observar a los diferentes agentes en funcionamiento una vez ya han sido entrenados.

Para la realización de los entornos, así como para el entrenamiento de los mismos, se ha utilizado un ordenador con las siguientes características

- Operating System: Windows 10 Home 64-bit (10.0, Build 19043)
- Processor: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz (12 CPUs), 2.6GHz
- RAM Memory: 16384MB
- Card name: NVIDIA GeForce RTX 2070

5.1. Diseño del agente final

Primero que nada es necesario hablar del agente final que he diseñado, el cual he reutilizado para la resolución de todos los entornos, tanto los monoagente como los multiagente.

Se decidió finalmente utilizar un agente con forma de cubo, a modo de símil del clásico robot móvil "Turtlebot 3". Este robot consta de un modelo de conducción diferencial mediante el cual puede moverse instantáneamente adelante o atrás, pero no lateralmente por el deslizamiento de las ruedas. (Fig.5.1)

En este tipo de modelo, el cambio de dirección se realiza modificando la velocidad relativa de las ruedas a Izquierda-Derecha, pero por simplicidad voy a permitir que el robot sea capaz de desplazarse hacia delante y hacia detrás en línea recta, y para girar es capaz de rotar respecto al eje Y. Así pues, los agentes pueden moverse aplicando una fuerza sobre sí mismos en las direcciones x e z, así como rotar a lo largo del eje y, pero además, se añadió al agente una funcionalidad de salto, de manera que sea capaz de sortear obstáculos y pueda moverse verticalmente con mayor facilidad por el entorno. (Fig.5.2)

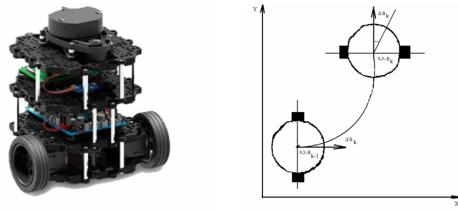


Figura 5.1: Modelo de conducción del robot Turtlebot3.

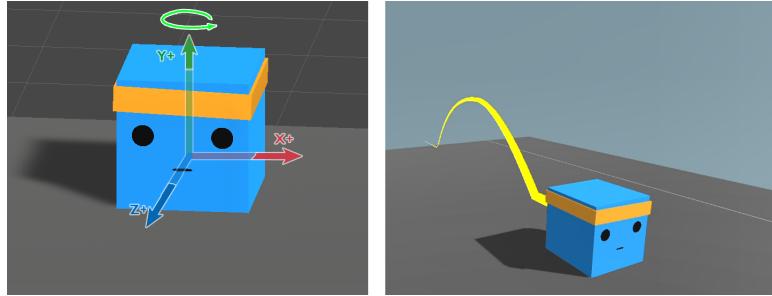


Figura 5.2: Modelo de conducción del Agente Final.

Para ello, se decidió que el agente se moviera utilizando tres ramas de acciones discretas, una de ellas para decidir si moverse recto, hacia atrás, o quedarse quieto, otra rama para decidir si rotar hacia la derecha, la izquierda, o no rotar, y finalmente otra rama para decidir si saltar o no.

Una vez hecho esto, procedimos a añadir sensores adicionales al agente, de manera que pueda obtener más información del entorno, además de simular de manera más realista el comportamiento de un robot móvil real. Los sensores extras que se añadieron son los siguientes:

- **Observaciones Raycast:** RayPerceptionSensorComponent3D.

Se decidieron añadir dos Raycast Sensors, de manera que actúen a modo de sensor Lidar, siendo capaces por lo tanto de medir a qué distancia están los diferentes objetos, paredes y otros agentes a su alrededor.

Por lo tanto, el **1er raycast** (Fig.5.3) tiene un total de 9 rayos que apuntan hacia el suelo en un rango de 180 grados, y se encarga de detectar la presencia del suelo así como de los elementos necesarios. Se utiliza para que el agente pueda detectar si tiene riesgo de caerse, o si encuentra algún elemento importante para el desarrollo de la tarea en su perímetro en caso de ser necesario.

El **2o raycast** (Fig.5.4), el cual tiene un total de 12 rayos esparcidos en un rango de 360 grados, se encarga de detectar la distancia a las paredes, los obstáculos, otros agentes, y en caso de ser necesario, la meta.

- **Observaciones visuales:** CameraSensor (Fig.5.5)

Se añadió también una **Cámara de 21x21 px**, de manera que el agente sea capaz de ver todos los elementos necesarios que se encuentren en su línea de visión. La información de la cámara se recopila y transforma en un tensor 3D, el cual como comentamos anteriormente

en el apartado de metodología, se puede alimentar a una red neuronal convolucional (CNN) para modificar la política del agente.

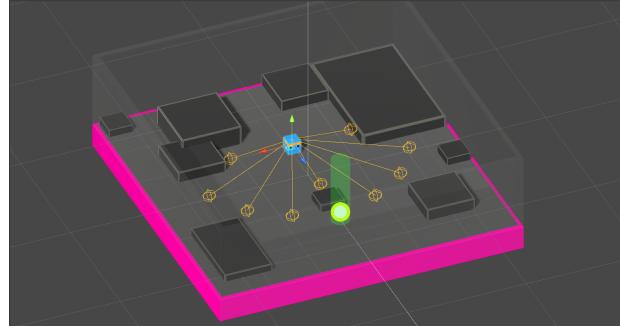


Figura 5.3: RayPerceptionSensorComponent3D nº1.

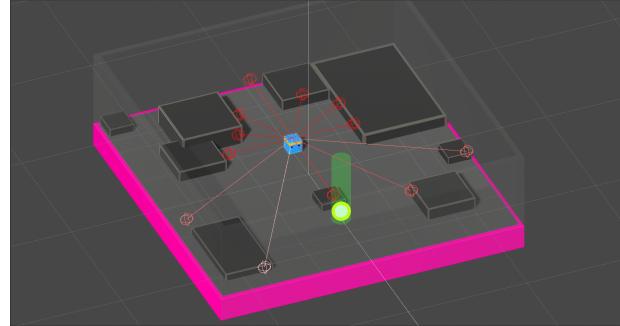


Figura 5.4: RayPerceptionSensorComponent3D nº2.

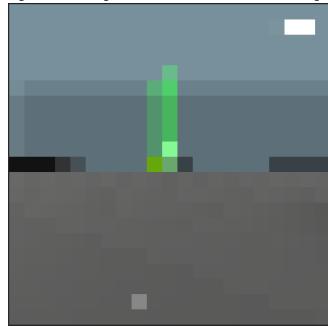


Figura 5.5: CameraSensor (21X21 px).

Una vez decidido el movimiento del agente, y configurados los sensores externos, nos quedaría todavía configurar los parámetros dependientes del entorno y de la tarea a realizar, que en este caso serían las observaciones vectoriales del agente, y las recompensas extrínsecas. Estos parámetros se configuran desde el script C# del agente, los cuales comentaremos en los apartados correspondientes a cada uno de los entornos.

Además, como comentamos en el apartado de Metodología, también será necesario crear un fichero YAML de configuración en el cual plasmamos los diferentes hiperparámetros necesarios para el correcto funcionamiento del algoritmo PPO o MA-POCA, y el correcto aprendizaje del agente.

El fichero YAML de configuración que se utilizó para resolver los entornos monoagentes (PPO) se puede encontrar en el Apéndice A (Fig.A.7), así como también el fichero utilizado para resolver los entornos multiagentes (MA-POCA) (Fig.A.8):

Para el fichero de configuración de los sistemas monoagente, ya que los entornos diseñados son relativamente complejos y constan de varias entradas y salidas, además de haber modificado la función de recompensas y las observaciones, se decidió aumentar el número de batches y el tamaño del buffer respecto al entorno de ejemplo, así como a introducir el módulo de Curiosity Rewards para incitar al robot a explorar.

Para el fichero de configuración de los sistemas multiagente, se decidió tomar valores incluso mayores para el número de batches y el tamaño del buffer, ya que la complejidad de dichos escenarios es considerablemente mayor. Además, también se configuró el parámetro learning_rate_schedule como constant, ya que debido a la complejidad del entorno y a la necesidad de que los agentes aprendan a cooperar, se consideró más útil mantener la tasa de aprendizaje constante durante todo el recorrido de entrenamiento.

Además de esto, comentar que en alguno de los entornos se probó a implementar las técnicas de curriculum learning, así como GAIL y Behavioral Cloning, con el fin de ver cómo afectan al aprendizaje, y si realmente facilitaban el aprendizaje del agente. Dichos ficheros YAML se mostrarán en los apartados correspondientes a los entornos donde se probaron.

En consecuencia, con la configuración del agente terminada, tanto de sensores como en el editor de Unity (Fig.5.5), y variando entre los diferentes entornos únicamente el nombre del comportamiento, el número de observaciones que reciben los agentes, y el diseño de las recompensas (además del TeamId en aquellos entornos adversariales que lo requieran), se podrá proceder a entrenar los agentes en cada uno de los entornos.

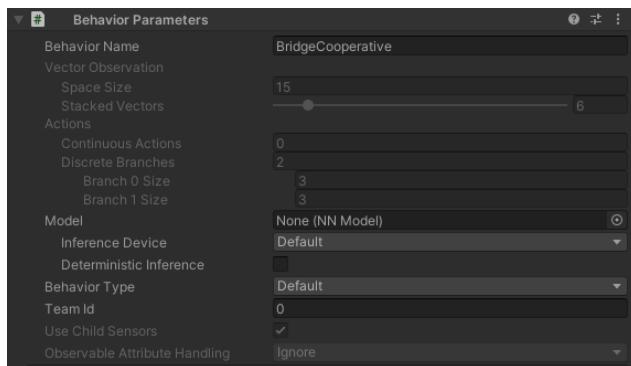


Figura 5.6: Configuración final de los parámetros del agente.

5.1.1. Sistema de Checkpoints

Antes de comentar el funcionamiento de los entornos, es necesario explicar cómo funciona el sistema de checkpoints que utilizaremos a lo largo de los diferentes entornos (Fig. 5.7), el cual nos servirá para guiar al agente hacia la solución. Este sistema usa una lista de objetos de Unity, y permite llevar un seguimiento acerca de si el agente pasa por los elementos de una lista de checkpoints en el orden correcto o no (Fig. 5.8). Así pues, se puede recompensar al agente cuando cruce por un checkpoint correcto, y si se desea, penalizarlo al cruzar por un checkpoint incorrecto.

Originalmente este sistema únicamente funcionaba en los sistemas monoagente, pero se expandió para sistemas multiagente, de manera que se pueda llevar el control de que agentes han pasado por los checkpoints, y de recompensarlos acorde a ello.

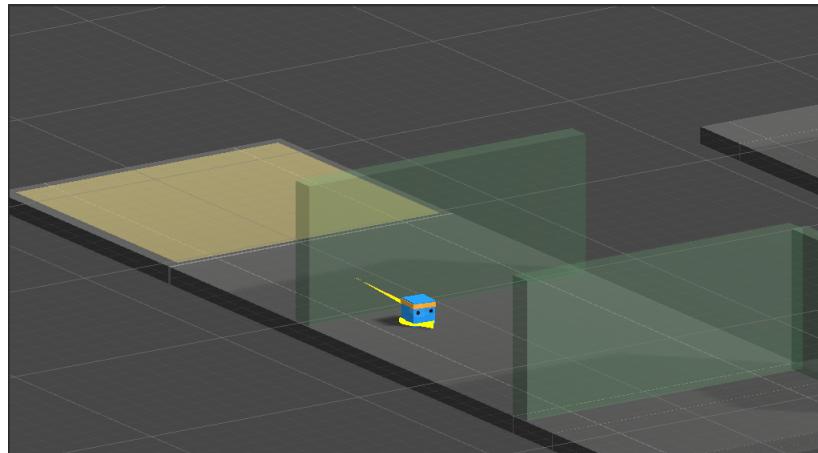


Figura 5.7: Sistema de Checkpoints: Imagen.

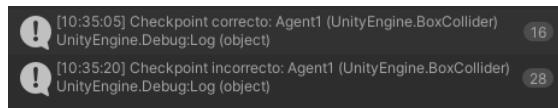


Figura 5.8: Sistema de Checkpoints: Mensajes durante la ejecución.

Por consiguiente, vamos a explicar el fichero mejorado, ya que se puede utilizar tanto en sistemas multiagente como en sistemas monoagente, añadiendo para ello los agentes necesarios a la lista. Cada uno de los sistemas consta de dos ficheros C#. El primer fichero, llamado '**CheckpointsMulti**' (Fig. A.15), se asigna a todos y cada uno de los checkpoints y se utiliza para detectar la

colisión del agente con los mismos. Al detectarse la colisión, se activa un evento de C# y se llama al segundo fichero.

El segundo fichero, llamado '**TrackCheckpoints**' (Fig. A.16) y el cual se añade a un objeto de Unity en el cual se encuentran agrupados todos los checkpoints (Fig. 5.9), se encarga de llevar el orden de los checkpoints para determinar si se cruzan los checkpoints por orden o no. En caso de cruzar un checkpoint correcto o incorrecto, se llamaría a una función determinada del script del agente, en este caso, **CheckpointCorrecto()** o **CheckpointIncorrecto()** (Fig. 5.10), las cuales principalmente se utilizan para recompensar al agente.

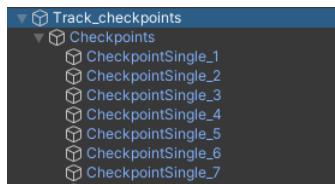


Figura 5.9: Sistema de Checkpoints: Agrupación de checkpoints en el editor de Unity.

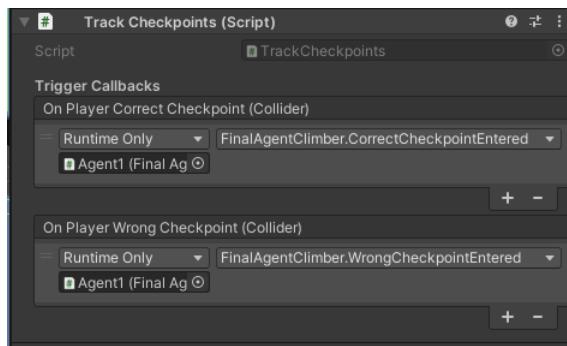


Figura 5.10: Sistema de Checkpoints: Configuración desde el editor de Unity.

Una vez comentado esto, estamos listos para comenzar a explicar el procedimiento llevado a cabo para cada uno de los entornos finales.

5.2. Entornos finales

Como he mencionado anteriormente, a continuación se procederán a mencionar los diferentes entornos que se han ido creando, así como los problemas y soluciones que se han aplicado a cada uno.

Recordar además, que como bien hemos comentado anteriormente, el objetivo del trabajo es principalmente el de analizar cómo se comporta un mismo agente entrenado en diferentes entornos y con diferentes tareas objetivo. Se busca por lo tanto enfatizar en la creación y configuración de los diferentes entornos, de manera que se pueda comprobar el comportamiento y entrenamiento realizado de un mismo agente en los diversos entornos y compararlo.

Primero procederemos a analizar los diferentes entornos monoagente, y finalmente analizaremos los entornos multiagentes.

5.2.1. Entornos monoagentes

Se decidió comenzar por el diseño de los entornos monoagente principalmente por su reducida complejidad respecto a los entornos que requieren cooperación, pero también se hizo para que estos entornos pudieran servir de base para los entornos multiagente.

Pensando en qué comportamientos básicos debería tener un agente móvil con un mínimo de inteligencia, he desarrollado 3 entornos finales. Dichos comportamientos básicos se relacionan cada uno directamente con uno de los entornos monoagente realizados, ya que han sido especialmente diseñados para poner esa tarea a prueba.

Los comportamientos básicos que se corresponden con los entornos finales son los siguientes:

- **Moverse hacia un objetivo esquivando obstáculos.** (Entorno 1: Climb, Fig. 5.11).
- **Mover un objeto hacia una posición esquivando obstáculos.** (Entorno 2: Push, Fig. 5.15).
- **Interactuar directamente con el entorno (Presionar Botones).** (Entorno 3: Bridge, Fig. 5.19).

5.2.1.1. Entorno 1: Climb

El primer entorno a comentar, y quizá el más sencillo de todos es el denominado como 'Climb' (5.7). Este entorno consiste en un circuito compuesto por un plano rectangular, seguido por una secuencia de giros y cuestas empinadas, tras las cuales se encuentra la meta. El objetivo del agente en este entorno concreto es de ser capaz de completar dicho recorrido en un tiempo adecuado, y posteriormente de aprender a completarlo esquivando una serie de obstáculos que pueden aparecer a lo largo del recorrido con cierta aleatoriedad.

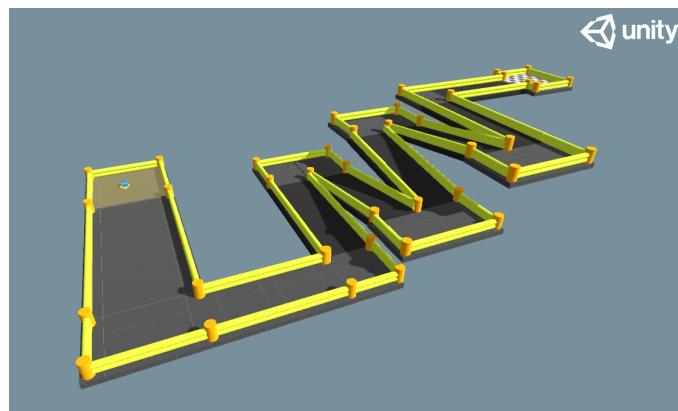


Figura 5.11: *Entorno monoagente 1: Climb.*

■ **Environment Randomization + Curriculum Learning:**

Así pues, en un comienzo el entorno no tendría ningún obstáculo y se entrenaría al agente para que aprenda únicamente a llegar a la meta. Una vez se consiga esto, mediante Curriculum Learning y Environment Randomization se establecería un sistema para que se generen obstáculos cada vez más altos conforme el agente aprende a avanzar en dirección de la meta, de manera que pueda aprender correctamente a saltar los obstáculos (Fig. 5.12).

Para esto fue necesario modificar el fichero de configuración YAML, y añadir una serie de lecciones (5 para este comportamiento), estableciendo en cada lección un mayor rango de la variable que establece la altura de los obstáculos (Fig. A.9). Además, es necesario incluir una nueva línea en el código del agente, la cual obtenga el valor de la variable altura directamente del fichero de configuración (Fig. A.10).

Además de esto, se configuró el entorno para que el agente aparezca al inicio del recorrido con una posición y orientación aleatoria dentro de la zona amarilla .

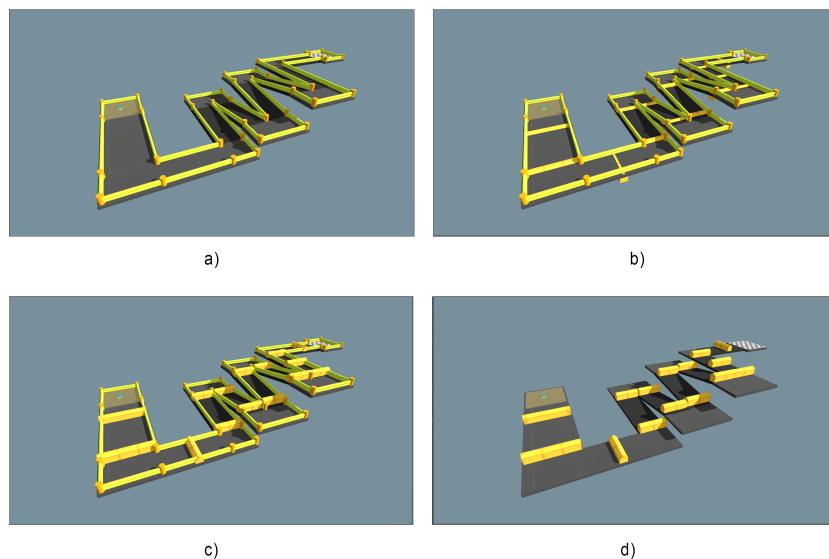


Figura 5.12: Curriculum Learning en Entorno monoagente 1: Climb.

■ **Observaciones:**

Respecto a las observaciones del agente en este entorno concreto, las cuales como bien hemos comentado en el apartado de metodología se sumarán a las observaciones de los sensores Raycast y la cámara de 21x21px, se seleccionaron las siguientes 8 variables:

- Posición del Checkpoint Siguiente (x,y,z)
- Posición del Agente (x,y,z)
- Velocidad del Agente en ejes x,z (x,z)

Se probó en un comienzo a añadir también la posición de la meta a las observaciones, pero ya que esta no varía a lo largo de los episodios, finalmente no fue necesario. Si se diseñara un entorno en el cual la posición de la meta cambiase en cada ejecución, sería recomendable añadirlo.

Por lo tanto, el agente es capaz de obtener información de sus sensores externos, de conocer tanto su posición como la posición del siguiente checkpoint, y de de conocer su velocidad, de manera que pueda controlar mejor para no caerse y sortear los obstáculos que puedan aparecer.

Finalmente, respecto a las observaciones del raycast, se configuraron de manera que el 1er raycast (Fig.5.3), el cual apunta hacia abajo, detecte tanto el suelo, como los posibles obstáculos, y el 2o raycast (Fig.5.4) se utilizará exclusivamente para detectar las paredes (si las hay), los checkpoints, y la meta.

■ **Recompensas:**

Respecto a las recompensas de este comportamiento y siguiendo las recomendaciones de la librería ML-Agents, inicialmente se hizo énfasis principalmente en las recompensas positivas, y una vez el agente fue capaz de aprender a moverse correctamente y consiguió un desempeño moderado, se agregaron el resto de recompensas negativas, de manera que desde un principio el aprendizaje del agente no se vea saturado por la recompensas negativas.

Así pues en un principio se configuraron tres recompensas positivas y una negativa. Las recompensas positivas se otorgan cuando el agente alcanza un checkpoint correcto, cuando consigue llegar a la meta, y cuando el agente se mueve recto sin saltar, para motivarlo a no saltar continuamente. La recompensa negativa en cambio, se otorga cuando el agente falla la tarea, lo cual ocurre al caerse del entorno, o al pasarse del tiempo máximo establecido (8000 steps).

Los valores de las recompensas iniciales son los siguientes

• **Recompensas positivas:**

- Recompensa al llegar a la meta (+0.5).
- Recompensa al cruzar un checkpoint correcto (+0.3 / NumCheckpoints).
- Recompensa al moverse recto sin saltar (+0.2 / MaxEnvironmentSteps).

• **Recompensas negativas:**

- Recompensa al caerse del entorno o pasarse del tiempo máximo (-1)

Una vez el agente consiguió avanzar hasta cierta parte del recorrido, se activaron el resto de recompensas. Las recompensas que se configuraron posteriormente se componen de una recompensa negativa a cada paso para motivarlo a acabar el recorrido lo antes posible, y otra recompensa negativa cuando el agente colisiona con las paredes, de manera que se mantenga lejos de estas (aunque si se quitan el agente es capaz de funcionar correctamente).

Comentar que la recompensa negativa a cada paso ayuda en cierta medida también a que el agente no esté continuamente saltando, ya que al saltar la velocidad del agente es menor que al moverse sin salto.

Se pensó también en añadir una recompensa negativa cuando el agente cruzaba un checkpoint incorrecto, pero finalmente no se implementó para motivar al agente a encontrar posibles atajos y exploits para llegar antes incluso a la meta (aunque eso le suponga saltarse checkpoints).

Las **recompensas finales** quedaron de la siguiente manera:

- **Recompensas positivas:**

Recompensa al llegar a la meta (+0.6).

Recompensa al cruzar un checkpoint correcto (+0.3 / NumCheckpoints).

Recompensa al moverse recto sin saltar (+0.1 / MaxEnvironmentSteps).

- **Recompensas negativas:**

Recompensa al caerse del entorno o pasarse del tiempo máximo (-1)

Recompensa de tiempo (-0.1 / MaxEnvironmentSteps)

Recompensa al colisionar con una pared (-0.1 / MaxEnvironmentSteps).

- **GAIL + BC:**

Finalmente se probó también el sistema de GAIL y Behavioral Cloning, para el cual hubo que previamente grabar una considerable cantidad de episodios de ejemplo controlando manualmente al agente. Para esto, se grabaron todos los episodios con la altura de los obstáculos establecida al máximo. Una vez hecho esto, se añadió la configuración de GAIL y de BC al fichero YAML, y se procedió a entrenar de nuevo. Para entrenarlo, se estableció en un comienzo la fuerza de GAIL y de BC a valores altos, de manera que el agente aprendiera a comportarse de manera igual a nosotros (BC), pero siendo capaz de superar incluso la puntuación de los comportamiento pregrabados (GAIL). Conforme el agente conseguía mejores puntuaciones, se fue disminuyendo el peso de los dos métodos comentados anteriormente, de manera que el Agente pueda desarrollar un comportamiento sobrehumano. Gracias a esto y como se verá más adelante en las gráficas del entrenamiento, se pudo comprobar cómo efectivamente el agente era capaz de aprender más rápido que comenzando a partir de un comportamiento completamente aleatorio.

- **Pruebas y resultados:**

Como se ha comentado anteriormente, se realizaron una considerable cantidad de pruebas en este entorno hasta encontrar una configuración que funcionaba correctamente, tanto de recompensas como de observaciones, pero finalmente con la configuración que acabamos de mencionar, el agente fue capaz de aprender correctamente a realizar el comportamiento deseado y de completar el circuito en un tiempo apropiado aun habiendo obstáculos de por medio.

A continuación mostraremos las gráficas con los resultados del entrenamiento obtenidas a través de Tensorboard, tanto del entrenamiento realizado sin GAIL y BC (Línea rosa), como del entrenamiento que si lo usa (Línea azul) (Fig. 5.13). Las diferencias más grandes entre estos entrenamientos es que como ya hemos comentado, en el entrenamiento que se utilizaron episodios pregrabados, se configuró para que desde el principio del entrenamiento el rango de variación de los obstáculos fuera el máximo posible.

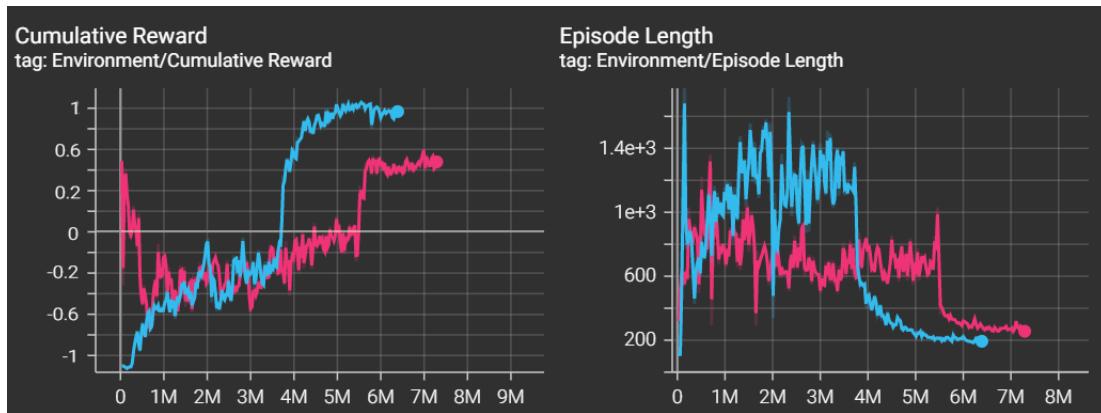


Figura 5.13: Resultados del entorno monoagente 1: Climb. Entrenamiento estándar (línea rosa) entrenamiento utilizando GAIL y BC (línea azul)

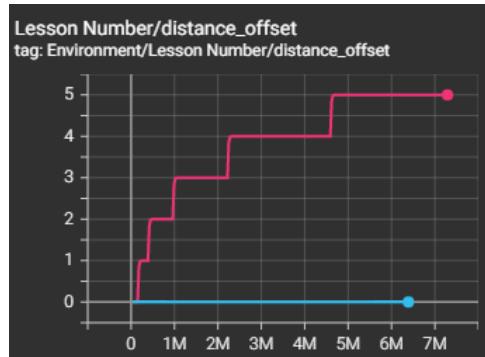


Figura 5.14: Resultados del Curriculum learning en el entorno monoagente 1: Climb.

Se puede ver como en los dos entrenamientos se consigue un comportamiento adecuado, necesitando de 6 y 4 millones de pasos de entrenamiento respectivamente en cada entrenamiento para obtener la máxima recompensa posible. Se puede ver en la figura 5.14 la cantidad de pasos necesarios para avanzar en cada lección del Curriculum Learning hasta entrenar al agente con los obstáculos en su máxima altura posible.

Se puede comprobar además cómo el haber incluido una recompensa negativa a cada paso, ha sido efectivo para reducir el tiempo de cada episodio, pero solo una vez el agente aprende a realizar la tarea con cierto nivel de precisión. Así pues, se puede observar como

conforme avanza el entrenamiento, el agente pasa de completar la tarea en más de 1.000 pasos por episodio, a aproximadamente una media de 200.

Finalmente, es necesario comentar también que el resultado entre los entrenamientos realizados pueden variar ampliamente aún sin modificar ningún parámetro. Esto principalmente se debe a la aleatoriedad del proceso de aprendizaje, y se puede solucionar simplemente ejecutando cada entrenamiento varias veces.

5.2.1.2. Entorno 2: Push

El segundo entorno a mencionar es el denominado como 'Push' (5.15). Este entorno consiste en un circuito compuesto por un rectángulo alargado, el cual tiene en un extremo la zona de inicio del agente y una esfera, y en la otra la meta. El objetivo del agente en este entorno es el de aprender a arrastrar una esfera desde la posición inicial hasta la meta, en un comienzo sin ningún obstáculo de por medio, y finalmente teniendo que empujar la esfera sorteando una serie de obstáculos que al igual que en el primer entorno, pueden aparecer a lo largo del recorrido con cierta aleatoriedad respecto a su altura.

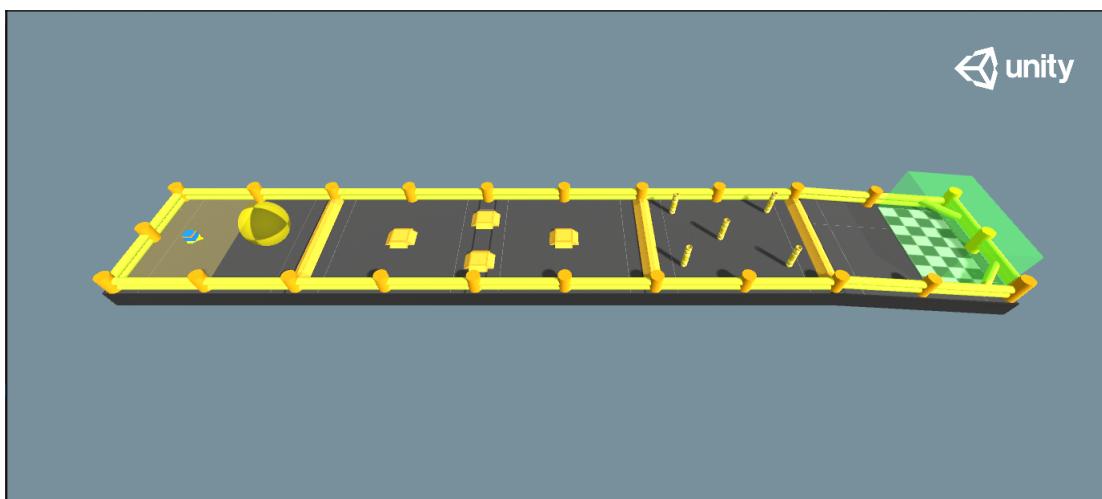


Figura 5.15: Entorno monoagente 2: Push.

- **Environment Randomization + Curriculum Learning:**

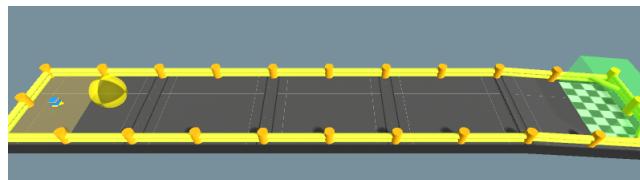
Al igual que en el entorno anterior, en este también se añadió un cierto nivel de aleatoriedad al agente, así como a los elementos del entorno. De esta manera, la posición y la rotación inicial del agente al comienzo del episodio es aleatoria (dentro de los límites de la zona amarilla). Además, se introdujeron una serie de obstáculos a lo largo del recorrido, de manera que gracias al curriculum learning, estos vayan aumentando poco a poco su altura conforme el agente aprende a acercar la esfera hacia la meta (Fig. 5.16).

■ Observaciones:

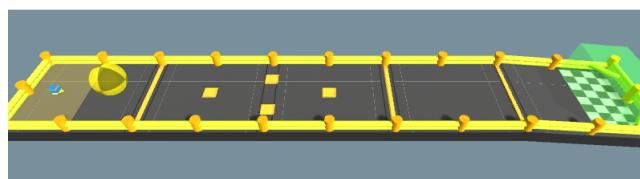
Respecto a las observaciones del agente, en este entorno se seleccionaron las siguientes 11 variables vectoriales:

- Posición del Checkpoint Siguiente (x)
- Posición del Agente (x,y,z)
- Posición de la esfera (x,y,z)
- Velocidad del Agente en ejes x,z (x,z)
- Velocidad de la esfera en ejes x,z (x,z)

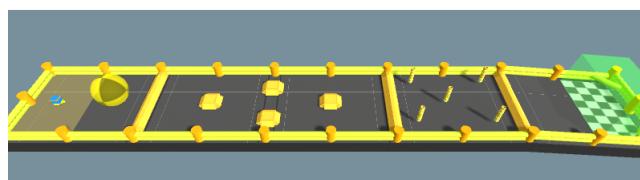
Con estas 11 observaciones, el agente es capaz de conocer la posición del siguiente checkpoint por el que tiene que cruzar la esfera, así como la posición y la velocidad de la esfera además de la suya.



a)



b)



c)

Figura 5.16: Curriculum Learning en Entorno monoagente 2: Push.

■ Recompensas:

Respecto a las recompensas de este comportamiento, se siguió una metodología similar a la utilizada en el entorno anterior. De esta manera se configuraron primero las recompensas positivas, así como una recompensa negativa para cuando el agente no conseguía completar la tarea en el tiempo máximo, y una vez el agente comienza a aprender correctamente, se

activaron el resto de recompensas negativas para optimizar el comportamiento del agente. Además, para este entorno se probó a incrementar el valor de las recompensas positivas por encima de 1 para comprobar si esto afectaba al aprendizaje, así como a la velocidad del entrenamiento.

La mayor diferencia respecto al comportamiento anterior es que en este caso la recompensa que se le entregaba al agente al cruzar un checkpoint, aquí se activa cuando es la esfera la que los cruza. De esta manera se motiva al agente a empujar la esfera hasta la meta, donde acabaría el episodio después de recompensar al agente.

Las **recompensas finales** quedaron por lo tanto de la siguiente manera:

- **Recompensas positivas:**

- Recompensa al llegar la esfera a la meta (+2).
- Recompensa al cruzar un checkpoint correcto (+0.5 / NumCheckpoints).
- Recompensa al moverse recto sin saltar (+0.5 / MaxEnvironmentSteps).

- **Recompensas negativas:**

- Recompensa al caerse del entorno o pasarse del tiempo máximo (-0.5)
- Recompensa al cruzar un checkpoint incorrecto (-0.3 / NumCheckpoints).
- Recompensa de tiempo (-0.1 / MaxEnvironmentSteps)
- Recompensa al colisionar con una pared (-0.1 / MaxEnvironmentSteps).

Como se puede comprobar y a diferencia del comportamiento anterior, aquí sí que activamos las recompensas negativas cuando se cruza un checkpoint incorrecto, ya que no nos interesa que el agente permita que la bola se mueva en dirección contraria a la meta.

- **GAIL + BC:**

Para este entorno también se probó la funcionalidad de entrenar al agente mediante comportamientos pregrabados, partiendo directamente con la altura de los obstáculos establecida al máximo. De nuevo, se pudo comprobar cómo efectivamente el uso de estas técnicas pueden ayudar ampliamente a la hora de que el agente pueda aprender a realizar tareas simples de manera mucho más rápida que partiendo de movimientos aleatorios (Fig. 5.17).

- **Pruetas y resultados:**

Tras configurar correctamente las recompensas y las observaciones necesarias para este entorno, el agente aprendió de manera adecuada a realizar el comportamiento deseado en un tiempo apropiado, aun sorteando los obstáculos de por medio, como se puede comprobar en las gráficas que se muestran a continuación.

A continuación se muestran las gráficas con los resultados del entrenamiento, obtenidas a través de Tensorboard, y de nuevo, tanto con el entrenamiento realizado sin GAIL y BC (Línea naranja), como con el entrenamiento que si lo usa (Línea gris). Se puede comprobar

que al igual que ocurría en el episodio anterior, el entrenamiento en el que utilizamos GAIL y BC alcanza la convergencia mucho antes que en el entrenamiento que no se usa dicha funcionalidad, aún cuando al habilitar GAIL y BC se usa la máxima aleatoriedad desde el principio (Fig. 5.17, Fig. 5.18).

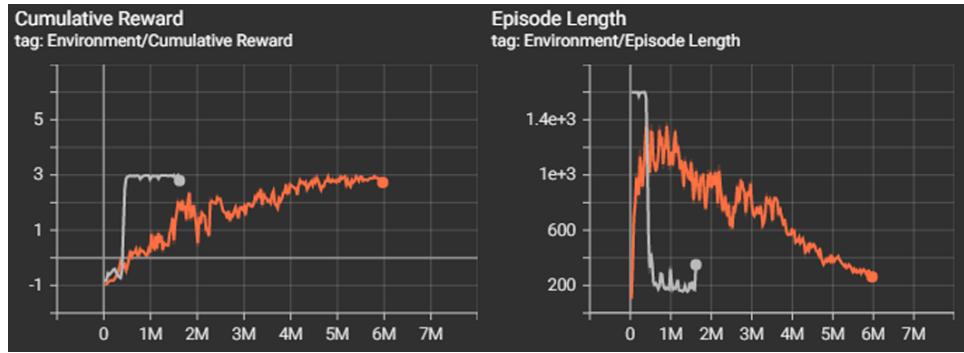


Figura 5.17: Resultados del entorno monoagente 2: Push. Entrenamiento estándar (línea naranja) entrenamiento utilizando GAIL y BC (línea gris)

Se puede ver como en los dos entrenamientos se consigue un comportamiento adecuado, necesitando de 6 y 2 millones de pasos de entrenamiento respectivamente en cada entrenamiento para obtener la máxima recompensa posible. También se puede ver cómo de manera similar al entorno anterior, el agente finalmente consigue completar los episodios en aproximadamente 200 pasos por episodio (Fig. 5.17), aunque este comportamiento todavía se podría mejorar si se siguiera entrenando con las recompensas adecuadas.

Para finalizar, se puede ver a continuación una imagen con la cantidad de pasos necesarias para avanzar en cada lección del Curriculum Learning hasta entrenar al agente con los obstáculos en su máxima altura posible (Fig. 5.18).

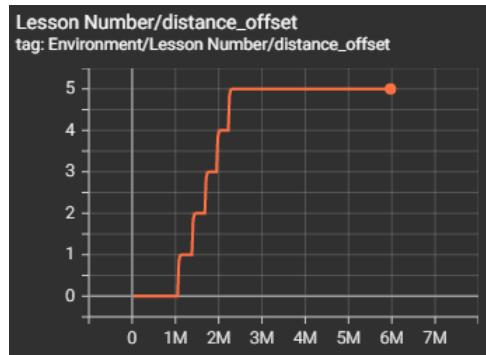


Figura 5.18: Resultados del Curriculum learning en entorno monoagente 2: Push.

5.2.1.3. Entorno 3: Bridge

Finalmente, el 3o y último de los escenarios monoagente es el denominado 'Bridge' (Fig 5.19). Este entorno consiste en dos áreas separadas por una zona de agua, y un puente entre las dos. En la primera de las áreas se puede ver de nuevo la zona amarilla donde aparecerá el agente, así como un botón rojo, el cual tras ser pulsado activará el puente, haciendo que este suba y permitiendo al agente cruzarlo y llegar a la segunda área, donde se encuentra la meta. El objetivo del agente por lo tanto es el aprender a interactuar con el entorno y a encontrar una estrategia para activar el puente y llegar a la meta lo antes posible sin caerse.

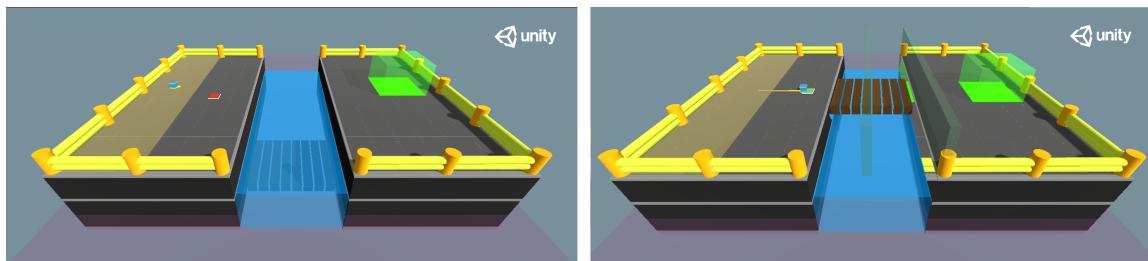


Figura 5.19: Entorno monoagente 3: Bridge.

Comentar antes que nada que para este entorno, así como para su versión multiagente, se deshabilitará la función del agente del salto, ya que no ayuda para la tarea, y además complica considerablemente el entrenamiento.

Este es el escenario monoagente sobre el que más pruebas se han realizado, por lo que a continuación se resume el proceso realizado hasta encontrar los mejores parámetros para este comportamiento.

- **Environment Randomization + Curriculum Learning:**

Para este entorno en un comienzo todas las posiciones, tanto del agente, como del botón y de la meta son estáticas, y conforme el agente comienza a conseguir una mayor recompensa se aumenta la aleatoriedad de dichos elementos.

De este modo, el área donde el agente puede generarse es cada vez mayor (llegando al final a abarcar toda la zona amarilla), y lo mismo ocurriría con el botón y con la meta. El puente se comporta de manera similar, pero únicamente variando su posición a lo largo de la separación entre las dos áreas, de manera que pueda aparecer en cualquier parte entre las dos zonas, pero sin llegar a salirse de las mismas (Fig 5.20).

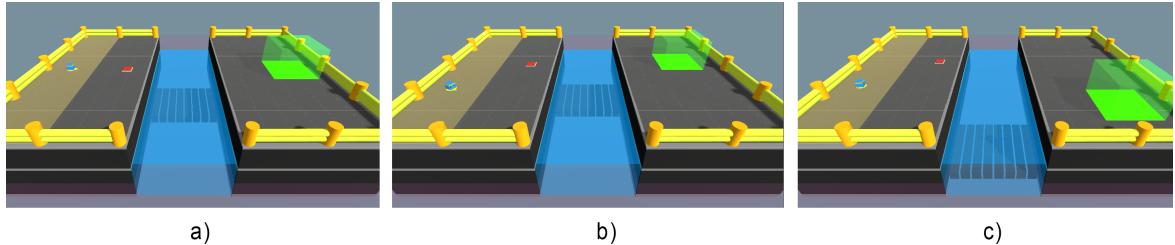


Figura 5.20: Curriculum Learning en Entorno monoagente 3: Bridge.

■ **Observaciones:**

Respecto a las observaciones del agente, y tras probar una larga variedad de combinaciones, finalmente se seleccionaron las siguientes 13 variables:

- Posición del Checkpoint Siguiente (x)
- Posición del Agente (x,y,z)
- Posición de la Meta (x,z)
- Posición del Botón (x,z)
- Posición del Puente (y,z)
- Puente activado (Bool)
- Velocidad del Agente en ejes x,z (x,z)

Dichas variables, de manera similar al resto de los entornos se seleccionaron para ayudar al agente a encontrar los diferentes elementos importantes que se encuentran en la escena, y sobretodo aquellos elementos que son móviles, o que en cada iteración tienen una posición diferente (como ocurre con el botón, el puente y la meta). Aún así, se probó también a entrenar al agente únicamente con las observaciones obtenidas de los sensores raycast y de la cámara, y aunque el tiempo de entrenamiento se alarga considerablemente, el agente seguía siendo capaz de aprender y con el tiempo, de cruzar el puente y llegar a la meta.

■ **Recompensas:**

Respecto a las recompensas de este comportamiento, podemos decir que la diferencia respecto al resto de comportamientos consiste en que se añadió una recompensa cuando el agente activa el puente. De esta manera se incita al agente a pulsar el botón, y que así pueda cruzar a la otra área y llegar hasta la meta, obteniendo además recompensas al cruzar los checkpoints que se encuentran de camino para guiarlo más todavía.

Las **recompensas finales** quedaron de la siguiente manera:

• **Recompensas positivas:**

- Recompensa al llegar a la tarea (+0.5).
- Recompensa al pulsar el botón (+0.1 / MaxEnvironmentSteps).
- Recompensa al cruzar un checkpoint correcto (+0.2 / NumCheckpoints).

• Recompensas negativas:

- Recompensa al caerse del entorno o pasarse del tiempo máximo (-0.5)
- Recompensa al cruzar un checkpoint incorrecto (-0.3 / NumCheckpoints).
- Recompensa de tiempo (-0.1 / MaxEnvironmentSteps)
- Recompensa al colisionar con una pared (-0.1 / MaxEnvironmentSteps).

Al igual que en el anterior escenario, comentar que aquí también hemos activado la recompensas negativas al cruzar un checkpoint incorrecto, de manera que el agente una vez cruzado el puente no sienta la necesidad de volver hacia atrás.

■ GAIL + BC:

En este entorno también se probaron las técnicas de GAIL y BC, probando también si se puede conseguir un buen entrenamiento si establecemos desde un principio la aleatoriedad al máximo. Se pudo comprobar como al activarlo desde un principio en este entorno, el agente no conseguía aprender tan rápido como cuando se grabaron episodios sin aleatoriedad y se permitió que el agente aprendiera a partir de ahí pero utilizando también curriculum learning. Esto se puede deber por ejemplo a que no se grabaran los suficientes episodios, o a que se introdujera demasiada complejidad desde el inicio del entrenamiento.

■ Pruebas y resultados:

Como ya se ha comentado, se realizaron una buena cantidad de pruebas sobre este entorno, pero únicamente se mostrarán a continuación las gráficas de 3 de los entrenamientos realizados. Todos ellos, utilizando la configuración comentada anteriormente.

A continuación se pueden ver las gráficas con los resultados del entrenamiento, obtenidas a través de Tensorboard (Fig. 5.21 y Fig. 5.22). Así pues, la línea de color verde corresponde al entrenamiento estándar sin clonación de comportamiento, la línea azul corresponde al entrenamiento aplicando GAIL y BC con la aleatoriedad máxima desde el principio, y finalmente, la línea de color naranja corresponde al entrenamiento en el que también se aplica GAIL y BC, pero esta vez aplicando Curriculum Learning de la misma manera que el entrenamiento estándar.

Se puede ver en las imágenes como en menos de 3 millones de episodios, todos los entrenamientos consiguen un buen comportamiento, llegando a obtener la recompensa máxima aun complicando paulatinamente el entorno mediante curriculum learning.

Finalmente, se puede comprobar como aquí también influye el incluir una recompensa negativa a cada paso, consiguiendo en los últimos millones de pasos que el agente aumente considerablemente la velocidad a la que completa la tarea hasta llegar en casi todos los comportamientos a una media de entre 50 y 100 pasos por episodio.

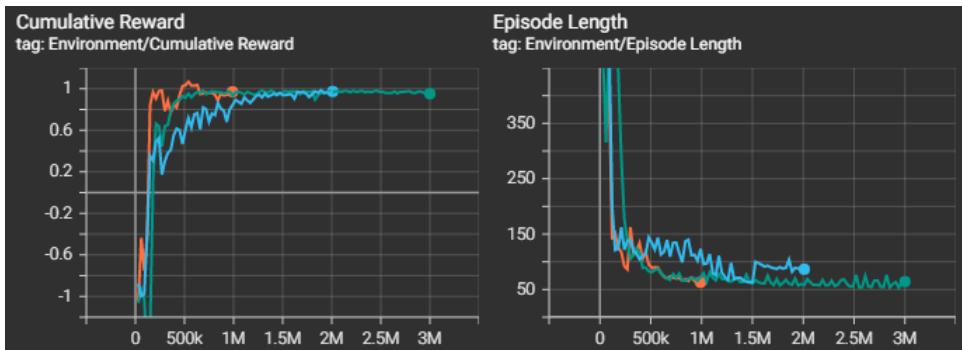


Figura 5.21: Resultados del entorno Bridge 3: Bridge. Entrenamiento estándar (línea verde), entrenamiento utilizando GAIL y BC con aleatoriedad máxima (línea azul), entrenamiento utilizando GAIL y BC con Curriculum Learning (línea naranja)

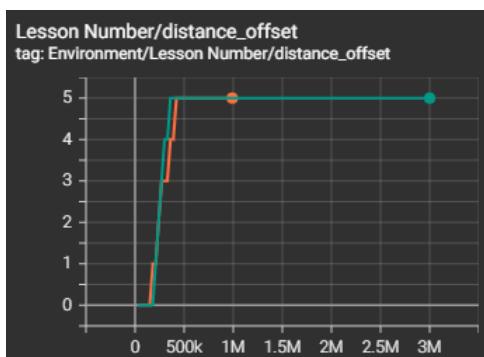


Figura 5.22: Resultados del Curriculum learning en entorno Bridge 3: Bridge.

5.2.2. Entornos multiagentes

A continuación vamos a proceder a describir los 3 diferentes entornos multiagentes desarrollados, así como las consideraciones necesarias a tener en cuenta para su correcto funcionamiento. Estos entornos se pueden considerar como una ampliación directa de cada uno de los entornos monoagente, habiendo probado además en cada uno una característica diferente.

Para ello, utilizamos el entorno multiagente 'Climb' para realizar un comportamiento adversativo donde un agente compita contra otro, haciendo para ello además uso del Self-Play. Respecto al entorno 'Push', lo utilizaremos para comparar el rendimiento de un solo agente para completar la tarea, frente al uso de dos o más agentes. Y finalmente, utilizaremos el entorno 'Bridge' para probar si mediante el uso de dos agentes, estos son capaces de aprender a realizar tareas para las cuales obligatoriamente deben aprender a cooperar y delegar tareas.

Es importante recordar que para estos entornos se va a proceder a utilizar el entrenador MAPPOCA, para el cual podemos definir recompensas intrínsecas para los agentes, así como recompensas grupales, las cuales es necesario configurar correctamente para que se produzca un aprendizaje adecuado. Las recompensas individuales utilizadas en cada uno de los entornos multiagente están diseñadas de manera similar a sus símiles monoagente, y se utilizan para motivar al agente a repetir una serie de acciones concretas y a orientarlo para completar la tarea.

Sin embargo y aunque se comportan de manera similar, las recompensas grupales se centran más en los hitos que se alcanzan mediante la cooperación entre los agentes.

Para todo esto es necesario modificar el código de los agentes, de manera que desde el código de cada uno de los agentes únicamente se controle el movimiento del agente, las observaciones que toma, y las recompensas individuales que obtiene (p. ej. al cruzar un checkpoint, caerse del entorno, o llegar a la meta). Por ello, es necesario crear para cada uno de los entornos un nuevo script global, desde el cual se controle todo el comportamiento del entorno, así como se monitorice el comportamiento de los agentes y se asignen las recompensas de grupo.

Una vez realizados estos cambios para cada uno de los entornos, y configurado el sistema de checkpoints para detectar varios agentes, podremos proceder a modificar las recompensas y las observaciones de cada uno de ellos. Por lo tanto, los diferentes entornos multiagentes que se probaron son los siguientes:

5.2.2.1. Entorno 1: Climb multiagente con self-play

Como se ha comentado al principio de este subapartado, se utilizará este entorno para probar la funcionalidad del Self-Play, y analizaremos si al implementarlo se consigue desarrollar un entorno con varios agentes, los cuales mediante la competición consigo mismos sean capaces de completar el circuito con mayor velocidad a los agentes entrenados en el entorno monoagente, o en su lugar, analizar si consiguen desarrollar algún otro tipo de estrategia. De esta manera, cuando uno de los agentes consiga llegar a la meta, el episodio finalizará, y se le recompensará positivamente, mientras que al otro agente se le recompensará negativamente (Fig. 5.23).

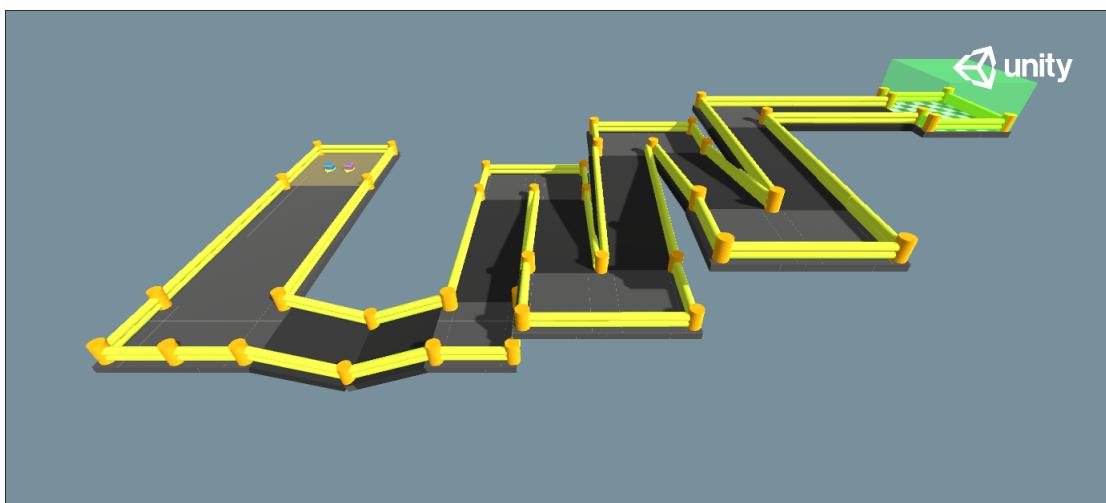


Figura 5.23: *Entorno multiagentes 1: Climb.*

Respecto a las **observaciones** que toma el agente, la única modificación realizada respecto al entorno base monoagente, es que se le ha añadido a cada uno de los agentes una nueva observación con la posición local del otro agente de la escena, de manera que en todo momento conozcan tanto su posición y la del otro agente.

Respecto a las **recompensas** que obtiene el agente, están se han dividido en dos tipos, recompensas individuales y de grupo. Además, se probó en este entorno a incrementar el valor de las recompensas positivas de manera que se motive más todavía a los agentes a llegar a la meta, ya que con las recompensas iguales al entorno monoagente y al activar el Self-Play, el entrenamiento se dificulta y tardaba demasiado en aprender. A continuación se muestran las recompensas que finalmente se establecieron para completar este entorno:

■ **Recompensas individuales:**

• **Recompensas positivas:**

- Recompensa al cruzar un checkpoint correcto (+3 / NumCheckpoints).
- Recompensa al moverse recto sin saltar (+0.5 / MaxEnvironmentSteps).
- Si un agente llega a la meta, (+3.0 para dicho agente, -1.0 para el otro).

• **Recompensas negativas:**

- Recompensa al caerse del entorno o pasarse del tiempo máximo (-1)
- Recompensa de tiempo (-0.5 / MaxEnvironmentSteps)
- Recompensa al colisionar con una pared (-0.25 / MaxEnvironmentSteps).

■ **Recompensas Grupales:** Estas recompensas se establecen de manera que se incite a los agentes a competir entre sí, además de motivarlos a completar la tarea, de manera que aquel que llegue primero a la meta debería obtener mayor recompensa.

• **Recompensas positivas:**

- Recompensa al cruzar un checkpoint correcto (+4f / NumCheckpoints).
- Si un agente llega a la meta (+2).

Una vez hecho todo esto, procedemos a realizar los cambios necesarios para activar el Self-Play. Dichas modificaciones como bien se mencionan en el apartado de metodología, consisten en primero modificar el 'Team ID' de cada uno de los agentes, de manera que se definan los equipos que se enfrentarán entre sí, y posteriormente en añadir un nuevo apartado al fichero de configuración YAML con los hiperparámetros de Self-Play.

Como se puede comprobar en la figura (Fig. A.8), para entrenar un sistema multiagente funcionando con el algoritmo MA-POCA y la funcionalidad de Self-Play, únicamente es necesario cambiar del fichero de configuración el tipo de entrenador, y añadir un apartado final con los parámetros del Self-Play. Respecto a los hiperparámetros utilizados, estos se modificaron respecto a los entornos monoagente debido a la complejidad extra que surge de utilizar más de un agente y el algoritmo de MA-POCA. Por lo tanto, se ha aumentado tanto el tamaño de batch, como el del buffer, además de

configurar el learning rate como constante, de manera que el agente pueda continuar aprendiendo hasta que converja naturalmente.

Comentar además que para el resto de entornos multiagentes, se utilizaran los mismos hiperparámetros que los mostrados en la figura (Fig. A.8), con la excepción del parámetro de Self-Play que únicamente se utilizará en este entorno.

Así pues, se comenzó a entrenar el entorno y tras muchas pruebas, se consiguió que los dos agentes fueran capaces de alcanzar la meta. Gracias a esto se puede comprobar como al entrenar los agentes de manera competitiva, estos siguen siendo capaces de alcanzar la meta y completar el recorrido.

A continuación se muestran en las siguientes gráficas (Fig. 5.24) los resultados del entrenamiento, y una comparativa con la duración de los episodios entre los modelos monoagente y este (Fig. 5.25).

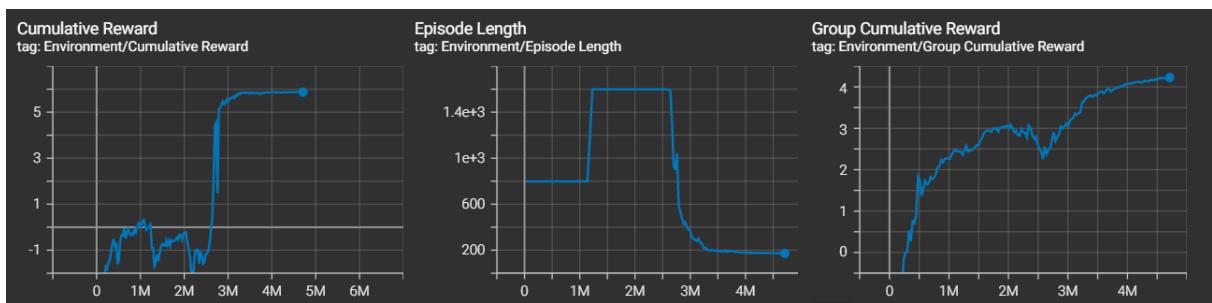


Figura 5.24: Gráficas del entrenamiento del entorno multiagente 2: Climb.

Name	Smoothed	Value	Step	Time	Relative
FinalMultiClimb_6\Climb	172.3	172.3	4.71M	Fri Aug 26, 23:05:45	13h 49m 30s
FinaClimbTest_3\Climb	254.1	254.1	7.29M	Wed Aug 10, 01:19:06	6h 59m 32s
FinaClimbTest_4\Climb	193.3	193.3	6.39M	Wed Aug 10, 19:00:32	5h 33m 21s

Figura 5.25: Comparación del resultado entre el entorno multiagente y los entorno monoagente Climb

Se puede observar por lo tanto, como gracias al self-play se consigue que los agentes aprendan a realizar la tarea, y sean capaces incluso de lograrlo en un tiempo menor al que se conseguía utilizando los modelos generados en el entorno monoagente, aunque no con una diferencia muy notoria.

5.2.2.2. Entorno 2: Push multiagente

Para este segundo entorno multiagente, y como se ha comentado anteriormente, se analizará si se puede mejorar el rendimiento de la tarea de empujar un objeto hacia la meta, únicamente aumentando los agentes que hay en el entorno. Para esto, se probará a entrenar el mismo entorno con 2, 3 y 4 agentes, y se le añadirán recompensas grupales para motivarlos a cooperar y trabajar en conjunto.

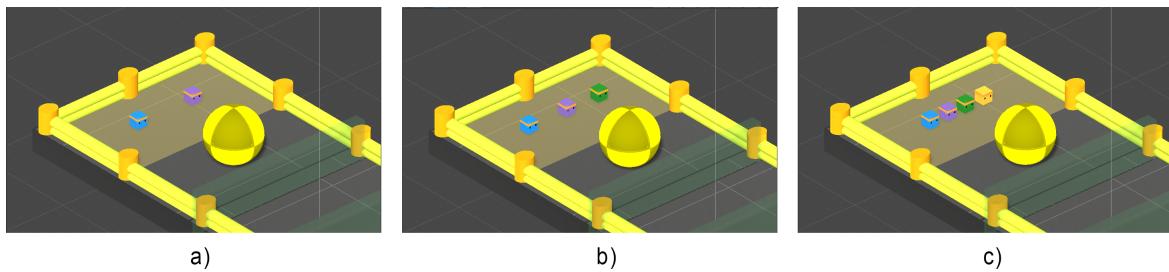


Figura 5.26: Entornos multiagente 2: Push. (a) 2 agentes, (b) 3 agentes, (c) 4 agentes.

Al igual que ocurrió con el entorno anterior, lo primero que hay que hacer es modificar el código de los agentes y adaptar el entorno para funcionar con más de un agente. Posteriormente adaptamos tanto las recompensas como las observaciones.

Respecto a las **observaciones** que toma el agente y en comparación a las configuradas para el entorno monoagente, únicamente añadimos para cada uno de los agentes la posición (x,z) del resto de los agentes. El resto de observaciones siguen siendo las mismas que en dicho entorno.

Respecto a las **recompensas** que obtiene el agente, estas también se han dividido en dos tipos, recompensas individuales y de grupo, aunque para este entorno y respecto a la configuración realizada para el monoagente, se volvieron a normalizar las recompensas individuales entre [-1,1]:

- **Recompensas individuales:** Las recompensas individuales se quedan iguales al entorno monoagente, pero esta vez normalizadas, de manera que se sigue motivando al agente a empujar la esfera hacia la meta:

- **Recompensas positivas:**

- Recompensa al llegar la esfera a la meta (+0.6).
 - Recompensa al cruzar un checkpoint correcto (+0.3 / NumCheckpoints).
 - Recompensa al moverse recto sin saltar (+0.1 / MaxEnvironmentSteps).

- **Recompensas negativas:**

- Recompensa al caerse del entorno o pasarse del tiempo máximo (-0.5)

- Recompensa al cruzar un checkpoint incorrecto (-0.3 / NumCheckpoints).
- Recompensa de tiempo (-0.1 / MaxEnvironmentSteps)
- Recompensa al colisionar con una pared (-0.1 / MaxEnvironmentSteps).

■ **Recompensas Grupales:** Para determinar las recompensas grupales se añadieron dos funciones extra, las cuales calculan la distancia media entre los agentes de la escena, así como la distancia media de todos los agentes a la esfera. De esta manera se puede controlar la agrupación de los agentes (tarea donde los miembros del enjambre, inicialmente dispersos en el entorno, deben agregarse para acercarse unos a otros en un mismo espacio físico), así como su agrupación respecto a la bola (distancia a la bola). De esta manera se incita a los diferentes agentes a no separarse, evitando que solo una parte de los agentes empujen a la esfera mientras el resto se encuentran dispersos por el resto del escenario.

• **Recompensas positivas:**

- Si se llega a la meta (+3)
- Recompensa de agrupación si distancia entre agentes <x
(+2 / MaxEnvironmentSteps)
- Recompensa de agrupación si distancia agentes-esfera <y
(+1 / MaxEnvironmentSteps)
- Recompensa al cruzar un checkpoint correcto (+1 / NumCheckpoints).

• **Recompensas negativas:**

- Recompensa de agrupación si distancia entre agentes >x
(-2 / MaxEnvironmentSteps)
- Recompensa de agrupación si distancia agentes-esfera >y
(-1 / MaxEnvironmentSteps)
- Recompensa al cruzar un checkpoint incorrecto (-1f / NumCheckpoints).

Una vez modificadas las recompensas y añadidas las observaciones necesarias, procedimos a configurar el fichero de configuración YAML con el entrenador MA-POCA, y una vez configurado todo correctamente, procedimos a entrenar los agentes.

Comentar que las diferencias que se observarán entre cada uno de los entrenamientos en las recompensas individuales de las gráficas se debe a que cuantos más agentes se encuentran en la escena, mayor es la recompensa acumulativa obtenida. Esto se principalmente a las recompensas obtenidas cuando la esfera cruza un checkpoint correcto, ya que además de obtener una recompensa grupal, también se otorga a cada uno de los agentes una recompensa positiva individual, haciendo que a más agentes en la escena, menos pesos tengan las recompensas negativas, y mayor sea por lo tanto la media. Además, en el entorno con únicamente 2 agentes, se comenzó entrenando sin la recompensa negativa a cada paso, pero se decidió añadir al resto de entrenamientos desde el principio, ya que se comprobó que no afectaba demasiado al entrenamiento,

Se muestran a continuación las gráficas con cada uno de los 3 entrenamientos (Fig. 5.27). La línea azul corresponde al entorno con un agente, la línea de color naranja al entorno con 3 agentes, y la línea verde al entorno con 4 agentes.

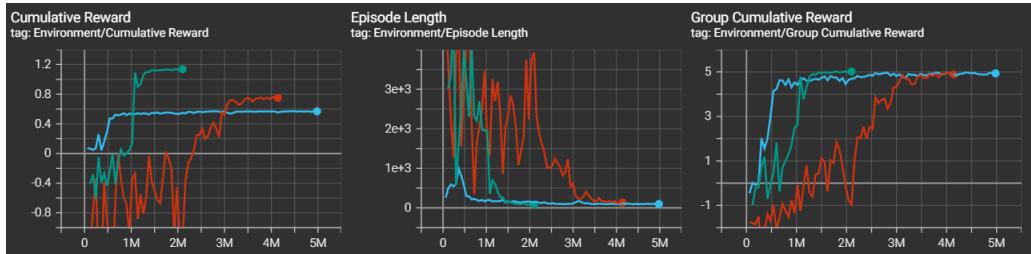


Figura 5.27: Gráficas del entrenamiento de los entornos multiagentes 2: "Push".

Name	Smoothed	Value	Step	Time	Relative
FinalMultiPushTest_2\Push	97.32	97.32	4.98M	Wed Aug 24, 16:07:04	7h 15m 16s
FinalMultiPushTest_6\Push	137.3	137.3	4.14M	Thu Aug 25, 02:27:32	5h 9m 39s
FinalMultiPushTest_7\Push	83.71	83.71	2.1M	Thu Aug 25, 10:02:08	2h 39m 22s

Figura 5.28: Resultado del entrenamiento de los entornos multiagentes 2: "Push".

Podemos observar en la figura 5.28 como el entorno con dos agentes consigue completar los episodios con una media de 97 pasos, el entorno con 3 agentes con una media de 137, y el entorno con 4 agentes con una media de 93. Todo esto en parte gracias a las funciones que se añadieron para motivar a los agentes a agruparse, dado que cuando no se incluyeron, únicamente uno o dos agentes empujaban la esfera en lugar de todos los agentes a la vez.

Además, comentar que una vez conseguido que los agentes alcancen el comportamiento deseado, se podría continuar entrenando los agentes y modificando las recompensas para conseguir un comportamiento incluso más veloz, si así se desea. Incluso se podría repetir el entrenamiento sin cambiar ningún parámetro, ya que como se comentó anteriormente, la inicialización aleatoria del entorno puede afectar en gran medida a los resultados.

Comparando estos resultados con los del entorno monoagente (Fig. 5.17), el cual completaba los episodios con una media superior a los 200 pasos, podemos observar como efectivamente al aumentar el número de agentes se consigue reducir el tiempo que se necesita para completar los episodios, siendo el entorno con 4 agentes el que consigue completarlo con la mayor velocidad posible.

Resulta muy interesante también comentar en este apartado la funcionalidad de ML-Agents, la cual nos permite inicializar un entrenamiento a partir del conocimiento obtenido de un entrenamiento pasado, siempre y cuando los parámetros entre los entrenamientos sean iguales o similares. Gracias a esta funcionalidad es posible iniciar cualquiera de los entrenamientos comentados a partir de

una de las versiones ya entrenadas anteriormente con menos agentes. Esto nos permite reutilizar todo ese conocimiento previamente obtenido, y acelerar considerablemente los entrenamientos si las diferencias entre los dos entrenamientos no son muy diferentes.

Para activar esta funcionalidad únicamente es necesario activar un flag extra en el comando que se introduce en la terminal de conda para iniciar el entrenamiento, como bien se comenta en el apartado de entrenamiento del Apéndice A. Dicho flag es '`--initialize-from=<ID_of_previous_training>`'.

A continuación se puede ver su funcionamiento, entrenando para ello el entorno con 4 agentes a partir del conocimiento del entrenamiento previo con 3 agentes, y comparándolo con el entrenamiento desde 0 mostrado anteriormente (Fig. 5.29, Fig. 5.30).

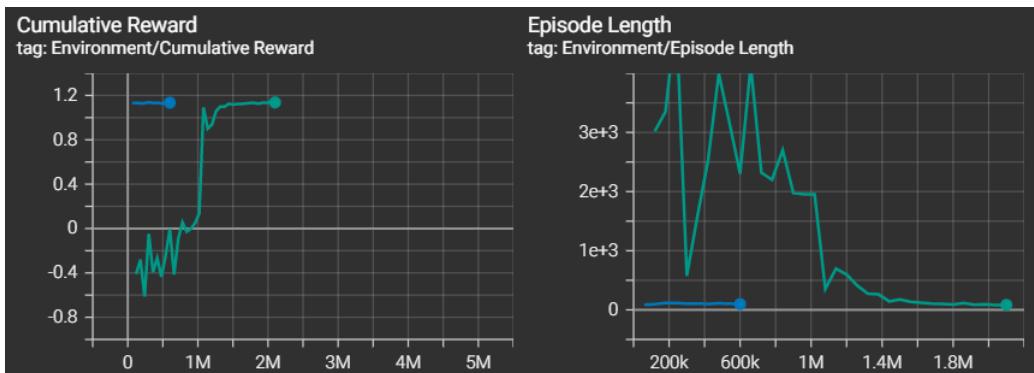


Figura 5.29: Comparativa del entrenamiento de los entornos multiagente 2: Push, iniciando el entrenamiento desde 0 y partiendo de un entrenamiento anterior.

	Name	Smoothed	Value	Step	Time	Relative
●	FinalMultiPushTest_7\Push	83.71	83.71	2.1M	Wed Aug 24, 16:07:04	2h 39m 22s
●	FinalMultiPushTest_5\Push	97.45	97.45	600k	Wed Aug 24, 20:35:03	54m 12s

Figura 5.30: Resultado del entrenamiento de los entornos multiagente 2: Push, iniciando el entrenamiento desde 0 y partiendo de un entrenamiento anterior..

Se puede comprobar como efectivamente al empezar el entrenamiento a partir de un conocimiento previo se consigue incrementar la velocidad del entrenamiento, así como obtener desde un principio del entrenamiento un buen nivel de recompensa y de longitud del episodio. Esta funcionalidad puede ser muy interesante cuando se ha conseguido realizar un entrenamiento fructífero sobre un entorno concreto, y tras alguna modificación sobre el agente o el escenario se quiere volver a entrenar. Gracias a esto se puede comenzar a entrenar a partir del conocimiento anterior, agilizando enormemente el proceso y ahorrando una gran cantidad tanto de tiempo como de recursos.

5.2.2.3. Entorno 3: Bridge multiagente

Finalmente, aquí tenemos el último de los entornos (Fig. 5.31), sobre el cual se han realizado la mayor cantidad de modificaciones y pruebas para intentar conseguir hacerlo funcionar. A continuación comentaremos por encima algunas de las diferentes pruebas que se realizaron, así como mostraremos la configuración final y las gráficas con el resultado.

El objetivo de este entorno como bien ya hemos comentado anteriormente, es el de que los dos agentes consigan cruzar el puente y llegar juntos a la meta. Para ello y de manera similar al entorno de 'Bridge' monoagente, es necesario presionar un botón para activar el puente, lo que permitirá que los agentes crucen de una zona del mapa a la otra. Así pues, y para incentivar la cooperación de los agentes, el puente únicamente se activa cuando alguno de los botones está presionado, y en caso de no ser así, el puente vuelve a su posición de reposo. De esta manera se busca que los agentes aprendan a cooperar entre sí, y que finalmente sean los dos capaces de cruzar el puente.

A diferencia del entorno monoagente 'Bridge' y debido a una mala funcionalidad de la función utilizada para mover el puente, fue necesario cambiar el funcionamiento del mismo. Por ello, se movió la posición de este para que en lugar de que se desplazase de manera vertical, se desplace en horizontal.

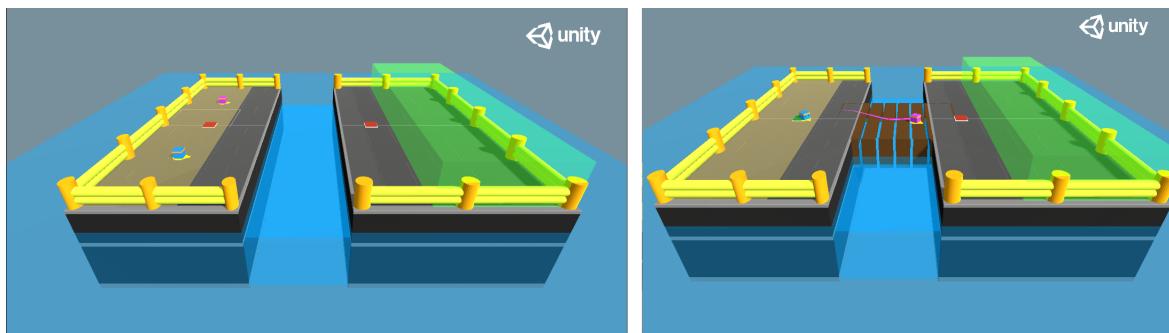


Figura 5.31: Entorno multiagente 3: Bridge.

Al igual que para la versión monoagente de este entorno, se deshabilitó la función de salto, y debido a la complejidad añadida del entorno al requerir que los agentes cooperen, y al tiempo gastado en encontrar un modelo funcional, no se llegó a entrenar el entorno con curriculum learning.

Comentar que después de todas las pruebas realizadas y debido a la complejidad del entorno, se consiguió obtener una configuración lo suficientemente buena como para que los agentes consigan aprender a cooperar y a cruzar el puente de manera estable, e incluso en numeradas ocasiones llegar a la meta juntos, pero hubo que simplificar en varias ocasiones el entorno para ello.

Para conseguir este resultado se realizaron una gran cantidad de pruebas sobre las recompensas e hiperparámetros del entrenador MA-POCA, pero finalmente el entrenamiento más prometedor surgió tras probar a realizar el entrenamiento del entorno con el entrenador PPO, realizando un ajuste al tamaño del batch así como al del buffer respecto al utilizado para resolver los entornos monoagentes. Fue necesario también modificar las recompensas inicialmente pensadas para el entorno grupal, de manera que se distribuyeran directamente a cada uno de los agentes por separado.

Se probó también a aumentar el decision requester de 5 a 10, de manera que los agentes tomen decisiones a intervalos más largos, y aunque se pudo comprobar como efectivamente ayudó en parte a que los agentes no se cayeran tanto al principio del entrenamiento, no llegó a influir demasiado. Por esto mismo finalmente se decidió volver al valor inicial de manera que el comportamiento del agente sea más similar al del resto de entornos.

Respecto a las **observaciones** del agente, se utilizaron las mismas que se comentan en la versión de Bridge monoagente, añadiendo para cada agente tanto la posición del nuevo botón, como la posición del otro agente, y quitando las observaciones de la velocidad del propio agente para reducir el número de observaciones totales y así agilizar el entrenamiento.

Respecto a las **recompensas** de este comportamiento, este fue el entorno con mayor complejidad para establecer una función de recompensas adecuada, dado que sin una configuración adecuada para motivar la cooperación no se conseguía que los agentes llegaran siquiera a cruzar el puente. Por consiguiente, el apartado más importante de este entorno recae en las recompensas grupales, las cuales se centran principalmente en hitos obtenidos cuando los agentes cooperan correctamente y se delegan las funciones necesarias para completar la tarea global. Dichas recompensas fue necesario adaptar ya que finalmente cambiamos el entrenador a PPO .

Se probaron una gran cantidad de funciones de recompensa diferentes, desde simplemente recompensar al agente al pulsar los botones y llegar a la meta, hasta a funciones más complejas que recompensaban a los agentes cuando se delegaban tareas y se iban acercando hacia la meta, o cuando un agente pulsa el botón mientras el otro cruza, etc.

A continuación se muestran las **recompensas finales** del entrenamiento en el que mejor se comportaron los agentes:

■ **Recompensas individuales:**

- **Recompensas positivas:**

- + Recompensa al cruzar un checkpoint correcto (+0.5 / NumCheckpoints).
- + Recompensa al pulsar el 1er botón si ninguno de los dos agentes ha cruzado (+0.25 / MaxEnvironmentSteps).
- + Recompensa si el otro agente comienza a cruzar el puente y estoy pulsado el 1er botón.
(+0.5 / MaxEnvironmentSteps).

+ Recompensa si el otro agente comienza a cruzar el puente y estoy pulsado el 2o botón.

(+1.0 / MaxEnvironmentSteps).

- **Recompensas negativas:**

+ Recompensa al cruzar un checkpoint incorrecto (-0.25 / NumCheckpoints).

+ Recompensa si el otro agente ha cruzado el puente y sigo pulsado el 1er botón (-0.5 / MaxEnvironmentSteps).

+ Recompensa al pulsar el 2o botón si los dos agentes han cruzado (-0.5 / MaxEnvironmentSteps).

■ **Recompensas grupales:**

- **Recompensas positivas:**

+ Los dos agentes sobre la 2a área (+2 / MaxEnvironmentSteps)

+ Agente1 en la meta (+0.5 / MaxEnvironmentSteps al Agente2)

+ Agente2 en la meta (+0.5 / MaxEnvironmentSteps al Agente1)

+ Recompensa al completar la tarea (Los dos agentes en la meta)

(+3 a los dos agentes)

- **Recompensas negativas:**

+ Recompensa al fallar o pasarse del tiempo máximo (-1f a cada uno)

Se utilizaron también las técnicas de GAIL y BC para hacer funcionar este entorno, pero para probarlo fue necesario modificar el código de los agentes, ya que al grabar los comportamientos los dos agentes se movían con las mismas teclas y esto dificulta completar la tarea. Una vez modificado, se grabaron más de 100 episodios donde los agentes conseguían llegar a la meta juntos en más de un 90 % de los intentos, y finalmente se pudo usar GAIL + BC en el entrenamiento. Se pudo comprobar como gracias a aplicar esta funcionalidad, tras unos pocos miles de episodios uno de los agentes ya había conseguido superar el puente y pulsar el segundo botón, cosa que en otros entrenamientos había costado millones de episodios, o directamente no había llegado siquiera a suceder.

Se procedió por lo tanto, y tras ver que el agente conseguía por fin llegar a la meta de manera más constante, a modificar los diferentes valores de las recompensas, así como los valores de GAIL y BC. Para ello, a lo largo del entrenamiento y conforme el agente conseguía obtener mejores recompensas y llegar en más ocasiones a la meta, se procedió a paulatinamente ir reduciendo la cantidad de las recompensas obtenidas a través de GAIL y BC, de manera que el agente en lugar de preocuparse por copiar los movimientos pregrabados pudiera aprender a comportarse incluso mejor que ellos. Además, respecto a las recompensas, se aumentó en un momento del entrenamiento tanto la recompensa obtenida al completar el objetivo, así como la de los checkpoints, de manera

que se motivó más al agente a cruzar y llegar a la meta juntos, en lugar de por ejemplo quedarse pulsando botones o dando vueltas simplemente.

A continuación se pueden ver los resultados de las diferentes pruebas previas realizadas para intentar hacer funcionar este entorno (Fig. 5.32, Fig. 5.33), los cuales fueron un total de 19 entrenamientos, que se traducen en aproximadamente 127 horas de entrenamiento. Se puede ver cómo a excepción de un entrenamiento, el resto no consiguió obtener una recompensa individual mayor a 1, así como una recompensa grupal mayor a 0, y por lo tanto, casi nunca llegaban a la meta o cruzaban siquiera el puente.

Name	re/distance_offset	Smoothed	Value	Step	Time	Relative
FinalCollabBridge_1\BridgeCooperative	FinalCollabBridge_1\BridgeCooperative	-1.675	-1.675	820k	Tue Aug 30, 16:09:30	46m 18s
FinalCollabBridge_2\BridgeCooperative	FinalCollabBridge_2\BridgeCooperative	-1.808	-1.808	350k	Tue Aug 30, 16:53:31	40m 25s
FinalCollabBridge_3\BridgeCooperative	FinalCollabBridge_3\BridgeCooperative	-1.543	-1.543	260k	Tue Aug 30, 17:35:27	27m 37s
FinalCollabBridge_4\BridgeCooperative	FinalCollabBridge_4\BridgeCooperative	-0.6615	-0.6615	560k	Tue Aug 30, 20:17:36	1h 26m 37s
FinalCollabBridge_5\BridgeCooperative	FinalCollabBridge_5\BridgeCooperative	-0.9378	-0.9378	410k	Tue Aug 30, 22:02:15	1h 12m 22s
FinalCollabBridge_6\BridgeCooperative	FinalCollabBridge_6\BridgeCooperative	-0.7044	-0.7044	5.67M	Thu Sep 1, 12:21:02	1d 13h 42m 22s
FinalMultiBridgeGAIL_ppo\BridgeCooperative_1	FinalMultiBridgeGAIL_ppo\BridgeCooperative_1	-0.8311	-0.8311	240k	Mon Aug 29, 13:09:50	15m 54s
FinalMultiBridgeGAIL_ppo_2\BridgeCooperative_1	FinalMultiBridgeGAIL_ppo_2\BridgeCooperative_1	-1.292	-1.292	2.4M	Mon Aug 29, 17:05:20	2h 23m 54s
FinalMultiBridgeTest_4\BridgeCooperative	FinalMultiBridgeTest_4\BridgeCooperative	-0.1041	-0.1041	660k	Mon Aug 22, 11:53:05	43m 36s
FinalMultiBridgeTest_5\BridgeCooperative	FinalMultiBridgeTest_5\BridgeCooperative	-0.1475	-0.1475	1.56M	Mon Aug 22, 17:34:48	4h 26m 0s
FinalMultiBridgeTest_7\BridgeCooperative	FinalMultiBridgeTest_7\BridgeCooperative	-0.5318	-0.5318	2.58M	Mon Aug 22, 20:48:19	2h 24m 59s
FinalMultiBridgeTest_8\BridgeCooperative	FinalMultiBridgeTest_8\BridgeCooperative	0.9812	0.9812	16.56M	Tue Aug 23, 17:57:38	20h 17m 57s
FinalMultiBridge_1\BridgeCooperative	FinalMultiBridge_1\BridgeCooperative	0	0	6.78M	Wed Aug 17, 01:43:08	13h 3m 42s
FinalMultiBridge_2\BridgeCooperative	FinalMultiBridge_2\BridgeCooperative	-0.5	-0.5	7.44M	Wed Aug 17, 16:02:54	13h 32m 26s
FinalMultiBridge_3\BridgeCooperative	FinalMultiBridge_3\BridgeCooperative	0.06211	0.06211	5.22M	Thu Aug 18, 21:31:13	11h 6m 27s
FinalMultiBridge_4\BridgeCooperative	FinalMultiBridge_4\BridgeCooperative	-0.0439	-0.0439	8.4M	Fri Aug 19, 11:50:12	12h 53m 19s
FinalMultiBridge_GAIL_ppo_1\BridgeCooperative_1	FinalMultiBridge_GAIL_ppo_1\BridgeCooperative_1	-1.936	-1.936	420k	Mon Aug 29, 17:58:34	23m 13s
FinalMultiBridge_GAIL_ppo_2\BridgeCooperative_1	FinalMultiBridge_GAIL_ppo_2\BridgeCooperative_1	-1.782	-1.782	420k	Mon Aug 29, 18:43:54	36m 10s
FinalMultiBridge_GAIL_ppo_3\BridgeCooperative_1	FinalMultiBridge_GAIL_ppo_3\BridgeCooperative_1	-0.3406	-0.3406	1.68M	Mon Aug 29, 22:59:49	2h 44m 44s

Figura 5.32: Resultado de las pruebas sobre el entorno multiagente 3: Bridge.

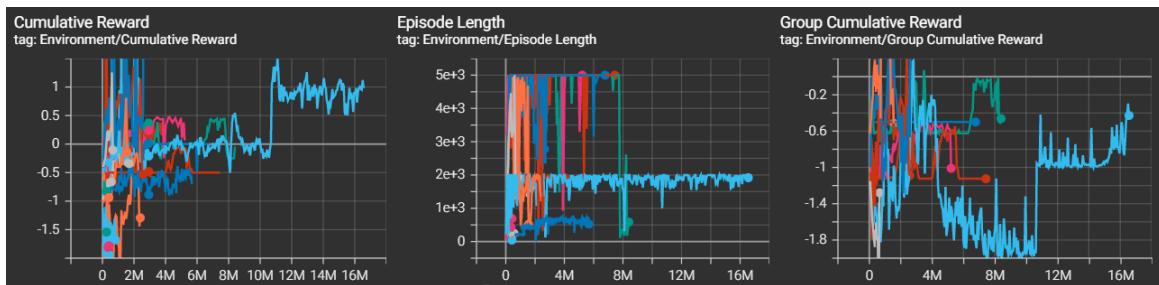


Figura 5.33: Resultado de las pruebas sobre el entorno multiagente 3: Bridge.

Se puede observar por lo tanto como para obtener un buen resultado en este entorno, ha sido necesario ejecutar más pruebas que en los entornos anteriores, así como configurar una función de recompensa más compleja, tanto por la complejidad añadida de la necesidad de cooperación entre los agentes para completar la tarea, así como por la complejidad extra que surge del propio diseño del entorno 3D.

Finalmente, se muestran a continuación las gráficas del mejor de todos los entrenamientos, el cual se consiguió tras entrenar el entorno únicamente durante 2.4M de pasos (13h), y cuya configuración de las observaciones así como de la función de recompensas se han comentado anteriormente (Fig. 5.34). Comentar que si continuamos entrenando durante unos cuantos millones más de pasos, y seguimos modificando la función de recompensa y las observaciones hasta encontrar una mejor configuración, existe la posibilidad de que finalmente aprendan a realizar la tarea en todos los intentos. Además también se podría continuar entrenando para probar a activar las posiciones del puente y los botones aleatorias, lo cual al final no se llegó a implementar en este entorno.

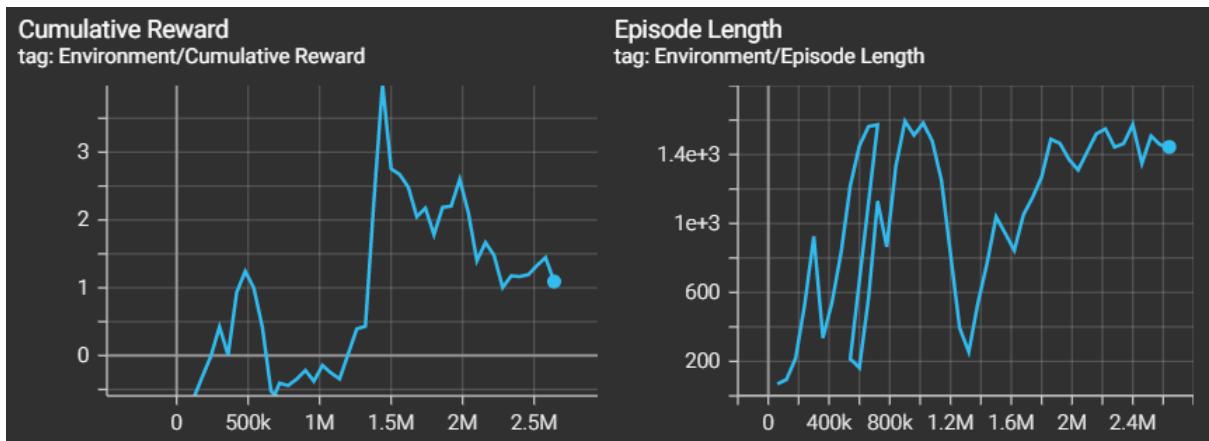


Figura 5.34: Resultado de las pruebas finales sobre el entorno multiagente 3: Bridge.

	Name	Smoothed	Value	Step	Time	Relative
FinalCollabBridge_Gail1\BridgeCooperative_2	2.286	2.286	1.38M	Fri Sep 2, 01:53:17	6h 14m 18s	

a)

	Name	Smoothed	Value	Step	Time	Relative
FinalCollabBridge_Gail1\BridgeCooperative_2	537	537	1.38M	Fri Sep 2, 01:53:17	6h 14m 18s	

b)

Figura 5.35: Mejor recompensa (a) y tiempo medio por episodio (b) en el entorno multiagente 3: Bridge.

Así pues y a diferencia de otros entornos donde nos quedamos con el último modelo guardado, para implementar el modelo final en este entorno seleccionamos el checkpoint del modelo más cercano al punto del entrenamiento donde se consigue una mayor relación recompensa-tiempo, lo cual aproximadamente sobre los 1.38M de pasos de comenzar el entrenamiento (Fig. 5.35).

Como conclusión, se ha podido comprobar gracias a este entrenamiento la importancia de configurar correctamente las recompensas y las observaciones de un agente, de manera que se pueda guiar correctamente a los agentes hacia la cooperación, y por lo tanto a alcanzar juntos la meta. Es muy importante también saber cómo establecer los valores de las recompensas para que los agentes no se 'distraigan' durante el aprendizaje, como por ejemplo pulsando únicamente un botón en lugar de cruzar el puente, o realizando otras tareas que lo mantengan lejos de completar la tarea final.

Resultados y Conclusiones

6

En este capítulo vamos a resumir todos los resultados obtenidos durante el desarrollo del proyecto, los cuales ya se han ido comentando a lo largo del apartado anterior. En concreto, vamos a hablar de los diferentes entrenamientos realizados en los escenarios 3D, del comportamiento del/-los agente/s en los diferentes entornos, y de los problemas que hemos encontrado y las soluciones aplicadas.

Así pues, podemos decir antes que nada y en vista de los resultados obtenidos en el apartado anterior, se ha logrado obtener un agente capaz de aprender de manera relativamente sencilla a realizar una serie de diferentes tareas, tanto en entornos monoagentes como multiagentes, e inclusive llegando a cooperar con otros agentes para ello.

Respecto a los entrenamiento monoagentes, recordar que fueron realizados con el objetivo principal de comprobar tanto si los algoritmos de RL implementados funcionan correctamente, como de desarrollar un agente capaz de aprender a realizar una serie de tareas básicas, aún entrenando desde un dispositivo con bajas especificaciones. Así pues, pudimos comprobar que los resultados obtenidos en estos entrenamientos fueron los esperados, y los agentes consiguieron aprender de manera correcta, obteniendo resultados satisfactorios en todos los escenarios.

Por otro lado, los entrenamientos multiagentes fueron realizados con el objetivo de analizar el comportamiento de los agentes en escenarios en los que su objetivo era cooperar para resolver el problema planteado, o mejorar el rendimiento de la tarea monoagente mediante el uso de agentes adicionales. En concreto, los entornos que utilizamos para comparar la recompensa obtenida al utilizar más de un agente, nos permitieron comprobar que los agentes entrenados, aunque no siempre obtienen resultados ampliamente mejores a cuando se tenía un solo agente, sí que consiguen completar la tarea de manera cooperativa. También se pudo comprobar con el entorno de Bridge cooperativo, donde los agentes deben cooperar de manera forzosa, que aunque todavía el proceso de entrenamiento para que los agentes aprendan realizar tareas conjuntas es complejo, y requiere de muchas pruebas, comienza a ser cada vez más viable y factible de realizar.

Gracias a todos estos entrenamientos y pruebas realizadas descubrimos la gran importancia que recae sobre el diseño del entorno, sobre las observaciones del agente y especialmente, sobre la función de recompensa. El diseño de dicha función de recompensa es muy importante para permitir que el agente aprenda adecuadamente. Además, en los entornos cooperativos es importante definirla de manera que se premie la cooperación entre los agentes o la delegación de subtareas, de modo que se guíe a los agentes para conseguir completar la tarea global.

Así pues y analizando los resultados obtenidos, podemos decir que aplicando las diferentes funciones de recompensa para cada uno de los entornos comentados anteriormente en el apartado de experimentación, conseguimos que los agentes entrenados fueran capaces de resolver los problemas planteados de manera satisfactoria, a excepción del último entorno, donde a pesar de todas las pruebas realizadas a lo largo de los entrenamientos, los agentes no llegaron a desarrollar una estrategia adecuada para completar de manera constante la tarea deseada.

A lo largo del desarrollo del proyecto, se han podido comprobar algunos de los problemas que el aprendizaje por refuerzo todavía presenta y que es importante tener en cuenta, siendo los más importantes los siguientes:

- Los problemas pueden ser muy difíciles de escalar, especialmente en entornos cada vez más grandes y complejos.
- El problema del aprendizaje por refuerzo puede requerir un gran número de pasos de simulación para aprender una política óptima, lo cual puede ser muy costoso en términos de tiempo y recursos.
- Existe una alta complejidad para determinar las recompensas, de manera que el agente sea capaz de aprender de manera adecuada a lo largo del entrenamiento. Siendo alguno de dichos problemas los siguientes:
 - Sparse Rewards: Se refiere a la dificultad de aprendizaje que surge cuando las recompensas son escasas o poco frecuentes. Esto puede hacer que el aprendizaje sea muy lento y difícil.
 - Curiosity Rewards: Se refiere a la capacidad de utilizar recompensas para explorar el entorno y aprender de él. Esto puede ser muy útil en entornos donde no se dispone de una fuente externa de recompensas.
- Los entrenamientos pueden ser propensos a resultados no deseados, como por ejemplo, en el caso de no asignar de manera adecuada las recompensas negativas, o de asignar demasiadas.

A modo de conclusión, podemos decir que hemos logrado los objetivos propuestos, habiendo obtenido resultados satisfactorios en la mayoría de los entornos realizados, además de que gracias a las diferentes pruebas realizadas, se ha conseguido profundizar en las técnicas de aprendizaje por refuerzo y obtener un mayor conocimiento acerca de los diferentes factores a tener en cuenta para la correcta implementación de este tipo de técnicas. Así pues y para finalizar, aunque aún queda mucho trabajo por hacer para mejorar el rendimiento de los agentes en los diferentes entornos, podemos decir que hemos llegado a un punto en el que se ha conseguido entrenar a uno o varios agentes, los cuales mediante diversas técnicas de aprendizaje por refuerzo son capaces de resolver una serie de problemas planteados.

Limitaciones y Perspectivas de Futuro

7

Por último, vamos a exponer una serie de trabajos que se podrían llevar a cabo en un futuro a modo de ampliación, y finalmente, hablaremos de la posibilidad de aplicar este tipo de técnicas de aprendizaje por refuerzo en entornos reales, y del posible uso de las mismas en la robótica.

A continuación, enumeramos una serie de trabajos futuros que podrían realizarse a modo de ampliación:

- Profundizar en las arquitecturas de aprendizaje por refuerzo utilizadas y en las técnicas de apoyo para facilitar el aprendizaje.
- Profundizar en el diseño de las funciones de recompensa así como en las observaciones que tiene el agente para mejorar el comportamiento de los agentes.
- Diseñar entornos monoagentes y multiagentes más complejos y que requieran de mayor interacción tanto entre el agente y el entorno, como entre los agentes.
- Aplicar las técnicas de aprendizaje por refuerzo en entornos reales mediante robots móviles y la utilización de la plataforma de robótica ROS para controlarlos ([Robotics \(a\)](#)).

Como bien comentamos anteriormente, se estudiaría también la posibilidad de implementar este tipo de algoritmos en un sistema robótico real. En consecuencia y respecto al uso del aprendizaje por refuerzo en aplicaciones de robótica y para el control de robots reales, podemos decir que el uso de este tipo de arquitecturas no es una tarea sencilla, principalmente debido a la necesidad de las aplicaciones de este tipo de operar en tiempo real, además de por la cantidad de datos que un sistema de estas características requiere y la necesidad de un resultado fiable y constante. Por ello, aunque ya existen varias arquitecturas de aprendizaje por refuerzo capaces de controlar robots de manera satisfactoria (p. ej. [OpenAI, Julian Ibarz \(2017\)](#)), aún hay mucho trabajo por hacer para mejorar el rendimiento de estos sistemas.

Por lo tanto, se ha encontrado un repositorio el cual mediante la plataforma robótica ROS ([Robotics \(a\)](#), [Robotics \(d\)](#), [Robotics \(b\)](#)) permite implementar una serie de algoritmos de RL tanto en entornos robóticos reales como simulados. Dicho comportamiento se puede lograr gracias a una librería especial de ROS desarrollada por el equipo de OpenAI ([Robotics \(c\), team](#)). Con esto podríamos ser capaces de diseñar entornos similares a los comentados en este trabajo, así como de poder utilizar finalmente un robot Turtlebot (u otro similar) para la realización de los mismos,

siendo posible incluso el añadir un manipulador robótico al robot móvil, habilitando así la posibilidad de manipular y transportar objetos de la escena.

Son muchos los campos de la robótica que se podrían beneficiar de este tipo de aprendizaje, siendo unos de los más notorios quizás el de los robots móviles y los robots colaborativos. Algunas de las posibles aplicaciones del RL en la robótica son las siguientes:

- Sistema de robots móviles los cuales tuvieran la capacidad de aprender de la información obtenida durante las misiones, de forma que cada vez se pudiera mover de manera más eficiente, o de forma que pudiera llegar a su objetivo en un tiempo mínimo.
- Aprendizaje de optimización la batería. En este caso, el objetivo sería que el robot aprendiera a minimizar el uso de batería durante sus misiones, de forma que el robot pudiera moverse por el entorno durante un periodo de tiempo más largo.
- Control de un robot móvil en un entorno desconocido. En este caso, el objetivo sería que el robot mediante la creación de un mapa o únicamente mediante información visual y redes LSTM aprendiera a moverse por un entorno con alta incertidumbre, de forma que pudiera encontrar una salida en un tiempo mínimo, o simplemente recabar información del entorno.
- Implementación de sistemas inteligentes multiagentes, donde los agentes aprenden a realizar una serie de tareas sencillas mediante la interacción directa y la cooperación.
- Sistemas con agentes con desarrollo sobrehumano para problemas donde se conoce el entorno (p. ej. [Alp \(b\)](#))
- Algunas otras posibles ideas son:
 - Aprendizaje de acciones a partir de vídeos pregrabados de objetos, personas o robots en movimiento.
 - Aprendizaje de las interacciones hombre-robot para robots colaborativos.
 - Aprendizaje de la manipulación de objetos mediante brazos robóticos.
 - Aprendizaje de la optimización del rendimiento de los brazos robóticos (actuadores y motores).

Se puede ver por lo tanto como el uso de algoritmos de RL en la robótica tiene mucho potencial, y se podría aplicar en muchos campos de la robótica, ya que permite a los robots aprender y mejorar su funcionalidad con la experiencia, y mediante la interacción directa con el entorno, entre otras cosas.

Respecto al uso del aprendizaje por refuerzo en sistemas inteligentes y para finalizar, podemos decir que existen muchas aplicaciones posibles para este tipo de algoritmos y en multitud de entornos diferentes. El uso de este tipo de algoritmos en los entornos inteligentes podría permitir el desarrollo de una serie de aplicaciones muy interesantes, las cuales en un futuro puede que comencemos a ver con mayor frecuencia.

CAPÍTULO 7. LIMITACIONES Y PERSPECTIVAS DE FUTURO

En conclusión, el aprendizaje por refuerzo es un campo de investigación muy activo en la actualidad y se espera que en un futuro cercano comience a tener una mayor presencia en el desarrollo de sistemas inteligentes, así como en el campo de la robótica.

Apéndize

A

A.1. Repositorio Github

A continuación se encuentra un link que lleva directamente a un repositorio de Github, en el cual se pueden encontrar los diversos códigos de cada uno de los agentes, así como los ficheros de configuración utilizados y los escenarios de Unity, y una carpeta donde se pueden encontrar ejecutables para cada uno de los entornos por si se desean probar (tanto la versión con el modelo de RL implementado, como una para controlar al agente directamente):

[Repositorio Github TFM_CarratalaRizzo-Valderico](#) .

En este repositorio podemos encontrar tres carpetas, cuyo contenido es el siguiente:

- **config:** en esta carpeta se encuentran todos los ficheros de configuración YAML utilizados para el desarrollo de este proyecto.
- **Examples:** en esta carpeta se encuentran los diferentes códigos y escenas de Unity utilizadas para cada uno de los entornos, así como los elementos necesarios para su funcionamiento.
 - Final Environments: aquí se encuentran cada uno de los entornos finales descritos en el apartado de experimentación.
 - Previous Environments: aquí se encuentran cada uno de los entornos de prueba realizados previamente a los entornos finales.
- **Enviroments:** en esta carpeta se encuentran los diferentes ejecutables de los entornos, dentro de cada uno encontramos tres opciones:
 - Camera_Agent: Entorno con el modelo de RL implementado y la cámara general habilitada.
 - Camera_Normal: Entorno con el modelo de RL implementado y la cámara del agente habilitada.
 - Heuristic: Entorno en modo de control manual (sin el modelo de RL implementado) y la cámara del agente habilitada.

Si se desean probar los escenarios, únicamente es necesario acceder a la carpeta de Environments y ejecutar el archivo 'UnityEnvironment.exe' del entorno deseado.

Si por otro lado se quisieran modificar o ampliar los escenarios, es preciso realizar previamente la configuración de Unity y de ML-Agents comentada en este trabajo en el Apéndice A, y posteriormente copiar el contenido de la carpeta 'Examples' donde se encuentran el resto de ejemplos del repositorio de ML-Agents. Una vez hecho esto, únicamente habrá que abrir el proyecto en Unity, seleccionar la escena de Unity del escenario que queramos probar.

A.2. Vídeo de demostración

Como se ha comentado al comienzo del apartado de experimentación, en el siguiente enlace se puede encontrar un vídeo de demostración en el cual se puede observar el comportamiento final de los agentes en cada uno de los entornos creados:

[Enlace a la carpeta de Drive que contiene el vídeo de demostración.](#)

A.3. Instalación y requisitos

A.3.1. Creación de un environment de conda

Primero que nada, procedemos a crear un entorno de conda, de manera que podamos instalar todas las librerías desde 0, y no incurrir en ningún problema por incompatibilidad de versiones entre librerías o programas. Para ello, instalaremos Python 3.7.13 y posteriormente descargamos la versión de Unity 2021.3.3f1.

Algoritmo 1: Código *Creacion de un enviroment de conda*

1. conda create -n TFM python=3.7.13 tensorflow=1.13.1 keras=2.2.4 h5py pillow
 2. conda activate TFM
 3. pip install keras-rl==0.4.2
 4. pip3 install torch =1.7.1 -f https://download.pytorch.org/whl/torch_stable.html
 5. python -m pip install mlagents==0.28.0
-

A.3.2. Instalación de Unity y ml-agents

Una vez creado el entorno de conda y instalado las librerías básicas necesarias, procederemos a instalar Unity y los paquetes necesarios. Así pues, primero descargamos e instalamos la versión de Unity 2021.3.3f1 accediendo desde la página web oficial.

Tras instalar Unity y Python correctamente, procedemos a instalar los paquetes extra necesarios para el correcto funcionamiento del proyecto.

Para ello, lo primero que haremos será clonar el repositorio de ml-agents de github, de manera que se puedan probar los entornos de ejemplo desarrollados por los creadores de este proyecto, tanto para experimentar con ellos y aprender más acerca de su funcionamiento., como para modificar o ampliar el kit de herramientas de ML-Agents para mis el desarrollo de este trabajo.

Algoritmo 2: Código *Clonación del repositorio de ml-agents.)*

1. cd E:\UA\MASTER\TFM
 2. git clone <https://github.com/Unity-Technologies/ml-agents.git>
-

Una vez hecho esto, procedemos a crear un proyecto de Unity, y dentro de él a instalar el paquete de Unity "com.unity.ml-agents": El paquete de Unity ML-Agents C# SDK "com.unity.ml-agents" se puede instalar directamente desde el panel de registro del Administrador de paquetes. Hay que asegurarse de habilitar la opción "Preview Packages" en el menú desplegable "Advanced" para encontrar la última versión de vista previa del paquete.

Una vez hecho esto, procedemos a hacer la instalación local para desarrolladores:

Se puede agregar el paquete com.unity.ml-agents de manera local proyecto de Unity de la siguiente manera:

1. Navegar al menú Window ->Package Manager.
2. En la ventana del administrador de paquetes, hacer clic en el botón -en la parte superior izquierda de la lista de).
3. Seleccionar Add package from disk... (Fig.A.1)
4. Navegar a la carpeta com.unity.ml-agents.
5. Seleccionar el archivo package.json. (Fig.A.2)

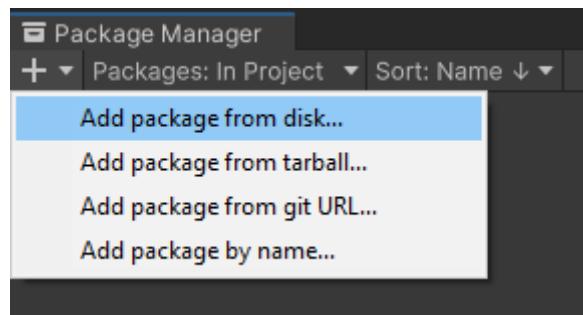


Figura A.1: Ventana del administrador de paquetes.

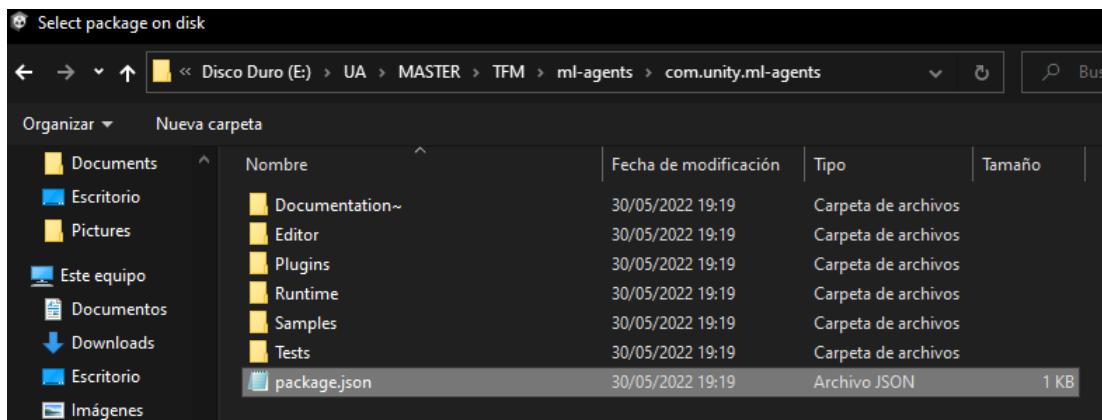


Figura A.2: Selección el archivo package.json.

Después de todo esto, estamos ya listos para comenzar a crear los diferentes entornos, así como a los agentes, pero antes que eso procederemos a explicar como crear el fichero de configuración necesario para llevar a cabo el entrenamiento.

A.3.3. Fichero de configuración YAML en el que se incluyen todas las configuraciones posibles usando un entrenador PPO

```
behaviors:  
  <BehaviorName>:  
    trainer_type: ppo  
  
  hyperparameters:  
    # Hyperparameters common to PPO and SAC  
    batch_size: 1024  
    buffer_size: 10240  
    learning_rate: 3.0e-4  
    learning_rate_schedule: linear  
  
    # PPO-specific hyperparameters  
    beta: 5.0e-3  
    beta_schedule: constant  
    epsilon: 0.2  
    epsilon_schedule: linear  
    lambd: 0.95  
    num_epoch: 3  
  
  # Configuration of the neural network (common to PPO/SAC)  
  network_settings:  
    vis_encode_type: simple  
    normalize: false  
    hidden_units: 128  
    num_layers: 2  
    # memory  
    memory:  
      sequence_length: 64  
      memory_size: 256  
  
    # Trainer configurations common to all trainers  
    max_steps: 5.0e5  
    time_horizon: 64  
    summary_freq: 10000
```

```
keep_checkpoints: 5
threaded: false
init_path: null

# behavior cloning
behavioral_cloning:
    demo_path: Project/Assets/ML-Agents/TFM/DemoRecordings.demo
    strength: 0.5
    steps: 150000
    batch_size: 512
    num_epoch: 3

reward_signals:
    # environment reward (default)
    extrinsic:
        strength: 1.0
        gamma: 0.99

    # curiosity module
    curiosity:
        strength: 0.02
        gamma: 0.99
        encoding_size: 256
        learning_rate: 3.0e-4

# GAIL
gail:
    strength: 0.01
    gamma: 0.99
    demo_path: Project/Assets/ML-Agents/TFM/DemoRecordings.demo
    learning_rate: 3.0e-4
    use_actions: false
    use_vail: false

# self-play
self_play:
    window: 10
    play_against_latest_model_ratio: 0.5
    save_steps: 50000
    swap_steps: 2000
    team_change: 100000
```

Nótese que la configuración de los parámetros del curriculum learning y de aleatorización del entorno no forman parte de este fichero YAML, sino que se encuentra en su propia sección llamada environment_parameters.

A.4. Ejecución entrenamiento

A continuación se adjunta el fichero de configuración utilizado para el entrenamiento del entorno de ejemplo explicado en la metodología, así como el resto de pasos que se llevaron a cabo para su funcionamiento.

A.4.1. Fichero de configuración YAML en el entorno de ejemplo mencionado en la metodología

```
behaviors:  
  MoveToGoal:  
    trainer_type: ppo  
    hyperparameters:  
      batch_size: 10  
      buffer_size: 100  
      learning_rate: 3.0e-4  
      beta: 5.0e-4  
      epsilon: 0.2  
      lambd: 0.99  
      num_epoch: 3  
      learning_rate_schedule: linear  
      beta_schedule: constant  
      epsilon_schedule: linear  
    network_settings:  
      normalize: false  
      hidden_units: 128  
      num_layers: 2  
    reward_signals:  
      extrinsic:  
        gamma: 0.99  
        strength: 1.0  
  max_steps: 500000  
  time_horizon: 64  
  summary_freq: 10000
```

Una vez está todo configurado, se puede comenzar a entrenar los agentes. El comando base para realizar dicho entrenamiento es el siguiente:

Algoritmo 3: Código Comando base para entrenar un agente.)

1. `mlagents-learn <trainer-config-file>--run-id=<run-identifier>--num-areas=<num-areas>`

Donde:

- **<trainer-config-file>**: ruta del archivo YAML de la configuración del entrenador. Contiene todos los valores de los diferentes hiperparámetros.
- **<run-identifier>** : nombre único que se usa para identificar las ejecuciones de cada entrenamiento. Permite continuar un entrenamiento desde el último step ejecutado, empezar a entrenar a partir de otro entrenamiento previo, y más.
- **<num-areas>** : número de instancias simultáneas de Unity a ejecutar durante el entrenamiento. A más instancias, menos pasos se necesitarán para que el entrenamiento converja, pero mayor carga computacional requerirá.

Para ejecutar el entrenamiento es necesario realizar los siguientes pasos:

1. Abrir una terminal de conda y activar el entorno.
2. Navegar a la carpeta donde se clonó el repositorio de ml-agents.
3. Ejecutar el comando `mlagents-learn` mencionado anteriormente con las banderas necesarias.
4. Cuando se muestre por pantalla el mensaje "Start training by pressing the Play button in the Unity Editor", se puede presionar el botón Play en Unity para comenzar a entrenar en el Editor. (Fig.[A.3](#))



```
(venv) (base) E:\UAVMASTER\TFP\ml-agents>mlagents-learn config\ppo\MoveToGoal.yaml --run-id=Test_Environment --num-areas=19

Version information:
  TensorFlow: 2.5.0-dev,
  ml-agents-envs: 0.29.0-dev@,
  Communicator: 0.29.0,
  PyTorch: 1.7.1+cu100
[INFO] Listening on port 5004. Start training by pressing the Play button in the Unity Editor.
[INFO] Connected to Unity Editor using package version 2.2.1-exp.1 and communication version 1.5.0
[INFO] Selected new brain MoveToGoalteam
[INFO] Hyperparameters for behavior name MoveToGoal:
```

Figura A.3: Ejecutar el entrenamiento en la terminal.

Independientemente de los métodos de entrenamiento, configuraciones o hiperparámetros que se proporcionen, el proceso de entrenamiento siempre generará tres artefactos, todos ellos encontrados en la carpeta **results/<run-identifier>**:

- **Summaries:** aquí se incluyen las métricas de entrenamiento que se actualizan a lo largo del proceso de entrenamiento. Son útiles para monitorizar el rendimiento del entrenamiento y pueden ayudar a informar cómo actualizar los valores de los hiperparámetros. En nuestro caso utilizaremos TensorBoard para visualizar dichas métricas.
- **Modelos:** aquí están contenidos los puntos de control del modelo que se actualizan a lo largo del entrenamiento y el archivo del modelo final (.onnx). Este archivo con el modelo final se genera tanto cuando finaliza el entrenamiento, como cuando se interrumpe, aunque se puede configurar para generarse cada cierto número de pasos, así como un número máximo de modelos a generar.
- **Timers file** (dentro de results/<run-identifier>/run_logs): aquí se incluyen el resto de métricas agregadas en el proceso de entrenamiento.

Estos artefactos se actualizan a lo largo del proceso de entrenamiento y finalizan cuando se completa o se interrumpe el entrenamiento.

Para interrumpir el entrenamiento y guardar el progreso actual, basta con presionar Ctrl+C desde la terminal de conda una vez y esperar a que se guarden los modelos.

Para reanudar una ejecución de un entrenamiento previamente interrumpida o finalizado, únicamente es necesario incluir el flag **-resume** al comando para iniciar el entrenamiento, y asegurarse de especificar correctamente el ID de la ejecución utilizada anteriormente.

Si se desea ejecutar un entrenamiento reutilizando el ID de una ejecución anterior (en este caso, sobrescribiendo los artefactos generados anteriormente), usar el indicador **-force** en lugar del **-resume**.

Como alternativa, es también posible iniciar un nuevo entrenamiento pero partiendo de un modelo ya entrenado. Es posible que se desee hacer esto, por ejemplo, si el entorno que se estaba entrenando cambió y se quiere obtener un nuevo modelo, pero el comportamiento anterior sigue siendo mejor que comenzar con un comportamiento aleatorio. Para utilizar esta funcionalidad, únicamente hay que usar el indicador **-initialize-from=<run-identifier>**, donde <run-identifier> es el ID de ejecución anterior.

A.5. Análisis del entrenamiento

Una vez terminado el entrenamiento, o durante el mismo, podemos proceder a visualizar los resultados mediante la herramienta de Tensorboard. Por ello y para observar el proceso de entrenamiento con más detalle, ejecutaremos el siguiente código desde la terminal:

Algoritmo 4: Código para visualizar los resultados mediante Tensorboard.)

```
1. tensorboard --logdir results --port 6006
```

Una vez hecho esto, entraremos desde el navegador a la dirección localhost:6006 para ver las estadísticas generadas durante el entrenamiento tal y como se muestra a continuación (Fig.A.4):

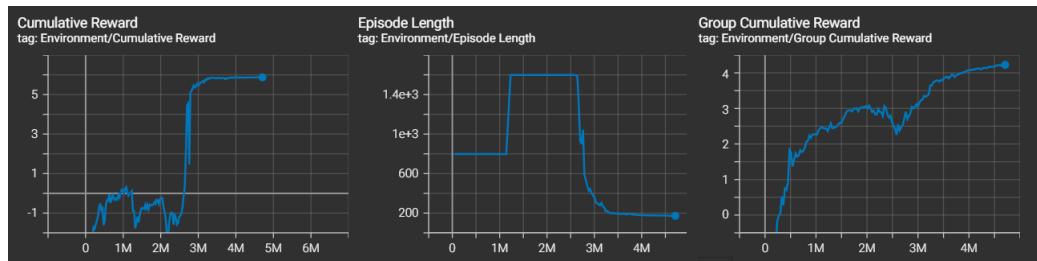


Figura A.4: Ejemplo de Tensorboard.

A.6. Implementación del entrenamiento

Finalmente, una vez que está completo el proceso de aprendizaje y se ha guardado el modelo, es posible añadirlo a Unity y usarlo con agentes compatibles (los agentes que generaron el modelo, o aquellos con las mismas recompensas y observaciones). El modelo entrenado se encontrará en la ruta **results/<run-identifier>/<behavior_name>.onnx** donde **<behavior_name>** es el nombre del comportamiento de los agentes correspondientes al modelo.

Se puede integrar el modelo entrenado en los agentes siguiendo los pasos que se describen a continuación:

1. Mover el archivo del modelo **<behavior_name>.onnx** a **Project/Assets/ML-Agents/TFM/<project_name>**
2. Abrir el Editor de Unity y seleccionar la escena adecuada.
3. Seleccionar el Agente.
4. Arrastrar el archivo **<behavior_name>.onnx** desde la ventana del Project window del editor hasta la casilla del Modelo del Agente en la ventana del inspector. (Fig.A.5)

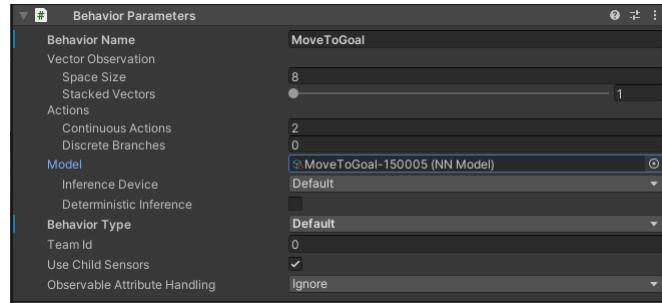


Figura A.5: Implementación del modelo ya entrenado.

Una vez hecho esto, el modelo estaría ya implementado en el agente, y únicamente presionando el botón de play del Editor de Unity seríamos por fin capaces de verlo en funcionamiento y tomar decisiones en tiempo real. (Fig.A.6)

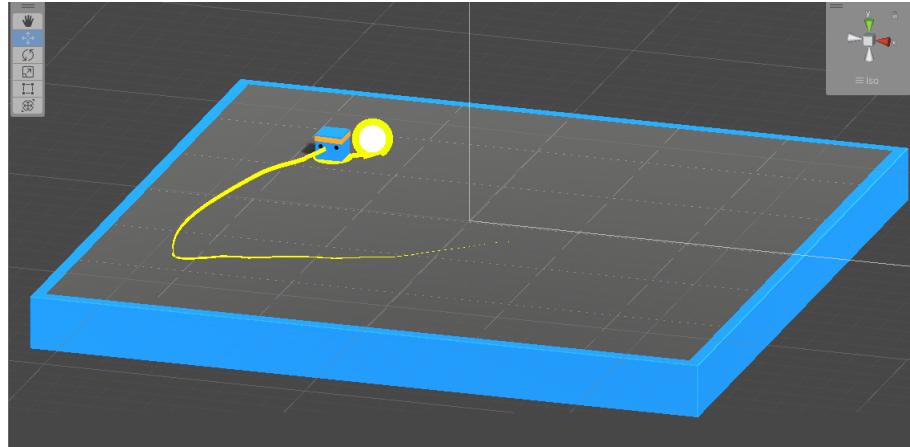


Figura A.6: Entorno de ejemplo en modo de inferencia.

A.7. Experimentación: 'Ficheros YAML'

A.7.1. Fichero de configuración YAML utilizado para resolver los entornos monoagente y multiagentes

```
behaviors:
  FinalAgent:
    trainer_type: ppo
    hyperparameters:
      batch_size: 256
      buffer_size: 4096
      learning_rate: 0.0003
      beta: 0.01
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 8
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 512
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
      curiosity:
        gamma: 0.99
        strength: 0.05
      network_settings:
        hidden_units: 256
        learning_rate: 0.0003
    keep_checkpoints: 50
    max_steps: 2000001
    time_horizon: 128
    summary_freq: 30000
```

Figura A.7: *Fichero YAML para entornos monoagente (PPO).*

```
behaviors:
  FinalMultiAgent:
    trainer_type: poca
    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: 50
  max_steps: 15000000
  time_horizon: 64
  summary_freq: 60000
```

Figura A.8: *Fichero YAML para entornos multiagentes (MA-POCA)*.

```
environment_parameters:
  distance_offset:
    curriculum:
      - name: Lesson0
        completion_criteria:
          measure: reward
          behavior: Climb
          signal_smoothing: true
          min_lesson_length: 300
          threshold: 0.3
        value: 0.0
      - name: Lesson1
        completion_criteria:
          measure: reward
          behavior: Climb
          signal_smoothing: true
          min_lesson_length: 300
          threshold: 0.6
        value:
          sampler_type: uniform
          sampler_parameters:
            min_value: 0
            max_value: 0.5
```

Figura A.9: *Parte del fichero YAML con Curriculum Learning para el Entorno monoagente 1: Climb.*

A.8. IMÁGENES DE LOS CÓDIGOS UTILIZADOS EN EL ENTORNO DE EJEMPLO MENCIONADO EN LA METODOLOGÍA

```
var randomDistance = Academy.Instance.EnvironmentParameters.GetWithDefault("distance_offset", RandomQuantity);
```

Figura A.10: *Código para obtener la altura al utilizar Curriculum learning.*

```
behaviors:
  Climb:
    trainer_type: poca
    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: 50
  max_steps: 21000001
  time_horizon: 64
  summary_freq: 60000
  self_play:
    save_steps: 50000
    team_change: 20000
    swap_steps: 2000
    window: 10
    play_against_latest_model_ratio: 0.5
  initial_elo: 1200.0
```

Figura A.11: *Fichero YAML para entornos multiagentes con Self-Play.*

A.8. Imágenes de los códigos utilizados en el entorno de ejemplo mencionado en la metodología

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// import ML-Agents package
using Unity.MLAgents;
using Unity.MLAgents.Sensors;
using Unity.MLAgents.Actuators;

// Delete Update() since we are not using it, but keep Start()
public class MoveToGoalAgent : Agent{
    // reference to the Rigidbody component to reset the Agent's velocity and later to apply force to it
    Rigidbody rBody;

    // public field of type Transform to the RollerAgent class for reference to move the target
    public Transform Target;

    public Material winMaterial;
    public Material loseMaterial;
    public Material normalMaterial;
    public MeshRenderer floorMeshRenderer;

    void Start(){
        rBody = GetComponent<Rigidbody>();
    }

    // set-up the environment for a new episode
    public override void OnEpisodeBegin()
    {
        // If the Agent fell off the platform, zero its momentum
        if (this.transform.localPosition.y < 0)
        {
            // reset the Agent's velocity and put back onto the floor
            this.rBody.angularVelocity = Vector3.zero;
            this.rBody.velocity = Vector3.zero;
            this.transform.localPosition = new Vector3(0, 0.8f, 0);
            this.transform.localRotation = Quaternion.Euler(0, 0, 0);
        }

        // Move the target to a new spot (moved to a new random location)
        Target.localPosition = new Vector3(Random.Range(-10.0f, +10.0f), 0.6f, Random.Range(-10.0f, +10.0f));
    }
}

```

Figura A.12: Código del agente: *OnEpisodeBegin()*.

```

// Observing the Environment (what information to collect), 8 values
public override void CollectObservations(VectorSensor sensor)
{
    // Target and Agent positions
    sensor.AddObservation(Target.localPosition);           // Target (x,y,z)
    sensor.AddObservation(this.transform.localPosition);    // Agent (x,y,z)

    // Agent velocity
    sensor.AddObservation(rBody.velocity.x);              // Agent Vel (x)
    sensor.AddObservation(rBody.velocity.z);              // Agent Vel (z)
}

```

Figura A.13: Código del agente: *CollectObservations()*.

```

public override void OnActionReceived(ActionBuffers actionBuffers)
{
    // Actions, size = 2
    Vector3 controlSignal = Vector3.zero;
    controlSignal.x = actionBuffers.ContinuousActions[0]; // force applied along the x-axis, MoveX
    controlSignal.y = 0;
    controlSignal.z = actionBuffers.ContinuousActions[1]; // force applied along the z-axis, MoveZ

    // Movemos el objeto
    this.transform.position += (controlSignal * moveSpeed * Time.deltaTime);
    // RollerAgent applies the values from the action[] array to its Rigidbody component rBody, using Rigidbody.Add
    rBody.AddForce(controlSignal * forceAmount); // * Time.deltaTime;

    // Rewards
    // calculates the distance to detect when it reaches the target
    float distanceToTarget = Vector3.Distance(this.transform.localPosition, Target.localPosition);

    // Reached target
    // The Agent is given a reward of 1.0 for reaching the Target cube
    if (distanceToTarget < 1.42f)
    {
        SetReward(1.0f);
        EndEpisode();
        StartCoroutine(
            GoalScoredSwapGroundMaterial(winMaterial, 2));
    }

    // Fell off platform
    // if the Agent falls off the platform, end the episode so that it can reset itself
    else if (this.transform.localPosition.y < 0)
    {
        SetReward(-0.5f);
        EndEpisode();
        StartCoroutine(
            GoalScoredSwapGroundMaterial(loseMaterial, 2));
    }
}

```

Figura A.14: Código del agente: *OnActionReceived()*.

A.9. Código del sistema de checkpoints utilizado

```

public class CheckpointMulti : MonoBehaviour {
    private Collider m_col; // Checkpoint Collider

    private TrackCheckpointsMulti trackCheckpointsMulti;

    private void OnTriggerEnter(Collider col) {
        if (col.gameObject.CompareTag("agent") == true) {
            trackCheckpointsMulti.PlayerThroughCheckpoint(this, col, m_col); //Send CheckpointMulti, agentCollider, checkpointCollider
        }
    }

    public void SetTrackCheckpoints(TrackCheckpointsMulti trackCheckpointsMulti){
        this.trackCheckpointsMulti = trackCheckpointsMulti;
    }

    // Start is called before the first frame update
    void Awake()
    {
        m_col = GetComponent<Collider>();
    }
}

```

Figura A.15: Sistema de Checkpoints: Función *CheckpointsMulti()*.

```
public class TrackCheckpoints : MonoBehaviour
{
    [System.Serializable]
    public class TriggerEvent : UnityEvent<Collider>
    {
    }
    [Header("Trigger Callbacks")]
    public TriggerEvent OnPlayerCorrectCheckpoint = new TriggerEvent();
    public TriggerEvent OnPlayerWrongCheckpoint = new TriggerEvent();

    private List <CheckpointSingle> checkpointSingleList;

    private int nextCheckpointSingleIndex;

    private void Awake()
    {
        Transform checkpointsTransform = transform.Find("Checkpoints");

        checkpointSingleList = new List <CheckpointSingle>();
        nextCheckpointSingleIndex = 0;

        foreach (Transform checkpointsSingleTransform in checkpointsTransform) {
            CheckpointSingle checkpointSingle = checkpointsSingleTransform.GetComponent<CheckpointSingle>();

            checkpointSingle.SetTrackCheckpoints(this);

            checkpointSingleList.Add(checkpointSingle);
        }
    }

    public void PlayerThroughCheckpoint(CheckpointSingle checkpointSingle, Collider m_col)
    {
        if (checkpointSingleList.IndexOf(checkpointSingle) == nextCheckpointSingleIndex)
        {
            // Correct checkpoint
            nextCheckpointSingleIndex = (nextCheckpointSingleIndex + 1) % checkpointSingleList.Count;
            OnPlayerCorrectCheckpoint.Invoke(m_col);
        }
        else {
            // Wrong checkpoint
            OnPlayerWrongCheckpoint.Invoke(m_col);
        }
    }

    public void OnEpisodeBegin()
    {
        Awake();
    }
    public CheckpointSingle GetNextCheckpoint(int indice) {
        return checkpointSingleList[indice];
    }
}
```

Figura A.16: Sistema de Checkpoints: Función *TrackCheckpoints()*.

Bibliografía

- Alphago. <https://www.deepmind.com/research/highlighted-research/alphago>.
- Alphastar: Grandmaster level in starcraft ii using multi-agent reinforcement learning. <https://www.deepmind.com/blog/alphastar-grandmaster-level-in-starcraft-ii-using-multi-agent-reinforcement-learning>.
- Alphazero: Shedding new light on chess, shogi, and go. <https://www.deepmind.com/blog/alphazero-shedding-new-light-on-chess-shogi-and-go>.
- Gym documentation. <https://www.gymlibrary.ml/>.
- Ma-poca (multiagent posthumous credit assignment). <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Design-Agents.md#groups-for-cooperative-scenarios>.
- Muzero: Mastering go, chess, shogi and atari without rules. <https://www.deepmind.com/blog/muzero-mastering-go-chess-shogi-and-atari-without-rules>.
- Openai: Emergent tool use from multi-agent interaction. <https://openai.com/blog/emergent-tool-use/>.
- Spinningup openai, part 1: Key concepts in rl. https://spinningup.openai.com/en/latest/spinningup/rl_intro.html.
- Spinningup openai, part 2: Kinds of rl algorithms. https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html.
- Unity ml-agents toolkit. <https://github.com/Unity-Technologies/ml-agents>.
- Unity ml-agents toolkit behavioral cloning (bc). <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md#behavioral-cloning-bc>.
- Unity ml-agents toolkit curriculum learning. <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md#solving-complex-tasks-using-curriculum-learning>.
- Unity ml-agents toolkit gail (generative adversarial imitation learning). <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md#gail-generative-adversarial-imitation-learning>.
- Unity ml-agents toolkit overview. <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md>.
- Unity ml-agents toolkit overview. <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md#imitation-learning>.
- Unity ml-agents toolkit: Walljump example. <https://github.com/Unity-Technologies/ml-agents/blob/main/docs/images/curriculum.png>.
- BSC y UPC. Deep reinforcement learning explained. <https://torres.ai/deep-reinforcement-learning-explained-series/>.
- Ho, J. y Ermon, S. (2016). Gail (generative adversarial imitation learning). <https://arxiv.org/abs/1606.03476>.
- Joseph, R. y Ali, F. (2017a). Ma-poca (multiagent posthumous credit assignment). <https://arxiv.org/abs/1707.06347>.
- Joseph, R. y Ali, F. (2017b). Ppo (proximal policy optimization algorithms). <https://arxiv.org/abs/1707.06347>.
- Julian Ibarz, Jie Tan, C. F. M. K. P. P. S. L. (2017). How to train your robot with deep reinforcement learning; lessons we've learned. <https://arxiv.org/abs/2102.02915>.
- OpenAI. Solving rubik's cube with a robot hand. <https://openai.com/blog/solving-rubiks-cube/>.
- Robotics, O. Ros oficial page. <https://www.ros.org>.
- Robotics, O. Ros rl documentation. https://theconstructcore.bitbucket.io/openai_ros/index.html.
- Robotics, O. Ros rl source. https://bitbucket.org/theconstructcore/openai_ros/src/solving-complex-tasks-using-curriculum-learning.

APÉNDICE A. APÉNDIZE

Robotics, O. Ros wiki page. <https://docs.ros.org/en/foxy/index.html>.

Sutton, R. S. y Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

team, O. Ros rl page. http://wiki.ros.org/openai_ros.

Tuomas Haarnoja, Aurick Zhou, P. A. S. L. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. <https://arxiv.org/abs/1801.01290>.

Vinicius G. Goecks, Gregory M. Gremillion, V. J. L. J. V. N. R. W. (2019). Integrating behavior cloning and reinforcement learning for improved performance in dense and sparse reward environments. <https://arxiv.org/abs/1910.04281>.

Volodymyr Mnih, Koray Kavukcuoglu, D. S. A. A. R. J. V. M. G. B. A. G. M. R. A. K. F. G. O. S. P. C. B. A. S. I. A. H. K. D. K. D. W. S. L. . D. H. (2015). Human-level control through deep reinforcement learning. <https://www.nature.com/articles/nature14236>.

Volodymyr Mnih, Koray Kavukcuoglu, D. S. A. G. I. A. D. W. M. R. Playing atari with deep reinforcement learning. <https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>.

VPetru Soviany, Radu Tudor Ionescu, P. R. N. S. (2021). Curriculum learning: A survey. <https://arxiv.org/abs/2101.10382>.