ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO UNIVERSITARIO EN INGENIERIA DE COMPUTADORES

Aprendizaje por refuerzo en Unity: MlAgents

UNIVERSIDAD DE SEVILLA

José Gallardo Harillo

Tutorizado y dirigido por

Dr. Fernando Sancho Caparrini

Índice general

Introducción y objetivos

El Trabajo Fin de Grado (TFG) que se presenta a continuación está centrado en el entrenamiento de agentes mediante aprendizaje por refuerzo mediante el uso de la herramienta **MlAgents** de Unity.

El objetivo principal del trabajo es hacer una revisión de los conocimientos previos, tanto del campo del aprendizaje por refuerzo como de la herramienta misma que se usa para su implementación práctica en Unity (analizar los límites que podemos encontrar, reconocer qué partes son las más importantes para su implementación, etc.), que pueden ser necesarios para posibles usuarios de esta metodología en la creación de experiencias multimedia.

En este contexto, marcamos las siguientes competencias a cumplir mediante la realización del trabajo:

- Conocimiento adecuado del marco de modelado y generación de modelos de **Aprendizaje por Refuerzo**.
- Comprensión del uso de las redes neuronales en el contexto del aprendizaje por refuerzo.
- Adquisición de experiencia en programación dentro del entorno de desarrollo **Unity**.
- Familiarización con la herramienta MlAgents del entorno Unity.

Aunque sabíamos al comienzo de la realización del TFG que la consecución de los siguientes objetivos no tendría cabida dentro del desarrollo por falta de tiempo, se plantearon también los siguientes extensiones, que han marcado muchas de las decisiones tomadas durante dicho desarrollo:

- Implementación de un videojuego/simulación en **Realidad Virtual** con agentes entrenados mediante MlAgents.

- Aplicación de Aprendizaje por Refuerzo a la creación de comportamiento **procedural** de los agentes involucrados en el sistema.
- Introducción al entrenamiento de sistemas multiagente, y no solo de agentes individuales.

Conceptos básicos generales

A lo largo del TFG vamos a hacer uso de un conjunto de conceptos propios de la Inteligencia Artificial, de Unity y de otras áreas y herramientas de la Computación. Para facilitar la lectura del texto, introducimos muy brevemente aquí algunos de los más repetidos, aunque muchos de ellos será introducidos con mayor extensión durante la memoria:

- **Machine Learning (ML, por sus siglas en inglés)**: Es uno de los campos fundamentales de la inteligencia artificial, donde se diseñan algoritmos y modelos que permiten a las máquinas/programas aprender y mejorar en las tareas que desempeñen por medio de la experiencia.
- **Redes neuronales**: Es uno de los marcos de creación de modelos/algoritmos ML más populares, donde su ventaja principal es su velocidad, adaptabilidad y rendimiento en el procesamiento de información.
- **Unity**[1]: Es uno de los motores de videojuego multiplataforma más populares (hoy en día, se ha convertido en una herramienta transversal de experiencias multimedia, no solo centrado en los videojuegos). Los componentes que vamos a usar en la práctica son:
  - **Escenario**: Entorno o espacio virtual donde se desarrolla el juego/simulación.
  - **Objeto**: Cada uno de los elementos pertenecientes al escenario.
  - **Componente**: Pieza de funcionalidad agregable a un objeto. Los componentes pueden estar creados ya, o bien pueden ser creados por el usuario por medio de scripts. Podemos destacar los siguientes componentes:
    - **Collider**: Componente que proporciona la capacidad de actuar en base a las colisiones del objeto al que pertenece. Hay dos tipos:

[1] https://unity.com/es

        - ◇ **Collider físico** (`is trigger` desactivado): Un objeto no puede traspasar al objeto que lo implemente.
        - ◇ **Collider desencadenador** (`is trigger` activado): Un objeto puede ser traspasado pero da una señal que puede ser tratada desde otras funciones asociadas, como `OnCollisionEnter`, `OnCollisionStay` o `OnCollisionExit`.
    - **Rigid Body**: Componente que proporciona al objeto un comportamiento físico y, por ende, la capacidad de que le afecten fuerzas externas como la gravitatoria o la de algún otro objeto.

- **Archivo de configuración .yaml**: Es un archivo de texto plano que se utiliza para almacenar datos estructurados. La sintaxis `.yaml` se utiliza comúnmente para archivos de configuración debido a su simplicidad y facilidad de lectura.

- **Anaconda**[2]: Es una plataforma de distribución y gestión de paquetes de software para varios lenguajes de programación (como `Python` y `R`).

- **TensorBoard**[3]: Es una herramienta de visualización interactiva para *TensorFlow*, un popular marco de trabajo de aprendizaje automático. Permite visualizar y entender el modelo de aprendizaje automático a través de gráficos, tablas y otras representaciones visuales:

    - **Visualizar el modelo de aprendizaje automático:** Se puede ver el gráfico del modelo de `TensorFlow` y ver cómo se conectan las diferentes capas y operaciones.
    - **Monitorear el rendimiento**: `TensorBoard` proporciona información en tiempo real sobre las métricas de entrenamiento y validación, lo que ayuda a evaluar y ajustar el rendimiento del modelo.
    - **Depurar problemas**: Los usuarios pueden utilizar `TensorBoard` para identificar problemas con el modelo, tales como gradientes que explotan o desaparecen, y ajustar la estructura del modelo en consecuencia.

    `TensorBoard` se ejecuta como una aplicación web que se conecta al modelo de `TensorFlow` a través de un servidor web local. Para utilizar `TensorBoard`, se deben agregar llamadas a la API de `TensorBoard` al código de `TensorFlow`, lo que permite a `TensorBoard` acceder a

[2] https://www.anaconda.com/
[3] https://www.tensorflow.org/tensorboard?hl=es-419

los datos de entrenamiento y validación y mostrarlos en la interfaz de usuario.

Capítulo 1

Fundamentos del Aprendizaje por Refuerzo

1.1. RL, la tercera joya de la corona del ML

En el área del *Machine Learning* (preferimos usar en la memoria el término en inglés, porque está mucho más extendido que el término correspondiente en español, *Aprendizaje Automático*), el cual abarca muchas ramas de estudio dentro de la propia Inteligencia Artificial, se pueden reconocer dos paradigmas principales de aprendizaje en donde se enmarcan la mayoría de soluciones propuestas: el **Aprendizaje Supervisado** (SL, por sus siglas en inglés), en el que, haciendo uso de conocimiento explícito de la salida que debe producirse en forma de etiquetas (sean numéricas o simbólicas), el algoritmo debe aprender a dar una respuesta correcta; y el **Aprendizaje no Supervisado** (UL, por sus siglas en inglés), en el que el algoritmo no tiene esa dirección adicional sobre la salida esperada, y debe identificar patrones existentes en las muestras de aprendizaje (muy comúnmente, en forma de agrupaciones de los datos en clústeres). Con estos dos paradigmas se han podido abordar una amplia gama de problemas, donde destacan los problemas de clasificación, regresión y clusterización.

En este trabajo nos vamos a centrar concretamente en las limitaciones que tienen estas aproximaciones a la hora de abordar problemas de toma de decisiones secuenciales, es decir, aquellos problemas donde se debe tomar decisiones en un entorno dinámico secuencialmente (en el siguiente apartado lo aplicaremos a un ejemplo). Para ello haremos uso de una de nuestras formas de aprender más primitivas y naturales, siguiendo mecanismos que son reconocibles en la forma en que aprenden muchos seres vivos (y en particular,

en los mecanismos que seguimos los seres humanos).Con este fin, y como uno de los condicionamientos operantes reconocibles, prestaremos especial atención a la relación existente entre acciones ejecutadas por el individuo que está realizando el aprendizaje y la toma de decisiones y las consecuencias que obtiene por las acciones que toma.

A partir de esta aproximación entra en juego un tercer paradigma, conocido como **Aprendizaje por Refuerzo** (RL, por sus siglas en inglés), y que proporciona un paradigma separado y distintivo que se centra específicamente en abordar este tipo de problemas mediante la interacción con un entorno y la retroalimentación en forma de recompensas o penalizaciones (ver Figura 1.1).

Figura 1.1: Paradigmas del Machine Learning

La información con la que hemos podido realizar los siguientes apartados de este capítulo (este inclusive) ha sido sacada del libro de *Reinforcement Learning, An introduction* [13].

1.2. RL vs SL y UL

Es interesante destacar que RL, a diferencia del SL, no tiene una función tan específica y clara, como clasificar, sino que abarca un escenario amplio donde se deben tener en cuenta múltiples factores que no son fácilmente medibles. Vamos a comenzar con un ejemplo que pretende aclarar el contexto en el que nos situamos:

Supongamos que somos un individuo que tiene como misión llegar desde nuestra casa a un lugar en concreto, el cual está a una determinada distancia. Nuestra primera toma de decisiones puede ser por ejemplo el medio por el que vamos a llegar, si tomaremos el bus, iremos en nuestro coche, o preferimos

acercarnos caminando. Podríamos imaginar que, en principio, cualquiera de las dos primeras opciones proporcionarían una recompensa mayor desde el punto de vista de uso del tiempo, pero si tenemos en cuenta que pueden darse condiciones de tráfico denso, estas dos primeras opciones podrían tener una penalización y puede ser mejor entonces ir caminando. Otra decisión a tomar puede ser si llevar o no paragüas, algo que proporciona una ventaja en caso de que llueva, pero se convierte en una carga en caso negativo. Como último ejemplo de toma de decisiones sería nuestra acción respecto a los semáforos existentes, que proporcionan condicionantes distintos dependiendo de si vamos caminando o en bus/coche.

Si quisiésemos modelar este problema como un problema SL tendríamos que hacer uso de varios sistemas que se enfoquen en cada una de las tomas de decisión (como las tres destacadas anteriormente), diseñando un sistema para cada una y necesitando conjuntos de datos etiquetados distintos que permitan diferenciar las buenas decisiones de las malas en cada una de las opciones (por ejemplo, para el caso de llevar paragüas o no, proporcionar varios casos donde llueve o no según la temperatura o el cielo suponiendo estas variables relacionadas).

La aproximación para resolver este tipo de problemas desde RL es distinta. En este caso, una opción sería que el individuo fuese tomando decisiones y aprendiendo por medio de las penalizaciones o recompensas que vaya obteniendo, y todo ello dinámicamente, sin necesidad de un supervisor que proporcione la casuística necesaria para cubrir tantas formas de tomar las decisiones como se vean necesarias, siendo capaz mediante penalizaciones de saber que si el cielo esta nublado y no llevas paragüas te vas a mojar o mediante recompensas saber que si esta soleado y no llevas paragüas, que entonces no te mojas y tienes las manos libres, además de solo necesitar un sistema de aprendizaje por refuerzo.

Enfocándonos ahora solo en RL y UL, podemos decir que el aprendizaje por refuerzo de algún modo toma un papel parecido, ya que no devuelve ningún resultado de salida, sino que estructura internamente el conocimiento en función de la recompensa acumulada, pero que no tiene como única función el solo estructurar, sino además tomar las decisiones en base a lo estructurado.

Vemos que la necesidad que cubre la aproximación RL está claramente en el núcleo de lo que entendemos por aprendizaje, pero ha tenido mucho menos desarrollo en los últimos años que el resto de aproximaciones y tareas reconocidas en ML. En este trabajo nos planteamos conocer más a fondo los fundamentos de esta aproximación y usaremos algún sistema de experimentación para poner a prueba su capacidad.

Pero antes, veamos algunas de sus aplicaciones y la evolución que ha tenido lo largo de la historia por medio de los métodos de aprendizaje que este aplica.

1.3. Aplicaciones en la vida cotidiana

El aprendizaje por refuerzo ha sido considerablemente implementado en otras ingenierías, disciplinas científicas o incluso estudios fuera del campo de la ciencia, mostrando un uso muy extendido que originalmente surgió lejos del campo de la computación.

Veamos varias aplicaciones, ordenadas según el campo donde se ha hecho uso:

- **Psicología**[1]:
  - **Tratamiento de adicciones**: Implementado para el desarrollo de programas de tratamiento de adicciones [10], como el alcoholismo, la ludopatía, o la adicción a las drogas.
  - **Terapia de comportamiento**: Implementado para el desarrollo de terapias de comportamiento para tratar trastornos como el autismo [5] y la depresión [1].
- **Neurofísica**:
  - **Control de prótesis** [2]: Implementado para permitir que las personas con amputaciones controlen prótesis mediante señales cerebrales.
  - **Neurociencia cognitiva**[2] Implementado para investigar la forma en que los seres humanos toman decisiones y aprenden de la experiencia.
- **Medicina**:
  - **Control de la glucemia en diabéticos** [8]: Implementado para el desarrollo de sistemas que ayuden a los diabéticos a controlar su glucemia mediante la administración de insulina.

[1] En el capítulo 14 del libro de *Reinforcement Learning, An introduction* [13] habla del RL con respecto este enfoque.

[2] En el capítulo 15 del libro de *Reinforcement Learning, An introduction* [13] habla del RL con respecto este enfoque.

- **Tratamiento del dolor** [6]: Implementado para el desarrollo de tratamientos para el dolor crónico.

- **Robótica**[3]:
  - **Control de robots**: Implementado para el desarrollo de algoritmos de control para robots autónomos.
  - **Navegación robótica**: Implementado para permitir que los robots aprendan a navegar en entornos desconocidos.

[3] El artículo [4] trata los dos puntos mencionados de este enfoque

- **Economía**:
  - **Control de inventario** [11]: Implementado para la optimización del control de inventario en empresas, permitiendo una gestión más eficiente de los recursos.
  - **Análisis de datos financieros** [7]: Implementado para el análisis de los mercados financieros y predecir las fluctuaciones de los precios de las acciones.
- **Arquitectura**:
  - **Diseño de edificios inteligentes** [12]: Implementado para el desarrollo de sistemas de automatización y control de edificios inteligentes, que ajustan automáticamente las condiciones ambientales, como la iluminación y la temperatura, para maximizar la eficiencia energética y el confort de los ocupantes.
  - **Diseño acústico** [9]: Implementado para el diseño de sistemas de reducción de ruido en edificios y espacios públicos, permitiendo una reducción efectiva del ruido ambiental.

1.4. Orígenes del RL: Las ramas principales

El aprendizaje por refuerzo, al igual que muchos campos de la ingeniería en general, ha sido una solución resultante que proviene de la unión de dos o más ramas distintas, una solución beneficiosa para tales ramas. En este caso, podemos destacar su origen en el estudio de dos ramas concretas del ML[4]:

- **El aprendizaje por ensayo y error**: una rama de la inteligencia artificial y el aprendizaje automático que se centra en la investigación y desarrollo de algoritmos de aprendizaje en los que un individuo intenta varias veces diferentes soluciones a un problema, evaluando cada intento y aprendiendo de los errores hasta que encuentra una solución efectiva.
- **La teoría del control**: una rama de las matemáticas aplicadas que se centra en el diseño de sistemas dinámicos para lograr un objetivo específico.

[4]Este apartado está principalmente influenciado por el capítulo 1 (Apartado 7 concretamente) de *Reinforcement Learning, An introduction* [13].

La razón de ser de estas dos ramas en el aprendizaje por refuerzo surge de la necesidad de abordar diferentes aspectos y desafíos en el proceso de aprendizaje y toma de decisiones de un individuo. El ensayo y error, por su parte, permite al agente descubrir y aprender mediante la experiencia a medida que interactúa con el entorno, estando así relacionada con la exploración y explotación, donde el agente debe encontrar un equilibrio entre probar nuevas acciones para obtener información que podría ser más ventajosa, y aprovechar el conocimiento adquirido para maximizar las recompensas a largo plazo; mientras que la rama de la teoría del control se enfoca en la planificación, en establecer un marco teórico sólido para la toma de decisiones óptimas y la adaptación del agente para maximizar las recompensas esperadas.

1.4.1. Aprendizaje por ensayo y error

Enfocándonos en la rama de aprendizaje por ensayo y error, la idea se remonta a la década de 1850 y a la discusión de **Alexander Bain** sobre el aprendizaje mediante *tentativas y experimentos*, y más explícitamente al uso que el etólogo y psicólogo británico **Conway Lloyd Morgan** hizo del término en 1894 para describir sus observaciones del comportamiento animal.

Quizá el primero en expresar de mejor forma la esencia del aprendizaje por ensayo y error como principio de aprendizaje fue **Edward Thorndike**:

> "De varias respuestas hechas a la misma situación, aquellas que van acompañadas o seguidas de satisfacción para el animal, otras cosas siendo iguales, estarán más firmemente conectadas con la situación, de modo que cuando vuelva a ocurrir, será más probable que se repitan aquellas que van acompañadas o seguidas de incomodidad para el animal, otras cosas siendo iguales, tendrán sus conexiones con esa situación debilitadas, de modo que cuando vuelva a ocurrir, serán menos probables que ocurran. Cuanto mayor sea la satisfacción o la incomodidad, mayor será el fortalecimiento o debilitamiento del vínculo." (Thorndike, 1911)

Thorndike llamó a esto la *Ley del Efecto*, ya que describe el efecto de los eventos de refuerzo en la tendencia a seleccionar acciones.

Más tarde, este autor modificaría la ley para actualizarlo con los datos posteriores sobre el aprendizaje animal, como las diferencias entre los efectos de la recompensa y el castigo. En cualquiera de sus formas, la ley ha generado considerable controversia entre los teóricos del aprendizaje. A pesar de esto, la *Ley del Efecto*, es ampliamente considerada como un principio básico subyacente a gran parte del comportamiento.

El término "refuerzo" en el contexto del aprendizaje animal comenzó a usarse mucho después de la expresión de la *Ley del Efecto* de Thorndike, apareciendo por primera vez en la traducción al inglés de 1927 de la monografía de **Pavlov** sobre reflejos condicionados.

Pavlov observó que los perros podían aprender a asociar un estímulo neutral (por ejemplo, el sonido de una campana) con un estímulo que naturalmente provocaba una respuesta (por ejemplo, la comida). Después de repetir esta asociación muchas veces, el sonido de la campana por sí solo era suficiente para provocar la respuesta (salivación) que antes sólo era provocada por la comida, describiendo Pavlov así el refuerzo como *el fortalecimiento de un patrón de comportamiento debido a que un animal recibe un estímulo, un reforzador, en una relación temporal apropiada con otro estímulo o con una respuesta.*

A partir de todo lo mencionado hasta el momento, podemos diferenciar notablemente dos tipos de condicionamiento:

- **Condicionamiento clásico o pavloviano**: Como su segundo nombre indica, esta inspirado en el trabajo de Pavlov, refiriéndose al proceso de asociar un estímulo neutral con un estímulo que naturalmente provoca una respuesta.
- **Condicionamiento operante o instrumental**: Se inspira en el trabajo de Thorndike, refiriéndose al proceso de asociar una respuesta con una consecuencia reforzante o castigadora.

Algunos psicólogos extendieron la idea de refuerzo para incluir tanto la debilitación como el fortalecimiento del comportamiento, y extendieron la idea de un reforzador para incluir posiblemente la omisión o la terminación de un estímulo.

Para ser considerado un reforzador, el fortalecimiento o debilitamiento debe persistir después de que se retira el reforzador. Un estímulo que simplemente atrae la atención del animal o que energiza su comportamiento sin producir cambios duraderos no se consideraría un reforzador.

La idea de implementar el aprendizaje por ensayo y error en una computadora apareció entre los primeros pensamientos sobre la posibilidad de inteligencia artificial. En un informe de 1948, **Alan Turing** describió un diseño para un *sistema de placer-dolor* que funcionaba siguiendo los lineamientos de la *Ley del Efecto*:

> "Cuando se alcanza una configuración para la cual la acción es indeterminada, se hace una elección aleatoria para los datos faltantes y se realiza la entrada correspondiente en la descripción, tentativamente, y se aplica. Cuando ocurre un estímulo doloroso, se cancelan todas las entradas tentativas, y cuando ocurre un estímulo placentero, se hacen todas las entradas permanentes." (Turing, 1948)

Fue en esa franja temporal cuando se construyeron muchas máquinas electro-mecánicas ingeniosas que demostraban el aprendizaje por ensayo y error. La primera de ellas fue una máquina construida por **Thomas Ross** en 1933, que era capaz de encontrar su camino a través de un laberinto simple y recordar el camino a través de la configuración de interruptores. En 1951, **W. Grey Walter** construyó una versión de su tortuga mecánica (Walter, 1950), que se trataba de un pequeño robot que se podía mover de forma autónoma y se alimentaba con una batería, siendo su movimiento controlado por circuitos electrónicos que imitaban la actividad de los nervios. En 1952, **Claude Shannon** mostró un ratón corriendo por un laberinto llamado **Teseo** que utilizó el ensayo y error para encontrar su camino a través de un laberinto, donde era el propio laberinto el que recordaba las direcciones exitosas a través de imanes y relés bajo su suelo. **J. A. Deutsch** (1954) describió una máquina de resolución de laberintos basada en su teoría del comportamiento (Deutsch, 1953) que tiene algunas propiedades en común con el **aprendizaje por refuerzo basado en modelos** (MBRL, por sus siglas en inglés), que es una variante del aprendizaje por refuerzo en la que el agente utiliza un modelo del entorno para tomar decisiones. **Marvin Minsky** (1954) discutió modelos computacionales de aprendizaje por refuerzo y describió su construcción de una máquina analógica compuesta por componentes que llamó **SNARCs** (*Stochastic Neural-Analog Reinforcement Calculators*) que se asemejaban a conexiones sinápticas modificables en el cerebro.

La construcción de máquinas de aprendizaje electro-mecánicas abrió el camino a la programación de computadoras digitales para realizar varios tipos de aprendizaje, algunos de los cuales implementaron el aprendizaje por ensayo y error.

Farley y Clark (1954) describieron una simulación digital de una máquina de aprendizaje de redes neuronales que aprendía por ensayo y error, pero

pronto su interés se desvió del aprendizaje por ensayo y error a la generalización y el reconocimiento de patrones, es decir, del RL al SL (Farley y Clark, 1955), iniciando así un patrón de confusión acerca de la relación entre estos tipos de aprendizaje.

Muchos investigadores parecían creer que estaban estudiando el aprendizaje por refuerzo cuando en realidad estaban estudiando el aprendizaje supervisado, como por ejemplo los pioneros de las redes neuronales artificiales como **Rosenblatt** (1962) y **Widrow y Ho** (1960), los cuales estaban claramente motivados por el aprendizaje por refuerzo, ya que utilizaban el lenguaje de las recompensas y los castigos, pero los sistemas que estudiaron eran sistemas de aprendizaje supervisado adecuados para el reconocimiento de patrones y el aprendizaje perceptual. Incluso hoy en día, algunos investigadores y libros de texto minimizan o difuminan la distinción entre estos tipos de aprendizaje. Por ejemplo, algunos libros de texto han utilizado el término "ensayo y error" para describir las redes neuronales artificiales que aprenden a partir de ejemplos de entrenamiento. Esta es una confusión comprensible porque estas redes utilizan información de error para actualizar los pesos de conexión, pero esto no capta la característica esencial del aprendizaje por ensayo y error como la selección de acciones basadas en la retroalimentación evaluativa que no depende del conocimiento de cuál debería ser la acción correcta.

En parte como resultado de estas confusiones, la investigación sobre el aprendizaje genuino por prueba y error se volvió rara en las décadas de 1960 y 1970, aunque hubo notables excepciones.

En los años 60, los términos "refuerzo" y "aprendizaje por refuerzo" se usaron por primera vez en la literatura de ingeniería para describir el uso del aprendizaje por prueba y error en ingeniería. Un artículo especialmente influyente fue uno de Marvin Minsky, *Pasos hacia la inteligencia artificial* (Minsky, 1961), que discutió varios temas relevantes para el aprendizaje por prueba y error, incluyendo la predicción, la expectativa y lo que llamó el problema básico de asignación de crédito para sistemas de aprendizaje por reforzamiento complejos: "¿Cómo se distribuye el crédito por el éxito entre las muchas decisiones que pueden haber participado en su producción?"

Entrando en las excepciones a la relativa falta de estudio teórico y computacional del aprendizaje por ensayo y error genuino (1960-1970), una excepción fue el trabajo del investigador neozelandés **John Andreae**, quien desarrolló un sistema llamado **STeLLA** que aprendía por ensayo y error en interacción con su entorno. Este sistema incluía un modelo interno del mundo y, más tarde, un "monólogo interno"para lidiar con problemas de estado

oculto.

El trabajo posterior de Andreae (1977) puso más énfasis en el aprendizaje a partir de un maestro, pero aún incluía el aprendizaje por ensayo y error, con la generación de eventos novedosos como uno de los objetivos del sistema. Una característica de este trabajo fue un "proceso de filtrado hacia atrás", desarrollado más completamente en Andreae (1998), que implementaba un mecanismo de asignación de crédito similar a las operaciones de actualización de retroceso implementadas en la **programación dinámica**, concepto que nos encontraremos de nuevo más tarde y que es clave en el aprendizaje por refuerzo.

Desafortunadamente, su investigación pionera no fue muy conocida y no tuvo un gran impacto en la investigación posterior sobre RL, siendo más influyente el trabajo de **Donald Michie** en 1961 y 1963, que describió un sistema simple de aprendizaje por ensayo y error para aprender a jugar al tres en raya (o tic tac toe) llamado **MENACE** (*Matchbox Educable Naughts and Crosses Engine*), que consistía en una caja de fósforos para cada posición posible del juego, conteniendo cada caja de fósforos un número de cuentas de colores, un color diferente para cada posible movimiento desde esa posición. Al sacar una cuenta al azar de la caja de fósforos correspondiente a la posición actual del juego, se podía determinar el movimiento de MENACE. Cuando terminaba un juego, se agregaban o quitaban cuentas de las cajas utilizadas durante el juego para recompensar o castigar las decisiones de MENACE.

Michie y Chambers (1968) describieron otro aprendiz de refuerzo para el tres en raya llamado **GLEE** (*Game Learning Expectimaxing Engine*) y un controlador de aprendizaje por refuerzo llamado **BOXES**, que se aplicó a la tarea de aprender a equilibrar un poste articulado a un carrito móvil sobre la base de una señal de fracaso que ocurría solo cuando el poste caía o el carrito llegaba al final de una pista. Esta tarea se adaptó del trabajo anterior de **Widrow y Smith** (1964), quienes utilizaron métodos de aprendizaje supervisado, asumiendo la instrucción de un maestro ya capaz de equilibrar el poste. La versión de Michie y Chambers de equilibrio del poste es uno de los mejores ejemplos tempranos de una tarea de aprendizaje por refuerzo en condiciones de conocimiento incompleto, influyendo en gran parte del trabajo posterior del aprendizaje por refuerzo.

Widrow, Gupta, y Maitra (1973) modificaron el algoritmo **LMS** (*Least-Mean-Square*) de Widrow y Ho (1960) para producir una regla de aprendizaje por refuerzo que pudiera aprender de señales de éxito y fracaso en lugar de a partir de ejemplos de entrenamiento, llamando a esta forma de aprendizaje **adaptación de arranque selectivo** y la describieron como *aprendizaje*

con un crítico en lugar de *aprendizaje con un maestro*. Analizaron esta regla y mostraron cómo podía aprender a jugar al blackjack, siendo esta una incursión aislada en el aprendizaje por refuerzo por parte de Widrow, cuyas contribuciones al aprendizaje supervisado fueron mucho más influyentes.

La investigación sobre autómatas de aprendizaje tuvo una influencia más directa en la rama de prueba y error que condujo a la investigación moderna sobre el aprendizaje por refuerzo. Estos son métodos para resolver un problema de aprendizaje no asociativo y puramente selectivo conocido como el **bandido de k brazos** por analogía a una máquina tragaperras "bandido de un solo brazo", solo que con k palancas.

Los autómatas de aprendizaje son máquinas simples de baja memoria para mejorar la probabilidad de recompensa en estos problemas, originándose con el trabajo en la década de 1960 del matemático y físico ruso **M. L. Tsetlin** (1973) y desarrollándose extensamente desde entonces dentro de la ingeniería. Estos desarrollos incluyeron el estudio de autómatas de aprendizaje estocásticos, que son métodos para actualizar las probabilidades de acción en función de señales de recompensa. Aunque no fue desarrollado tradicionalmente dentro de los autómatas de aprendizaje estocásticos, el algoritmo **Alopex** de **Harth y Tzanakou** (1974) (para algoritmos de extracción de patrones) es un método estocástico para detectar correlaciones entre acciones y refuerzo que influyó en algunas de las investigaciones tempranas. Los autómatas de aprendizaje estocásticos fueron prefigurados por trabajos anteriores en psicología, comenzando con el esfuerzo de **William Estes** (1950) hacia una teoría estadística del aprendizaje y desarrollado aún más por otros.

Harry Klopf fue quizás la persona responsable de revivir la rama de prueba y error del aprendizaje por refuerzo dentro de la inteligencia artificial (1972, 1975, 1982), que reconoció que los aspectos esenciales del comportamiento adaptativo se estaban perdiendo a medida que los investigadores del aprendizaje se enfocaban casi exclusivamente en el aprendizaje supervisado. Según Klopf, lo que faltaba eran los aspectos hedónicos del comportamiento, que eran los siguientes:

- La motivación para lograr algún resultado del entorno.
- El control el entorno hacia fines deseados.
- El alejamiento de fines no deseados.

Esta es la idea esencial del aprendizaje por prueba y error.

Las ideas de Klopf fueron especialmente influyentes porque la evaluación de ellas llevó a la apreciación de la distinción entre el SL y RL, y al eventual

enfoque en el aprendizaje por refuerzo. Otros estudios mostraron cómo el aprendizaje por refuerzo podría abordar problemas importantes en el aprendizaje de redes neuronales artificiales, en particular, cómo podría producir algoritmos de aprendizaje para redes multicapa.

1.4.2. Teoría del control óptimo

A continuación nos vamos a enfocar en la segunda rama principal del aprendizaje por refuerzo: el control óptimo.

El término "control óptimo" comenzó a utilizarse a finales de los años 50 para describir el problema de diseñar un controlador que minimizase o maximizase una medida del comportamiento de un sistema dinámico a lo largo del tiempo. Una de las aproximaciones a este problema fue desarrollada a mediados de los años 50 por **Richard Bellman**, un matemático aplicado que desempeñó un papel fundamental en la formulación y desarrollo de la **programación dinámica** en la década de los años 50 (veremos en profundidad en qué consiste), a través de la extensión de una teoría del siglo XIX de **Hamilton y Jacobi**.

Este enfoque utiliza los conceptos de estado de un sistema dinámico y de una función de valor, o *función de retorno óptimo*, para definir una ecuación funcional, ahora llamada **ecuación de Bellman** (la ecuación más importante del aprendizaje por refuerzo).

Bellman también introdujo la versión estocástica discreta del problema de control óptimo, los **procesos de decisión de Markov** (MDP, por sus sigas en inglés, otro concepto clave que veremos más tarde), el marco en el que **Ronald Howard** ideó el método de iteración de políticas para MDP.

Antes de la contribución de Ronald Howard, la resolución de los problemas de control óptimo y MDPs era a través de técnicas *ad hoc*, es decir, heurísticas y aproximaciones numéricas que no garantizaban la optimalidad de la solución. La contribución de Howard fue proporcionar un método sistemático y garantizado para encontrar la política óptima para un MDP, en 4 pasos:

1. Se comienza con una política arbitraria que determine la acción a tomar en cada estado del entorno de Markov.

2. Se evalúa la política actual, es decir, se calcula el valor esperado de la recompensa total acumulada al seguir esa política a lo largo del tiempo.

3. Se mejora la política actual, es decir, se actualizan las acciones que se toman en cada estado del entorno de Markov para maximizar el valor esperado de la recompensa total acumulada.

4. Se repiten los pasos 2 y 3 hasta que la política converja a una política óptima.

Podemos considerar que todo el trabajo en control óptimo también es, en cierto sentido, trabajo en aprendizaje por refuerzo.

Si definimos un **método de RL** como cualquier forma efectiva de resolver problemas de RL, está claro que estos problemas están estrechamente relacionados con problemas de control óptimo, en particular problemas de control óptimo estocástico como los formulados como MDPs (próximamente ahondaremos en los métodos de aprendizaje por refuerzo más famosos).

En consecuencia, debemos considerar que los métodos de solución de control óptimo, al igual que la programación dinámica, también son métodos de RL.

1.5. RL y Búsqueda

A partir de la introducción anterior sobre los métodos de aprendizaje por refuerzo, podemos preguntarnos, "¿no disponemos ya de un surtido suficientemente variado de métodos de búsqueda que se encargan ya de por si de explorar un espacio multidimensional en busca de un punto o puntos que minimicen la función objetivo?"; y es que técnicamente es cierto, ya disponemos de estos, como pueden ser A^*, *Templado Simulado*, o los *Algoritmos Evolutivos*, donde encontramos algoritmos como *PSO*, *Algoritmo Genéticos* o el *Algoritmo Diferencial*.

Como hemos comentado, la rama de la teoría de control estaba en esos momento intentando abordar los problemas dinámicos secuenciales, problema en el que se veía involucrado el concepto de aprendizaje por refuerzo, pero ninguno de estos aportaban la solución esperada, ya que estos son métodos más estáticos y donde se necesita un conocimiento amplio del espacio de búsqueda. En RL siempre había una constante actualización del espacio, por lo que una complejidad considerable sería inabordable.

Es por ello que sí, los métodos propios de RL han tenido su razón de ser y por ello tendrían y tienen un papel clave distinguible de los métodos generales de búsqueda.

1.6. Procesos de Decisión de Markov, el esqueleto de RL

Daremos en este apartado una introducción a los **Procesos de Decisión de Markov** (MDP, por sus siglas en inglés), que proporcionan el marco formal más extendido dentro de la rama de RL[5].

Este modelo, como mencionamos anteriormente, fue presentado por el matemático **Richard Bellman** para modelar y resolver problemas de toma de decisiones secuenciales en entornos estocásticos, como es la **programación dinámica**, ya que abordar esos entornos con modelos deterministas era una opción inabordable debido a la complejidad total del entorno dinámico que era necesario modelar. La variabilidad del entorno da al agente un nivel adicional de incertidumbre sobre el entorno.

1.6.1. Componentes

En primer lugar, vamos a contemplar los dos componentes principales por los que se rige la metodología del RL y los MDP:

1. El **agente**, que se trata del individuo/máquina que se desenvuelve por el entorno para maximizar su recompensa por medio de la toma de decisiones que aprenda a lo largo del tiempo.

2. El **entorno**, el espacio estocástico donde el agente debe amoldarse y adquirir una buena toma de decisiones.

A partir de estos componentes principales, ¿Cómo interactúa el agente con el entorno? ¿y cómo indica el entorno al agente que sus decisiones son correctas? Los componentes secundarios utilizados para conectar agente con el entorno son los siguientes:

- El **estado** (S_t), que se trata de la información útil que el agente recolecta del entorno en un determinado momento t para así determinar las decisiones a tomar.

- La **acción** A_t, que es la forma por la que el agente interactúa con el entorno en un momento t para así pasar a un estado S_{t+1}. Por cada

[5]Este apartado está principalmente influenciado por el capítulo 3 de *Reinforcement Learning, An introduction* [13].

estado el agente optará de varias acciones a tomar.

$$S_{t+1} = A_t(S_t)$$

- La **recompensa** ($R(S_t, A_t)$), que es el valor cuantitativo que recibe el agente del entorno y el cuál indica lo correcto o no que es la acción tomada en un estado t determinado.

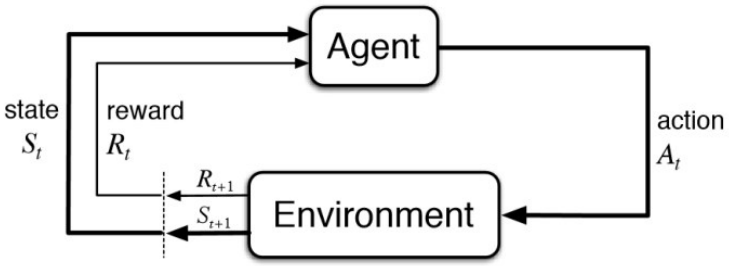

Figura 1.2: Ciclo por el que pasa un proceso de RL

Por tanto, podemos definir un MDP como un modelo matemático que describe un entorno estocástico en el que un agente toma decisiones secuenciales para maximizar su recompensa a lo largo del tiempo.

1.6.2. Funcionamiento

El objetivo del agente será maximizar la recompensa acumulada siendo capaz de tomar las acciones que den mayores recompensas en cada instante t de su dinámica y para los distintos estados del entorno en el que se desenvuelve. Obsérvese que se habla de recompensa acumulada, por lo que no es necesariamente la acción de mayor recompensa puntual.

Entorno

Enfocándonos primeramente en el contexto del problema desde el punto de vista del entorno, en un determinado instante t, la probabilidad de pasar a un nuevo estado s' obteniendo una recompensa r tras haber tomado una acción a en un estado actual s formalmente se puede expresar de la siguiente forma:

$$p(s', r|s, a) = \mathbb{P}(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a)$$

La función p define la dinámica del MDP, es decir, cómo el estado del sistema cambia a lo largo del tiempo en respuesta a las acciones tomadas por el agente y las transiciones de estado resultantes, y dominio viene definido por los posibles estados de transición s' y las posibles recompensas r':

$$\sum_{s' \in S} \sum_{r \in R} p(s', r|s, a) = 1$$

Para que esta ecuación se verifique, es necesario que el estado actual del agente incluya información sobre todos los aspectos de la interacción pasada entre el agente y el entorno que tienen algún tipo de relevancia para el futuro, siendo solo necesaria de esa forma la acción actual como parámetro. Si lo hace, entonces se dice que el estado tiene la **propiedad de Markov**.

A partir de la función de dinámica del MDP de cuatro argumentos, p, vista anteriormente se puede calcular cualquier otro valor sobre el entorno, como las probabilidades de transición de estado, donde esta vez el dominio se ve comprendido solo por las posibles recompensas r:

$$p(s'|s, a) = \mathbb{P}(S_{t+1} = s'|S_t = s, A_t = a) = \textstyle\sum_{r \in R} p(s', r|s, a)$$

También podemos calcular las recompensas esperadas para pares estado-acción como una función de dos argumentos r:

$$r(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$$

O la recompensa esperada dado un estado s, una acción a y un estado de transición s':

$$r(s, a, s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'] = \sum_{r \in R} \frac{p(s', r|s, a)}{p(s'|s, a)}$$

Como podemos ver, al igual que en el caso de las probabilidades dinámicas, el dominio está formado por las distintas recompensas, pero al ser la esperanza lo que buscamos es necesario normalizar la función sobre el estado de transición s'.

Las recompensas por su parte, como indicamos anteriormente, deberán ser ganadas de forma que se maximice la **recompensa acumulada final** o **retorno total**, que denotaremos por G_t:

$$G_t = R_t + R_{t+1} + R_{t+2} + ... + R_T$$

Donde T es el tiempo total comprendido a lo largo del episodio, concepto que aprovecharemos para introducirlo a continuación.

Teniendo en mente la estructura y los componentes del MDP/RL, el agente tomará acciones sobre el entorno dentro de un margen de tiempo T, tiempo comprendido en un episodio, por lo que podemos decir que un episodio es un periodo de tiempo en el que el agente trata de realizar la tarea encomendada en el entorno (a este tipo de tareas las llamaremos **tareas episódicas**). También hay casos en los que T no se cnoce a priori, su valor límite, si existe puede ser incierto, o incluso podría ser una dinámica de tiempo infinito. El concepto de episodio será fundamental cuando hablemos de la fase de entrenamiento del agente.

En el caso de episodios no acotados, es necesario introducir un nuevo concepto, puesto que si no limitamos de alguna forma la recompensa que se adquiera a lo largo del episodio entonces podríamos llegar a adquirir una recompensa acumulada no acotada (y, por lo tanto, sería imposible comparar la acción más adecuada porque la mayoría de ellas darían un valor no acotado en el episodio). A este concepto lo llamaremos **tasa de descuento**, y la penalización que dará a las recompensas dependerá de t, de forma que cuanto mayor sea t mayor será la penalización, o en otras palabras, para estados futuros de valor t alto la penalización será mayor.

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ... + \gamma^k R_T = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

La tasa de descuento es un valor real, $\gamma \in [0, 1]$. Una tasa de descuento alta significa que las recompensas a largo plazo tienen una mayor importancia, mientras que una tasa de descuento baja significa que solo se consideran las recompensas inmediatas. Este concepto será importante recordarlo para cuando veamos la ecuación clave del aprendizaje por refuerzo, la ecuación de Bellman.

Agente

Centrándonos ahora en la perspectiva del agente, es común que al principio no conozca el entorno en el que se encuentra, ya que no ha podido aprender nada de momento, por lo que la toma de decisiones tendrá que ser empleando acciones de forma aleatoria (o con información local), y poco a poco ir adquiriendo la experiencia esperada para la próxima vez repetir o no la acción en ese determinado estado. A la estrategia de decisión de acciones se le denomina norma, o más comúnmente, **política**, y se denota como π.

Más formalmente, una política es una función que mapea estados del entorno en las probabilidades de ser tomadas cada una de las acciones que el agente puede tomar en ese estado ($\pi(a|s)$).

A partir del concepto de política, se puede definir la **función de valor** v ($v_\pi(s)$), que calcula la recompensa acumulada esperada que se obtiene al seguir una política determinada en un estado determinado:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

De forma similar, podemos definir una variación, la función de valor q, que se diferencia de la función v en que contempla además del estado actual la acción tomada para indicar la recompensa acumulada esperada:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

La Figura 1.3 representa visualmente lo tratado hasta ahora desde el punto de vista del agente, donde las flechas de las funciones v y q indican el punto del diagrama de donde sacan el valor, viéndose que la función q, como habíamos indicado, al disponer de a como parámetro no necesita contemplar las demás acciones desde el estado actual, cosa que sí debe hacer la función v:

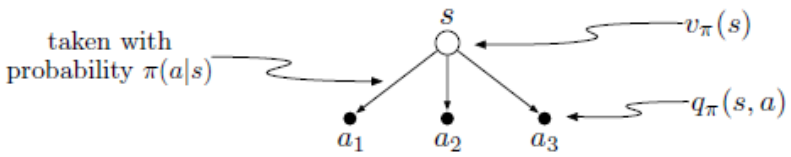

Figura 1.3: Representación de la función $v_\pi(s)$ y $q_\pi(s, a)$ en el diagrama MDP

El retorno total G_t se puede descomponer en dos componentes:

1. La recompensa inmediata (R_{t+1}) que se recibe en el siguiente instante de tiempo $t + 1$.

2. El retorno descontado en el siguiente instante de tiempo (γG_{t+1}), o lo que es lo mismo, la suma de todas las recompensas futuras.

Lo que nos da una expresión distinta de la función v:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s]$$

Si expandimos la esperanza condicional ($\mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s]$) en términos de las probabilidades de transición ($p(s', r|s, a)$) y la política ($\pi(a|s)$), nos quedaría:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_{r \in R} p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_t + 1 = s']]$$

Si adicionalmente tenemos en cuenta que la esperanza futura puede ser ser representada recursivamente como el valor de la función v:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

Teniendo en cuenta de donde venía esta descomposición final nos queda la **ecuación de Bellman** para una función v bajo una política π:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

Esta ecuación expresa la relación recursiva entre el valor del estado actual y el valor futuro en futuros estados.

La expansión y descomposiciones realizadas para obtener la ecuación de Bellman son de igual modo aplicables a la función q, pero en este caso no contemplamos las posibles acciones actuales, sino que contemplaremos para los futuros estados las posibles acciones a' a tomar, es decir:

$$q_\pi(s, a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')]$$

En la Figura 1.4 se puede ver mejor el papel de $v_\pi(s')$ como el representante de los valores futuros debido a que la actualización se realiza de forma recursiva, es decir, llegado a lo más abajo del diagrama, en ese momento se dispondrá a ir marcha atrás para dar los valores recursivamente a los respectivos estados. r_i serían los valores inmediatos, y los que comprende el estado actual sin necesidad de comprender los valores futuros:

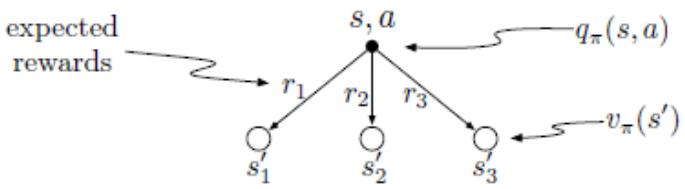

Figura 1.4: Representación de la función $q_\pi(s,a)$ y $v_\pi(s')$ en el diagrama MDP

Como podemos intuir, cuanto mayor sea el valor de la función v, mejor será la política que se ha seguido. Por tanto, en un estado determinado existirá una política óptima (π_*) que proporciona el valor v óptimo (v_*),tal que:

$$v_*(s) = \max_\pi v_\pi(s)$$

Igualmente aplicable a la función q, la cuál sería q_* y sería definida tal que:

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

Si recordamos la ecuación de Bellman para las funciones v y q, como queremos optar por la toma de decisiones más óptima, es decir, v_* y q_*, podemos sacar la ecuación óptima de Bellman en base a estas dos:

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]$$

Donde, esta vez, en vez de contemplarse las distintas acciones se contemplará la acción óptima.

Partiendo de esa premisa, para el caso de la ecuación óptima de Bellman para la función q sería igual solo que en vez de contemplar la mejor acción actual se contemplará la futura acción temporal a', ya que la función q dispone de la actual como parámetro (por algo se utiliza en la ecuación v de forma recursiva), quedando de la siguiente forma:

$$q_*(s,a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')]$$

En la Figura 1.5 se representa la elección de rama/arista que se quiere tomar de forma óptima en los dos casos, en el caso de v_* las acciones actuales y en el caso de q_* las acciones futuras.

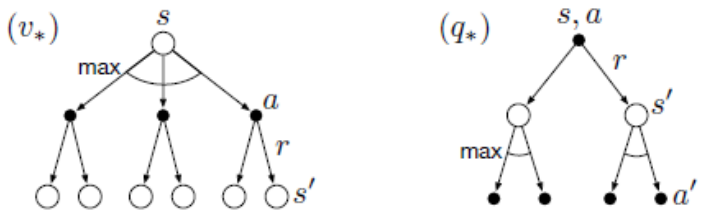

Figura 1.5: Diagrama MDP de las funciones v_* y q_*

El hecho de que la función q permita seleccionar la acción óptima de manera más directa, y sin necesidad de realizar búsquedas a un paso adelante, es el motivo por el que es más utilizado en algoritmos más modernos como **Q-Learning**, que veremos más adelante.

1.6.3. Métodos de RL

[**?**] Para terminar con el apartado de MDP ahondaremos en los **métodos de aprendizaje por refuerzo**, un concepto que vimos ya en los orígenes cuya función principal es especificar cómo se cambia la política del agente como resultado de su experiencia.

En este sentido, se pueden clasificar por las siguientes categorías:

- La forma de representar y actualizar los valores o funciones, donde tenemos:
  - **Tabulares**, que se pueden estructurar mediante una tabla estado-acción (Q-tabla) o estado-Valor (V-tabla) y son capaces de resolver el problema de forma exacta llegando a una política pi_* o valor q_*/v_* óptimo.
  - **Aproximados**, que mediante una función aproximadora son capaces de devolver una política/valores q/v buenos pero sin tener por qué ser los óptimos.
- El uso o no de un modelo explícito del entorno:
  - **Model-free**, que se centra en aprender una política pi_* o valor q_*/v_* directamente de la interacción con el entorno, sin requerir un modelo explícito del entorno.
  - **Model-based**, que se basan en la construcción y utilización de un modelo explícito del entorno para planificar y tomar decisio-

nes óptimas en base a simulaciones de posibles estados y acciones futuras.

- El enfoque algorítmico y conceptual utilizado para resolver los problemas:
  - **Value-based**, que se enfoca en aprender y estimar los valores q o v.
  - **Policy-based**, que se enfoca en aprender directamente la política óptima.
- Cómo se utiliza la experiencia recolectada para mejorar la política de toma de decisiones:
  - **On-policy**, donde el agente aprende y mejora su política de toma de decisiones utilizando la experiencia recolectada siguiendo directamente la política actual sin cambiar la política.
  - **Off-policy**, donde el agente aprende y mejora una política objetivo utilizando la experiencia recolectada siguiendo una política de comportamiento diferente, es decir, cambiando la política.

Los métodos se pueden diferenciar mediante la combinación de características de cada categoría.

Centrándonos en los métodos tabulares, independiente de las demás características, podemos contemplar principalmente cuatro métodos:

- **Programación Dinámica (DP)**.
- **Aprendizaje por diferencias temporales (LD)**.
- **Búsqueda exhaustiva**.
- **Monte Carlo**.

Donde los DP y búsqueda exhaustiva necesitan un conocimiento completo de los MDP (conocimiento de las probabilidades de transición), mientras que los métodos de Monte Carlo y TD van explorando el espacio mediante la prueba y error.

Desde el punto de vista de manipulación de información en el árbol de estados, Monte Carlo y búsqueda exhaustiva operan a mayor profundidad en comparación con TD y DP, los cuales operan con información parcial o estimada.

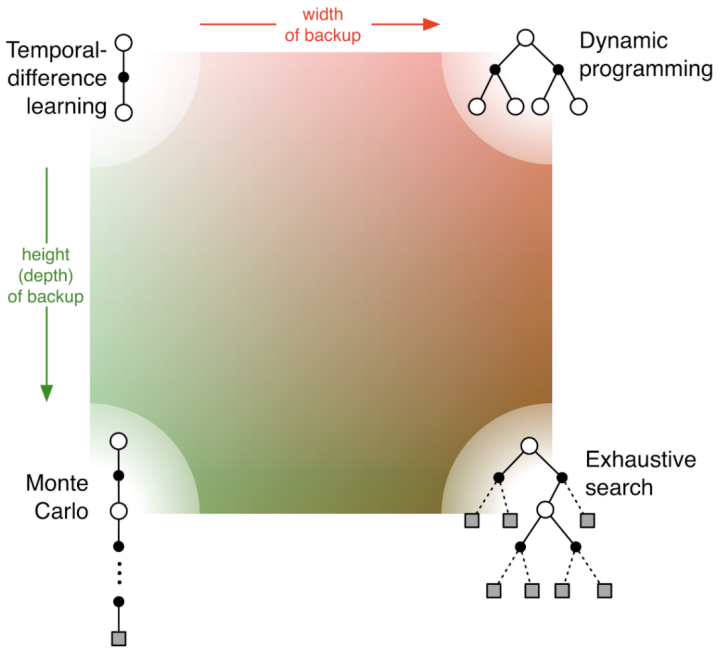

Figura 1.6: Diferenciación entre los métodos tabulares

Para el resto de la memoria nos vamos a centrar en DP y TD, en los cuales profundizaremos en los próximos apartados.

Como ya sabemos, el objetivo del agente es encontrar una política, valor v, o valor q, óptima, pero esto es difícil y costoso computacionalmente. Incluso si se tiene un modelo completo y preciso del entorno, no siempre es posible calcular la política óptima. Además, la cantidad de memoria disponible también es una restricción importante, por lo que en muchos casos se deben utilizar aproximaciones para las funciones de valor y las políticas (métodos aproximados).

Aunque esto significa que no se puede alcanzar el caso óptimo absoluto, se pueden lograr aproximaciones útiles ya que, además, en muchos casos hay estados que se enfrentan con una probabilidad tan baja que seleccionar acciones subóptimas para ellos tiene poco impacto en la cantidad de recompensa que recibe el agente.

El hecho de que haya opciones con casos óptimos pese a no ser los absolutos será algo que podamos ver en el siguiente apartado.

1.7. Programación dinámica (DP)

[13, Capítulo 4] [**?**] Tras ver cómo se puede fundamentar el aprendizaje por refuerzo sobre los MDP, con la importancia que adquiere así la ecuación de Bellman, que es la ecuación por la que se rigen los valores de las funciones v (dado un estado) y q (dado un estado y una acción) en una determinada política π; ahora nos introduciremos al funcionamiento detallado de todo el sistema y en diversos métodos de cómo se computa de modo efectivo todo ello. Comenzaremos por uno de los métodos fundamentales que ya hemos mencionado en el apartado anterior, y que constituye uno de los más importantes aplicados a RL, la **programación dinámica** (DP, por sus siglas en inglés)[6].

DP es una técnica de optimización que se basa en la idea de dividir un problema en subproblemas más pequeños para resolverlos de forma independiente, para así combinar luego las soluciones de los subproblemas en una solución global óptima.

El motivo por el cuál se divide el problema a abordar en distintos subproblemas es para reducir la complejidad computacional del problema original en términos de tiempo y espacio, que forma que se permita una resolución eficiente del mismo.

La estrategia de dividir un problema en subproblemas se va a aplicar a la tarea de encontrar la política óptima en el MDP (π_*), ya que es la parte clave para optimizar la ecuación de Bellman y acercarnos a la ecuación óptima, tanto en la función v como en la función q.

Para ello, primero consideramos cómo calcular la función de valor de estado v_π para una política arbitraria π, lo que se conoce como **evaluación de política**, o también **problema de predicción**:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$$

Para poder resolver la ecuación de Bellman es necesario conocer completamente la dinámica del entorno, es decir, la probabilidad de transición de un estado a otro estado para cada acción tomada por el agente. En caso contrario, se pueden utilizar métodos de aproximación para obtener una solución aproximada de la función de valor óptima o de la política óptima.

En principio, la solución que indica la ecuación es directa, pero al mismo

[6] Este apartado está principalmente influenciado por el capítulo 4 de *Reinforcement Learning, An introduction* [13].

tiempo es muy tediosa desde el punto de vista computacional. Es por eso que se utilizan métodos de solución iterativos para aproximar los valores de la función de valor de estado, dando lugar a la primera forma de evaluarla y modificarla: la **política de evaluación iterativa**.

1.7.1. Política de evaluación iterativa

Para explicar el siguiente algoritmo usaremos un ejemplo pequeño pero con el que se puede entender mejor lo que se pretende hacer y, de paso, todo lo explicado en este apartado y al final del apartado de Markov. Supongamos el siguiente problema:

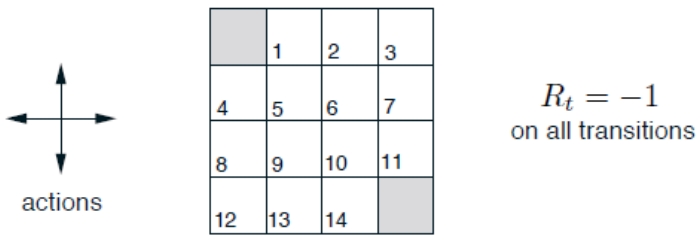

Figura 1.7: Problema de evaluación iterativa

Donde:

- Las celdas son los estados de nuestro problema: 14 estados intermedios y 2 terminales (que una vez llegados a ellos el episodio terminaría), dando un total de 16 estados ($s \in S^+$).
- Las acciones son los movimientos que puede hacer el agente en el entorno: los 4 movimientos básicos (arriba, abajo, izquierda, derecha).
- La recompensa por cada transición (R_t) es de -1: si el agente se encuentra en una celda s y debe llegar a una de las celdas terminales la política más óptima sería la que llevase al agente a pasar por el mínimo número de celdas posibles.

Comenzamos con una estimación inicial de la función de valor v_0, donde nos encontramos en la iteración $k = 0$, que será aleatoria por no haber podido el agente aprender nada del entorno (excepto en el estado terminal, donde el valor será 0).

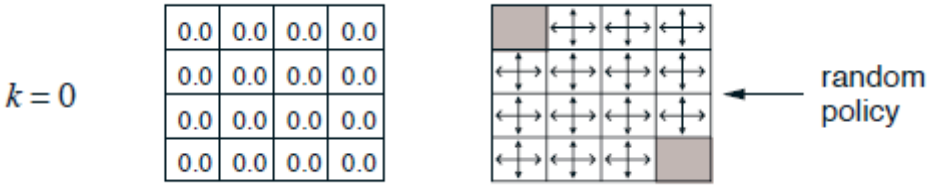

Figura 1.8: Problema en la iteración $k = 0$

La tabla de la derecha indica la política aleatoria π que hemos impuesto inicialmente, donde hay un 25 % de probabilidades de usar cada una de las acciones en un estado s cualquiera, es decir, que sin criterio alguno el agente se moverá a una celda vecina cualquiera.

Por otro lado, la tabla de la izquierda pondera en cada celda s el valor $v_\pi(s)$ que le corresponde, osea, como ya sabemos, la recompensa acumulada esperada si nos situamos en esa celda, siendo inicialmente todo 0 porque no hemos aprendido/evaluado aún (como ya hemos dicho, los terminales serán 0, porque el agente ya se encontrará en su objetivo final).

Si realizamos una nueva iteración sobre la función v (v_{k+1}), partiendo de v_1, tendremos lo siguiente:

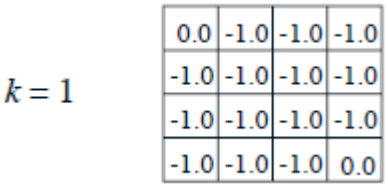

Figura 1.9: Problema en la iteración $k = 1$

Manteniendo la misma política (es decir, la misma tabla de la derecha en la Figura 1.8), podemos ver ahora en la de evaluación v que todos los estados intermedios tienen valor -1, ya que cada una ha realizado una sola acción y, por tanto, en promedio de momento se calcula que en un paso son capaces de llegar a la meta. Los valores terminales, como hemos indicado, se mantienen a 0.

Si iteramos una vez más (v_2), nuestra tabla de evaluación v nos queda de la siguiente forma:

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

Figura 1.10: Problema en la iteración $k = 2$

En este caso, podemos ver cómo las celdas vecinas de las terminales han sido menos penalizadas que las demás, y esto se debe a que en las no vecinas se estima que en dos pasos (por las dos iteraciones que llevamos) se llega a uno de los dos terminales. Como en las vecinas cabe la posibilidad entre las cuatro acciones posibles de poder llegar a la terminal con una ellas, entonces la penalización media se ve reducida, ya que una de cada cuatro acciones llegaremos al estado terminal.

Realicemos una tercera iteración (v_3), pero además a partir de aquí vamos a realizar también una iteración más lejana (v_{10}), además de una iteración que tienda a infinito (v_∞):

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Figura 1.11: Problema en la iteración $k = 3$, $k = 10$ y $k = \infty$

Como podemos ver, en v_3 están más definidas las celdas que necesitarían

un esfuerzo mayor para llegar a las celdas terminales, donde el punto mínimo sería -3, ya que la diagonal secundaria de la tabla es la que alberga las celdas más lejanas tanto de una celda terminal como de la otra. En v_{10} podemos ver la misma tendencia, pero más acentuada: cuanto más lejos se encuentre la celda s de las terminales, mayor será su penalización.

Podemos decir que con este número de iteraciones ya hemos podido evaluar al completo el entorno, y efectivamente es así, ya que si nos fijamos en la tabla de v_∞ la conclusión es la misma solo que con penalizaciones acumuladas mayores por haber iterado más veces. A este punto podemos decir que la evaluación ha sido realizada y que el valor v_π está ya determinado para π.

La actualización se realiza utilizando la siguiente fórmula, que es la ecuación de Bellman pero para la siguiente evaluación $k+1$, necesitando iterar sobre lo conocido en la iteración pasada (k):

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')]$$

Este proceso se repite hasta que la función de valor v_k converge a v_π o, desde un punto de vista práctico, hasta que el cambio entre dos iteraciones consecutivas es menor que un cierto valor de umbral que tendremos que indicar.

El pseudocódigo del algoritmo de evaluación iterativa es el siguiente:

Entrada: π (la entrada a ser evaluada).

Parámetros del algoritmo: Umbral ($\theta : \theta > 0$).

1. Inicializar $v(s)$ arbitrariamente, para el estado s y el valor estimado v (terminal) a 0.

2. Mientras $\triangle < \theta$ (Siendo $\triangle$ el nuevo valor asignado a $v(s)$):

   - $\triangle \leftarrow 0$
   - Para cada estado $s \in S^+$:
     - $V \leftarrow v(s)$
     - $v(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$
     - $\triangle \leftarrow \text{máx}(\triangle, |V - v(s)|)$

De esta forma, para concluir, podremos actualizar el valor de estado de la función v (o q con sus pequeñas diferencias) bajo la política a evaluar.

1.7.2. Mejora de la política (Policy Iteration)

Recordando el objetivo de DP en el contexto del agente, debemos encontrar la política óptima π_* (o una aproximación). El motivo por el que hemos comenzado explicando el algoritmo iterativo para evaluar y conocer v_π es porque nos va a servir de ayuda, y es que es posible que tengamos una política elegida que se considere buena, pero, ¿es óptima?, ¿cómo podemos comparar cada política para actualizar la política óptima?, y es lo que a continuación abordaremos, donde v_π determinará qué tan buena es la política utilizada actualmente como hemos visto hasta ahora.

Empecemos mencionando el **Teorema de Mejora de Política** (*Policy Improvement Theorem*) el cuál establece que, para cualquier política, siempre y cuando no se haya alcanzado aún la política óptima, existe una política que es al menos tan buena como esta y que puede ser mejor en al menos un estado. Esto significa que siempre es posible mejorar una política si no se ha alcanzado todavía la política óptima al menos en un estado mediante la selección de una acción diferente en este. De esta forma, si se aplica repetidamente este proceso de mejora de la política, eventualmente nos aproximaremos más a la política óptima. El algoritmo que a continuación trataremos y que se encargará de ejecutar la mejora de política junto a la evaluación de política anteriormente descrita es el algoritmo de **Policy Iteracion**.

Si volvemos a nuestro ejemplo de la tabla:

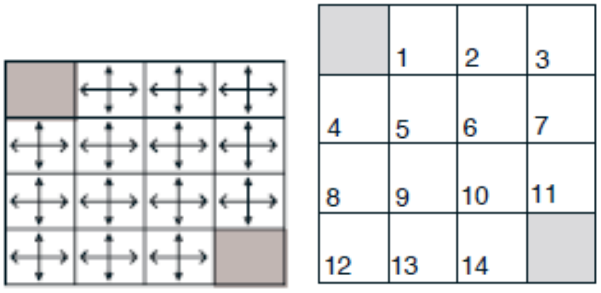

Figura 1.12: Problema aplicado a Policy Iteration

En este caso la política era arbitraria, por lo que las penalizaciones tras k iteraciones en las distintas celdas podrían reducirse si el agente usase algún mecanismo de optimización y, por ejemplo, se moviese siempre a la izquierda cada vez que nos encontrásemos en la celda 1 o, por ejemplo, se moviese abajo siempre que estemos en la celda 11.

La política aspirante a sustituir nuestra política actual π, π', tomará una acción distinta de la que tomaría de forma predeterminada π.

La comparación que vamos a realizar será el valor v de la política π con respecto el valor v de π', siendo este el valor q bajo la política π optando por la acción que indique π' $(q_\pi(s, \pi'(s)))$:

$$q_\pi(s, \pi'(s)) = v_{\pi'}(s) \geq v_\pi(s)$$

Si la condición indicada se cumple, entonces se mejora la política empleando esa acción y, por tanto, podemos considerarla por el momento como la nueva política mejorada, hasta que una nueva política π'' sea comparada con esta para ver si la supera.

A partir de aquí, podemos utilizar empleando el método greedy, que no es más que una mejora que hace que, en vez de que la política se compare con las distintas políticas aspirantes, directamente se sustituya por la política que seleccione la acción que emplee la acción que maximice el valor v, de forma que, en este caso, π' será la política óptima del estado s, tal que:

$$\pi'(s) = \arg\max_a q_\pi(s, a)$$

De este modo, el valor v óptimo se describiría de la siguiente forma:

$$v_{\pi'}(s) = \text{máx}\, q_\pi(s, a)$$

Volviendo con nuestro ejemplo, vamos a proponer la siguiente política π':

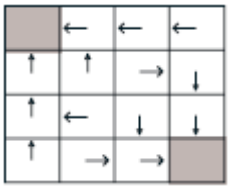

Figura 1.13: Política propuesta π'

Si iteramos infinitas veces de nuevo nuestro problema pero partiendo de esta nueva política, nuestra evaluación quedaría de la siguiente forma:

$k = \infty$

0.0	-1.0	-2.0	-3.0
-1.0	-2.0	-3.0	-2.0
-2.0	-3.0	-2.0	-1.0
-3.0	-2.0	-1.0	0.0

Figura 1.14: Convergencia de la evaluación cuando $k \to \infty$

Como vemos, si nos regimos por las acciones de mayor ganancia (o menor pérdida, mejor dicho en este caso), representadas por esta nueva política, la estimación ahora es más lógica y óptima, de tal forma que, a simple vista, podemos representar el valor de cada celda por el número mínimo de pasos que hacen falta para llegar a la celda terminal más cercana.

En este caso, solo ha hecho falta realizar una vez el proceso de evaluación y mejora, de forma que $\pi' = \pi_*$ y, análogamente, $v_{\pi'} = v_*$, ya que el ejemplo es muy sencillo (es más, no es la única política óptima de este problema). Normalmente, en casos más complejos, habrá que repetir la evaluación y comparación para conseguir aproximarnos más.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \cdots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_*$$

Figura 1.15: Transiciones de Policy Iteration

El pseudocódigo del algoritmo de Policy Iteration, con la evaluación iterativa junto al de mejora de política incluidos dentro, quedaría de la siguiente forma:

1. Inicializar $v(s)$ arbitrariamente, para el estado s y el valor estimado v (terminal) a 0, y $\pi(s) \in A(s)$, arbitrario también.

2. Política de Evaluación:

   - Mientras $\triangle < \theta$ (con $\triangle$ el nuevo valor asignado a $v(s)$):
     - $\triangle \leftarrow 0$
     - Para cada estado $s \in S^+$:
       - $V \leftarrow v(s)$
       - $v(s) \leftarrow \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$
       - $\triangle \leftarrow \text{máx}(\triangle, |V - v(s)|)$

3. Política de mejora:

   - $policy\_stable \leftarrow true$
   - Para cada $s \in S^+$:
     - $old\_action \leftarrow \pi(s)$
     - $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$
     - Si $old\_action \neq \pi(s)$, entonces $policy\_stable \leftarrow false$
   - Si $policy\_stable$, entonces para y devuelve $v \approx v_*$ y $\pi \approx \pi_*$. Si no, vuelve al paso 2.

1.7.3. Value Iteration

Aunque el algoritmo de Policy Iteration tenga sus ventajas y prometa ser un algoritmo eficiente para calcular, o por lo menos aproximar, una política óptima, tiene una gran desventaja, y es que computacionalmente, si el problema es más complejo, puede llegar a ser incluso ineficiente el algoritmo debido a que necesitamos emplear dos bucles, uno para la evaluación, y otro para la comparación/mejora. A esto se le suma el hecho de que, a veces, no es necesario repetir las iteraciones tantas veces solo por aproximar lo máximo posible la evaluación, por ello que en el caso de $k = 3$, $k = 10$, y $k \to \infty$ en el ejemplo de la tabla cualquiera de las tres pudiese haber sido una buena evaluación óptima.

Value Iteration es una mejora del algoritmo de Policy Iteration que se diferencia en realizar la evaluación y mejora en una sola iteración, de forma que, cuando converja la evaluación, a la hora de comparar y mejorar la política también se tendrá en cuenta, y por tanto la política también convergerá.

Volvamos al ejemplo de la tabla, el caso de $k = 0$ (Figura 1.12).

Con $k = 1$ empleando este algoritmo quedaría de la siguiente forma:

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

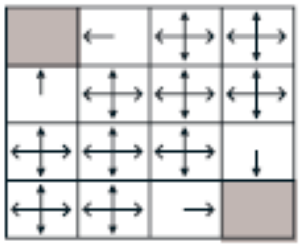

Figura 1.16: Problema aplicado a Value Iteration en la iteración $k = 1$

Como podemos ver, la política ha sido mejorada en la primera iteración de la evaluación, sirviendo de apoyo para la siguiente iteración de la evaluación para la siguiente aproximación de la política.

$k = 2$

0.0	-1.0	-2.0	-2.0
-1.0	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.0
-2.0	-2.0	-1.0	0.0

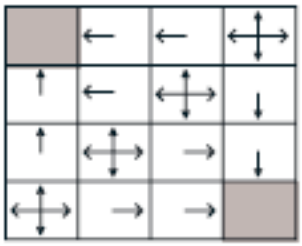

Figura 1.17: Problema aplicado a Value Iteration en la iteración $k = 2$

Aquí podemos ya observar cómo las celdas vecinas se quedan con su valor al ser óptimas ya, y ahora además tenemos una política que ha fijado las celdas inferiores de las vecinas, quedando como las más lejanas, las correspondientes a la diagonal secundaria.

Con la tercera debería bastarnos para completar la política óptima:

$k = 3$

0.0	-1.0	-2.0	-3.0
-1.0	-2.0	-3.0	-2.0
-2.0	-3.0	-2.0	-1.0
-3.0	-2.0	-1.0	0.0

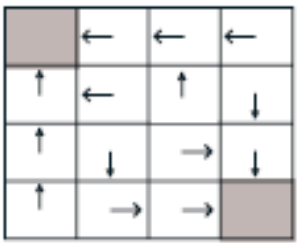

Figura 1.18: Problema aplicado a Value Iteration en la iteración $k = 3$

Con solo 3 iteraciones hemos conseguido llegar a los valores óptimos de v, que no puede ser mejorada con más iteraciones (coincide con el valor óptimo que se obtenía con Policy Iteration para $k \to \infty$).

El pseudocódigo del algoritmo es el siguiente:

1. Inicializar $v(s)$ arbitrariamente, para el estado s y el valor estimado V (terminal) a 0, y el umbral θ>0.

2. Bucle:

   - $\triangle \leftarrow 0$
   - Para cada estado $s \in S^+$
     - $v \leftarrow v(s)$
     - $v(s) \leftarrow \text{máx}_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$
     - $\triangle \leftarrow \text{máx}(\triangle, |v - V(s)|)$

   Hasta que $\triangle < \theta$

3. Como salida devuelve una política $\pi \approx \pi_*$, tal que:

$$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$$

1.7.4. Programación dinámica asíncrona

La programación dinámica, a diferencia de otros métodos de aprendizaje aplicados al RL necesita conocer todos los estados para poder actualizarlos en cada iteración. Esto, al igual que el problema de los bucles de Policy Iteration, implica que para problemas complejos con muchos estados provoque que el algoritmo sea ineficiente una vez más, sufriendo de lo que Bellman llamó "la maldición de la dimensionalidad", lo que significa que sus requisitos computacionales crecen exponencialmente con el número de variables de estado (por ejemplo, el juego del backgammon tiene del orden de 10^{20} estados distintos, un número que podría provocar que una simple iteración pueda durar incluso mil años).

La **programación asíncrona** se creó con el fin de poder abordar este tipo de problemas, permitiendo así actualizar los estados necesarios sin tener que recorrerlos todos por cada iteración. No obstante, es interesante que contemplemos un método hermano de este con una metodología que esquiva sus desventajas, un método que ya comentamos que no necesitaba conocer el MDP y que, además, no es tan agresivo como Mote Carlo: el **aprendizaje por diferencias temporales**.

1.8. Aprendizaje por diferencias temporales (TD)

Si uno tuviera que identificar una idea central y novedosa en el aprendizaje por refuerzo, sin duda esta sería el **aprendizaje por diferencias temporales** (TD, por sus siglas en inglés)[7].

TD es una combinación de ideas del método de Monte Carlo y DP, adquiriendo las ventajas de no necesitar conocer el MDP para poder abordar los problemas como en Monte Carlo, y al mismo tiempo pudiendo tratar el problema mediante subproblemas como en DP.

Al igual que cuando explicamos DP, nos vamos a centrar en la evaluación o predicción de políticas, es decir, el problema de estimar la función de valor v para una política dada. Las diferencias entre los métodos se encuentra, básicamente, en sus enfoques para el problema de predicción.

1.8.1. Predicción TD

Como ya sabemos, TD emplea la experiencia que adquiere durante los episodios, pudiendo así una vez entrenado ser capaz de predecir las transiciones más óptimas del problema.

Tras adquirir algo de experiencia siguiendo una política π, TD actualiza su estimación v de v_π para los estados no terminales S_t que se ven involucrados en esa experiencia. Un método simple para todas las visitas, adecuado para entornos no estacionarios, y que es una actualización del que se emplea en Monte Carlo, es:

$$v(S_t) \leftarrow v(S_t) + \alpha[R_{t+1} + \gamma v(S_{t+1}) - v(S_t)]$$

Que es una adaptación de la ecuación de Bellman para TD conocida como TD(0), o TD($\lambda : \lambda = 0$), donde $\lambda \in [0, 1]$ es la **elegibilidad de traza**, una medida de cuántos pasos en el tiempo se tienen en cuenta al realizar una actualización. Un valor $\lambda = 0$ significa que solo se tiene en cuenta una actualización de un solo paso, mientras que un valor $\lambda = 1$ conlleva tener en cuenta todas las actualizaciones de múltiples pasos hasta el final del episodio (lo que nos lleva a un método más cercano y parecido a Monte Carlo). Para esta explicación nos limitaremos a TD(0).

[7] Este apartado está principalmente influenciado por el capítulo 6 de *Reinforcement Learning, An introduction* [13].

Volviendo a la ecuación de Bellman aplicada a TD, podemos ver que al valor actual v se le añadirá la experiencia $(R_{t+1} + \gamma v(S_{t+1}))$, que es ajustable mediante un nuevo parámetro α, denominado la **tasa de aprendizaje**, que indica la velocidad a la que aprende, o lo que es lo mismo, el peso que se le da a la información nueva en relación con la información antigua, un valor $\alpha \sim 1$ puede dar lugar a un aprendizaje rápido y a una mayor exploración, pero quizás propenso al sobreajuste y no converger en una solución óptima, mientras que un valor $\alpha \sim 0$ puede ser menos propenso al sobreajuste pero bastante lento y con poca exploración del entorno.

Por otro lado, podemos ver de nuevo la tasa de descuento (γ), que indica la importancia de la recompensa futura $(v(S_{t+1}))$ como ya comentamos en el apartado de MDP. Un valor $\gamma \sim 1$ indica un mayor peso a la recompensa futura, aprendiendo más rápido a largo plazo pero pudiendo ser más inestable y no converger a una solución óptima, mientras que un valor $\gamma \sim 0$ indica un menor peso a la recompensa futura, teniendo un aprendizaje más estable pero más lento y que quizás se aleja de la solución óptima en lugar de acercarse a ella.

El pseudo código de del algoritmo TD(0) es el siguiente:

Entrada: π a ser evaluada.

Parámetros: $\alpha \in (0, 1]$.

1. Inicializar $v(s)$, arbitrariamente (menos para $v(terminal) = 0$).
2. Bucle para cada episodio:
   - Inicializar s.
   - Bucle para cada paso del episodio:
     - $a \leftarrow$ Acción dada para el estado s.
     - Tomada a, observar r y s'.
     - $v(s) \leftarrow v(s) + \alpha[r + \gamma v(s') - v(s)]$
     - $s \leftarrow s'$

     Hasta que s sea terminal.

   Fin del episodio.

Figura 1.19: Diagrama MDP de TD(0)

Como podemos ver en la Figura 1.19, el algoritmo evalúa en un episodio una acción y su estado de transición sin contemplar otras posibles acciones a tomar y actualizando el estado actual en base a estos dos factores, que es donde se diferencia de DP, que contemplaba cada una de las acciones y optaba por la política que tuviese una toma de acciones mejor.

1.8.2. TD vs DP

Como hemos visto a lo largo del apartado anterior, TD posee características que son capaces de superar a DP:

- La no necesidad del conocimiento completo del MDP.
- Las actualizaciones incrementales, lo que permite un aprendizaje más eficiente y en tiempo real,
- Mayor flexibilidad para los entornos cambiados.

Asimismo, también presenta una serie de desventajas considerables en comparación con DP, entre estas:

- La sensibilidad a la iniciación, pudiendo conllevar a una convergencia más lenta o a estimaciones inexactas.
- La varianza, siendo mayor TD por tratar muestras individuales.

Es posible que nos preguntemos lo sólido que puede ser TD en comparación con DP, ya que el desconocimiento del entorno, sumado a las desventajas mencionadas, pueden decir indirectamente que no. Afortunadamente, sí que podemos garantizar la convergencia al igual que con DP, ya que para cualquier política fija π, se ha demostrado que TD(0) converge a v_π, en promedio, para un parámetro de tamaño de paso constante si es suficientemente

pequeño, y con probabilidad 1 si el parámetro de tamaño de paso disminuye de acuerdo con las condiciones habituales de aproximación estocástica:

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \text{ and } \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty$$

Estas condiciones indican que la suma de las tasas de aprendizaje a lo largo del tiempo debe ser infinita, lo que asegura que el algoritmo explore suficientemente el espacio de estados, y que la suma de los cuadrados de las tasas de aprendizaje sea finita, lo que garantiza una convergencia estable.

Además, las nuevas posibilidades que da el enfoque de TD ha llegado crear dos de los algoritmos más populares del área: el **algoritmo Sarsa** y el **algoritmo Q-Learning**.

1.8.3. Control TD

Hasta ahora, hemos visto dentro de TD el método TD(0), un algoritmo de predicción *Value-based*, que busca aproximar la función v/q óptima. A continuación, vamos a tornar el enfoque a su contraparte, es decir, *Policy-based*, que busca aproximar la política óptima. De los algoritmos más populares que han salido de TD que abordan este enfoque están **Sarsa**, que lo hace con una orientación *On-Policy* (manteniendo la política tras la experiencia), y **Q-Learning**, con una orientación *Off-Policy* (cambiando la política tras la experiencia).

Sarsa

El algoritmo Sarsa, siglas provenientes de los elementos que se emplean para calcular los valores q $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ es una derivación que emplea la función q para evaluar y ajustar la política π, diferenciándose del algoritmo TD(0) en no usar la función v. Por ende, el diagrama que podría resumirlo sería el siguiente:

Figura 1.20: Diagrama MDP de Sarsa

donde el proceso de iteración equivalente a la ecuación de Bellman sería:

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha[R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)]$$

Como en todos los métodos On-Policy, continuamente estimamos q_π para la política π, y al mismo tiempo cambiamos π de forma avariciosa (greedy) con respecto a q_π. Las propiedades de convergencia del algoritmo Sarsa dependen de la naturaleza de la dependencia de la política en q. Por ejemplo, se podrían usar las siguientes políticas:

- ϵ-greedy: donde se elige la acción óptima con una probabilidad de $(1-\epsilon)$, lo que representa la explotación. Sin embargo, se selecciona una acción aleatoria con una probabilidad de ϵ.
- ϵ-soft: donde también se elige la acción óptima con una probabilidad de $(1 - \epsilon)$, pero la exploración se realiza distribuyendo la probabilidad ϵ entre todas las acciones no óptimas de manera equitativa. Esto permite una exploración más equilibrada y controlada, donde las acciones no óptimas tienen la oportunidad de ser elegidas, aunque con una probabilidad más baja que las acciones óptimas.

Estas políticas pueden ser aplicadas en los algoritmos vistos y en el de Q-learning.

Sarsa converge con probabilidad 1 hacia una política óptima bajo las condiciones habituales sobre la tasa de aprendizaje (las condiciones mencionadas anteriormente), osea, siempre y cuando se visiten un número infinito de veces todos los pares estado-acción y la política converja en el límite hacia la política greedy.

Pseudocódigo del algoritmo Sarsa para estimar $q \simeq q_*$:

Entradas: s, a, r, s', y a'

Parámetros: $\alpha \in (0, 1], \epsilon > 0$.

1. Inicializar $q(s, a)$ para todo $s \in S^+$ y $a \in A(s)$, arbitrariamente (menos para $q(terminal, \cdot) = 0$).

2. Bucle para cada episodio:

   - Inicializar s.
   - Elegir una acción a de s usando una política derivada de q (p.ej ϵ-greedy).
   - Bucle para cada paso del episodio:
     - Tomada a, observar r y s'.
     - $q(s, a) \leftarrow q(s, a) + \alpha[r + \gamma q(s', a') - q(s, a)]$
     - $s \leftarrow s'$; $a \leftarrow a'$

     Hasta que s sea terminal.

   Fin del episodio

Q-Learning

Es un algoritmo más antiguo que Sarsa, y de los algoritmos más conocidos del RL debido a su simplicidad y claridad conceptual, y es que si lo comparamos con DP veremos que su comportamiento puede ser hasta más comprensible e intuitivo, ya que proporciona una base sólida para comprender los fundamentos del aprendizaje por refuerzo, incluyendo los conceptos básicos de MDP como los estados, acciones, recompensas y la función q.

Se gestiona mediante una Q-tabla con los estados como filas, y las acciones como columnas (ver Figura 1.21).

Acción / Estado	A_1	A_2	...	A_n
S_1	$q_{1,1}$	$q_{1,2}$	...	$q_{1,n}$
S_2	$q_{2,1}$	$q_{2,2}$	...	$q_{2,n}$
⋮	⋮	⋮	⋱	⋮
S_m	$q_{m,1}$	$q_{m,2}$	...	$q_{m,n}$

Figura 1.21: Q-tabla

Para saber cómo ir completando la tabla, como hemos hecho hasta ahora dentro del enfoque TD, haremos uso de la ecuación de Bellman:

$$q(S_t, A_t) = q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_A q(S_{t+1}, A_{t+1}) - q(S_t, A_t) \right]$$

La diferencia con respecto a Sarsa es que, en este caso se seleccionará el valor q de la acción tomada que dé un valor q mayor, de igual forma que se hizo en DP. Hay que destacar que su diferencia principal con Sarsa está en que es Off-Policy, por lo que la acción que toma variará dinámicamente independientemente de la acción que el agente esté tomando en ese momento, cambiando así la política.

El pseudocódigo correspondiente a Q-Learning es el siguiente:

Parámetros: $\alpha \in (0, 1], \epsilon > 0$.

1. Inicializar $q(s, a)$ para todo $a \in S^+$ y $a \in A(s)$, arbitrariamente menos para $q(terminal, \cdot) = 0$.

2. Bucle para cada episodio:

   - Inicializar s.
   - Bucle para cada paso del episodio:
     - Elegir una acción a de s usando una política derivada de q (p.ej ϵ-greedy)
     - Tomada a, observar r y s'.

- $q(s,a) \leftarrow q(s,a) + \alpha\left[r + \gamma \max_A q(s',a') - q(s,a)\right]$

Hasta que s sea terminal.

Fin del episodio.

La diferencia clave con Sarsa está en el paso de elegir una acción, la cuál se realiza por cada iteración del episodio y no se mantiene.

Consideramos interesante en este punto de la explicación comentar los problemas de viabilidad con las tablas en los métodos tabulares, disponiendo como alternativa de los métodos aproximados. Estos problemas de viabilidad vienen por el hecho de que las tablas q deben albergar, como ya se sabe, todas las combinaciones entre los estados y las acciones, implicando que para un entorno mucho mayor, donde el número de estados y acciones sean enormes, la tabla es imposible de tratar de un modo efectivo, por lo que se necesita una forma más eficaz de ajustar los valores de q. Una solución alternativa a los métodos aproximados fue dado con Q-Learning mediante el uso de redes neuronales, mezclando así Q-Learning con las técnicas de las redes neuronales profundas (deep neural networks), dando lugar esta mejora a un nuevo algoritmo, el **algoritmo Deep Q-Network** (DQN).

Deep Q-Network (DQN)

DQN es un algoritmo que, mediante el uso de redes neuronales, es capaz de potenciar la estimación de los valores q, siendo capaz de resolver el problema de viabilidad que había con los métodos de aprendizaje tabulares.

La red neuronal que este algoritmo maneja tiene como entrada los estados, y como salida los valores ajustados de q para, de esa forma, acelerar la elección de la acción disponiendo de los valores de q aproximados por la propia red sin necesidad de disponer de la tabla completa.

Teniendo en cuenta esta aproximación, ahora tendremos que considerar los pesos de las distintas neuronas que componen la red (w_t) ajustándose los pesos a lo largo de de las iteraciones t en la ecuación de Bellman:

$$w_{t+1} = w_t + \alpha\left[R_{t+1} + \gamma \max_A \hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t)\right] \nabla \hat{q}(S_t, A_t, w_t)$$

Podemos ver cómo la función q se ve reflejada en la función w, donde el cambio más notable está en el gradiente $\nabla \hat{q}(S_t, A_t, w_t)$, que indica la dirección de mayor aumento de q.

A modo de anécdota, este algoritmo surgió de un artículo donde sus desarrolladores, unos investigadores de *DeepMind*, lo aplicaron para entrenar

a un agente para que fuese capaz de jugar a varios juegos de la Atari 2600, suponiendo un rotundo éxito que demostró que el DQN era capaz de aprender directamente de información visual compleja para así lograr un rendimiento sobrehumano en una variedad de juegos.

Originalmente, como la mayoría de las redes neuronales, comenzó aprendiendo mediante el algoritmo de *backpropagation*, pero con el paso del tiempo, como es natural, se empezó a implementar algoritmos más sofisticados y adaptados al RL, como son *Double DQN*, *Dueling DQN*, *Rainbow*, *A3C* (*Asynchronous Advantage Actor-Critic*), entre otros.

DQN es un algoritmo ya más cercano a lo que nos vamos a encontrar en la parte práctica de este TFG, siendo uno de los algoritmos más usados dentro de **MlAgents**, pero que ha sido superado por el que veremos al final de este capítulo.

Entre los problemas que podemos destacar de DQN, encontramos:

- Inestabilidad, debido a su naturaleza no estacionaria.
- Aprendizaje offline, que introduce problemas adicionales como la correlación temporal y la falta de exploración activa, dificultando el aprendizaje efectivo.
- Value Bootstrapping, heredado de TD. Esta técnica hace que, en cada paso, se estime el valor q objetivo utilizando el máximo valor q para el siguiente estado, pudiendo provocar la posible introducción de sesgos y errores, especialmente cuando la función de valor está mal inicializada o cuando se encuentran estados poco comunes o de alto riesgo. Los errores de Bootstrapping pueden propagarse y afectar la convergencia y estabilidad del algoritmo.

Uno de los algoritmos que ha sido capaz de cubrir esos problemas, y además es el que se usa en la librería MlAgents, es el **algoritmo de Proximal Policy Optimization** (PPO).

1.9. Proximal Policy Optimization (PPO)

Hasta ahora, en la explicación de los fundamentos del RL hemos hablado de los métodos de aprendizaje que han establecido los cimientos del aprendizaje en agentes. Todo lo explicado nos lleva a este algoritmo, que será el que empleemos para entrenar agentes en MlAgents.

El algoritmo PPO, surgido en 2017,fue un algoritmo propuesto para mejorar DQN. Este nuevo algoritmo aborda los problemas que tenía DQN del siguiente modo:

- Para la inestabilidad: PPO utiliza un enfoque de optimización de políticas que garantiza que las actualizaciones de la política sean graduales y estables, haciendo que, en lugar de realizar grandes cambios en las políticas en cada iteración, se limiten las actualizaciones a una región *segura* alrededor de la política actual, evitando así fluctuaciones bruscas y una fuerte mejora de la estabilidad del algoritmo.

- Para el problema del aprendizaje offline: PPO hace uso de muestras de experiencia recolectadas en tiempo real durante la interacción del agente con el entorno, lo que permite una exploración activa y una reducción considerable de la correlación temporal, haciéndolo a diferencia de DQN un algoritmo online.

- Por último, para el problema que da el uso de Bootstrapping: PPO utiliza en su función de valor aproximada (como una red neuronal al igual que DQN) la ventaja estimada de los estados para actualizar la política, ayudando así a mitigar los problemas de sesgo y errores introducidos por Bootstrapping.

A diferencia de DQN, PPO hace uso del valor v en vez del valor q, como hacía a su manera TD(0), solo que v es usado esta vez como complemento para la función de ventaja ($\hat{A}$), la cual indica la diferencia entre el valor de acción q y el valor de estado en un estado y tiempo específicos, lo que nos permite evaluar la calidad de una acción en relación con el valor general del estado, indicando si hay que reforzar la acción si da beneficio o si hay que penalizarla en caso de empeorar el valor:

$$\hat{A}_t = q(s_t, a_t) - v(s_t)$$

Podríamos preguntarnos, si se había dicho que se hacía uso de v, ¿por qué no se dice que se usa también q en la función?, y es que si bien el valor de acción q puede proporcionar información adicional, su uso en PPO es más secundario y se enfoca más en la estimación del valor del estado v para la toma de decisiones de política.

Con la función de ventaja introducimos la función principal y objetivo del algoritmo PPO, la función de pérdida (L), que emplea el enfoque CPI (Conservatice Police Iteration) que, como dice su nombre, es un enfoque

para la actualización de políticas en el que las actualizaciones se limitan a una región segura alrededor de la política actual (cosa que mencionamos con respecto la resolución de la inestabilidad).

La función de pérdida sería la siguiente:

$$L^{CPI}(\pi) = \mathbb{E}_t \left[r_t(\theta) \cdot \hat{A}_t \right]$$

Donde:

- θ es el conjunto de parámetros que ajustan la función objetivo para así ajustar su rendimiento.
- $r(\theta)$ es el ratio de probabilidad entre la probabilidad actual y la anterior:

$$r(\theta) = \pi_\theta(a_t|s_t)/\pi_{\theta old}(a_t|s_t)$$

Así pues, esta función objetivo se interpreta como el valor esperado del ratio de probabilidad a lo largo de las estimaciones de la función de mejora a lo largo de t.

La función objetivo que se va a mostrar a continuación emplea un enfoque distinto, el enfoque Clip (que viene del término *acotar*) y que, a diferencia del enfoque CPI, limita el tamaño de las actualizaciones a través de una función de acotamiento, reduciendo la inestabilidad y la falta de control:

$$L^{Clip}(\theta) = \mathbb{E}_t \left[\text{mín} \left(r(\theta) \cdot \hat{A}_t, \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon) \cdot \hat{A} \right) \right]$$

Donde se compara el ratio de probabilidad que considerábamos en CPI con respecto al ratio de probabilidad en el caso de verse acotado por la cota $[1-\epsilon, 1+\epsilon]$, donde ϵ ajusta la cota, siendo mayor la cota cuanto mayor sea ϵ. De esta forma, si el ratio supera el umbral que permite la cota, entonces se limita devolviendo el umbral.

Capítulo 2

Introducción a MlAgents

2.1. Introducción a MlAgents

Antes de pasar a la parte práctica de este trabajo es imprescindible dedicar un capítulo exclusivo a presentar la herramienta por la que vamos a ser capaces de aplicar los conceptos de RL en el entorno de Unity: **MlAgents**.

MlAgents (*Machine Learning Agents*) es un framework de Unity desarrollado por *Unity Technologies* que permite entrenar agentes inteligentes haciendo uso de técnicas de RL (aunque añade algunas técnicas más generales de ML, en este capítulo exploraremos la herramienta únicamente desde el punto de vista de RL)[1].

La aproximación que ofrece MlAgents es sencilla y permite ver distintos elementos de la herramienta desde un ángulo que ya conocemos gracias a la teoría MDP.

Dentro del contexto de Unity, lo primero con lo que nos topamos es el **escenario**, que es el espacio virtual donde se desarrolla el juego (o simulación) que hagamos. El resto de objetos serán elementos del escenario.

El escenario juega un papel similar al **entorno** que hemos visto en el capítulo anterior, conformado por los objetos (**elementos**) con los que interactúe el agente, y también **otros agentes existentes en el espacio**.

En los siguientes apartados veremos más detallada la implementación del **agente**.

[1]Para más detalle de su implementación, el proyecto oficial de MlAgents puede encontrarse en el repositorio correspondiente de Unity [14].

2.2. MlAgents implementado en agente

2.2.1. Script MlAgents

Lo primero que vamos a ver es la parte principal que debemos añadir y configurar para el entrenamiento que va a realizar nuestro agente, ubicado en un script programado en *C#* que debemos crear y añadir como complemento del agente. En este fichero se debe incluir la librería `Unity.MlAgents` y su clase además heredará de la clase `Agent` de la misma librería.

En este script debemos redefinir alguna de las funciones que contiene la librería, y que permitirán especificar el comportamiento concreto del agente. Para ello debemos indicar los componentes secundarios, que son configurados mediante las siguientes funciones:

- `void CollectObservations(VectorSensor sensor)`: Esta función depende de los "sensores" del agente, un objeto de la clase `VectorSensor`), que es un vector donde podremos añadirle el número de atributos (o características) del entorno que veamos importante para realizar una observación cada episodio. Sería el equivalente al entorno percibido por el agente en ese episodio determinado, lo que conocemos como el **estado**. Las observaciones deben proporcionar suficiente información relevante para permitir que el agente aprenda a realizar la tarea objetivo del aprendizaje.

- `void OnActionReceived(ActionBuffers actions)`: Esta función realizará las acciones que se indiquen en el buffer `actions`. Juega el papel de la **acción**. Aquí solo se indica la acción elegida, no es la responsable de definir las acciones disponibles en nuestro problema (veremos pronto dónde se define `actions`).

- `void AddReward(float increment)`: A diferencia de los anteriores, este no es necesario redefinirlo. La función toma el papel de la **recompensa**, y añade al agente el puntuaje que le pasemos a la función como parámetro (Si le metemos un valor positivo será una recompensa y si es negativo una penalización.)

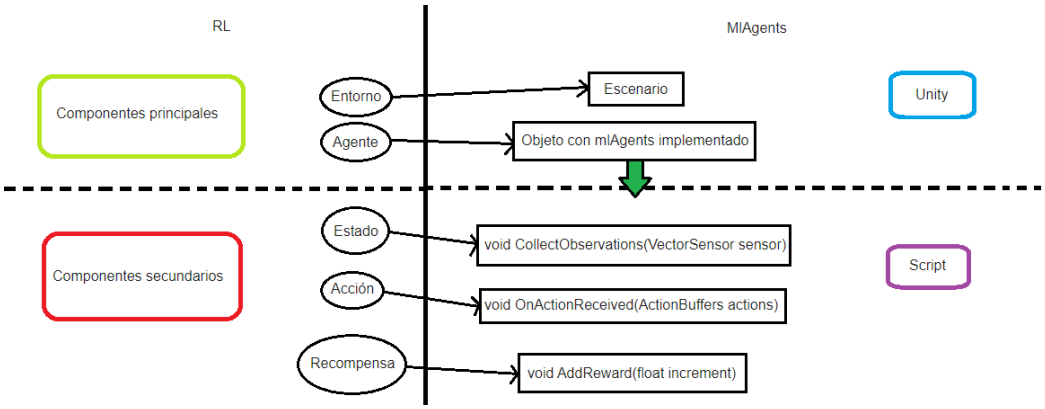

Figura 2.1: Comparación gráfica entre los compones de RL y sus reflejos en MlAgents.

Con los componentes ya identificados, vamos a mencionar aquellas funciones de la librería que utilizaremos/redefiniremos en el script:

- `void OnEpisodeBegin()`: Esta función se utiliza cada vez que se comience un episodio. Es como una función `setup` aplicado a MlAgents.
- `void Heuristic(in ActionBuffers actionsOut)`: Esta función toma el comportamiento de la heurística que le queramos añadir. Pese a ser mencionada y declarada/definida en la práctica, no se utilizará.
- `void EndEpisode()`: Cuando se llame a esta función el episodio actual finalizará, a diferencia de las anteriores no necesita ser redefinida.

Una vez implementado el script solo tendremos que tener en cuenta un parámetro dentro de este (aparte de los que se introduzcan en el propio script), que es la variable `MaxStep`, que define el número de pasos máximos que hay en un episodio.

2.2.2. Behavior Parameters

El componente `Behavior Parameters` se genera automáticamente tras incluir el script de MlAgents, y es donde se configura el comportamiento que va a tener el agente.

Podemos diferenciar en el componente los siguientes parámetros de configuración:

- `Behavior Name`: Es el nombre del comportamiento e identificador para el fichero `.yaml` que debemos usar.

- `Vector Observation`: Es el parámetro donde debemos dar las especificaciones del vector sensor, siendo sus especificaciones:

  - `Space Size`: Tamaño del vector y número máximo de atributos para una observación (recordemos que luego el vector se llena haciendo uso de la función `CollectObservations`)
  - `Stacked Vector`: Indica cuántas observaciones consecutivas se toman/apilan en el vector `Stack` como máximo. Permite al agente acceder a información histórica y secuencial del entorno, pudiendo acceder a un historial más profundo cuanto mayor sea este, y siempre eliminando la observación del episodio más antiguo en caso de estar lleno (una estructura de pila FIFO, ver Figura 2.2).

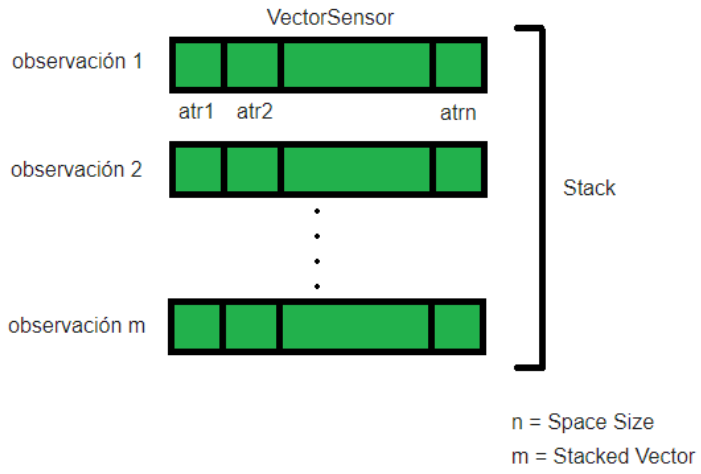

Figura 2.2: Ilustración gráfica de los conceptos de `VectorSensor`

- `Actions`: En este parámetro especificamos cómo va a ser nuestro buffer de acciones (`actions`), siguiendo las especificaciones:

  - `Continuous actions`: Número de acciones continuas. En este caso, las acciones que puede tomar el agente son valores continuos dentro de unos rangos específicos que se definen más tarde en el `.yaml`.

    - Discrete Branch: Número de acciones discretas. En lugar de tener un rango continuo de valores para elegir, el agente puede seleccionar solo una acción de un conjunto predefinido de opciones discretas con las siguientes subespecificaciones:
        - Branch 0 Size: Número de opciones para la acción 0.
        - Branch 1 Size: Número de opciones para la acción 1.
          ...
        - Branch (Discrete Branch - 1) Size: Número de opciones para la última acción del buffer.

- `Model`: Apartado para añadir el modelo/red neuronal entrenado para aplicar ese entrenamiento al agente.

- `Inference Device`: Dispositivo o plataforma en el que se ejecuta el modelo de agente entrenado (o a entrenar sin modelo añadido) para tomar decisiones en tiempo real. Puede ser ser la CPU, la GPU o incluso dispositivos de inferencia especializados como TPUs (Unidades de Procesamiento Tensorial). Depende de las características del computador disponible.

- `Behavior Type`: Especifica el tipo de comportamiento que va a tener el agente, pudiendo ser:

    - `Default`: Es la configuración predeterminada y la que hace que el agente entrene.
    - `Inference Only`: Ignora el entrenamiento para solo implementar el modelo entrenado (en caso de que tenga uno).
    - `Heuristic Only`: El agente se basa en heurísticas predefinidas (en la función `Heuristic`) en lugar de entrenar (no haremos uso de esta).

- `Team ID`: Indica la identificación del equipo al que pertenece el agente en un entorno multiagente. En un entorno con múltiples agentes es común agruparlos en equipos para permitir la colaboración o competencia entre ellos.

- `Use Child Sensors`: Indica si el agente debe utilizar los sensores de sus hijos en el entorno. En un entorno jerárquico, donde hay una estructura de agentes y subagentes, los sensores de los subagentes pueden proporcionar información adicional relevante para el agente principal,

por lo que habilitar esta opción permite al agente principal tener acceso a los sensores de sus subagentes, lo que le permite tomar decisiones informadas basadas en la información recopilada por ellos.

- `Observable Attribute`: Indica la información específica del entorno que el agente puede percibir y utilizar en su proceso de toma de decisiones. Nosotros tendremos este parámetro desactivado (`ignore`), ya que todo lo relacionado a las observaciones lo hacemos usando VectorSensor.

2.2.3. Desicion Recuester

El componente `Desicion Requester` es la parte de la implementación de MlAgents que se encarga de tomar las decisiones. Es esencial para que se produzca el entrenamiento y actividad del agente.

Este componente actúa como una interfaz entre el agente y el entorno, solicitando decisiones en cada paso del episodio, recopilando las observaciones del entorno (sacándolas de Stack) y enviándolas al modelo de aprendizaje (la red neuronal) para que genere una acción o decisión. Posteriormente, esa acción se ejecuta en el entorno y se continúa con el siguiente episodio.

Los dos únicos parámetros de configuración que tiene este componente son los siguientes:

- `Desicion Period`: Número de iteraciones en el que toma decisiones durante un episodio (no tienen por que ser iteraciones por episodio).
- `Take Actions Beetween`: Especifica el número de iteraciones en el que una acción se mantiene aplicando durante un episodio.

2.2.4. Ray Perception Sensor 3D

Este componente actúa de sensor adicional con el que podemos dar información adicional a MlAgents con respecto los elementos que se le indique mediante el uso de **etiquetas**.

`Ray Perception Sensor 3D` es un sensor que mediante el uso de rayos de colisión (*raycasts*) o esferas de colisión (*spherecasts*) es capaz de recolectar información de la distancia a la que se encuentra con respecto los elementos detectables por este. Una vez los rayos/esferas colisionan con algunos de los elementos detectables, esa observación/distancia se registra.

La diferencia entre usar *raycast* o *spherecast* está en el volumen de detección utilizado para la percepción del entorno, siendo en el caso de *spherecast* mayor, pero teniendo un coste computacional mayor.

Pese a no ser realmente un componente de MlAgents, es un componente bastante útil si queremos abordar entornos más complejos donde las observaciones explicitadas en el script no sean suficientes para el MDP del problema.

Este componente puede estar tanto incorporado en el agente, pudiéndolo llamar `Sensor POV` (Point Of View), como colocado en algún punto del entorno, pudiéndose llamar en este otro caso `Sensor TP` (Third Person). Esta elección depende del problema que estemos tratando ya que, por ejemplo, si en el problema el agente supuestamente no conoce el entorno donde está y debe explorarlo, no es normal que pueda percibir cosas más allá de lo que pueda percibir mediante sus sensores (POV). Por otra parte, si en el problema el agente conoce el entorno (por ejemplo, el agente es un guardia de un recinto que conoce, o bien tiene un mapa del sitio) es posible hacer "trampas" y colocar el sensor en una zona que a simple vista el agente no pueda percibir con tan solo la vista (TP).

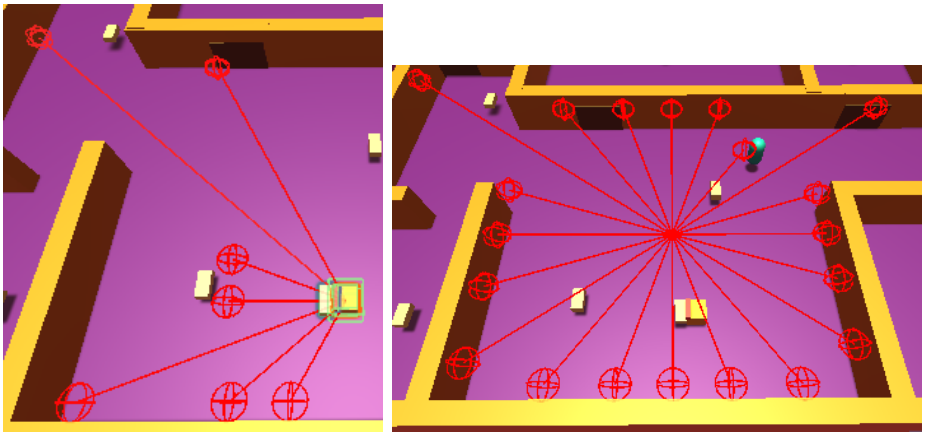

Figura 2.3: Ejemplos sensores configurados como POV (imagen de la izquierda) y como TP (imagen de la derecha).

Los parámetros son los siguientes:

- `Sensor Name`: El nombre del sensor.
- `Detectable Tags`: Este parámetro sirve para indicar el número de etiquetas/elementos observables por el sensor, necesitando posteriormente añadir las etiquetas en concreto que queremos que observe, ignorando así todo objeto que no lleve ninguna de las etiquetas indicadas.

- `Rays Per Direction`: Sirve para ajustar el número de rayos de percepción que va a tener el sensor.
- `Max Ray Degrees`: Esparce a modo de abanico los rayos en base a los grados indicados. Puede tomar un valor entre 0 y 180 grados.
- `Sphere Cast Radius`: Ajusta el radio de los *spherecasts*, donde si ese radio es cero entonces indicamos que los colisionadores son *raycasts*.
- `Ray Layer Mask`: Especifica qué objetos, de los observables, deben ser considerados y cuáles deben ser ignorados. Esto puede ser util para realizar pruebas de colisión.
- `Stacked Raycasts`: Indica el número de rayos/esferas apilados, siendo estos rayos/esferas superpuestos para aumentar así la precisión del sensor a cambio de un mayor coste computacional.
- `Start Vertical Offset`: Ajusta el ángulo de los rayos con respecto el objeto que implemente el sensor.
- `End Vertical Offset`: Ajusta la distancia a la que el sensor es capaz de percibir colisiones mediante las esferas de colisión.

Aparte tenemos también los parámetros de coloreado de las líneas de sensor, pero nos son triviales.

2.3. Parámetros de la red neuronal

Por el momento hemos visto cómo configurar los componentes principales y secundarios del MDP de nuestro problema en Unity, así como los componentes MlAgents para el comportamiento, toma de decisiones y sensores del agente (o entorno si el sensor no está acoplado al agente).

Si recordamos lo visto en el capítulo anterior, para los métodos de aprendizaje son necesarios ciertos parámetros aparte junto a las especificaciones de los episodios ($t \in T$). Es por ello que, de alguna forma, necesitamos indicar a MlAgents esta configuración restante.

Para este conjunto de parámetros de aprendizaje se hace uso de un fichero .yaml, `Behavior Name`, donde especificarán los siguientes parámetros:

- `trainer_type`: Aquí especificamos el método de aprendizaje que se va a utilizar para el entrenamiento del modelo. Para los entrenamientos

de un solo agente que vamos a realizar vamos a hacer uso del método **PPO** comentado en el capítulo anterior.

- `hyperparameters`: Esta agrupación de parámetros afectan al comportamiento y el rendimiento del agente y que no pueden ser modificados a lo largo de los episodios.
  - `batch_size`: Número de muestras utilizadas antes de realizar una actualización de los parámetros del modelo.
  - `buffer_size`: Tamaño del buffer de replay utilizado en el algoritmo de aprendizaje. El buffer de replay almacena transiciones de experiencia pasadas para entrenar al agente de manera más eficiente.
  - `learning_rate`: Tasa de aprendizaje utilizada en la ecuación de Bellman (α). Controla qué tan rápido se actualizan los parámetros del modelo.
  - `beta`: Factor de entropía utilizado en la función de pérdida. La entropía se utiliza para fomentar la exploración del agente durante el entrenamiento. Este hiperparámetro no es visible en las ecuaciones vistas en el capítulo anterior.
  - `epsilon`: hiperparámetro de clip utilizado en el algoritmo PPO (ϵ). Limita la magnitud del cambio de los parámetros del modelo para evitar actualizaciones demasiado grandes.
  - `lambd`: Elegibilidad de la tasa (λ). Indica cuántos pasos en el tiempo se tienen en cuenta al realizar una actualización.
  - `num_epoch`: Número de veces que iterará/hará uso de las muestras durante una actualización de los parámetros del modelo.
  - `learning_rate_schedule`: Tipo de programación de la tasa de aprendizaje. Nosotros haremos uso de un programa lineal (`linear`), lo que significa que la tasa de aprendizaje se reducirá de manera lineal a lo largo del tiempo de entrenamiento.
- `network_settings`: Esta agrupación de parámetros define la configuración de la red neuronal utilizada por el agente:
  - `normalize`: Indica si las observaciones del entorno deben normalizarse antes de ser procesadas por la red neuronal.
  - `hidden_units` Especifica el número de unidades (neuronas) en cada capa oculta de la red neuronal.

- `num_layers`: Especifica el número de capas ocultas en la red neuronal.
- `vis_encode_type`: Especifica el tipo de codificación visual utilizado para procesar las observaciones visuales, pudiendo ser:
  - `simple`: Indica que las observaciones visuales se codifican mediante un enfoque simple, sin aplicar ninguna transformación adicional. En este caso, las imágenes o datos visuales se envían directamente a la red neuronal sin cambios.
  - `Nature`: Utiliza la codificación visual inspirada en los algoritmos del campo de visión de los juegos de Atari (usado en el algoritmo DQN). Esto incluye la conversión de las imágenes en escala de grises y el reescalado a una resolución más baja.
  - `NatureDepth`: Similar a la codificación visual "Nature", pero también incluye la información de profundidad en la representación visual. Puede ser útil en entornos donde la percepción de la profundidad es relevante para el agente.
  - `ResNet`: Utiliza una red neuronal convolucional de tipo ResNet para extraer características visuales de las imágenes antes de enviarlas a la red neuronal principal. Esto puede ayudar a capturar características visuales más complejas y discriminativas.
  - `Custom`: Permite definir una codificación visual personalizada específica para el problema o entorno de aprendizaje por refuerzo en cuestión. Aquí se puede implementar una lógica personalizada para procesar y codificar las observaciones visuales según sea necesario.
    En nuestro caso haremos uso de una red simple.

- `reward_signals`: Esta agrupación de parámetros define las señales de recompensa adicionales que se utilizan en el entrenamiento de un agente de aprendizaje por refuerzo.

  Podemos diferenciar estos parámetros de recompensa entre dos tipos:

  - **Recompensas extrínsecas (`extrinsic`)**: Referido a las recompensas proporcionadas directamente por el entorno para evaluar el desempeño del agente en el problema específico que se está abordando. Este tipo de recompensa está diseñada para guiar al agente hacia la consecución de los objetivos generales del problema.
  - **Recompensas intrínsecas (`intrinsic`)**: Referido a unas recompensas adicionales que se agregan al proceso de entrenamiento del

agente y que se utilizan para guiar el aprendizaje interno del agente. Esta recompensa es generada internamente por el agente, generalmente basada en factores como la exploración, la curiosidad o la novedad. La recompensa intrínseca puede ayudar al agente a descubrir y aprender nuevas estrategias o comportamientos que no están directamente relacionados con la recompensa extrínseca, pero que pueden ser beneficiosos para el aprendizaje y la exploración del entorno.

En el presente trabajo solo vamos a hacer uso de recompensas extrínsecas, en concreto de las siguientes:

- `strength`: Factor de ponderación o importancia de la señal de recompensa extrínseca. Puede ajustarse para equilibrar la influencia de diferentes señales de recompensa en el entrenamiento.
- `gamma`: Tasa de descuento utilizada en la ecuación de Bellman (γ). Controla qué tan importantes son las recompensas futuras en comparación con las recompensas inmediatas.

- `keep_checkpoints`: Especifica la cantidad máxima de puntos de control (checkpoints) que se guardarán durante el entrenamiento.

  Estos puntos de control se utilizan para almacenar los parámetros y el estado del modelo en un determinado punto durante el entrenamiento, siendo útiles para:

  - **Respaldo del modelo**: Los puntos de control permiten guardar el progreso del modelo durante el entrenamiento, siendo especialmente útil en entrenamientos largos o costosos en recursos computacionales, ya que si ocurre algún fallo o interrupción, puedes retomar el entrenamiento desde el último punto de control guardado en lugar de comenzar desde cero.
  - **Evaluación del rendimiento**: Los puntos de control pueden servir para evaluar el rendimiento del modelo en diferentes etapas del entrenamiento. Al cargar un punto de control y ejecutar el modelo con datos de prueba, se puede evaluar su desempeño y compararlo con otros puntos de control o con el rendimiento final del modelo.
  - **Ajuste fino (*fine-tuning*)**: Los puntos de control también son útiles cuando se desea ajustar o continuar el entrenamiento de un modelo pre-entrenado, pudiéndose cargar un punto de control previo y continuar el entrenamiento con nuevos datos o con un

enfoque de aprendizaje diferente, lo que permite refinar y mejorar el modelo existente.

- **Comparación de experimentos**: Al guardar puntos de control en diferentes momentos o con diferentes configuraciones de hiperparámetros, se pueden comparar y analizar los resultados de diferentes experimentos. Esto permite tomar decisiones informadas sobre la configuración óptima del modelo y entender cómo se comporta el modelo a medida que avanza el entrenamiento.

- `max_steps`: Especifica el número máximo de iteraciones de un episodio ($t \in T$).

- `time_horizon`: Especifica el número máximo de pasos de tiempo consecutivos que se consideran en una actualización de la red neuronal durante el entrenamiento. Representa la ventana temporal utilizada para calcular las actualizaciones y retropropagar los gradientes.

- `summary_frec`: Especifica cada cuántas iteraciones se guardará un resumen del rendimiento y las métricas del agente.

- `threaded`: Indica si el entrenamiento debe realizarse en un entorno multihilo o no.

Para su uso práctico las modificaciones en estos parámetros van a ser nulas, teniendo el valor que indiquemos al principio de la práctica para todos los comportamientos que entrenemos.

2.4. Fase de entrenamiento

Una vez configurado todo, podemos pasar al entrenamiento del agente y, con ello, a la generación del modelo entrenado.

Para la fase de entrenamiento, cada comportamiento distinto que se quiera entrenar debe hacer uso de un agente representante para que lo realice. Una vez tengamos el modelo entrenado, si queremos tener n agentes configurados con el mismo comportamiento, lo único que se le tiene que incluir a estos es el modelo entrenado en `Behavior Parameters`. De esta forma, los agentes toman directamente la mejor política elegida durante el entrenamiento, teniendo así un comportamiento modular.

Es necesario tener en cuenta que este entrenamiento puede resultar muy lento, tardando un entrenamiento de recolecta sencillo (como el primero que

haremos en la práctica) aproximadamente 5 minutos para que el agente obtenga destreza. Aunque puede no parecer excesivo, una alta complejidad en el problema podría llevar a tiempos de entrenamiento severamente largos, incluso para aquellos sin problemas de viabilidad.

Este problema se puede resolver parcialmente paralelizando el entrenamiento mediante múltiples agentes representantes con sus respectivos entornos, todos ellos copias exactas entre sí. De esta forma, técnicamente, si tenemos dos agentes al mismo tiempo realizando el mismo entrenamiento provocará que la búsqueda de la política óptima se vea acelerada por dos, durando así la mitad de tiempo. Del mismo modo, si usamos más copias, nuestro tiempo se verá reducido, pudiendo así esta vez obtener tiempos de entrenamiento razonables para comportamientos más interesantes de desarrollar que conlleven a una mayor complejidad.

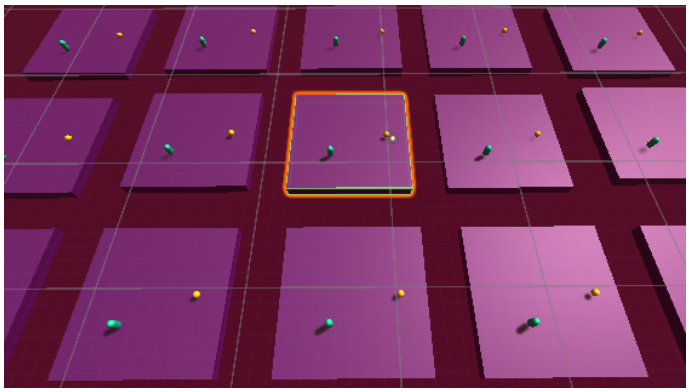

Figura 2.4: Escenario haciendo uso de múltiples copias del entornos

Así pues, tras este proceso de copia, la definición de escenario varía, pasando de ser un término equivalente de entorno a ser un espacio que alberga múltiples entornos y agentes iguales.

Como es natural, el límite de paralelización dependerá de la capacidad del computador donde se esté realizando el entrenamiento, pudiendo albergar una mayor paralelización si las características son de media aceptables o muy buenas.

En nuestro caso, los computadores donde ejecutaremos los distintos entrenamientos tienen de media unas características bastante buenas, las cuales veremos en el próximo capítulo cuando introduzcamos la fase práctica del trabajo. Esto nos permite albergar una paralelización aceptable, dándonos terreno suficiente para poder poner a prueba los límites del aprendizaje por MlAgents y así cumplir el objetivo principal de este trabajo.

Capítulo 3

Experimentos

3.1. Marco Experimental

Hemos comenzado nuestro recorrido en esta memoria desde los contenidos más teóricos para ir desembocando en las aplicaciones y contenidos más prácticos. En este capítulo nos introducimos en la parte práctica de la memoria donde, mediante MlAgents, vamos a configurar y entrenar una serie de comportamientos, y así poner a prueba los modelados, los algoritmos y a MlAgents. De esta forma, pretendemos verificar la posibilidad de crear comportamientos entrenados complejos por medio del Aprendizaje por Refuerzo.

Con este fin, presentaremos de forma detallada un conjunto de experimentos, de dificultad creciente, que pondrán a prueba algunas de las características que queremos estudiar.

3.1.1. Criterios de evaluación

A lo largo del capítulo describiremos los distintos experimentos realizados, detallando sus características principales a nivel conceptual, y sus respectivas implementaciones en Unity, para así posteriormente evaluar de forma supervisada el entrenamiento mediante TensorBoard enfocando el análisis en:

- **La eficiencia del entrenamiento**, donde aspiramos a que el agente acabe teniendo destreza en el problema tratado.
- **La velocidad y conformidad de la destreza con respecto al problema entrenado**, de forma que podamos evaluar si el tiempo necesario para entrenar al agente es razonable.

- **La complejidad**, donde evaluamos de forma subjetiva las conclusiones obtenidas con respecto la complejidad del problema. Es una conclusión orientada a arrojar luz sobre el próximo entrenamiento. En caso de no haber podido realizarse el entrenamiento de forma satisfactoria consideraremos la complejidad del entrenamiento actual como complejidad umbral del problema planteado usando tales configuraciones.

La explicación de las especificaciones de cada entrenamiento se centrará en cómo se configura cada entrenamiento en los scripts, y a su vez en la identificación y modificación por cada entrenamiento a nivel conceptual de los componentes primarios y secundarios del MDP. Para los detalles de configuración de todos los componentes de MlAgents (inclusive `Ray Perception 3D`) queda disponible el acceso al proyecto de la práctica en mi github:

https://github.com/JoseGallardoHarillo/RL-MlAgents

3.1.2. Entorno de Computación

Para estos entrenamientos disponemos de dos entornos de computación de distinta capacidad: uno de características medias con el que esperamos abarcar un conjunto de comportamientos más o menos complejos, al que llamamos **PC1**; y otro con características altas para aquellos comportamientos donde aspiremos abarcar una mayor complejidad, al que llamamos **PC2**.

Las características específicas de cada uno de ellos son:

- **PC1**
  - CPU = *AMD Ryzen 5 3400G Radeon Vega Graphics*
  - RAM = 16GB
  - SSD = *KINGSTON SA400S37480G*
  - GPU = *NVIDIA GeForce GTX 1650*
- **PC2**
  - CPU = *AMD Ryzen 7 3700X 8-Core Processor*
  - RAM = 32GB
  - SSD = *WDC WDS500G2B0C-00PXH0*
  - GPU = *NVIDIA GeForce RTX 3080*

3.2. Guía de instalación y entrenamiento

En este apartado vamos ver los pasos a realizar para poder tener operativo MlAgents y ejecutar un entrenamiento completo, con el fin de hacer reproducible la parte experimental[1].

3.2.1. Aplicaciones

Las aplicaciones principales que se han utilizado son los siguientes (se indican las versiones exactas que se han utilizado, téngase presente que los tiempos y resultados obtenidos pueden variar mucho de la versión):

1. **Unity**: `versión 2021.2.9f1`.
2. **Anaconda**: `versión 22.9.0`.

3.2.2. Paquetes y comandos

Una vez instaladas las aplicaciones mencionadas, es necesario instalar una serie de paquetes para que los entrenamientos, visualización y demás se puedan realizar sin problemas.

Por parte de Unity, desde `Window/Package manager` debemos instalar el paquete correspondiente a MlAgents (la versión utilizada es la indicada en la Figura 3.1):

[1] La guía de instalación general que hemos usado se encuentra en el curso [3].

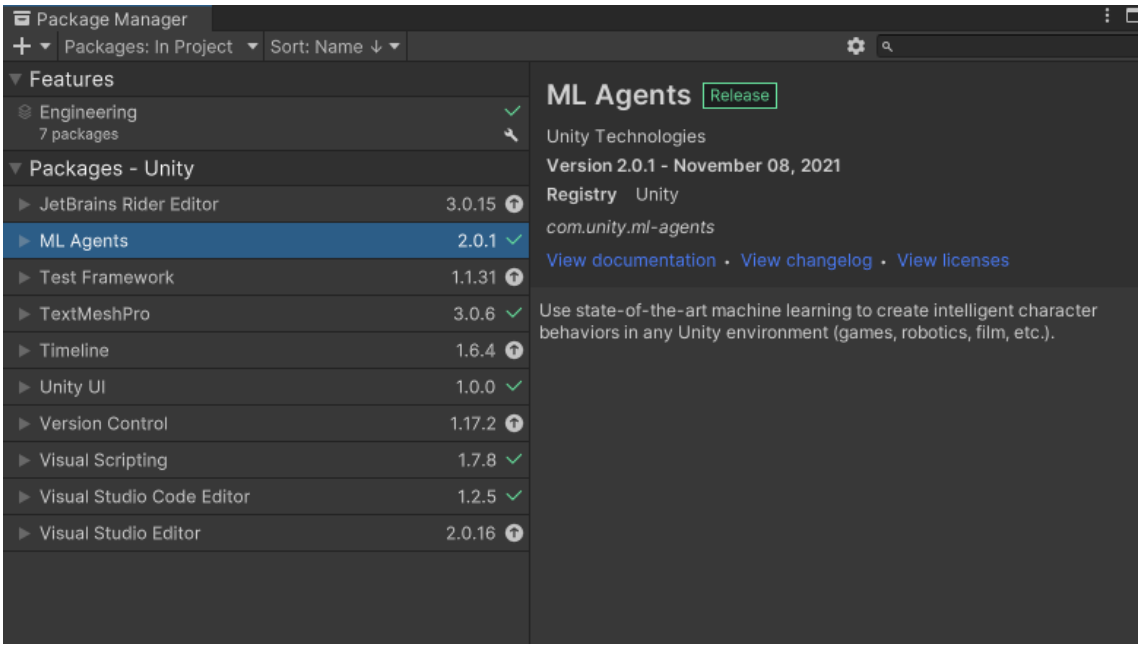

Figura 3.1: Paquete de MlAgents instalado en Package Manager.

Desde la terminal de Anaconda, lo primero y recomendable antes de instalar cualquier paquete necesario es crear un entorno virtual propio para así aislar las instalaciones del entorno `Base`.

Para crear un entorno virtual se utiliza el siguiente comando:

```
conda create -n *nombreEntorno* python=3.9
```

Donde `*nombreEntorno*` es el nombre que le queramos poner a nuestro entorno. Con este comando se crea el entorno con la versión `3.9` de `python`, que es una de las versiones recomendables y de la que hemos hecho uso. Actualmente, hacer uso de una versión más actual (como la `3.11`) conlleva, por ejemplo, que no se puedan instalar paquetes que posteriormente necesitaremos para que el entrenamiento se pueda hacer, visualizar, e implementar).

Con el entorno creado, siempre que queramos activarlo o desactivarlo se hace uso de los siguientes comandos respectivos:

```
conda activate *nombreEntorno*
conda deactivate *nombreEntorno*
```

Para ver los entornos disponibles en nuestro computador se hace uso de:

```
conda env list
```

Dentro del entorno donde vayamos a realizar los entrenamientos, los paquetes a instalar son los siguiente:

- **MlAgents**: `pip install mlAgents`
- **pytorch**: `conda install -c pytorch pytorch`
- **numpy**: `pip install numpy==1.19.0`
- **TensorBoard**: `pip install tensorboard`
- **onnx**: `pip install onnx`

Una vez configurados todos los componentes para el entrenamiento (script, Behavior Parameters, etc), dentro de nuestro entorno en la terminal de Anaconda debemos ubicarnos en la capeta donde tengamos el `.yaml` que vamos a utilizar y donde queramos que se genere la carpeta **results**, que contendrá los ficheros de los entrenamientos que se vayan haciendo mediante la configuración indicada en el `.yaml`.

El comando a utilizar para arrancar un entrenamiento es el siguiente:

```
mlagents-learn ./trainer_config.yaml --run-id *procAprend*
```

donde `*procAprend*` es el nombre que le daremos al entrenamiento que estemos ejecutando y con el que lo identificaremos en TensorBoard.

Figura 3.2: Pantalla resultante a la espera del comienzo de un entrenamiento

Si tras ejecutar el comando sale lo que aparece en la Figura 3.2, entonces MlAgents estará ya a la espera de realizar un entrenamiento. Para ello, seguidamente debemos arrancar el escenario concreto que define el experimento.

Para visualizar cómo va evolucionando el entrenamiento a lo largo del tiempo debemos abrir otra terminal ubicada en el entorno `base`. Dentro de este (ubicado en la misma carpeta donde se ubica **results**) utilizamos el siguiente comando para activar TensorBoard para los entrenamientos de la carpeta **results**:

```
tensorboard --logdir results
```

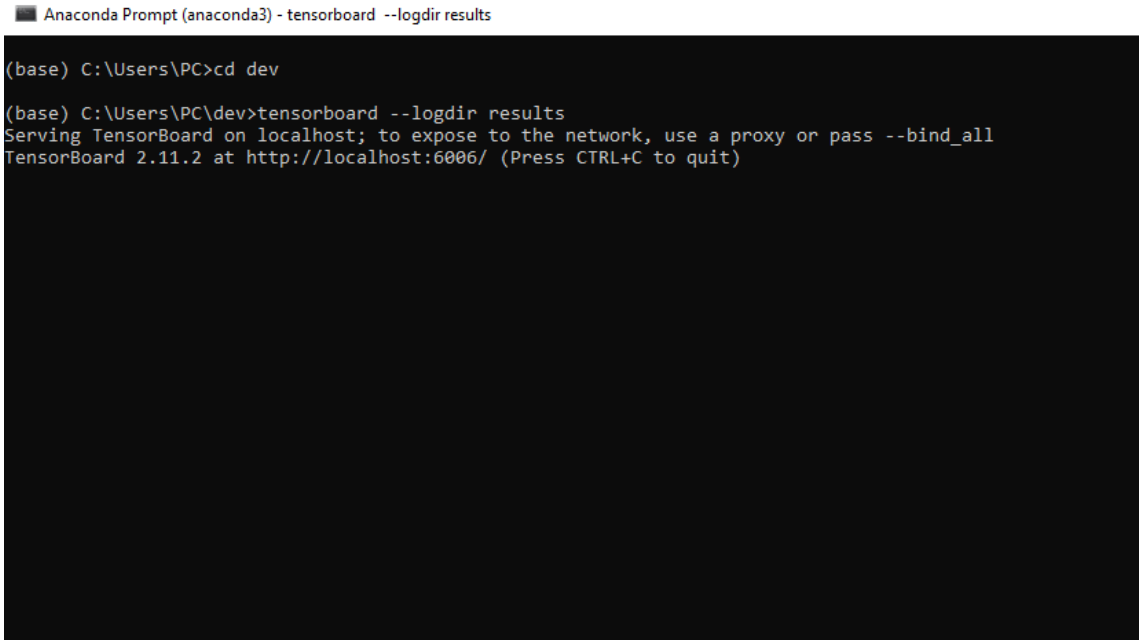

Figura 3.3: Pantalla resultante con la dirección localhost donde se puede visualizar TensorBoard

Desde el navegador podremos, acceder a la información servida por TensorBoard:

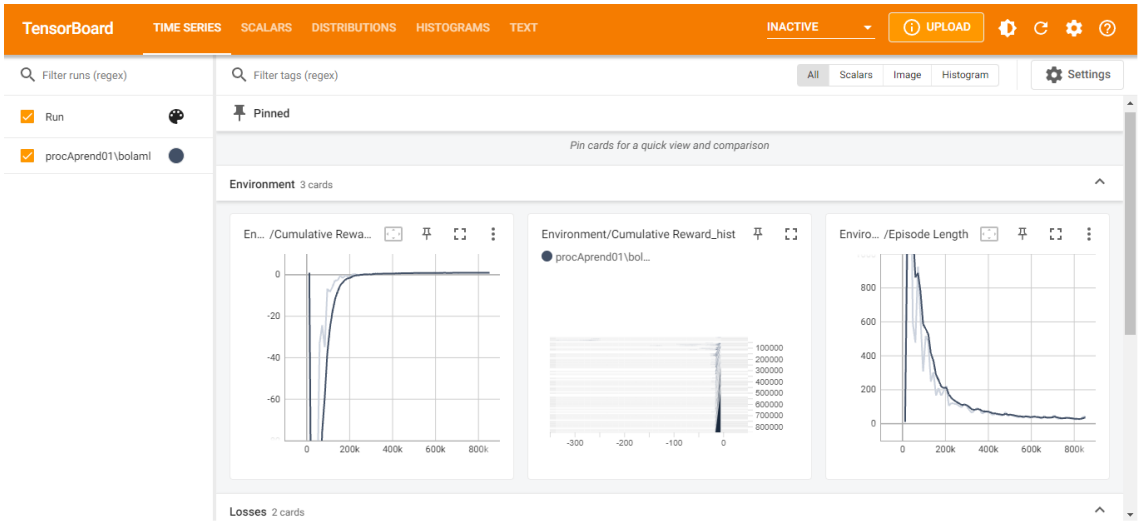

Figura 3.4: Visualización inicial en TensorBoard

Las gráficas que se muestran en la Figura 3.4 son las que iremos analizando analizando a lo largo de los entrenamientos. En concreto, las que ahí se

muestran son las correspondientes al primer experimento que presentaremos.

3.3. Script de apollo: MoveAgent

Aunque no está entre los experimentos de MlAgents, es interesante mencionar el script con el que se han testeado los entornos: `moveAgent`, que implantamos en el objeto que actuará como jugador.

El código es el siguiente:

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class MoveAgent : MonoBehaviour
6  {
7      Rigidbody rb = null;
8      public float speed = 400;
9
10     void Start()
11     {
12         //Almacenamos el rigidbody del jugador
13         rb = GetComponent<Rigidbody>();
14     }
15
16     void FixedUpdate() //Damos el movimiento
17     {
18         float moveHorizontal = Input.GetAxis("Horizontal
               ");
19         float moveVertical = Input.GetAxis("Vertical");
20
21         Vector3 movement = new Vector3(moveHorizontal, 0
               f, moveVertical);
22         rb.AddForce(movement * speed * Time.deltaTime);
23
24     }
25 }
```

Este jugador puede servir además para interactuar con el/los agentes del entorno, ya sea para obstaculizar o para ayudar. Este tipo de entrenamiento puede resultar más interesante si aplicamos un jugador que implemente **VR**, siendo algo en lo que no ahondaremos en este trabajo pero que podría ser una posible extensión (como ya comentamos en la introducción).

3.4. PC1

Vamos a empezar a realizar los experimentos con PC1 de forma que, cuando encontremos el entrenamiento que sirva de umbral para los límites, pasaremos a utilizar PC2[2].

3.4.1. Experimento 1: Bola recolectora

[3] [14] Para este primer experimento nos centraremos en uno de los objetivos más primitivo que se nos puede ocurrir cuando pensamos en un problema de RL: **recolectar un objeto del entorno**, al que llamaremos **ítem**.

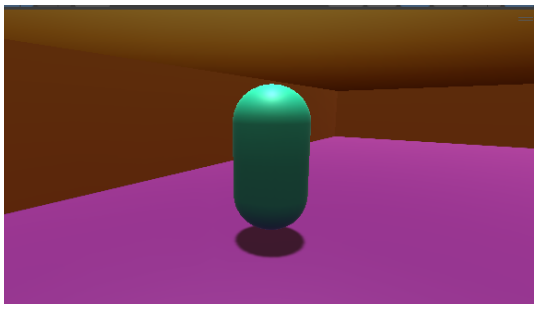

Figura 3.5: Objeto `item` en Unity

El agente a entrenar se trata de una esfera a la cuál se le aplicará una fuerza predefinida para que así se pueda desplazar por el entorno:

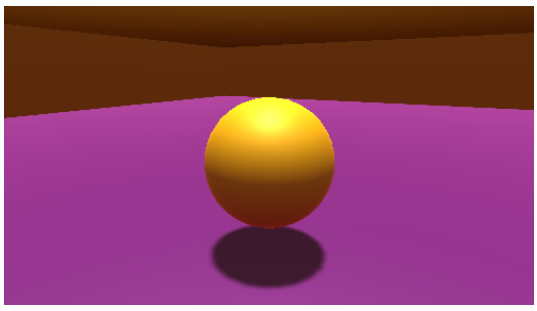

Figura 3.6: Objeto `agente` en Unity

[2]De igual modo que la guía de instalación, los dos primeros experimentos han sido realizados haciendo uso del curso [3]

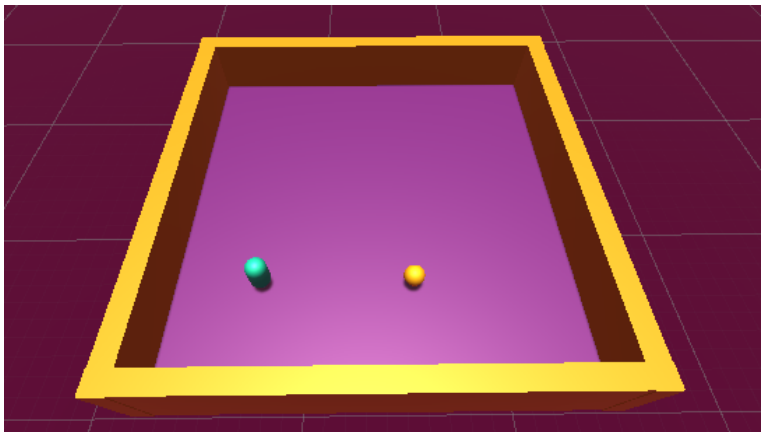

Figura 3.7: Entorno para el experimento 1

El entorno, además del agente y el ítem, está compuesto por dos tipos de objetos distintos:

- **El suelo** (`Ground`).
- **Las barreras** (`Barriers`), formada por 4 barreras (`Barrier`).

Esto es importante tenerlo en cuenta cuando aparezcan a lo largo del código del script.

Sin entrar todavía en los componentes de MlAgents, el agente lleva los siguientes componentes funcionales:

- **Rigid Body**.
- **Collider físico**.
- **Collider desencadenador**.

Hacemos uso de los dos tipos de collider para impedir que traspase objetos (con el físico), y reaccione a colisiones por medio del desencadenador (el collider desencadenador debe ser es un poco más grande que el físico para que funcione correctamente).

Una vez presentado el problema, vamos a adentrarnos en el script identificando los componentes secundarios.

Atributos del script

Los atributos del script son los siguientes:

```
[SerializeField]
private float Fmov = 200;

[SerializeField]
private Transform Target;

[SerializeField]
private GameObject Ground;

[SerializeField]
[Range(0f, 1f)]
private float MulFactor;

[SerializeField]
private bool Training = true;

Bounds areaBounds;
Rigidbody rb;
}
```

Como podemos ver, a excepción de las barreras (`areaBounds`) y del Rigidbody del agente (`rb`), que son simples variables globales, los atributos son configurables desde la interfaz del propio editor de Unity.

En el código podemos identificar los siguientes atributos:

- `Fmov`: Fuerza que se le va a ejercer al agente (veremos su aplicación en detalle luego).
- `Target`: Objeto que se identifica para el agente como el objetivo a recolectar (el objeto `item`).
- `Ground`: Identifica al objeto que toma el papel de suelo para el agente (cuyo objeto correspondiente tiene el mismo nombre) y que será necesario para colocar tanto el ítem como el agente en el entorno de forma aleatoria.
- `MulFactor`: El factor de multiplicación sirve para ajustar el rango que dispone el agente y el ítem para la colocación aleatoria. Luego detallaremos el significado de su valor (dentro del rango) cuando lo veamos aplicado. Mantendremos este atributo con **0.5** (50 % del rango).

- `Training`: Sirve para indicar si el agente está o no entrenando (en este caso, `MaxStep = 0`, lo que significa que sus pasos son infinitos).

`MaxStep` está configurado con **10000** pasos como máximo por episodio.

Inicialización

Como paso previo al inicio del primer episodio, y solo ejecutándose una vez durante el entrenamiento, se dispone de la siguiente función (su explicación se detalla en los comentarios):

```
1 public override void Initialize(){
2     //Definimos el Rigidbody del agente y los limites
          del entorno
3     rb = GetComponent<Rigidbody>();
4     areaBounds = Ground.GetComponent<Collider>().bounds;
5
6     //MaxStep forma parte de la clase Agent
7     //Si no esta entrenando los pasos son infinitos,
          osea, no se resetea
8     if(!Training) MaxStep = 0;
9 }
```

Inicialización de episodios

El código correspondiente a la inicialización de los episodios es el siguiente:

```
1  public override void OnEpisodeBegin(){
2
3      //Reseteo agente
4      transform.position = GetRandomSpawnPos();
5      rb.velocity = Vector3.zero;
6      rb.angularVelocity = Vector3.zero;
7
8      //Reseteo item
9      Target.transform.position = GetRandomSpawnPos();
10 }
```

Se hace uso de una función auxiliar llamada `GetRandomSpawnPos()`, que busca una posición para el objeto que la llama donde no se solape con ningún otro objeto, y eso dentro del rango de `areaBounds`:

```
public Vector3 GetRandomSpawnPos()
{
    var foundNewSpawnLocation = false;
    var randomSpawnPos = Vector3.zero;
    while (foundNewSpawnLocation == false)
    {
        //Se saca una posicion x dentro del limite del
           entorno
        var randomPosX = Random.Range(-areaBounds.
           extents.x * MulFactor,
            areaBounds.extents.x * MulFactor);

        //Se saca una posicion z dentro del limite del
           entorno
        var randomPosZ = Random.Range(-areaBounds.
           extents.z * MulFactor,
             areaBounds.extents.z * MulFactor);

        //Ubicamos la posicion en base al suelo
        randomSpawnPos = Ground.transform.position + new
            Vector3(randomPosX, 1f, randomPosZ);
        //Si no hay colisiones con el objeto cuyas
           dimensiones se indica en Vector3
        if (Physics.CheckBox(randomSpawnPos, new Vector3
           (2.5f, 0.01f, 2.5f)) == false)
        {
            foundNewSpawnLocation = true;
        }
    }
    return randomSpawnPos;
}
```

Como podemos ver, el atributo `MulFactor` ajusta el rango inferior y superior de la generación de posibles posiciones del objeto.

Estado

El estado debe contemplar la distancia a la que se encuentra el agente con respecto el ítem.

- **Sensor usado**: `VectorSensor`.
- **Nº de atributos**: 3 (la distancia se descompone en los ejes `x`, `y`, y `z`).

```
public override void CollectObservations(VectorSensor
   sensor){
    //Calcular cuanto nos queda hasta el objetivo
    Vector3 distance = Target.position - transform.
       position;

    //Un vector ocupa 3 atributos por observacion
    sensor.AddObservation(distance.normalized);
}
```

Acciones

Las acciones de las que dispone el agente son las de moverse por el plano XZ (manteniendo a 0 el eje Y al no haber acciones que implique elevaciones del agente). Aunque la fuerza que se le ejerza (`Fmov`) sea predefinida, las acciones controlan esa fuerza constante en sus respectivos ejes. Si se decide en una iteración poner una de las acciones con valor 0, entonces no se ejerce ninguna fuerza en el eje que tenga asignada esta acción.

- **Nº de acciones**: 2.
- **Tipo de la acción 1**: Continua.
- **Tipo de la acción 2**: Continua.

```
public override void OnActionReceived(ActionBuffers
   actions){
    //Construimos un vector con el vector recibido.
    Vector3 move = new Vector3(actions.ContinuousActions
       [0],
    0f, actions.ContinuousActions[1]);

    //Sumamos el vector construido como fuerza
    rb.AddForce(move * Fmov * Time.deltaTime);
}
```

Recompensas

Las recompensas y penalizaciones van a estar gestionadas por las funciones del Collider desencadenador del agente principalmente, usando las funciones `OnTriggerEnter` y `OnTriggerStay`:

```
1  private void OnTriggerEnter(Collider other)
2  {
3      //Si esta en fase de entrenamiento
4      if (Training)
5      {
6          if (other.CompareTag("Item"))
7          {
8              //Si choca con el item gana 1 punto
9              AddReward(1f);
10             EndEpisode();
11         }
12         if (other.CompareTag("Barrier"))
13         {
14             //Si choca con la barrera pierde 0.1 punto
15             AddReward(-0.1f);
16         }
17     }
18
19     else{
20         if (other.CompareTag("Item"))
21         {
22             EndEpisode();
23         }
24     }
25 }
26
27 private void OnTriggerStay(Collider other)
28 {
29     if (Training)
30     {
31         if (other.CompareTag("Barrier"))
32         {
33             //Se va penalizando el tiempo que este
                   chocando con la barrera
34             AddReward(-0.05f);
35         }
36     }
37 }
```

.yaml

El fichero de configuración `.yaml` que utilizaremos durante el experimento es el fichero **bolaml**, el cual tiene la configuración para realizar un aprendizaje mediante **PPO**. El fichero se puede encontrar como **3DBall** en la siguiente dirección de github en el repositorio oficial de MlAgents [14]:

https://github.com/Unity-Technologies/ml-agents/tree/develop/config/ppo

```yaml
behaviors:
  bolaml:
    trainer_type: ppo
    hyperparameters:
      batch_size: 64
      buffer_size: 12000
      learning_rate: 0.0003
      beta: 0.001
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 128
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 5
    max_steps: 10000000
    time_horizon: 1000
    summary_freq: 12000
    threaded: true
```

Resultados

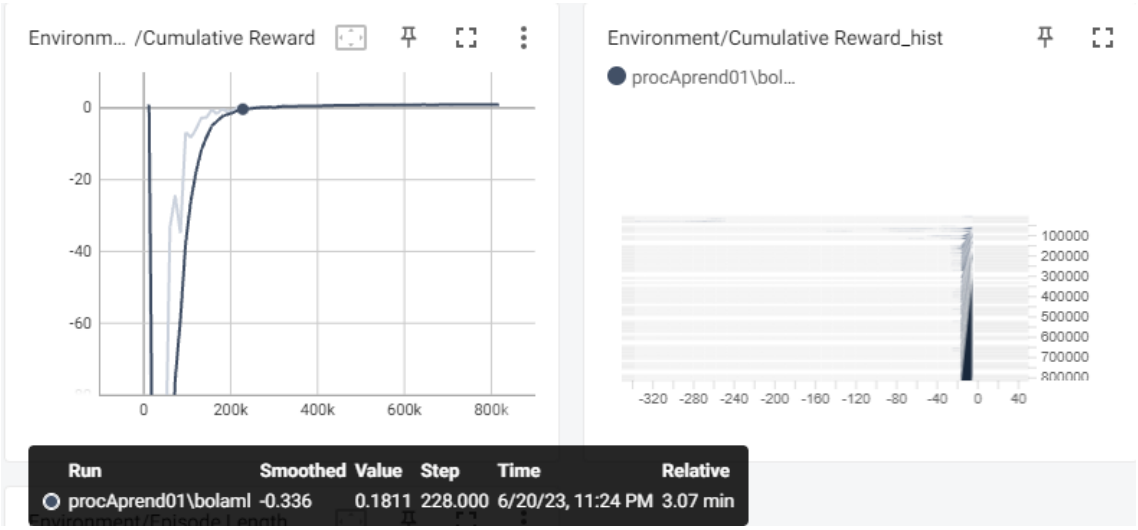

Figura 3.8: Recompensa acumulada del entrenamiento 1

Con una paralelización de **21** entrenamientos/entornos dentro del escenario, el entrenamiento ha tenido una mejora creciente durante los primeros 3 minutos para luego mantenerse en un valor cercano a 0.

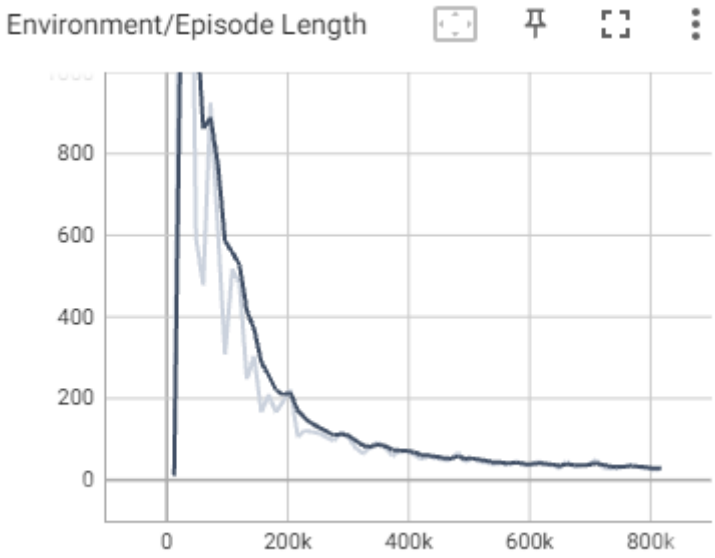

Figura 3.9: Peso del experimento 1

El tiempo de los episodios ha decrementado de forma considerable, y a las 800 mil iteraciones el número de iteraciones por episodio es considerablemente reducido.

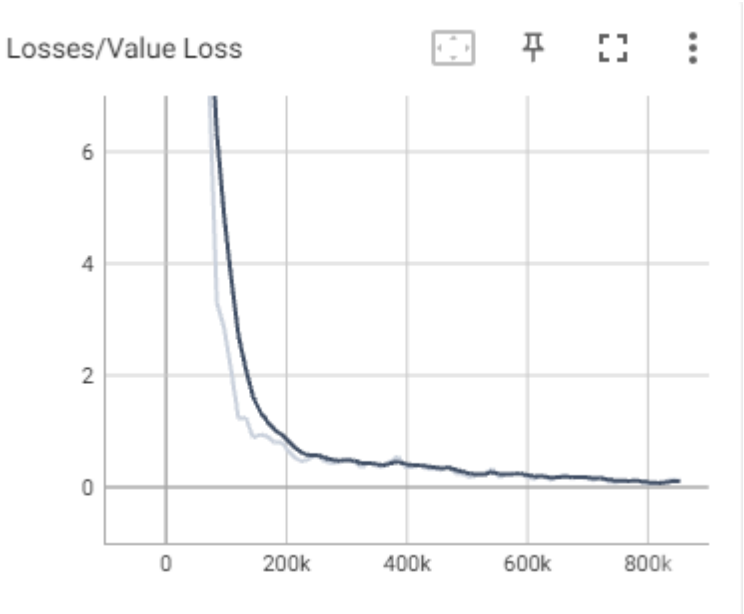

Figura 3.10: Pérdidas del experimento 1

La función de pérdida muestra un valor estimado que decrece junto con el peso de los episodios, lo que significa que la toma de decisiones va mejorando hasta un punto donde la política actual $\pi \approx \pi_*$.

Con las gráficas podemos concluir que este experimento ha sido satisfactorio en un tiempo considerablemente bajo ($\approx$ 3 minutos). Si los entrenamientos hechos en PC1 son exitosos a nivel temporal y de eficiencia, en PC2 pueden ser mejorados. Como nuestro objetivo es hallar los límites de cada computador descartaremos el repetir estos entrenamientos en PC2.

3.4.2. Experimento 2: Recolecta simple

Para este experimento, partimos del experimento 1, pero al agente se le dota de nuevas acciones, observaciones y de una apariencia física más compleja que una simple bola.

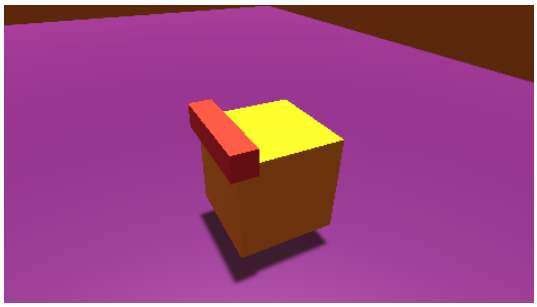

Figura 3.11: Versión 2 del agente

Así, podemos saber la orientación del agente, además de que esta vez las acciones que toma son más cercanas a los movimientos que haría un NPC sin animación procedural.

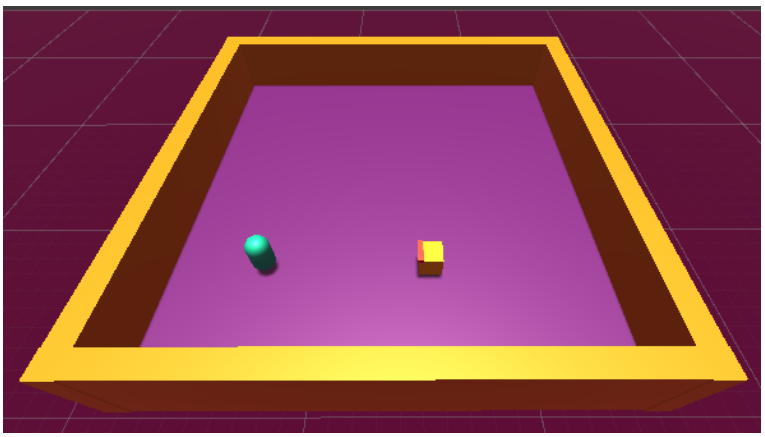

Figura 3.12: Entorno para el experimento 2

Atributos del script

Los atributos de los que dispone el agente ahora son los siguientes:

```
1 [SerializeField]
2 private Transform Target;
3
4 [SerializeField]
5 private GameObject Ground;
6
7 [Header("Factor multiplicador de spawn")]
8 [SerializeField]
9 [Range(0f, 1f)]
```

```
10 private float MulFactor;
11
12 [Header("Velocidad")]
13 [SerializeField]
14 [Range(0f, 3000f)]
15 public float Speed;
16
17 [Header("Velocidad de giro")]
18 [SerializeField]
19 [Range(50f, 300f)]
20 public float TurnSpeed;
21
22 [SerializeField]
23 private bool Training = true;
24
25 Bounds areaBounds;
26 Rigidbody rb;
```

Los nuevos atributos, que han sustituido al atributo `Fmov`, son:

- `Speed`: Velocidad de avance.
- `TurnSpeed`: Velocidad de giro.

Acciones

- **Nº de acciones**: 2.
- **Tipo de la acción 1**: Discreta [0,1].
- **Tipo de la acción 2**: Discreta [0,2].

La configuración de las acciones en el script es ahora la siguiente:

```
1 public override void OnActionReceived(ActionBuffers
     actions)
2 {
3     //Esta variable determina si el agente da un paso o
        no
4     float lForward = actions.DiscreteActions[0];
5
6     //Esta variable determina si el agente gira, y en
        caso de que si, en que sentido.
7     float lTurn = 0;
```

```
8
9       if(actions.DiscreteActions[1] == 1)
10      {
11          lTurn = -1;
12      }
13      else if(actions.DiscreteActions[1] == 2)
14      {
15              lTurn = 1;
16      }
17
18      rb.MovePosition(transform.position +
19          transform.forward * lForward * Speed * Time.
               deltaTime);
20      transform.Rotate(transform.up * lTurn * TurnSpeed *
           Time.deltaTime);
21
22      //Penaliza por cada paso que da
23      if(Training) AddReward(-1f / MaxStep);
24  }
```

Hay que destacar que ahora se realizará una penalización por cada iteración que se realice en el episodio. De esta forma se verá premiada la velocidad de resolución de la tarea.

Estado

El estado debe ahora contemplar los siguientes atributos:

- Distancia del agente con respecto al item (**distancia escalar**).
- La dirección a la que se va al ítem (ejes X, Y, Z).
- La orientación del agente (ejes X, Y, Z).

- **Sensor usado**: `VectorSensor`.
- **Nº de atributos**: 7.

```
1   public override void CollectObservations(VectorSensor
       sensor)
2   {
3       //Distancia al target
4       //Float de 1 posicion
```

```
5       sensor.AddObservation(
6           Vector3.Distance(Target.transform.position,
                transform.position));
7
8       //Direccion al target
9       //Vector 3 posiciones
10      sensor.AddObservation((Target.transform.position -
            transform.position).normalized);
11
12      //Vector de orientacion del agente
13      //Vector de 3 posiciones
14      sensor.AddObservation(transform.forward);
15  }
```

Heurística

Aunque no nos sirva para este experimento, disponemos de una función heurística, donde podemos proporcionar las acciones que toma el agente:

```
1   public override void Heuristic(in ActionBuffers
        actionsOut)
2   {
3       var discreteActionsOut = actionsOut.DiscreteActions;
4       if(Input.GetKey(KeyCode.D))
5       {
6           discreteActionsOut[0] = 3;
7       }
8       else if(Input.GetKey(KeyCode.W))
9       {
10          discreteActionsOut[0] = 1;
11      }
12      else if (Input.GetKey(KeyCode.A))
13      {
14          discreteActionsOut[0] = 4;
15      }
16      else if (Input.GetKey(KeyCode.S))
17      {
18          discreteActionsOut[0] = 2;
19      }
20  }
```

Resultados

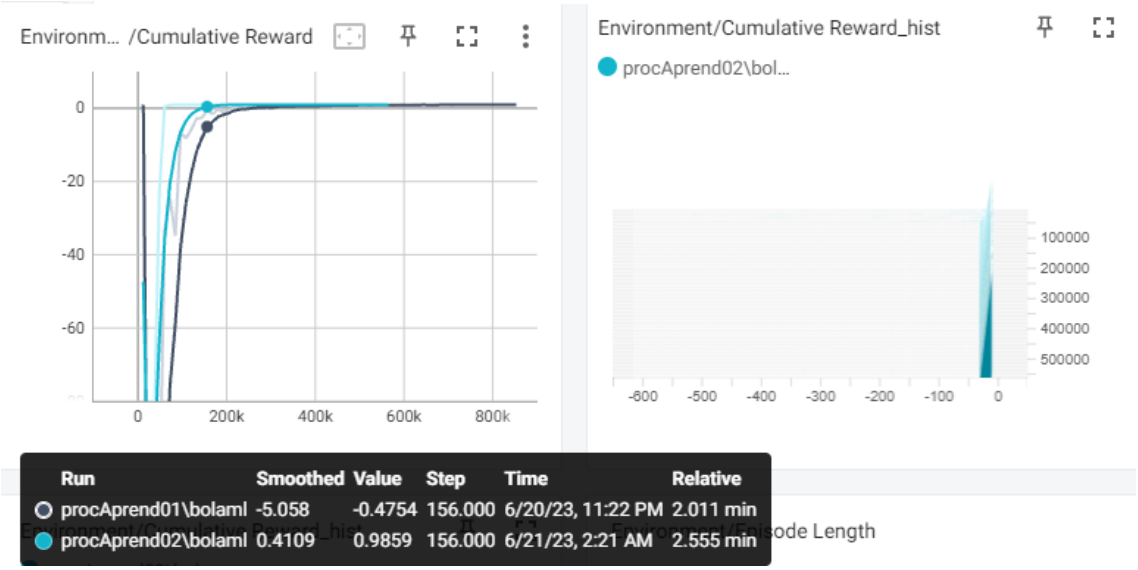

Figura 3.13: Recompensa acumulada del experimentos 1 y 2

La evolución que sigue este segundo experimento es parecida a la obtenida en el primero, solo que gracias a las mejoras del agente el tiempo de aprendizaje se ha reducido a 2 minutos. Esto es debido a que las acciones y observaciones, pese a ser más complejas, conllevan a una mayor precisión a la hora de realizar la tarea de recolección.

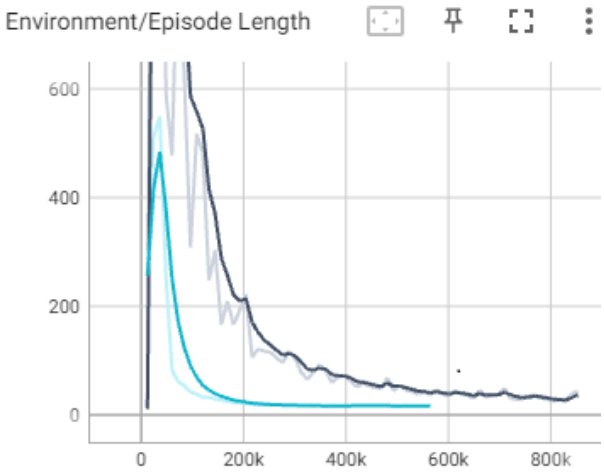

Figura 3.14: Peso de los experimentos 1 y 2

El peso afirma de forma más notable la afirmación anterior. Podemos ver que, además de partir con un número de iteraciones por episodio menor que el experimento 1, su aproximación a 0 se realiza de forma mucho más clara y limpia.

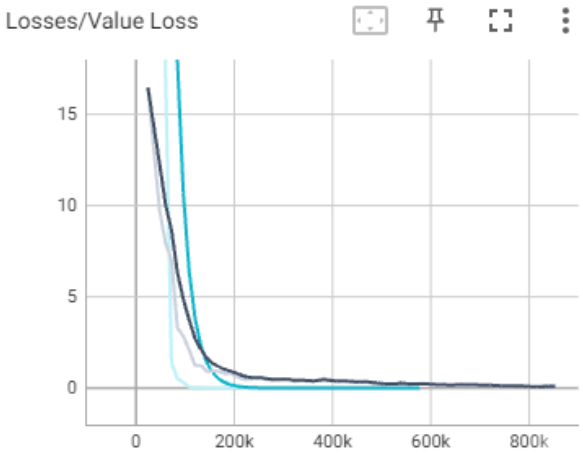

Figura 3.15: Pérdidas del experimento 1 y 2

En la gráfica de la función de pérdida, podemos observar que el segundo experimento tiene una reducción más pronunciada. Pese a aparentar que comienza con un valor mucho mayor que el primer experimento, se ve que en pocas iteraciones llega antes al valor aproximado a 0.

Como conclusión de este segundo experimento, con el nuevo agente somos capaces de realizar la tarea de recolección con una mejora considerable, además de dotar al agente de un comportamiento más realista.

3.4.3. Experimento 3: Recolecta simple con Ray Perception Sensor 3D

El tercer experimento que vamos a realizar solo va a tener una variación con respecto al experimento 2, y es el sensor. Vamos a realizar la recolección de los atributos del estado esta vez con `Ray Perception Sensor 3D`.

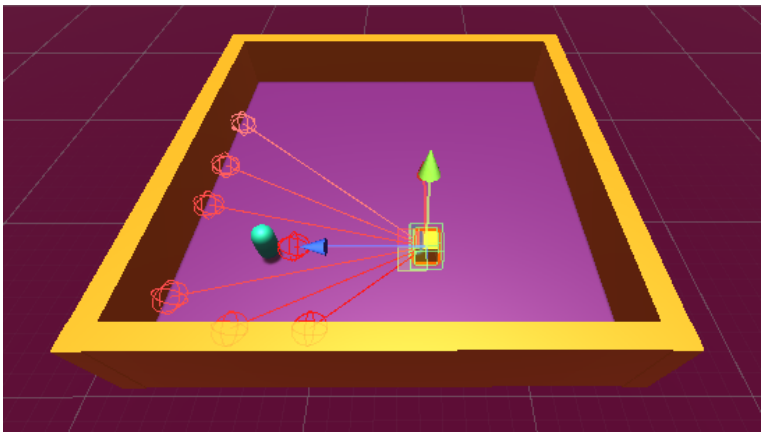

Figura 3.16: Entorno para el experimento 3

Estado

- **Sensor usado**: `Ray Perception Sensor 3D`.
- **Nº de atributos**: 7.

A diferencia de `VectorSensor`, con `Ray Perception Sensor 3D` no debemos de preocuparnos por saber el número de atributos que tiene una observación. Solo es necesario especificar al sensor los objetos que debe identificar. La identificación de los objetos, como mencionamos en el capítulo anterior, se hace mediante el uso de etiquetas, las cuales hemos colocado a cada objeto del entorno, y cuyo nombre coincide con el objeto para menor confusión. Los objetos que debemos hacer que detecte el sensor son:

- El ítem.
- Las barreras.

Que podamos identificar varios objetos el mismo tipo sin necesidad de especificar cada uno en la función de observación del script es la razón por la que hemos equipado al sensor para que pueda detectar las barreras. Esto permite prever colisiones con estas en caso de que el agente haya recibido un mínimo de penalización desde ellas. Por todo ello, la función `CollectObservations` se puede quitar del script.

Resultados

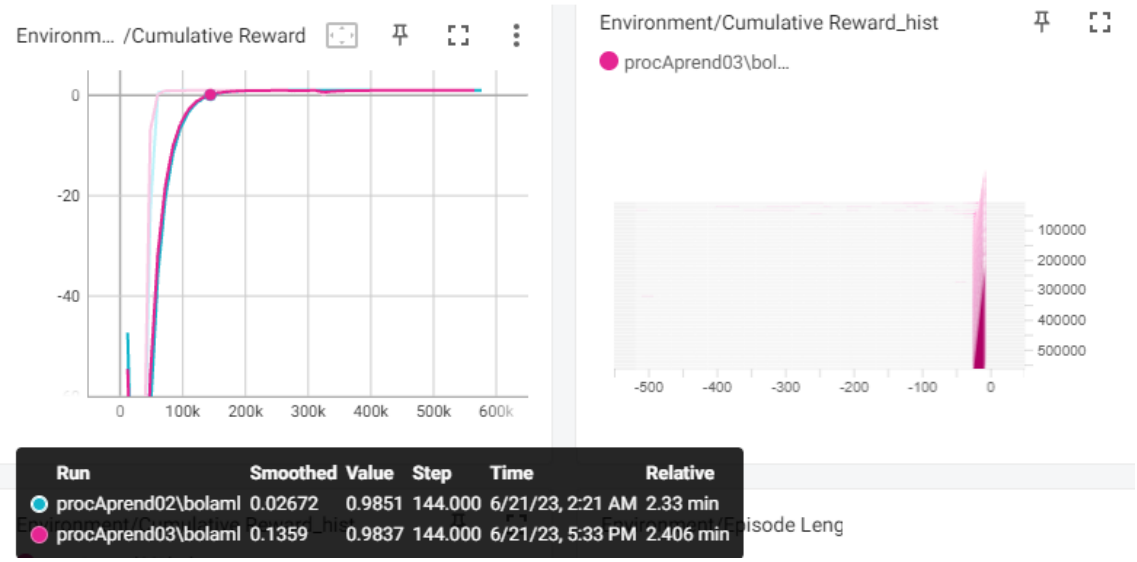

Figura 3.17: Recompensa acumulada del entrenamiento 2 y 3

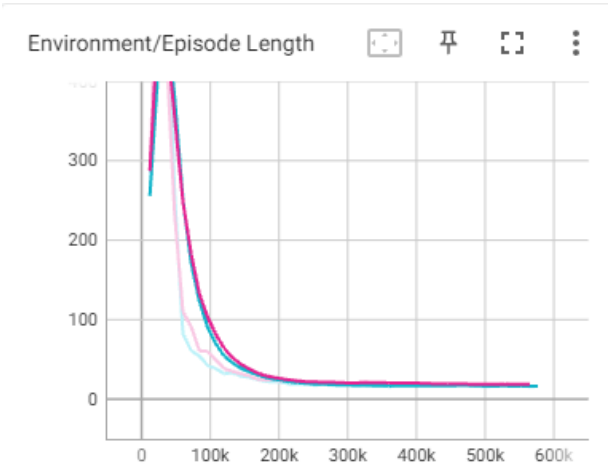

Figura 3.18: Peso de los entrenamientos 2 y 3

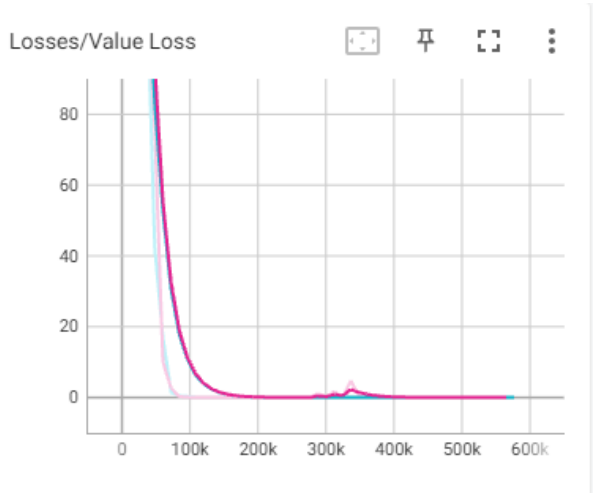

Figura 3.19: Pérdidas del entrenamiento 2 y 3

Como vemos en las gráficas, la variación que tiene con respecto al experimento anterior es insignificante. El uso de un sensor u otro no provoca una variación notable. Sí consideramos para futuros experimento que el uso de `Ray Perception Sensor 3D` podrá abordar de forma más realista los entornos, ya que en entornos más complejos puede ser que el agente a simple vista no vea el ítem por haber una pared delante; entonces optamos por usar `Ray Perception Sensor 3D` para los siguientes experimento.

3.4.4. Experimento 4: Abrir puerta

El cuarto experimento, como hemos hecho hasta ahora, parte del experimento anterior (experimento 3).

En este experimento vamos a añadir complejidad a la tarea de recolección de ítem, cambiando el objeto `item` por los siguientes dos objetos:

- El objeto **llave** (`Key`).
- El objeto **puerta** (`Door`).

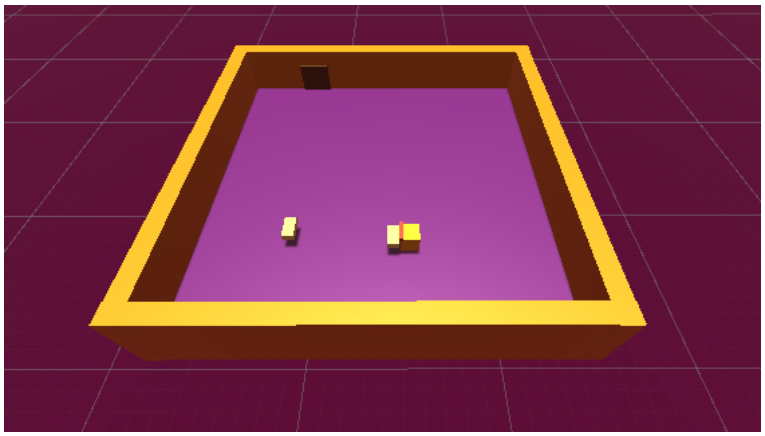

Figura 3.20: Entorno para el experimento 4

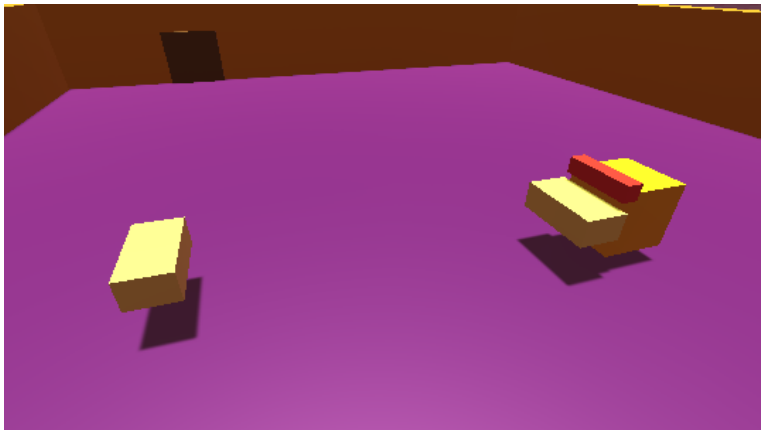

Figura 3.21: Agente con llave, y objetos `Key` y `Door`

En la Figura 3.21 podemos visualizar la puerta en la barrera del fondo, mientras que la llave es el objeto rectangular de color amarillo a la izquierda del agente. También podemos ver que el agente tiene otra llave, pero esta solo funciona como objeto que indica de forma gráfica que el agente ya ha recogido la llave.

La tarea que debe realizar el agente esta vez es recolectar la llave para posteriormente abrir la puerta y alcanzar la puerta.

Esta vez hay dos recolecciones, por lo que el agente debe ser capaz de priorizar la recolecta de la llave.

Para un mejor entrenamiento esta vez vamos, además de variar la posición inicial del agente y del ítem (la llave, en este caso), a rotar el entorno para así variar la posición de la puerta.

Atributos del script

Para poder modificar los ajustes básicos del agente modularmente, tenemos ya creados varios entornos para el entrenamiento en paralelo (`MulFactor`, `Speed`, `TurnSpeed`), hemos reunido tales atributos en un script aparte añadido como componente de un nuevo objeto invisible llamado `AgentSettings`, al cual llamaremos desde el script de `MlAgents`:

```
using UnityEngine;

public class AgentSettings : MonoBehaviour
{
[Header("Velocidad")]
[SerializeField]
public float AgentRunSpeed;

[Header("Velocidad de giro")]
[SerializeField]
public float AgentRotationSpeed;

[Header("Factor multiplicador de spawn")]
[SerializeField]
public float SpawnAreaMarginMultiplier;
}
```

Los atributos presentes en el script son ahora los siguientes:

```
[SerializeField]
private bool Training = true;
[SerializeField]
private GameObject MyKey;
[SerializeField]
private GameObject Key;
[SerializeField]
private GameObject Environment;
[SerializeField]
private GameObject Ground;

private AgentSettings M_AgentSettings; //Ajustes
Bounds areaBounds;
Rigidbody rb;
bool IHaveAKey = false;
```

Donde los nuevos atributos son:

- `MyKey`: La llave del agente.
- `Key`: La llave a recolectar.
- `Environment`: El objeto entorno al completo, y nos sirve para poder rotarlo en los inicios de episodio.
- `IHaveAKey`: Indica si la llave ha sido o no recolectada.

Inicialización

La función de inicialización queda ahora de la siguiente forma:

```
public override void Initialize()
{
    M_AgentSettings = FindObjectOfType<AgentSettings>();
    areaBounds = Ground.GetComponent<Collider>().bounds;
    rb = GetComponent<Rigidbody>();
    MyKey.SetActive(false);
    IHaveAKey = false;

    if (!Training) MaxStep = 0;
    }
```

Inicialización de episodios

La inicialización de episodios queda ahora de la siguiente forma:

```
public override void OnEpisodeBegin()
{
    //Reseteo propiedades
    MyKey.SetActive(false);
    IHaveAKey = false;

    //Reseteo entorno
    var rotation = Random.Range(0, 4);
    var rotationAngle = rotation * 90f;
    Environment.transform.Rotate(new Vector3(0f,
       rotationAngle, 0f));

    //Reseteo llave
    Key.transform.position = GetRandomSpawnPos();
    Key.SetActive(true);
```

```
15
16      //Reseteo agente
17      transform.position = GetRandomSpawnPos();
18      rb.velocity = Vector3.zero;
19      rb.angularVelocity = Vector3.zero;
20  }
```

La función `GetRandomSpawnPos` se ve afectada por el `AgentSettings` de la siguiente forma:

```
1  public Vector3 GetRandomSpawnPos()
2  {
3      var foundNewSpawnLocation = false;
4      var randomSpawnPos = Vector3.zero;
5
6      //Mientras no se haya encontrado una posicion
7      while (foundNewSpawnLocation == false)
8      {
9          //Se saca una posicion x dentro del limite del
               entorno
10         var randomPosX = Random.Range(-areaBounds.
               extents.x * M_AgentSettings.
               SpawnAreaMarginMultiplier,
11             areaBounds.extents.x * M_AgentSettings.
                   SpawnAreaMarginMultiplier);
12
13         //Se saca una posicion z dentro del limite del
               entorno
14         var randomPosZ = Random.Range(-areaBounds.
               extents.z * M_AgentSettings.
               SpawnAreaMarginMultiplier,
15             areaBounds.extents.z * M_AgentSettings.
                   SpawnAreaMarginMultiplier);
16
17         //Ubicamos la posicion en base al suelo
18         randomSpawnPos = Ground.transform.position + new
               Vector3(randomPosX, 1f, randomPosZ);
19
20         //Si no hay colisiones con el objeto cuyas
               dimensiones se indica en Vector3
21         if (Physics.CheckBox(randomSpawnPos, new Vector3
               (2.5f, 0.01f, 2.5f)) == false)
22         {
23             foundNewSpawnLocation = true;
```

```
24          }
25      }
26      return randomSpawnPos;
27  }
```

Estado

- **Sensor usado**: `Ray Perception Sensor 3D`.
- **Nº de atributos**: 11.

Tendríamos que definir 11 atributos distintos: por cada objeto que tenga que alcanzar, el agente necesita un total de 4 atributos (distancia escalar y dirección x,y,z), y otros tres atributos para definir la orientación del agente. Como tenemos 2 objetos, 4 x 2 = 8, que sumado con los 3 del agente dan 11.

Realmente, como hemos comentado, al usar `Ray Perception Sensor 3D` no es necesario saber el número de atributos, solo los tipos de objetos que debe detectar el agente. En este caso los objetos detectables son:

- La llave.
- Las barreras.
- La puerta.

Acciones

Las acciones no han sido modificadas, pero sí el código debido al uso de `AgentSettings`:

```
1  public override void OnActionReceived(ActionBuffers
       actions)
2  {
3      float lForward = actions.DiscreteActions[0];
4
5      float lTurn = 0;
6
7      if(actions.DiscreteActions[1] == 1)
8      {
9          lTurn = -1;
10     }
```

```
    else if(actions.DiscreteActions[1] == 2)
    {
        lTurn = 1;
    }
    rb.MovePosition(transform.position +
        transform.forward * lForward * M_AgentSettings.
           AgentRunSpeed * Time.deltaTime);
    transform.Rotate(transform.up * lTurn *
       M_AgentSettings.AgentRotationSpeed * Time.
       deltaTime);

    if(Training) AddReward(-1f / MaxStep);
}
```

Recompensas

Los colliders quedan ahora de la siguiente forma:

```
void OnTriggerEnter(Collider col)
{
    if(Training){
    //Si el agente se encuentra la llave, este la coge
        if (col.CompareTag("Key"))
        {
            if(Training) AddReward(5f);
            MyKey.SetActive(true);
            IHaveAKey = true;
            col.gameObject.SetActive(false);
        }

        if (col.CompareTag("Door"))
        {
            if (IHaveAKey) //Se abre la puerta
            {
                MyKey.SetActive(false);
                IHaveAKey = false;
                if(Training) AddReward(5f);
                EndEpisode();
            }
        }
    }
}

```

```
26 private void OnTriggerStay(Collider other)
27 {
28     if (Training)
29     {
30         if (other.CompareTag("Barrier"))
31         {
32             AddReward(-0.05f);
33         }
34     }
35 }
```

Resultados

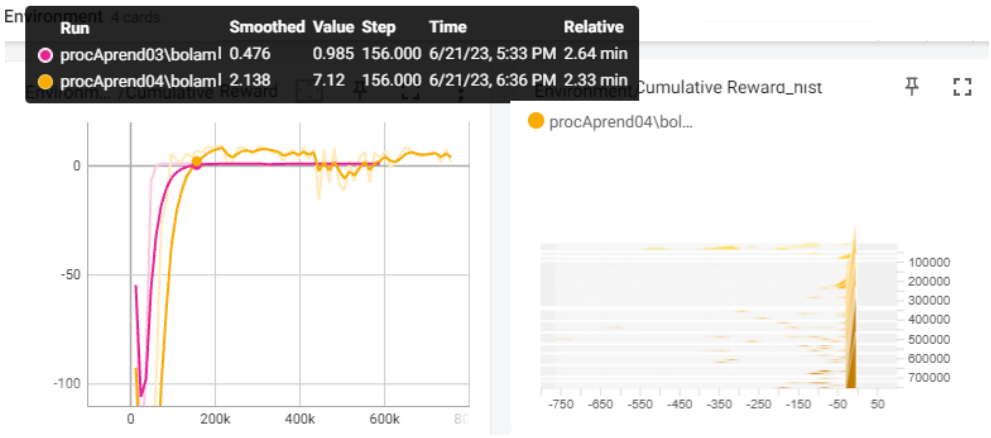

Figura 3.22: Recompensa acumulada del entrenamiento 3 y 4

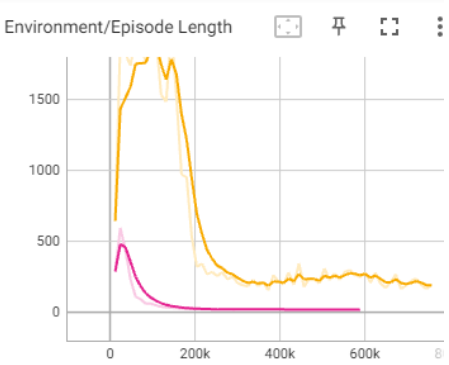

Figura 3.23: Peso de los entrenamientos 3 y 4

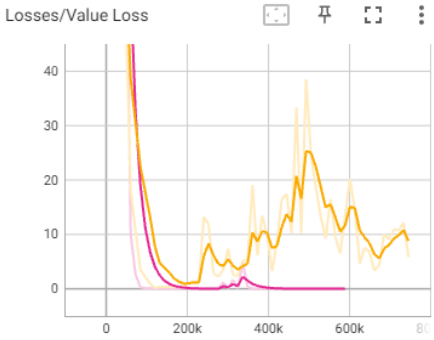

Figura 3.24: Pérdidas del entrenamiento 3 y 4

Como vemos en las gráficas, el entrenamiento esta vez no ha sido tan simple como en los casos anteriores, debiéndose al considerable aumento de complejidad que le hemos metido en este experimento.

Tanto para la recompensa acumulada como para el peso de los episodios se ha visto un empeoramiento, justificado por la complejidad, pero demostrando cómo una simple variación de la tarea, como es la recolección por prioridad de dos ítems, puede variar los resultados finales.

En la gráfica de las pérdidas se presenta un empeoramiento considerable a partir de la iteración 400 mil, lo que implica que los contratiempos en el entrenamiento sucedan con más frecuencia. Pese a ello, se ve igualmente que por la iteración 500 mil empieza a mejorar de nuevo.

Como conclusión a este experimento, pese al empeoramiento debido al aumento de complejidad, su eficiencia sigue siendo aceptable, no siendo un entrenamiento óptimo, pero sí satisfactorio, si consideramos además que el tiempo sigue rondando los 3 minutos, tiempo medio de espera que hemos contemplado hasta ahora.

3.4.5. Experimento 5: Recolecta tras puerta

Este experimento va a añadir un poco más de complejidad al caso anterior. Ahora, tras llegar a la puerta, `hay una habitación extra` que queda disponible tras sobrepasarla, donde se va a ubicar de forma aleatoria un ítem como el que vimos en los experimentos 1,2 y 3.

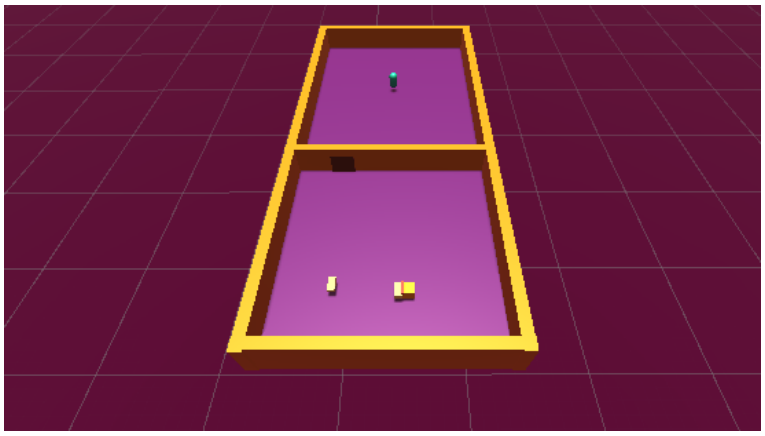

Figura 3.25: Entorno para el experimento 5

Debemos de destacar que el suelo de la habitación es un nuevo objeto tipo `Room`, que hemos creado independiente del otro suelo (`Ground`) para separar las generaciones del agente y la llave de la generación del ítem.

En este caso, no vamos a rotar el entorno, y además para dar más tiempo de resolución hemos aumentado el valor de `MaxStep` a **100000**.

Atributos del script

En este caso solo vamos a tener como atributo adicional, en comparación al caso anterior, el objeto Room:

```
1 [SerializeField]
2 private GameObject Room;
```

Inicialización

La inicialización se diferencia en que no inicializa `areaBounds`. Debido a que esto se hace en la función `GetRandomSpawnPos(Ground)`, ahora necesita especificar en qué suelo se debe generar el objeto que lo llame:

```
1 public Vector3 GetRandomSpawnPos(GameObject Base)
2 {
3     areaBounds = Base.GetComponent<Collider>().bounds;
4     var foundNewSpawnLocation = false;
5     var randomSpawnPos = Vector3.zero;
6
7     //Mientras no se haya encontrado una posicion
```

```
8     while (foundNewSpawnLocation == false)
9     {
10        //Se saca una posicion x dentro del limite del
              entorno
11        var randomPosX = Random.Range(-areaBounds.
              extents.x * M_AgentSettings.
              SpawnAreaMarginMultiplier,
12            areaBounds.extents.x * M_AgentSettings.
                  SpawnAreaMarginMultiplier);
13
14        //Se saca una posicion z dentro del limite del
              entorno
15        var randomPosZ = Random.Range(-areaBounds.
              extents.z * M_AgentSettings.
              SpawnAreaMarginMultiplier,
16            areaBounds.extents.z * M_AgentSettings.
                  SpawnAreaMarginMultiplier);
17
18        //Ubicamos la posicion en base al suelo
19        randomSpawnPos = Base.transform.position + new
              Vector3(randomPosX, 1f, randomPosZ);
20
21        //Si no hay colisiones con el objeto cuyas
              dimensiones se indica en Vector3
22        if (Physics.CheckBox(randomSpawnPos, new Vector3
              (2.5f, 0.01f, 2.5f)) == false)
23        {
24            foundNewSpawnLocation = true;
25        }
26    }
27    return randomSpawnPos;
28 }
```

Inicialización de episodios

Teniendo en cuenta lo comentado en el apartado de incialización, y que ahora debemos restaurar el ítem, el comienzo de episodio es ahora:

```
1 public override void OnEpisodeBegin()
2 {
3     //Reseteo propiedades
4     MyKey.SetActive(false);
5     IHaveAKey = false;
```

```
6
7       //Reseteo llave
8       Key.transform.position = GetRandomSpawnPos(Ground);
9       Key.SetActive(true);
10
11      //Reseteo Item
12      Target.transform.position = GetRandomSpawnPos(Room);
13
14      //Reseteo Agente
15      transform.position = GetRandomSpawnPos(Ground);
16      rb.velocity = Vector3.zero;
17      rb.angularVelocity = Vector3.zero;
18
19      Door.SetActive(true);
20  }
```

Estado

Al tener ahora al ítem de nuevo en el entorno, debemos configurar `Ray Perception Sensor 3D` para que lo identifique como `Item`.

Recompensas

A diferencia de los anteriores, esta vez hemos prescindido de `OnTriggerStay`. Esto se debe a que, si se utiliza, el agente no pasará a la habitación del ítem debido a que ahora debe cruzar una barrera a través del hueco de la puerta.

```
1  void OnTriggerEnter(Collider col)
2  {
3      //Si el agente se encuentra la llave, este la coge
4      if (col.CompareTag("Key"))
5      {
6          if(Training) AddReward(20f);
7          MyKey.SetActive(true);
8          IHaveAKey = true;
9          col.gameObject.SetActive(false);
10     }
11
12     if (col.CompareTag("Door"))
13     {
14         if (IHaveAKey) //Se abre la puerta
15         {
```

```
16              MyKey.SetActive(false);
17              IHaveAKey = false;
18              if(Training) AddReward(30f);
19              Door.SetActive(false);
20          }
21      }
22
23      if (col.CompareTag("Item"))
24      {
25          if(Training) AddReward(50f);
26          EndEpisode();
27      }
28  }
```

Resultados

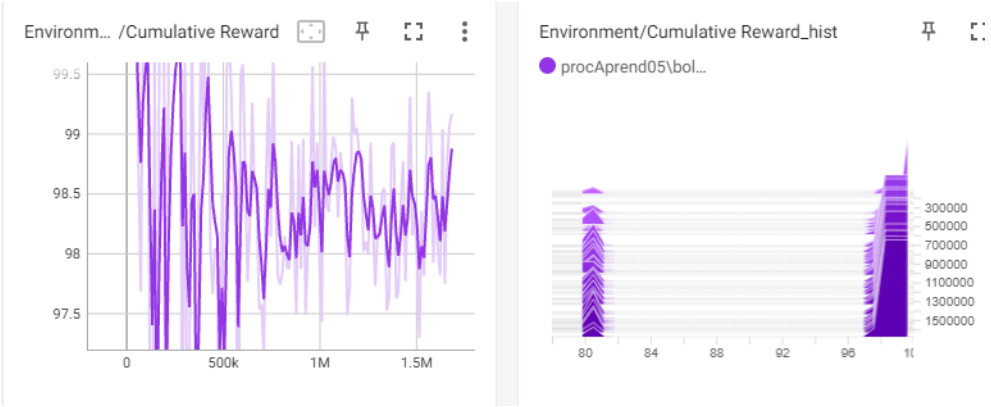

Figura 3.26: Recompensa acumulada del experimento 5

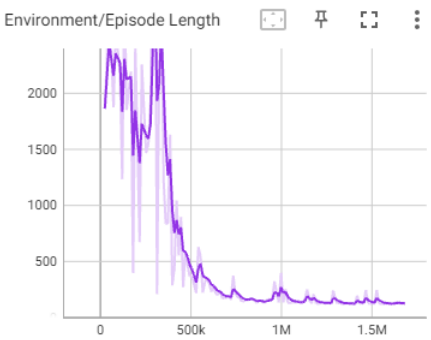

Figura 3.27: Peso del experimento 5

Figura 3.28: Pérdidas del experimento 5

Como podemos ver en las gráficas, esta vez no hemos comparado el caso con el experimento anterior debido a que la complejidad y transmisión de recompensa/penalización han sido modificadas.

Podemos ver que la recompensa acumulada se altera muchísimo más a lo largo de las iteraciones. Pese a ello, a partir de las 500 mil iteraciones podemos notar que se mantiene en unos valores aceptables entre 98 y 99 puntos de recompensa.

El peso sorprendentemente se reduce de forma considerable decreciendo tal y como hemos visto en los experimentos 1, 2 y 3.

Las pérdidas también se ven reducidas pese al crecimiento considerable que tiene entre las iteraciones 500 mil y 1 millón.

Como conclusión, podemos decir que a pesar del aumento de complejidad da un resultado aceptable y eficiente. Su tiempo se ve incrementado considerablemente en comparación con los anteriores, si tenemos en cuenta que la

duración del entrenamiento ha sido de, más o menos, entre 15 y 30 minutos. Es un tiempo y un número de iteraciones que puede ser ya cuestionable si queremos comportamientos entrenados en menos tiempo, pero es un tiempo abordable desde un punto de vista práctico.

3.4.6. Experimento 6: Recolecta dentro o fuera

Hasta ahora, para aumentar la complejidad de la tarea hemos estado añadiendo objetos nuevos que recolectar de forma consecutiva. En el experimento 6 vamos a mantener todos los elementos de forma conceptual, teniendo una llave, una puerta, una habitación y un ítem. La diferencia va a estar ahora en que el ítem podrá estar en la habitación **o bien fuera de esta**. Esto significa que si el ítem no está dentro de la habitación, entonces el agente debe buscarlo en su entorno **ignorando la llave y la puerta**.

Ahora volvemos a tener un solo suelo (`Ground`), eliminando así el objeto `Room`. En su lugar vamos a tener un objeto invisible que hace de barrera y delimita el terreno de la habitación, cuyo nombre es el mismo que su objeto predecesor (`Room`).

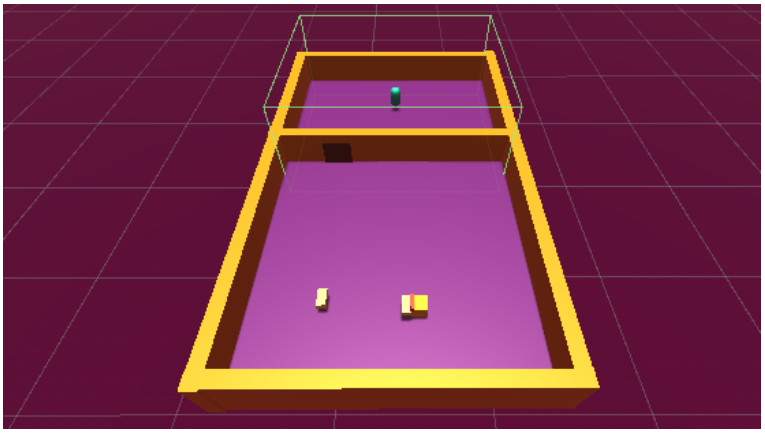

Figura 3.29: Entorno para el experimento 6

Para mayor libertad de pasos por episodios hemos aumentado `MaxStep` a **500000**.

Atributos del script

Los atributos en este entrenamientos son los siguientes:

```
[SerializeField]
private bool Training = true;
[SerializeField]
private GameObject MyKey;
[SerializeField]
private GameObject Key;
[SerializeField]
private GameObject Target;
[SerializeField]
private GameObject Ground;
[SerializeField]
private GameObject Door;

private AgentSettings M_AgentSettings; //Ajustes
Bounds areaBounds;
Rigidbody rb;
bool IHaveAKey = false;
bool InRoom;
```

Donde el atributo para identificar a la habitación (`Room`) se ha quitado, y ahora tenemos además de los atributos del caso anterior el atributo `InRoom`, que servirá para indicar que el ítem está dentro o fuera de la habitación.

Inicialización de episodios

El código de la inicialización de los episodios queda de ahora de la siguiente forma:

```
public override void OnEpisodeBegin()
{
    //Reseteo propiedades
    MyKey.SetActive(false);
    IHaveAKey = false;

    //Reseteo llave
    Key.transform.position = GetRandomSpawnPos();
    Key.SetActive(true);

    //Reseteo Agente
    transform.position = GetRandomSpawnPos();
```

```
13      rb.velocity = Vector3.zero;
14      rb.angularVelocity = Vector3.zero;
15
16      //ReseteoItem
17      Target.transform.position = GetRandomSpawnPosItem();
18      InRoom = Physics.CheckBox(Target.transform.position,
            new Vector3(2.5f, 0.01f, 2.5f), transform.
           rotation, LayerMask.GetMask("Rooml"));
19
20      Door.SetActive(true);
21  }
```

Donde, además de quitar el parámetro que se le añadió a la función de spawn, ahora hacemos uso de **dos** funciones de spawn distintas: una para los objetos que no deban ir dentro de la habitación (`GetRandomSpawnPos`), y otra para el ítem, que puede generarse tanto dentro como fuera (`GetRandomSpawnPosItem()`).

La función de `GetRandomSpawnPos` tiene la misma estructura que hemos visto en la mayoría de experimentos, pero `GetRandomSpawnPosItem` tiene un par de variaciones clave:

```
1   public Vector3 GetRandomSpawnPosItem()
2   {
3       //Mascara para la capa numero 6 (Rooml)
4       int layerMask = ~(1 << 6);
5
6       areaBounds = Ground.GetComponent<Collider>().bounds;
7       var foundNewSpawnLocation = false;
8       var randomSpawnPos = Vector3.zero;
9       while (foundNewSpawnLocation == false)
10      {
11          //Se saca una posicion x dentro del limite del
                entorno
12          var randomPosX = Random.Range(-areaBounds.
               extents.x * M_AgentSettings.
               SpawnAreaMarginMultiplier,
13              areaBounds.extents.x * M_AgentSettings.
                   SpawnAreaMarginMultiplier);
14
15          //Se saca una posicion z dentro del limite del
                entorno
16          var randomPosZ = Random.Range(-areaBounds.
               extents.z * M_AgentSettings.
               SpawnAreaMarginMultiplier,
```

```
17              areaBounds.extents.z * M_AgentSettings.
                    SpawnAreaMarginMultiplier);
18
19          //Ubicamos la posicion en base al suelo
20          randomSpawnPos = Ground.transform.position + new
                Vector3(randomPosX, 1f, randomPosZ);
21
22          //Si no hay colisiones IGNORANDO a los objetos
                de la capa 6.
23          if (Physics.CheckBox(randomSpawnPos, new Vector3
                (2.5f, 0.01f, 2.5f), transform.rotation,
                layerMask) == false)
24          {
25              foundNewSpawnLocation = true;
26          }
27      }
28      return randomSpawnPos;
29  }
```

La diferencia principal está en el uso de una **capa** (`layer`), que se debe crear en el editor de Unity en la posición que se le indica (línea 6) y atribuirla a los objetos que la usen (como si fuese una etiqueta). De esta forma, aunque se siga comprobando que el ítem no se solapa con ningún objeto, para el caso de aquellos objetos que tengan atribuida la capa 6 (`Roomml`) se permitirá el solape.

Estado

Adicional al estado del caso anterior, vamos a utilizar de forma simultánea de nuevo `VectorSensor`, al cuál vamos a asignarle los atributos `inRoom` y `IhaveAKey`. Esto dotará al agente de mayor conocimiento del problema: saber si el ítem está dentro o fuera; y saber si tiene o no la llave.

```
1  public override void CollectObservations(VectorSensor
       sensor)
2  {
3      sensor.AddObservation(IHaveAKey);
4      sensor.AddObservation(InRoom);
5  }
```

Esto implica que en la configuración de `VectorSensor` hay que indicar que hay 2 atributos por observación. El resto de atributos los controla `Ray Perception Sensor 3D`.

Recompensas

El código de la función de entrada del collider del agente queda ahora de la siguiente forma:

```
void OnTriggerEnter(Collider col)
{
    //Si el agente se encuentra la llave, este la coge
    if (col.CompareTag("Key"))
    {
        //Si esta verdaderamente en la habitacion
        if(InRoom ==true){
            if(Training) AddReward(20f);
            MyKey.SetActive(true);
            IHaveAKey = true;
            col.gameObject.SetActive(false);
        }
        else{
            if(Training) AddReward(-20f);
            MyKey.SetActive(true);
            IHaveAKey = true;
            col.gameObject.SetActive(false);
        }

    }

    if (col.CompareTag("Door"))
    {
        if (IHaveAKey) //Se abre la puerta
        {
            if(InRoom == true){
                MyKey.SetActive(false);
                IHaveAKey = false;
                if(Training) AddReward(30f);
                Door.SetActive(false);
            }

            else{
                MyKey.SetActive(false);
                IHaveAKey = false;
                if(Training) AddReward(-30f);
                Door.SetActive(false);
            }
        }
```

```
40      }
41
42      if (col.CompareTag("Item"))
43      {
44          if(Training) AddReward(50f);
45          EndEpisode();
46      }
47  }
```

Resultados

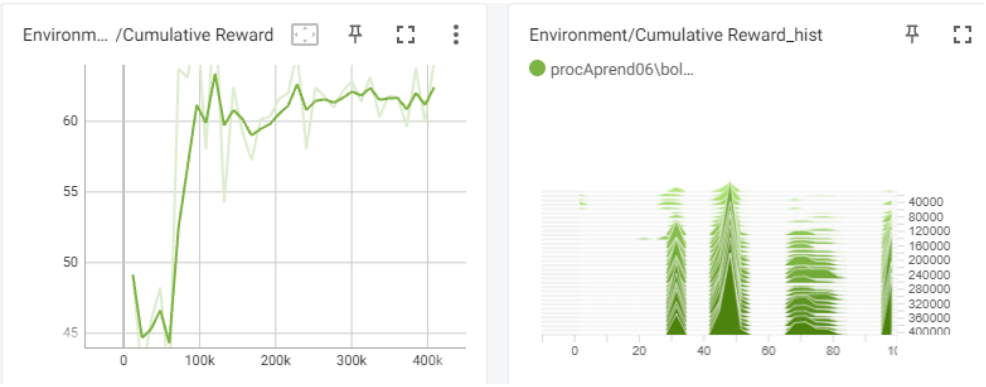

Figura 3.30: Recompensa acumulada del experimento 6

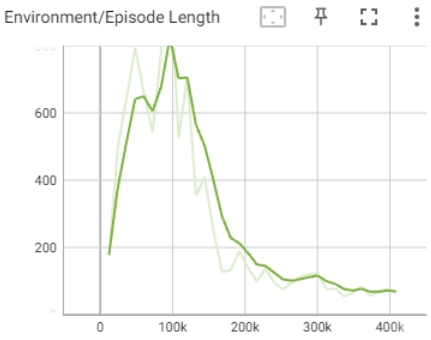

Figura 3.31: Peso del experimento 6

Figura 3.32: Pérdidas del experimento 6

Este experimento, a diferencia de los anteriores, ha sido limitado a más o menos 408.000 iteraciones por los límites de PC1. Pese a ello, como podemos ver en las dos primeras gráficas, tanto la recompensa como el peso mantienen el correcto comportamiento para considerar este un entrenamiento eficiente.

Por parte de la gráfica de pérdidas, hay un incremento considerable a partir de las 100 mil iteraciones, pero manteniéndose posteriormente en el rango de entre los 60 y 70 valores de pérdida. Podemos considerarlo como la franja constante de la función de pérdida.

El tiempo de entrenamiento ronda los 7 minutos, menos que el entrenamiento anterior, pero recordemos que no puede entrenar durante más tiempo sin que se congele `Unity`, por lo que para el apartado de entrenamientos de PC2 volveremos a repetir este entrenamiento para analizar su potencial. Parece que estamos acercándonos al límite de PC1.

3.4.7. Experimento 7: Recolecta en recinto grande

Este experimento parte del experimento 6, y no modifica el script, sino que amplía el entorno de forma que el agente deba explorar más el entorno para cumplir la tarea.

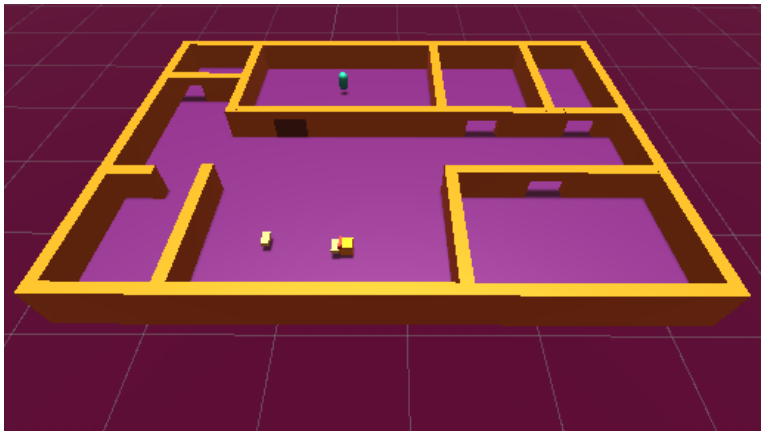

Figura 3.33: Entorno para el experimento 7

Resultados

Ni con un solo entorno responde el editor de `Unity` ante tal entrenamiento, por lo que la capacidad de aprendizaje con un entorno de tal tamaño pese a su mismo código es inabordable para el PC1. Esto nos lleva a un experimento límite del PC1. Repetiremos este experimento en el PC2.

3.4.8. Experimento 8: Recolecta multiItem y multiLlave

Como experimento derivado del experimento 6, en vez de ampliar el entorno como hizo el experimento 7, se aumenta el número de llaves e ítems a recolectar, de tal forma que el se mantienen las normas del experimento 6 pero con la adición de recolectar todos los ítems y, en caso de necesitar una llave, solo recolectar una llave.

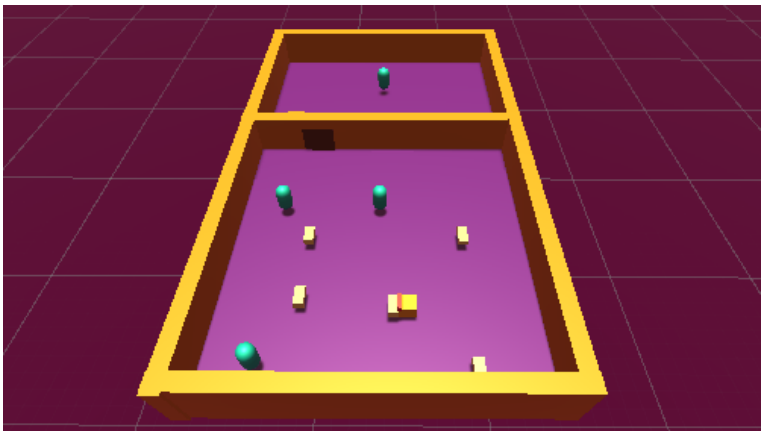

Figura 3.34: Entorno para el experimento 8

Atributos del script

Los atributos ahora son los siguientes:

```
1  [SerializeField]
2  private bool Training = true;
3  [SerializeField]
4  private GameObject MyKey;
5  [SerializeField]
6  private List<GameObject> KeyList = new List<GameObject
       >();
7  [SerializeField]
8  public List<GameObject> TargetList = new List<GameObject
       >();
9  [SerializeField]
10 private GameObject Ground;
11 [SerializeField]
12 private GameObject Door;
13
```

```
14 private AgentSettings M_AgentSettings; //Ajustes
15 Bounds areaBounds;
16 Rigidbody rb;
17 bool IHaveAKey = false;
18 bool InRoom;
19
20 List<GameObject> TargetListCopia;
21
22 //Lista de items donde 0 Significa fuera y 1 significa
      dentro (inRoom)
23 List<int> IteminRoomList = new List<int>();
```

La diferencia con respecto el experimento 6 radica en haber cambiado los atributos de los objetos tipo `Item` y `Key` por listas de estos tipos de objetos, dando así modularidad en lo que respecta al número que hay de cada objeto en el entorno.

Además, disponemos de dos listas adicionales:

- `TargetListCopia`, que servirá para recargar la lista de ítems.
- `IteminRoomList`, que indica si los ítems se encuentran dentro (1) o fuera (0).

Inicialización

En la iniciación solo se añade de más la descripción de `TargetListCopia`:

```
1 TargetListCopia = new List<GameObject>(TargetList);
```

Inicialización de episodios

La función de inicialización de los episodios se ve ahora de la siguiente forma:

```
1 public override void OnEpisodeBegin()
2 {
3
4     //Reseteo propiedades
5     MyKey.SetActive(false);
6     IHaveAKey = false;
7
8     //Evaluacion episodio anterior
```

```
9       if(TargetList.Count !=0){
10          AddReward(-100f);
11      }
12
13      //Recarga de la lista de items y puertas, ya que se
            ir n eliminando
14      TargetList.Clear();
15      TargetList.AddRange(TargetListCopia);
16
17      foreach(var it in TargetList){
18          it.SetActive(true);
19      }
20
21      //Se reparten las llaves por el  rea
22      foreach(var key in KeyList){
23          key.transform.position = GetRandomSpawnPos();
24          key.SetActive(true);
25      }
26
27      //Reseteo Agente
28      transform.position = GetRandomSpawnPos();
29      rb.velocity = Vector3.zero;
30      rb.angularVelocity = Vector3.zero;
31
32      //ReseteoItem
33      IteminRoomList.Clear(); //Al principio no sabemos
            que items est n dentro o fuera
34
35      //Para cada item de la lista de items a obtener
36      foreach(var item in TargetList){
37          Collider col_item = item.GetComponent<Collider
                >();
38          //Colocamos aleatoriamente los items
39          item.transform.position = GetRandomSpawnPosItem
                ();
40
41          InRoom = Physics.CheckBox(item.transform.
                position, new Vector3(2.5f, 0.01f, 2.5f),
                transform.rotation, LayerMask.GetMask("Rooml"
                ));
42
43          if(InRoom == true){
44              IteminRoomList.Add(1); //Dentro
```

```
45          }
46          else{
47              IteminRoomList.Add(0); //Fuera
48          }
49      }
50      //Tras este bucle tendremos una lista IteminRoomList
            con un valor 0 o 1 en cada casilla
          correspondiente a cada Item (Coge el mismo orden
          que TargetList)
51
52      Door.SetActive(true);
53  }
```

Estados

En `Ray Percetion Sensor 3D` no hay que cambiar nada, ya que los tipos de objetos que deben detectar son los mismos. Por otra parte, en `VectorSensor`, cambiamos la asignación como atributo de `inRoom` a los valores de cada ítem en `IteminRoomList`.

```
1 public override void CollectObservations(VectorSensor
     sensor)
2 {
3      sensor.AddObservation(IHaveAKey);
4      //Tiene tantos atributos como elementos/items haya
          en la lista
5      sensor.AddObservation(IteminRoomList.ConvertAll(n =>
           (float)n));
6 }
```

La indicación del número de atributos correspondiente al número de ítems sí que tendremos configurarlo de forma manual, siendo el número total de atributos por observación:

$$\text{nº atributos/observación} = \text{nº de ítems} + 1 \text{ (IHaveAKey)}$$

Recompensas

Para el collider la función de entrada es ahora la siguiente:

```
void OnTriggerEnter(Collider col)
{
    if (col.CompareTag("Key"))
    {
        if(IHaveAKey){ //Si ya tengo una llave
        if(Training) AddReward(-20);
        }

        else{
            if(Training){
                //Si no hay ningun item en una
                   habitacion (0) o ya se tiene una
                   llave en posecion
                if((ItemInRoomList.All(n => (n == 0)))
                   ||     IHaveAKey == false){
                    //No es necesaria la llave o ya
                       tienes una acaparador
                    AddReward(-20f);
                }

                else{
                    AddReward(20f);
                }
            }
        }

        //Si no tengo ya la animacion de la llave en el
           agente entonces se la pongo
        if(MyKey.activeSelf != true) MyKey.SetActive(
           true);
        //Si no ten a ninguna llave entonces ahora si
           indico que al menos tengo una
        if(IHaveAKey != true) IHaveAKey = true;
        col.gameObject.SetActive(false);
    }

    if (col.CompareTag("Door"))
    {
        if (IHaveAKey) //Si tengo una llave al menos
        {
            if(Training){
                //Si no hay ningun item en una
                   habitacion (0) o ya se tiene una
```

```
                    llave en poseci n
36                  if((IteminRoomList.All(n => (n == 0)))
                    || IHaveAKey == false){
37                      //No es necesaria la llave o ya
                            tienes una acaparador
38                      AddReward(-30f);
39                  }
40
41                  else{
42                      AddReward(30f);
43                  }
44              }
45
46              //Ya no tengo llave
47              MyKey.SetActive(false);
48              IHaveAKey = false;
49
50              col.gameObject.SetActive(false);
51          }
52
53          else{
54              if(Training) AddReward(-30);
55          }
56      }
57
58      if (col.CompareTag("Item"))
59      {
60          AddReward(30f);
61          //Descarto el valor correspondiente al item que
                se ha eliminado IteminRoomList.RemoveAt(
                TargetList.IndexOf(col.gameObject));
62          TargetList.Remove(col.gameObject); //Descarto el
                item que se ha eliminado
63          col.gameObject.SetActive(false);
64
65          if(TargetList.Count == 0){ //Si obtenemos todos
                los items
66              AddReward(50f); //Enhorabuena, has ganado
67              EndEpisode(); //SE FINALIZA
68          }
69      }
70  }
```

Resultados

En este caso, al igual que en el experimento 7, el editor Unity colapsa, haciendo inabordable también este experimento en el PC1.

Visto que ninguno de estos dos experimentos funciona, tendremos que pasar al PC2, ya que para querer ver comportamientos "interesantes" que aporten mejoría a los 6 experimentos realizados con éxito.

3.5. PC2

3.5.1. Experimento 6: Recolecta dentro o fuera

Vamos a repetir el experimento 6. Aunque PC1 realizó el experimento con éxito, nos interesa saber si podemos mejorarlo disponiendo de una capacidad mayor con PC2 como para que no se limite a 408 mil iteraciones.

Resultados

Podemos ver que usar PC2 ha incrementado el número de iteraciones máximas de entrenamiento a 100 mil iteraciones más. Es sorprendente ver que, pese a estar entrenando actualmente con PC2, la complejidad del experimento 6 se sigue viendo limitada.

3.5.2. Experimentos 7 y 8: Los límites de MlAgents

Si ya nos pareció decepcionante que el entrenamiento 6 acabase colapsando todavía `Unity` con tan solo 100 mil iteraciones, era de esperar que el entrenamiento 7 y 8 no nos iban a dar los resultados esperados. Ninguno de los dos entrenamientos se ha podido realizar con éxito colapsando al momento `Unity` al igual que pasó con el PC1. Esto pone de forma abrupta el límite de complejidad de `MlAgents` en el PC2.

3.5.3. Entenamiento 9: Experimento 7 con VectorSensor

Como experimento extra, vamos a repetir el experimento 7, pero esta vez aplicando todo el peso del estado a `VectorSensor`:

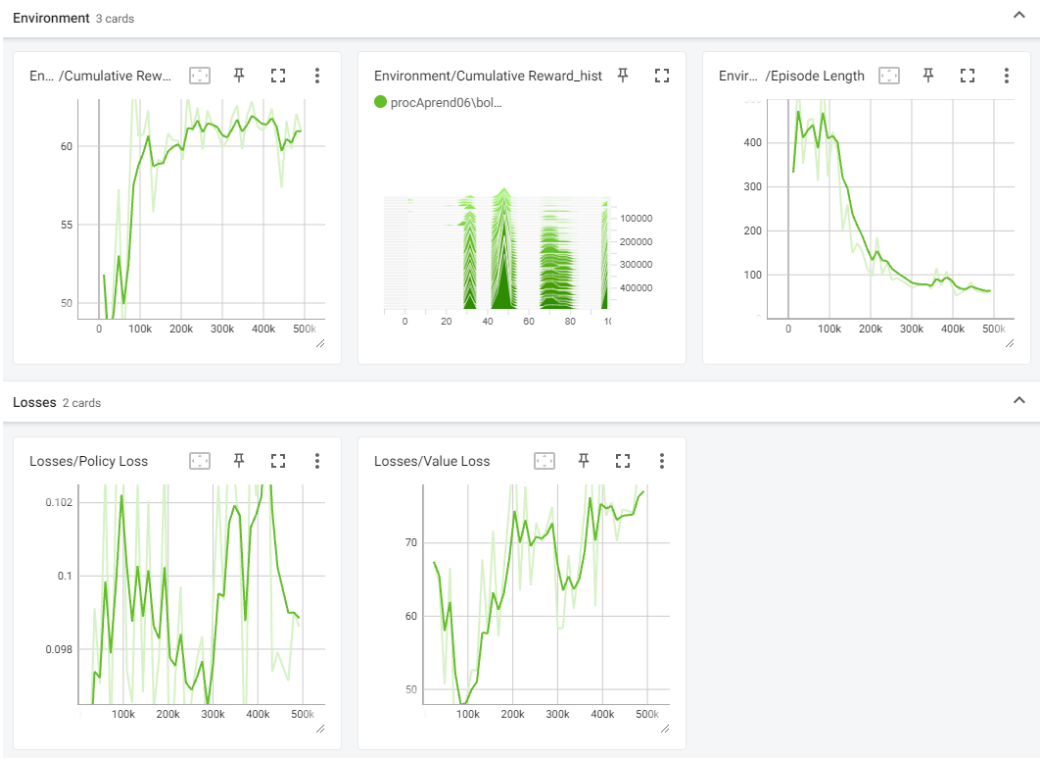

Figura 3.35: Resultados del experimento 6bis en PC2

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

    //Distancia al target.
    //Float de 1 posicion.
    sensor.AddObservation(
    Vector3.Distance(Target.transform.position,
        transform.position));

    //Direccion al target.
    //Vector 3 posiciones.
    sensor.AddObservation((Target.transform.position -
        transform.position).normalized);

    //Orientacion del agente.
    //Vector de 3 posiciones.
```

```
17      sensor.AddObservation(transform.forward);
18  //------------------------------------------------
19
20      //Distancia a key.
21      //Float de 1 posicion.
22      sensor.AddObservation(
23      Vector3.Distance(Key.transform.position, transform.
            position));
24
25      //Direccion a key.
26      //Vector 3 posiciones.
27      sensor.AddObservation((Key.transform.position -
            transform.position).normalized);
28  //------------------------------------------------
29
30      //Distancia a door.
31      //Float de 1 posicion.
32      sensor.AddObservation(
33      Vector3.Distance(Door.transform.position, transform.
            position));
34
35      //Direccion a door.
36      //Vector 3 posiciones.
37      sensor.AddObservation((Door.transform.position -
            transform.position).normalized);
38  }
```

$$\text{nº de atributos/observación} = (4 \text{ x } 3) + 3 + 2 = \mathbf{17}$$

Resultados

Reducido a solo 3 entornos en paralelo y con `maxStep` = **10000**, ahora por lo menos se mantiene un tiempo entrenando. `VectorSensor`, al tener un menor coste computacional en comparación con `Ray Perception Sensor 3D` es capaz de abordar mejor este experimento.

Dentro de lo que cabe, en su reducido número de iteraciones, las gráficas muestran una evolución optimista del entrenamiento.

Como conclusión, tanto para este experimento como para el experimento 8, podemos deducir que los experimento sin el límite que nos interpone `Unity` con su motor gráfico podrían ser realizados sin ningún problema, viéndose

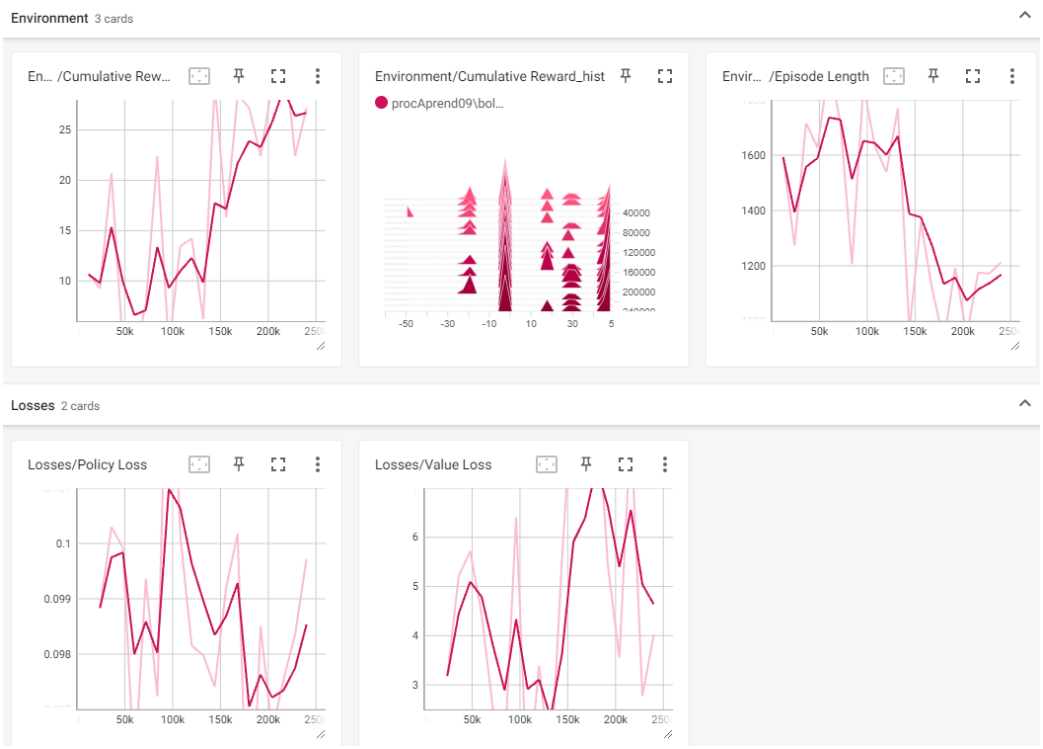

Figura 3.36: Resultados del experimento 9 en PC2

cómo reduciendo un mínimo el coste computacional puede llevara un avance en el entrenamiento de estos.

3.5.4. Experimento 10: Recolecta múltiple en recinto multihabitación y multillave

Este experimento es el último que vamos a presentar, pero sin ejecutarlo. Este experimento reúne los avances que querían aportar los experimentos 7 y 8, sumados con la modularización de puertas/habitaciones.

El script parte del experimento 8.

Atributos del script

```
[SerializeField]
private bool Training = true;
[SerializeField]
private GameObject MyKey;
```

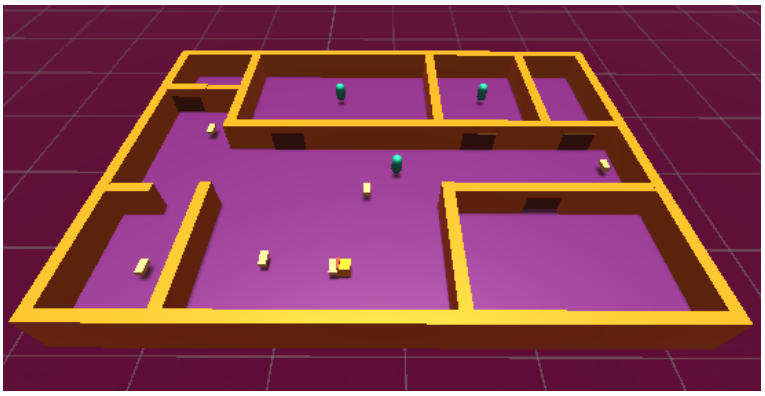

Figura 3.37: Entorno para el experimento 10

```
5  [SerializeField]
6  private List<GameObject> KeyList = new List<GameObject
       >();
7  [SerializeField]
8  public List<GameObject> TargetList = new List<GameObject
       >();
9  [SerializeField]
10 private GameObject Ground;
11
12 private AgentSettings M_AgentSettings; //Ajustes
13 Bounds areaBounds;
14 Rigidbody rb;
15 bool IHaveAKey = false;
16 bool InRoom;
17
18 //Los items, las habitaciones a las que corresponden,
       las habitaciones en si, y las puertas, van
       relacionadas con el orden en el que aparecen en las
       listas
19
20 //La posicion de cada puerta corresponde a la posicion
       de cada habitacion de la que pertenece
21 [SerializeField]
22 public List<GameObject> DoorList = new List<GameObject
       >();
23 //Solo se utiliza para saber que items estan en que
       habitaciones
24 [SerializeField]
25 public List<GameObject> RoomList = new List<GameObject
```

```
    >();

List<GameObject> TargetListCopia;
//Lista de colliders de cada habitacion
List<Collider> CollRoomList = new List<Collider>();
//Distinto a E8: Lista de items que se han detectado que estan en habitaciones y no en el pasillo
List<int> IteminRoomsList = new List<int>();
```

Inicialización

Se le añade adicionalmente:

```
//A adimos los colliders de cada habitacion
foreach(var room in RoomList){
    CollRoomList.Add(room.GetComponent<Collider>());
}
```

Inicialización del episodio

```
public override void OnEpisodeBegin()
{

    //Reseteo propiedades
    MyKey.SetActive(false);
    IHaveAKey = false;

    //Evaluacion episodio anterior
    if(TargetList.Count !=0){
        AddReward(-100f);
    }

    //Recarga de la lista de items y puertas, ya que se ir n eliminando
    TargetList.Clear();
    TargetList.AddRange(TargetListCopia);

    //Reseteo llave
    foreach(var it in TargetList){
        it.SetActive(true);
    }

```

```
22      //Se reparten las llaves por el area
23      foreach(var key in KeyList){
24          key.transform.position = GetRandomSpawnPos();
25          key.SetActive(true);
26      }
27
28      //Reseteo Agente
29      transform.position = GetRandomSpawnPos();
30      rb.velocity = Vector3.zero;
31      rb.angularVelocity = Vector3.zero;
32
33      //ReseteoItem
34      //Al principio no sabemos que items estan en
           habitaciones
35      IteminRoomsList.Clear();
36
37      //Para cada item de la lista de items a obtener
38      foreach(var item in TargetList){
39          Collider col_item = item.GetComponent<Collider
               >();
40          //Colocamos aleatoriamente los items
41          item.transform.position = GetRandomSpawnPosItem
               ();
42
43          //Para cada collider de las habitaciones
44          foreach(var col in CollRoomList){
45              //Si el collider del item actual esta en
                   contacto con la habitacion actual
46              if(col_item.bounds.Intersects(col.bounds))
47              {
48                  //[item1,item2.....itemn],donde el valor
                       de cada uno de la habitacion a la
                       que corresponden
49                  IteminRoomsList.Add(CollRoomList.IndexOf
                       (col));
50                  //Importante mencionar que esto es asi
                       porque el orden de las habitaciones
                       en la lista de habitaciones y el
                       orden de los colliders de estos es el
                        mismo
51
52                  break;
53              }
```

```
54
55              //Si ha llegado a la ultima habitacion y no
                    ha sido detectada en una
56              else if(CollRoomList.IndexOf(col) == (
                    CollRoomList.Count - 1)){
57                  IteminRoomsList.Add(-1); //Esta en el
                        pasillo principal
58              }
59          }
60          //Con este foreach ahora disponemos de una lista
                que nos indicara las habitaciones en las que
                estan los items en el episodio actual
61      }
62
63      //Activacion de las puertas
64      foreach(var door in DoorList){
65          door.SetActive(true);
66      }
67  }
```

Recompensas

```
1   void OnTriggerEnter(Collider col)
2   {
3       //Si el agente se encuentra la llave, este la coge
4       if (col.CompareTag("key"))
5       {
6           if((IteminRoomsList.All(n => (n == -1))) ||
7           //Si no hay ningun item en una habitacion o ya
                se tiene una llave en posecion
8           IHaveAKey == false){
9               //No es necesaria la llave o ya tienes una
                    acaparador
10              AddReward(-20f);
11          }
12
13          else{
14              AddReward(20f);
15          }
16
17          //Si no tengo ya la animacion de la llave en el
                agente entOonces se la pongo
```

```
18          if(MyKey.activeSelf != true) MyKey.SetActive(
                true);
19
20          //Si no ten a ninguna llave entonces ahora si
                indico que al menos tengo una
21          if(IHaveAKey != true) IHaveAKey = true;
22              col.gameObject.SetActive(false);
23          }
24
25          if (col.CompareTag("door"))
26          {
27              if (IHaveAKey) //Si tengo una llave al menos
28              {
29                  //Puerta 0..4
30                  //Si ningun item esta en la habitacion
                        que le corresponde
31                  if(IteminRoomsList.All(n => (n !=
                        DoorList.IndexOf(col.gameObject)))){
32                   //No es necesaria la llave
33                   AddReward(-30f);
34              }
35
36              else{
37                  AddReward(30f);
38              }
39
40              //Ya no tengo llaves
41              MyKey.SetActive(false);
42              IHaveAKey = false;
43
44              col.gameObject.SetActive(false);
45          }
46      }
47
48      if (col.CompareTag("Item"))
49      {
50
51          AddReward(30f);
52
53          //Lo descartamos de las listas
54          IteminRoomsList.RemoveAt(TargetList.IndexOf(col.
                gameObject));
55          TargetList.Remove(col.gameObject);
```

```
56          col.gameObject.SetActive(false);
57
58          //Si obtenemos todos los items
59          if(TargetList.Count == 0){
60              AddReward(50f); //Enhorabuena, has ganado
61              EndEpisode(); //SE FINALIZA
62          }
63      }
64  }
```

Capítulo 4

Conclusiones

Para cerrar esta memoria, reuniremos todas las observaciones claves que se han ido recopilando a lo largo de los diversos experimentos y estudios realizados.

4.1. Estados en mlAgents

Como hemos visto, la configuración del estado en `mlAgents` depende de si se hace uso de `VectorSensor` o `Ray Perception Sensor 3D`, donde:

- `Ray Perception Sensor 3D` es un sensor sencillo de configurar, con solo la necesidad de las etiquetas de los tipos de objetos que debe detectar, pero que implica un coste de computación considerable por la precisión que ofrece.
- `VectorSensor` es un sensor más primitivo en el que tenemos que configurar de forma más manual los atributos de las observaciones que percibe, pero que necesita un menor costo computacional, siendo útil para problemas con alto coste computacional, como pasó en el experimento 7.

4.2. Acciones en mlAgents

Las acciones, pese a ser una parte clave en el MDP y que puede a simple vista resultar una parte complicada de modelar, es de hecho la más simple de todas las configuraciones. Para tareas de recolección, que es a lo que se

reducen la mayoría de los problemas analizados, el agente toma un comportamiento más humano, donde la animación procedural se descarta, y las acciones se reducen al movimiento bidimensional (ver el experimento 2). Exceptuando que se quiera añadir una acción que permita al agente desplazarse en las tres dimensiones (con una acción de salto, por ejemplo), con las acciones vistas en los experimentos posteriores al segundo podemos ser capaces de abarcar una gran gama de problemas implementados mediante RL.

4.3. Recompensas en mlAgents

Las recompensas se ven concentradas en las funciones collider ya que, por definición, son respuestas cuantificadas de la interacción entre el agente y el entorno, donde el collider es el componente estrella de Unity.

También hemos visto el uso de recompensas en zonas donde no nos esperaríamos su uso, como en las acciones como punto de descuento por cada iteración pasada, o en la iniciación de episodios para evaluar el resultado final del episodio anterior. Esto implica que el uso de recompensas, pese a centrarse en los colliders, pueden estar presentes donde el programador quiera implementarlas.

4.4. La buena práctica del RL

Como último punto, retomamos el final del capítulo anterior. El motivo por el cuál no se ha podido pasar del experimento 6 es por el alto coste que lleva de por sí ejecutar cualquier cosa en Unity al ser un motor gráfico. MlAgents, que no deja de ser parte de Unity, no se libra de ese alto costo en gráfica, motivo el cuál no ha sido muy exitoso en el mercado y casi ha funcionado más como una curiosidad.

A pesar de ello, MlAgents es una buena herramienta para introducir a usuarios en el aprendizaje por refuerzo, y donde se pueden realizar problemas sencillos para comprender sus fundamentos. Más allá de eso, es mejor disponer de herramientas no gráficas para realizar los entrenamientos RL más complejos (aunque luego se implementen los modelos entrenados en Unity), solo así veremos el verdadero potencial que promete el Aprendizaje por refuerzo.

De todas formas, eso no significa que no podamos explorar más aún el potencial de MlAgents mediante las extensiones que comentamos al principio

de este trabajo.

Bibliografía

[1] Pedro Castro-Rodrigues and Albino J. Oliveira-Maia. Exploring the effects of depression and treatment of depression in reinforcement learning. *frontiers*, 2013. 16

[2] Himanshu Pareek Deepali Salwan, Shri Kant and Roopali Sharma. Challenges with reinforcement learning in prosthesis. *Science Direct*, 2022. 16

[3] Uadla Games. Curso de inteligencia artificial con ml-agents de unity, 10/8/2020. 75, 81

[4] J. Andrew Bagnell Jens Kober and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 2013. 17

[5] Deborah Dewey Adam McCrimmon Manuela Schuetze, Christiane S Rohr and Signe Bray. Reinforcement learning in autism spectrum disorder. *frontiers*, 2017. 16

[6] Phuong Dinh Ngo Andrius Budrionis Asbjørn Johansen Fagerlund Maryam Tayefi Johan Gustav Bellika Marit Dagny Kristine Jenssen, Per Atle Bakkevoll and Fred Godtliebsen. Machine learning in chronic pain research: A scoping review. *MDPI*, 2021. 17

[7] Terry Lingze Meng and Matloob Khushi. Reinforcement learning in financial markets. *MDPI*, 2019. 18

[8] Ashenafi Zebene Woldaregay Miguel Tejedor and Fred Godtliebsen. Reinforcement learning application in diabetes blood glucose control: A systematic review. *PubMed*, 2020. 16

[9] John Kennedy Muhammad and C.W. Lim. Machine learning and deep learning in phononic crystals and metamaterials – a review. *ScienceDirect*, 2022. 18

[10] P.Dayan. Dopamine, reinforcement learning, and addiction. *Thieme*, 2009. 16

[11] Willem van Jaarsveld Robert N. Boute, Joren Gijsbrechts and Nathalie Vanvuchelen. Deep reinforcement learning for inventory control: A roadmap. *ScienceDirect*, 2022. 18

[12] Will Serrano. Deep reinforcement learning algorithms in intelligent infrastructure. *MDPI*, 2019. 18

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, An introduction (second edition)*. The MIT Press, 2020. 14, 16, 18, 27, 37, 48

[14] Unity-Technologies. ml-agent. https://github.com/Unity-Technologies/ml-agents, 29/9/1017. Fecha última de acceso: 2/7/2023. 59, 81, 88