

DISEÑO DE ENTORNOS SIMULADOS Y ENTRENAMIENTO DE AGENTES CON APRENDIZAJE POR REFUERZO

TITULACIÓN:
Inteligencia Artificial

Curso académico:
2019/2020

**Lugar de residencia,
mes y año:**
Barcelona, Julio 2020

Alumno/a:
Toscano Pla, Nico

D.N.I:
47853752K

Director:
Muñoz Ríos, Gabriel

Convocatoria:
Segunda

Orientación:
Aplicación

Créditos:
60

Índice

1. Introducción	7
2. Aprendizaje por refuerzo	8
2.1 Conceptos	8
3. Algoritmos	12
3.1 Clasificaciones	12
3.2 Q-Learning	14
3.3 Policy Gradients	17
4. Unity	24
4.1 ML-Agents	24
4.2 Entorno Unity	25
4.3 Diseño del Agente y el Entorno	26
4.4 Configuración del Agente	27
4.5 Entrenamiento del Agente	28
5. Entornos	30
5.1 Entorno 1	30
5.2 Entorno 2	32
5.3 Entorno 3	34
5.4 Entorno 4	36
5.5 Entorno 5	38
6. Implementación	41
6.1 Comunicación entre Python y Unity	41
6.2 Estructura del código	41
6.3 Visualización del entrenamiento	49
7. Resultados	51
7.1 Experimentos	51
8. Conclusiones	89
9. Trabajo a futuro	91
10. Bibliografía	95

Anexos.....	99
Pseudocódigos de Algoritmos	99
Unity.....	102
Hiperparámetros	109
Código y vídeos	112

Índice de imágenes

Imagen 1 Brazo robot simulado.....	9
Imagen 2 Brazo robot real.....	9
Imagen 3 Esquema general del aprendizaje por refuerzo	10
Imagen 4 Entorno con Model Free (Pong)	13
Imagen 5 Entorno con Model Based (Ajedrez).....	13
Imagen 6 Clasificación algoritmos de aprendizaje por refuerzo	14
Imagen 7 Esquema del algoritmo de DQN	16
Imagen 8 Esquema del algoritmo de Actor-Critic.....	19
Imagen 9 Esquema del algoritmo de A3C.....	20
Imagen 10 Esquema del algoritmo de A2C	21
Imagen 11 Entorno de Unity	25
Imagen 12 Parámetros del Agente.....	28
Imagen 13 Entorno 1	30
Imagen 14 Entorno 1 (Raycast).....	31
Imagen 15 Entorno 2	32
Imagen 16 Entorno 2 (Raycast).....	33
Imagen 17 Entorno 3	34
Imagen 18 Entorno 3 (Raycast).....	35
Imagen 19 Entorno 4	36
Imagen 20 Entorno 4 (Raycast).....	37
Imagen 21 Entorno 5	38
Imagen 22 Entorno 5 (Raycast).....	39
Imagen 23 Flujo de información entre Python y Unity	41
Imagen 24 Información del entrenamiento.....	49
Imagen 25 Entrenamiento agentes	50
Imagen 26 Entorno 5, posiciones iniciales	74
Imagen 27 Entorno 5 - Nivel 1, posiciones iniciales	76
Imagen 28 Entorno 5 - Nivel 2, posiciones iniciales	78
Imagen 29 Entorno 5 - Nivel 3, posiciones iniciales	80
Imagen 30 Entorno 5 - Nivel 4, posiciones iniciales	82
Imagen 31 Entorno 5 - Nivel 5, posiciones iniciales	84
Imagen 32 Entorno 5 - Nivel 6, posiciones iniciales	86
Imagen 33 Entorno con agentes compitiendo entre ellos	92
Imagen 34 Flujo de información entre Python y Unity	92
Imagen 35 Flujo de información entre Python y Unity (2)	93
Imagen 36 Entorno con el espacio de acciones en continuo.....	94

Imagen 37 Parámetros del vector de observaciones y de acciones	105
Imagen 38 Parámetros generales del Agente.....	106
Imagen 39 Parámetros de Raycast	107

Índice de gráficas

Gráfica 1 Entorno 1, DQN	54
Gráfica 2 Entorno 1, Actor-Critic.....	55
Gráfica 3 Entorno 1, A3C.....	56
Gráfica 4 Entorno 1, A2C.....	57
Gráfica 5 Entorno 1, PPO.....	58
Gráfica 6 Entorno 2, DQN	59
Gráfica 7 Entorno 2, Actor-Critic.....	60
Gráfica 8 Entorno 2, A3C.....	61
Gráfica 9 Entorno 2, A2C.....	62
Gráfica 10 Entorno 2, PPO	63
Gráfica 11 Entorno 3, DQN	64
Gráfica 12 Entorno 3, Actor-Critic	66
Gráfica 13 Entorno 3, A3C.....	68
Gráfica 14 Entorno 3, A2C.....	69
Gráfica 15 Entorno 3, PPO	70
Gráfica 16 Entorno 4, A3C.....	71
Gráfica 17 Entorno 4, A2C.....	72
Gráfica 18 Entorno 4, PPO	73
Gráfica 19 Entorno 5, PPO	75
Gráfica 20 Entorno 5 - Nivel 1, PPO.....	77
Gráfica 21 Entorno 5 - Nivel 2, PPO.....	79
Gráfica 22 Entorno 5 - Nivel 3, PPO.....	81
Gráfica 23 Entorno 5 - Nivel 4, PPO.....	83
Gráfica 24 Entorno 5 - Nivel 5, PPO.....	85
Gráfica 25 Entorno 5 - Nivel 6, PPO.....	87

1. Introducción

En los últimos años, en el campo de la inteligencia artificial, se han logrado grandes progresos gracias al gran avance que se ha hecho en el ámbito de las redes neuronales y son particularmente útiles en el campo del aprendizaje por refuerzo.

La finalidad de este trabajo, es el estudio e implementación de diferentes algoritmos de aprendizaje por refuerzo, con el uso de redes neuronales como funciones aproximadoras, siendo así aprendizaje por refuerzo profundo.

Para probar el funcionamiento de estos algoritmos, es común utilizar librerías externas que contienen entornos simulados, como por ejemplo juegos de Atari. Pero en este trabajo, en vez de utilizar dichas librerías, se crearán los entornos desde cero. De esta manera se tiene un mayor control sobre el entorno y así poder optimizar el proceso de aprendizaje de la inteligencia artificial encargada, de aprender a realizar una tarea determinada. Esto implica, el diseño del entorno con sus mecánicas y objetos, que van interactuar directamente con la inteligencia artificial.

La parte de la creación de entornos simulados es importante, porque si se quiere llevar a cabo una aplicación, como podría ser el movimiento de un brazo robótico, primero se puede probar en simulaciones a través de un ordenador y luego llevarlo al mundo real. Las aplicaciones suelen ser bastante concretas y, por lo tanto, no se puede recurrir a la utilización de librerías para realizar las simulaciones, de modo que se tienen que diseñar desde cero, de la misma forma que se va hacer en este trabajo.

Primero se empezará explicando de manera general, el aprendizaje por refuerzo y los conceptos más importantes, seguidamente los algoritmos principales que hay, después la parte de creación de los entornos y finalmente la implementación de los algoritmos en los entornos creados, exponiendo los resultados obtenidos.

2. Aprendizaje por refuerzo

El aprendizaje por refuerzo [Sutton & Barto, 2018] es un campo dentro de la inteligencia artificial, que intenta emular la manera en que las personas aprenden, para aplicarlo en máquinas y poderse beneficiar de las ventajas que proporcionan.

En psicología se intenta describir las leyes generales que rigen la conducta voluntaria de las personas y el condicionamiento operante es uno de los conceptos que se utiliza para ello. Es una forma de aprendizaje mediante la cual, un sujeto tiene más probabilidades de que repita una acción en particular si es seguida por algo deseable y con menos probabilidad, por algo no deseable, disuadiendo de realizar dicha acción.

En aprendizaje por refuerzo, se utiliza este mismo concepto a la máquina, para que las inteligencias artificiales sean capaces de aprender por sí mismas. Las IAs respecto a los humanos, tienen la ventaja que no se cansan y realizan las tareas extraordinariamente rápido.

Las IAs aprenden con un feedback de lo acertado de sus decisiones, en el entorno que se están entrenando y de esta manera pueden llegar a correlacionar acciones inmediatas, con sus consecuencias a largo plazo.

2.1 Conceptos

Antes de entrar en más detalle sobre el funcionamiento del aprendizaje por refuerzo, se explicarán a continuación los conceptos básicos.

Agente

Se denomina Agente, a la IA que se encarga en descubrir que acciones se deben tomar en el entorno, para maximizar la señal de recompensa. Al agente no se le dice que acciones tomar, sino que debe experimentar por sí mismo y encontrar que acciones lo llevan a una mayor recompensa.

Entorno

El entorno es dónde se encuentra el agente, aprendiendo e interactuando con los diferentes elementos que hay. A parte de los componentes que conforman el entorno, también es parte de él su funcionamiento y sus reglas, como por ejemplo el tiempo que tiene el agente para alcanzar su objetivo, la vida que dispone, la manera en la que se puntuá, etc. Generalmente los entornos son simulaciones, como por ejemplo entrenar IAs para videojuegos o simulaciones para el funcionamiento de brazos robóticos, que luego se prueban en el mundo real.

La imagen de la izquierda se muestra un brazo robótico en una simulación y en la imagen de la derecha el mismo brazo robótico trasladado a la realidad, físicamente.

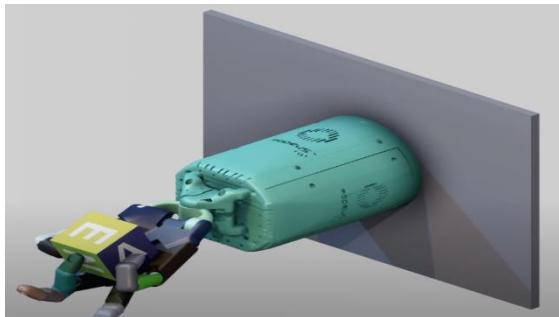


Imagen 1 Brazo robot simulado

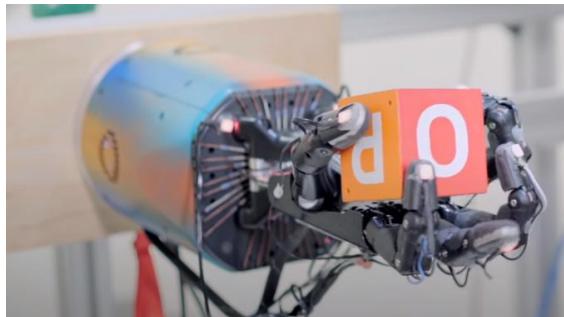


Imagen 2 Brazo robot real

Acciones

Las decisiones que el agente va tomando en el entorno, son las acciones. El agente dispone de una lista de acciones que puede realizar, y en cada decisión intenta escoger la mejor entre ellas, dependiendo del estado en el que se encuentre el entorno.

Estado

Las acciones que va tomando el agente afectan al entorno, y el estado es como queda el entorno afectado después de cada acción del agente.

Observación

Es la información sobre el entorno en un estado, a partir de la cual el agente toma una acción.

Recompensa

La recompensa es la manera de evaluar lo buena o mala que ha sido la acción que ha tomado el agente, dependiendo del estado en el que se encontraba el entorno.

En la siguiente imagen se puede observar el esquema que se sigue en aprendizaje por refuerzo:

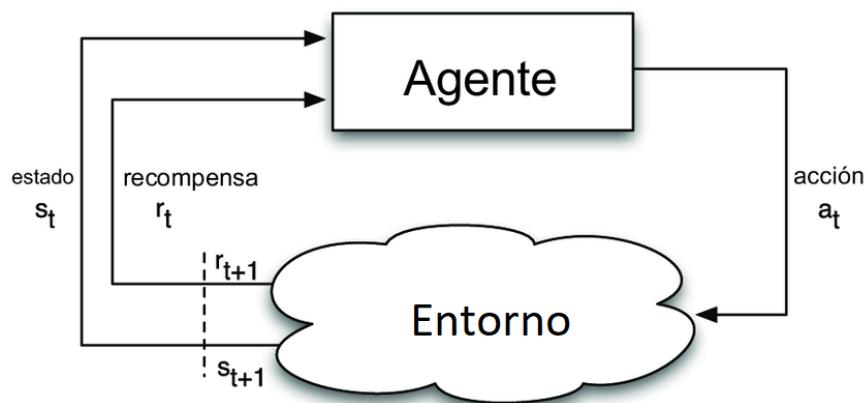


Imagen 3 Esquema general del aprendizaje por refuerzo

El agente (IA) toma una acción a partir del estado (s_t) en el que se encuentra actualmente el entorno, recibe la recompensa (r_{t+1}) por dicha acción y el estado de cómo queda el entorno (s_{t+1}).

Iteración (step)

El conjunto de los pasos del proceso que se acaba de describir, vendría a ser una iteración (step) y el agente va aprendiendo a medida que se van ejecutando los steps.

Episodio

El entorno se reinicializa cuando el agente ha logrado su objetivo o cuando han pasado una cierta cantidad de steps, y el agente no ha alcanzado el objetivo. El episodio es el transcurso de steps que se van ejecutando antes de que se reinicialice el entorno.

Estrategia (policy)

El agente lo que termina haciendo, es aprender sobre un entorno cual es la mejor estrategia (policy en inglés) para lograr su objetivo de la manera más optima y eficiente. La policy del agente, decide cual es la mejor acción a tomar en los distintos estados que se vaya encontrando, para completar la tarea que tiene asignada. El encargado de modelar las acciones que se toman dependiendo del estado del entorno, es un modelo de Deep Learning.

Experiencia

La experiencia es la información que utiliza el agente para su aprendizaje. Esta información dependiendo del algoritmo que se aplique puede variar, pero en general acostumbra a ser las acciones que ha ido tomando el agente, los estados en los cuales se encontraba y las recompensas obtenidas por dichas acciones.

Ahora que se ha visto la base de aprendizaje por refuerzo y sus conceptos principales, en el siguiente capítulo se explicaran los algoritmos de forma teórica, que se utilizan para que el agente aprenda.

3. Algoritmos

Los algoritmos de aprendizaje por refuerzo, se centran en cómo utilizar la experiencia del agente para que encuentre la policy óptima, haciendo actualizaciones del modelo de Deep Learning durante el transcurso del aprendizaje.

Como se puede ver en [OpenAI, 2018], se pueden clasificar los algoritmos de dos maneras distintas, según si el comportamiento del agente está basado en un modelo o no y dependiendo del tipo de estrategia que utiliza el agente para aprender con la experiencia que va recolectando. Cabe decir que existen más maneras para clasificar los algoritmos de aprendizaje por refuerzo, pero las que se han mencionado son las más importantes.

3.1 Clasificaciones

Basada en modelo

Con la clasificación basada en modelo se pueden encontrar las siguientes opciones:

- **Model free:** en este caso el modelo del agente desconoce el funcionamiento del entorno, esto sucede en entornos no deterministas y por lo tanto no se puede conocer un modelo para predecir los siguientes estados del entorno. Un ejemplo de este tipo de entornos serían los videojuegos, en que el jugador tiene que ir aprendiendo como jugar, como en el juego del pong.
- **Model based:** en cambio en este caso, si se conoce el funcionamiento del entorno, por lo tanto, con las acciones que toma el agente se pueden predecir los siguientes estados del entorno. Se suelen combinar algoritmos de aprendizaje por refuerzo con métodos heurísticos o de inteligencia artificial, para mejorar la búsqueda de las soluciones óptimas durante el proceso de aprendizaje. Un ejemplo de este tipo de entornos serían juegos como el ajedrez, en que se puede saber cómo quedará el entorno después de una acción. Cuando se mueve una pieza del tablero de una casilla a otra, se sabe el conjunto de acciones que puede llegar a tomar el contrincante, y por lo tanto, se puede predecir qué acción va a tomar, en otras palabras, el próximo estado del entorno.

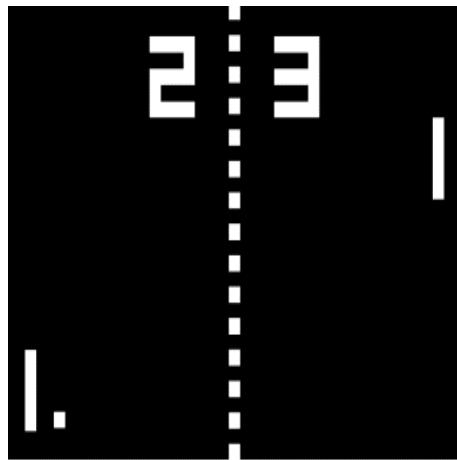


Imagen 4 Entorno con Model Free
(Pong)



Imagen 5 Entorno con Model Based
(Ajedrez)

Basada en estrategia

El agente cada vez que actualiza su modelo de Deep Learning cambia su policy, y en esta clasificación trata sobre que experiencias se utilizan para actualizar el modelo, dónde se pueden encontrar las siguientes opciones:

- **Off policy:** se hace uso de experiencias de distintas policies para el aprendizaje del agente. Al utilizar información del pasado, puede representar más tiempo de convergencia para llegar a la policy óptima. Los algoritmos de la familia Q-Learning siguen este mismo principio.
- **On policy:** en este caso solo se hace uso de las experiencias de la policy actual del agente. Este tipo de aprendizaje implica más tiempo de entrenamiento para llegar a la solución óptima y una varianza en los datos mayor. Los algoritmos de la familia de Policy Gradients siguen este principio.

En la siguiente imagen se muestra una clasificación de los principales algoritmos en aprendizaje por refuerzo:

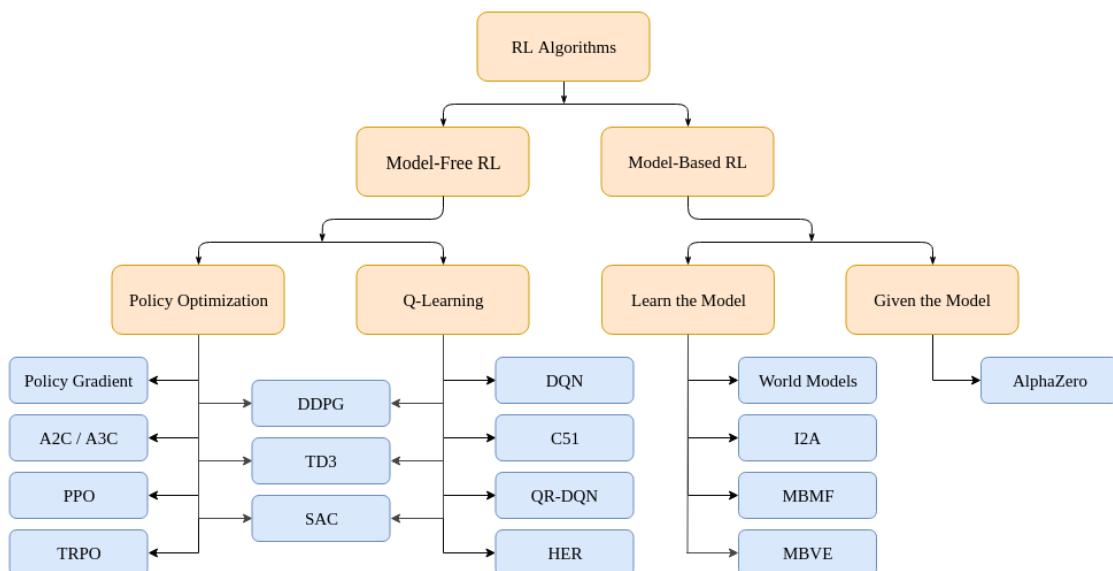


Imagen 6 Clasificación algoritmos de aprendizaje por refuerzo

Este trabajo se centra en los algoritmos de model free, con el algoritmo de DQN de la familia de Q-Learning y los algoritmos Policy Gradient, A2C/A3C y PPO de la familia de Policy Gradients.

3.2 Q-Learning

Los algoritmos de la familia Q-Learning [Sutton & Barto, 2018] utilizan experiencias que pueden ser de la policy actual o no, para el aprendizaje del agente (off policy). El objetivo de Q-Learning es aprender una serie de normas que le diga a un agente que acción tomar dado un estado del entorno, de manera que es capaz de llegar a una policy óptima, que maximiza el valor esperado de la recompensa total sobre todos los pasos sucesivos, empezando desde el estado actual. La "Q" en Q-learning viene de "quality", calidad, y en este caso representa como de útil es una acción dada, para obtener alguna recompensa futura.

Lo que se hace, es utilizar una tabla que relacione los estados con las acciones. De modo que, para cada par estado-acción, tiene asociado el valor esperado de la suma de las recompensas futuras (q_value). De esta manera, se puede saber en cada estado, cual es la mejor acción a tomar por el agente para que pueda obtener la máxima recompensa futura.

Para encontrar la policy óptima, este algoritmo se basa en el proceso de exploración – explotación:

- **Exploración:** en esta primera etapa, el agente explora el entorno tomando acciones de manera aleatoria y va almacenando la experiencia obtenida. La exploración es controlada por una variable que va disminuyendo en el tiempo y acostumbra adoptar valores entre 0.99, que representa una acción siempre aleatoria, hasta un valor de 0.05, que significa que la acción será tomada por la policy mayoritariamente, aunque con una pequeña probabilidad de que se tome una acción aleatoria. En este intervalo de tiempo, entre estos dos valores, el agente estará en la etapa de exploración.
- **Eplotación:** una vez el agente ha explorado el entorno durante un tiempo determinado, de modo que haya aprendido lo suficiente, para que la toma de decisiones se base directamente en la policy, empieza la etapa de explotación. En esta etapa, es común dejar un cierto grado de aleatoriedad, una probabilidad muy pequeña, para que el agente tome una acción de manera aleatoria.

De la familia de algoritmos de Q-Learning, solo se utilizará el algoritmo DQN en este trabajo, debido a su sencillez.

DQN

En este algoritmo [Mnih et al, 2013], el agente va recolectando experiencias primero en la etapa de exploración, dónde sus acciones son más aleatorias, y luego a partir de la etapa de explotación ya empieza a tomar decisiones en base a su policy. En ambas etapas, el modelo del agente se va actualizando cada cierto número determinado de steps, a partir de las experiencias que ha ido recolectando, cabe decir que en la primera actualización hay que tener un número mínimo determinado de experiencias.

Para modelar la recompensa esperada a futuro, se utiliza la ecuación de Bellman:

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

Es una ecuación recursiva, en la que se podrá encontrar la función óptima Q. Dónde $Q^{\pi^*}(s, a)$ es la suma de la recompensa obtenida en el estado actual "r" y la recompensa futura esperada del siguiente estado " $\gamma \max Q(s', a')$ ". Gamma " γ " es el factor de descuento para futuras recompensas.

El problema de usar una tabla como función aproximadora para los q_values, es que, en problemas complejos en términos de dimensionalidad, el aprendizaje del agente se vuelve imposible. Por este motivo, en vez de una tabla de q_values, se use una red neuronal como función aproximadora. Y en vez de aprender los q_values, se aprenderán los parámetros del modelo de la red neuronal.

Es por ello, que se requiere de una función de coste para que mida el error que se está cometiendo. La función de coste en el caso del algoritmo de DQN es la siguiente:

$$\text{loss} = (r + \gamma \max \hat{Q}(s', a') - Q(s, a))^2$$

Lo que se pretende con esta ecuación, es minimizar el error que se produce entre la ecuación de Bellman (target) " $r + \gamma \max \hat{Q}(s', a')$ " y el término que se predice " $Q(s, a)$ ".

En la siguiente imagen, se puede observar la implementación de DQN de forma esquemática, dónde se puede apreciar que se tienen dos modelos (redes neuronales). Un modelo para la predicción de $Q(s, a)$ (Q-Network) y otro como target de la ecuación de Bellman (Target Q-Network) tal y como se indicaba en la función de coste:

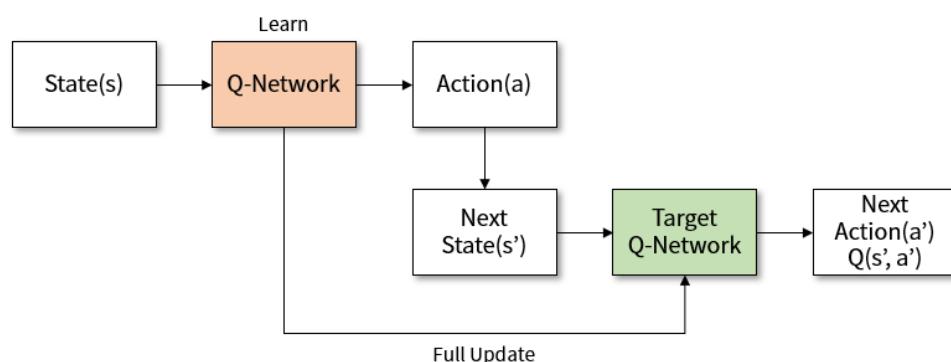


Imagen 7 Esquema del algoritmo de DQN

3.3 Policy Gradients

Los algoritmos de Policy Gradients [Sutton & Barto, 2018], optimizan la policy directamente y utilizan las experiencias actuales del agente para el aprendizaje (on policy), aunque también hay variaciones que utilizan experiencias pasadas (off policy). La policy, en Deep Reinforcement Learning se utiliza una red neuronal, para predecir las acciones del agente en función del estado en el que se encuentre el entorno. La policy se va optimizando con las recompensas que va obteniendo el agente, para maximizar la recompensa esperada, y se busca el gradiente de la policy para maximizar este valor.

El agente en cada toma de decisión, es decir, la elección de una acción en función del estado del entorno, lo que obtendrá de la red neuronal es una lista de todas las acciones que tiene disponible, con una probabilidad asociada a cada una de ellas. Y la acción con más probabilidad, será la más probable que escogerá el agente, dentro de una cierta aleatoriedad. De esta manera se evita que, en el proceso de aprendizaje, durante el descenso de gradiente, no se quede estancado en un máximo o mínimo local. A medida que se vaya entrenando el agente, se irán potenciando positivamente aquellas acciones que devuelvan una recompensa positiva y se ponderarán negativamente las que devuelvan una recompensa negativa.

Las probabilidades de cada acción se van actualizando durante el entrenamiento, a través de la maximización de la recompensa esperada. Cuando el agente obtiene una recompensa, se recompensan todas las acciones que han llevado a esta recompensa (cuando se dicen “todas las acciones”, se refieren a las acciones tomadas por el agente entre cada actualización del modelo). Las acciones justo antes de la recompensa, tendrán una ponderación más elevada en comparación a las primeras. En la siguiente fórmula se puede apreciar el factor de descuento para futuras recompensas:

$$R_t = \sum_{i=t}^T \gamma^{i-t} r_i = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^{T-t} r_T$$

Dónde r es la recompensa obtenida en un estado en concreto y γ el factor de descuento para futuras recompensas.

La finalidad en este trabajo es llegar al algoritmo de PPO y para ello primero se tienen que tener claros los algoritmos predecesores, Actor-Critic y sus variaciones de A3C y A2C.

Actor-Critic

El primer algoritmo que es el Actor-Critic [Konda & Tsitsiklis, 2002] y que sirve como base para los algoritmos posteriores, tiene dos elementos importantes, el Actor y el Critic:

- **Actor:** es el modelo que predice las acciones del agente en función de los estados en los que se encuentra el entorno. Y se actualiza en la dirección sugerida por el critic.
- **Critic:** es el modelo que evalúa lo bueno o malo que es para el agente, estar en los estados que se va encontrando durante el entrenamiento.

El agente durante el proceso de aprendizaje, va interactuando con el entorno, de manera que va recolectando experiencias (estados, acciones y recompensas) y cada cierto número determinado de steps, se hace una actualización en ambos modelos. Cada vez que se hace una actualización, la experiencia se resetea, y de esta manera siempre se actualizan los dos modelos con experiencias de la policy actual (on policy).

En Policy Gradients, se usa la recompensa como factor de las probabilidades de las acciones, esto implica una varianza muy grande en los datos. Porque de esta manera, la probabilidad de una acción en un estado determinado, dependerá también si la recompensa cambia. Esto provoca que no haya una correlación directa entre estado y probabilidad de acción, y dificulta el proceso de aprendizaje.

Para suavizar este problema del uso de la recompensa como factor, existen otras soluciones, y en este trabajo se utilizará la denominada GAE [Schulman et al, 2016], Generalized Advantage Estimation:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$$

Dónde $Q^\pi(s_t, a_t)$, es el valor del par estado-acción del modelo Actor y $V^\pi(s_t)$, es el valor del estado del modelo Critic.

La función de coste, con la cual el algoritmo Actor-Critic utiliza, para la optimización del modelo de la red neuronal, es la siguiente:

$$L^{PG}(\theta) = E[\log \pi_\theta(a_t | s_t) \hat{A}_t]$$

Que viene a ser el logaritmo del valor del par estado-acción multiplicado por la ventaja (GAE).

La función de coste también se le añade un término de error, en la estimación del valor del Critic (fórmula en rojo) y un término de entropía (fórmula en azul) para que el agente tenga la exploración suficiente del entorno:

$$L^{PG}(\theta) = E[L^{PG}(\theta) - c_1(V_\theta(s) - V_{target})^2 + c_2 H(s, \pi_\theta(.))]$$

Dónde c_1 y c_2 son dos hiperparámetros constantes.

En la siguiente imagen se puede observar el esquema de actor-critic:

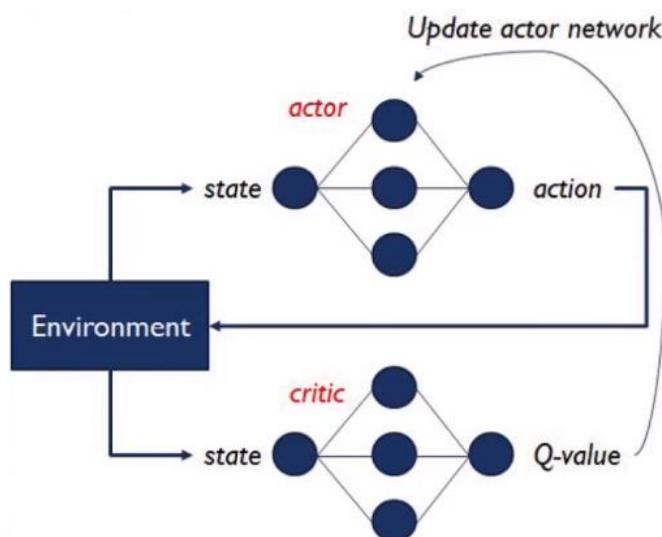


Imagen 8 Esquema del algoritmo de Actor-Critic

A3C

En Actor-Critic un único agente se encarga de recolectar las experiencias, en cambio en A3C [Mnih et al, 2016], se utiliza más de un agente. De esta manera, se consigue explorar más estados a la vez y esto implica, obtener una gran variedad de experiencias, en comparación a con un solo agente.

Cada agente tiene su propio modelo, que va actualizando con sus propias experiencias al igual que en Actor-Critic. Y el modelo global se va actualizando, con la experiencia de cada uno de los agentes por separado y de manera asíncrona.

En la siguiente imagen se puede apreciar de manera conceptual, como los agentes se van entrenando de manera paralela y van transfiriendo los parámetros de sus modelos al modelo global, a través de sus propias experiencias.

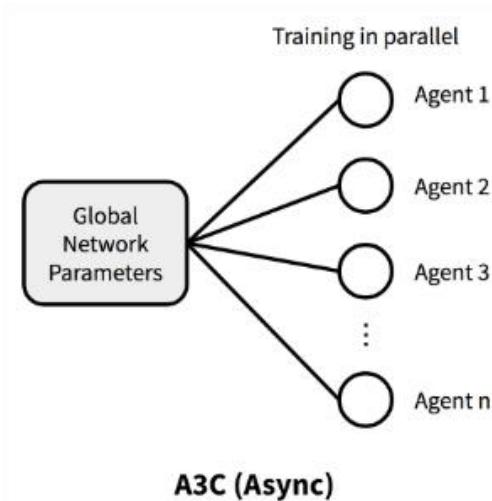


Imagen 9 Esquema del algoritmo de A3C

Implementación

Este algoritmo se ha implementado en código de la siguiente manera:

Cuando un agente completa un episodio, ese agente transfiere los parámetros de su modelo al modelo global. Cuando un agente completa 5 episodios, se transfieren los parámetros del modelo global al modelo de ese agente. Y así sucesivamente, durante todo el aprendizaje, con todos los agentes. De esta manera se consigue que los agentes vayan transfiriendo sus parámetros al modelo global y viceversa en momentos distintos y de manera asíncrona, debido a que cada agente finalizará sus episodios en distintos momentos de tiempo, respecto a los demás agentes.

A2C

El algoritmo A2C [OpenAI, 2017] sigue la misma filosofía que A3C, pero con la diferencia, que los parámetros del modelo global, se actualizan de manera síncrona.

Al igual que en A3C, los agentes se van entrenando en paralelo y recolectando experiencias. La diferencia viene a la hora de actualizar el modelo global, que en A2C todos los agentes transfieren sus conocimientos al modelo global a la vez, a través de un coordinador cada cierto número determinado de steps.

En la siguiente imagen se puede apreciar el A2C de manera gráfica:

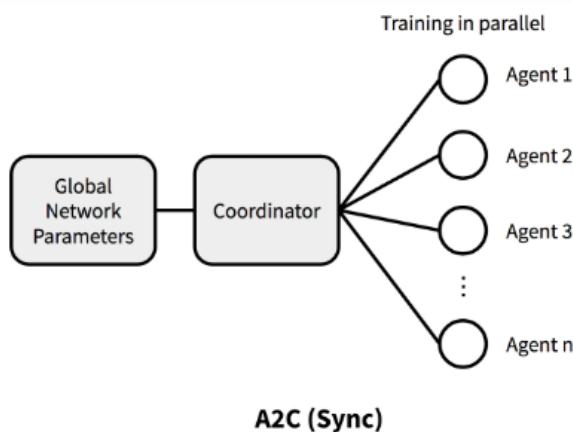


Imagen 10 Esquema del algoritmo de A2C

Implementación

El algoritmo de A2C se ha implementado en código de la siguiente manera:

En este caso, solo hay un único modelo que es el global, en vez de tener un modelo por cada agente y el global, como en A3C. De esta manera, las acciones de todos los agentes, siempre son tomadas por el modelo global.

Cada agente va recolectando experiencias por separado al igual que en A3C, y en vez de ir actualizando el modelo de cada agente, se van guardando los gradientes de cada agente, para actualizar el modelo global cada cierto determinado número de steps, con la media de los gradientes de todos los agentes, de manera sincronizada.

PPO

En PPO [Schulman et al, 2017] lo que se pretende, es que entre las distintas actualizaciones que se van realizando durante el entrenamiento del agente, no cambie mucho la nueva policy respecto a la anterior. Esto se hace calculando la ratio entre la antigua policy y la nueva:

$$r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta old}(a|s)}$$

Luego en la función de coste de PPO, se hace un clip “recortar” a la ratio para que la variación entre la policy antigua y la nueva, no sea más grande que $1 + \varepsilon$ ni más pequeña que $1 - \varepsilon$, dónde ε es un hiperparámetro:

$$L^{CLIP}(\theta) = E[\min(r(\theta)\hat{A}_{\theta old}(s, a), \text{clip}(r(\theta), 1 - \varepsilon, 1 + \varepsilon)\hat{A}_{\theta old}(s, a))]$$

Se toma el mínimo, entre el valor original y la versión recortada, por lo tanto, como se había comentado previamente, se pierde la motivación de incrementar la actualización de la policy para que obtenga mejores recompensas.

Al igual que en Actor-Critic, la función de coste también se le añade un término de error, en la estimación del valor del Critic (fórmula en rojo) y un término de entropía (fórmula en azul) para que el agente tenga la exploración suficiente del entorno:

$$L^{CLIP}(\theta) = E[L^{CLIP}(\theta) - c_1(V_\theta(s) - V_{target})^2 + c_2 H(s, \pi_\theta(.))]$$

Dónde c_1 y c_2 son dos hiperparámetros constantes.

En este algoritmo respecto a Actor-Critic, varia un poco la manera en que se actualiza el modelo. Uno o varios agentes van recolectando experiencias y actualizan el modelo cada cierto número determinado de steps, con la experiencia obtenida entre actualización y actualización (on policy). La actualización se hace mediante batches “fragmentos” aleatorios de la experiencia, en varias pasadas, denominadas épocas. El tamaño del batch y el número de épocas son hiperparámetros.

Implementación

PPO se ha implementado en código de la siguiente manera:

Se utilizan varios agentes para recolectar experiencias, y en la actualización del modelo, se utilizan todas las experiencias obtenidas por todos los agentes. Cada actualización consta de un número determinado de épocas, y en cada época se hacen actualizaciones con batches aleatorios, hasta cubrir toda la experiencia.

Por ejemplo, si la actualización es cada 2048 steps, que vendría a ser el número de experiencias, y el tamaño del batch es de 256, y el número de épocas es 3. En una época, para recorrer toda la experiencia, se tendrán que hacer 8 pasadas ($2048/256 = 8$). Y en cada pasada se utilizará un batch de 256 experiencias aleatorias, sin repetir experiencias durante todas las pasadas. Este mismo proceso se repetirá 3 veces, que viene a ser el número de épocas que se ha establecido en este caso.

En el anexo Pseudocódigos de Algoritmos se pueden encontrar todos los pseudocódigos de los algoritmos de aprendizaje por refuerzo que se han explicado.

Una vez explicado el funcionamiento de los algoritmos de aprendizaje por refuerzo que se van a tratar en este trabajo, es turno de explicar el programa Unity, que es con el cual se crearan los entornos para probar los diferentes algoritmos.

4. Unity

Unity [1] es un motor de videojuegos (2D y 3D) multiplataforma creado por Unity Technologies. Se utiliza para la creación de juegos o aplicaciones para cualquier plataforma (PC, PlayStation, Xbox, Smartphones, etc.) y los lenguajes de programación que se pueden utilizar son C# y JavaScript.

En setiembre de 2017 sacaron una librería llamada ML-Agents [2], donde permitía a los desarrolladores de juegos y a Investigadores de Inteligencia Artificial entrenar agentes utilizando técnicas de aprendizaje por refuerzo.

4.1 ML-Agents

Con esta librería se puede programar un entorno, con uno o varios agentes que tienen que realizar una tarea determinada, y después automáticamente Unity y ML-Agents se encargan de entrenar a los agentes.

En este trabajo se utilizará la versión 0.13.1 (diciembre 2019) de ML-Agents. No se ha actualizado a versiones más recientes por el hecho de que cada mes o dos meses han ido sacando una nueva actualización, dónde había cambios importantes en la manera de cómo se programaba el agente en Unity, y para no cambiar la programación del agente en cada actualización, se ha mantenido una única versión en todo el trabajo.

Desde 2017 ha estado en versión beta y han ido sacando actualizaciones dónde se iban solucionando errores, hasta en mayo de 2020 que ha salido de la beta con una versión bastante estable, la 1.0.

El lenguaje para la creación del entorno y la programación del agente con la librería de ML-Agents es C#.

La librería que utiliza en relación a los modelos de las redes neuronales, es TensorFlow [3].

A continuación, se explicarán las principales zonas del programa de Unity para la creación del entorno y el agente, y luego como se diseñan ambas partes.

4.2 Entorno Unity

En Unity se tiene una pantalla principal con sus respectivas zonas para la creación de un entorno y todos los elementos que lo conforman, incluyendo también las físicas entre los distintos objetos que están en el entorno y su visualización.

En la siguiente imagen se pueden observar las principales zonas de Unity:

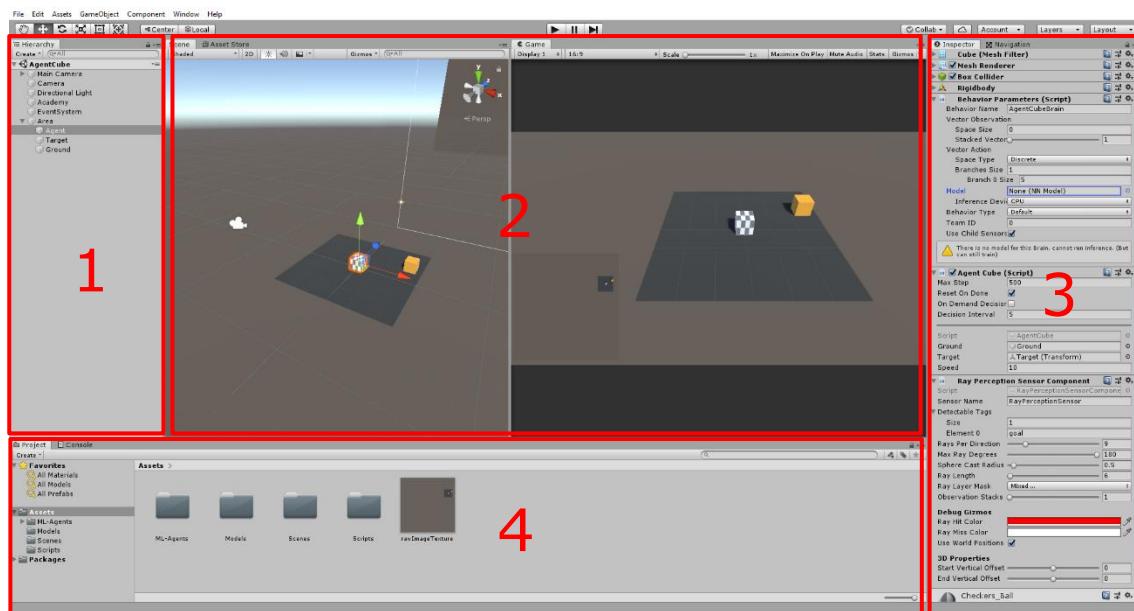


Imagen 11 Entorno de Unity

- **Zona 1:** es en forma de árbol y sirve para la organización de todos los objetos que se encuentran en el entorno; el agente, objetos con los que interactúa, iluminación, cámaras, etc.
- **Zona 2:** es la representación del entorno de todos los objetos y elementos de la zona 1. En la parte izquierda de esta zona, es la parte dónde se monta el entorno y la parte de la derecha es como queda cuando se simula.
- **Zona 3:** son todos los elementos que contiene cada objeto del entorno, como por ejemplo las físicas de los objetos, la visualización, los scripts, etc. En el caso del agente se detallará esta parte en el apartado 4.4 Configuración del Agente.
- **Zona 4:** son los recursos del proyecto, como por ejemplo las librerías, las texturas de los objetos, los modelos de las redes neuronales entrenadas de los agentes, etc.

4.3 Diseño del Agente y el Entorno

Para el diseño del agente y del entorno se tienen que seguir unas pautas, que vienen explicadas en la documentación de la librería de ML-Agents, que se puede encontrar en su repositorio de GitHub [4].

Se diseñan ambas partes a través de scripts, que luego se insertan en los respectivos objetos del entorno que se hagan referencia.

Seguidamente se explica de manera conceptual, la forma en que se programa el agente y el entorno siguiendo las indicaciones de la librería de ML-Agents.

El esquema principal que se sigue en la programación del agente y el entorno es el siguiente:

- **Start:** se empieza con la función Start para inicializar todas las variables del agente y de los distintos objetos que se encuentran en el entorno, dependiendo del entorno pueden variar.
- **AgentReset:** cuando un episodio finaliza, bien porque se ha completado la tarea, se han llegado los máximos steps por episodio, o el agente ha hecho alguna acción en la que se tenga que finalizar el episodio, antes de empezar el siguiente, se tiene que resetear el agente y todas las variables de los diferentes objetos del entorno. De esta manera cada vez que se empieza un nuevo episodio, se inicializa con las mismas condiciones, aunque dependiendo del entorno, las posiciones iniciales del agente y los objetos del entorno pueden variar debido a una cierta aleatoriedad, de esta manera se crea una dificultad para el agente ya que no siempre se encontrará con el mismo estado del entorno y tendrá que aprender a completar la tarea en distintos casos.
- **MoveAgent:** en esta función se programan todas las posibles acciones que puede realizar el agente. Como por ejemplo los movimientos que puede hacer, ir hacia adelante, atrás, izquierda, derecha, etc.
- **AgentAction:** es la acción que toma el agente en cada step, dentro de la lista de acciones de la anterior función, MoveAgent.
- **Otras funciones:** se utilizan otras funciones para detectar cuando se tiene que recompensar el agente, positivamente o negativamente, dependiendo de si ha tomado una buena o mala acción. Por ejemplo, la detección de la colisión del agente con otro objeto del entorno. También se emplean funciones para gestionar el funcionamiento del entorno, como podría ser el control de la vida del agente.

En el anexo de Unity, apartado Script del Agente y el Entorno, se puede encontrar de manera más detallada la programación de ambas partes.

Una vez se tiene programado el agente y el entorno en uno o diversos scripts, se insertan en sus lugares correspondientes en el entorno de Unity. En el caso del agente, también se configuran los parámetros para el entrenamiento, la manera en la que se captan las observaciones y parámetros propios del agente y el entorno, como podría ser la velocidad del agente.

4.4 Configuración del Agente

Antes del entrenamiento del agente, se tienen que configurar diversos parámetros. Y también, la manera en que el agente va a obtener las observaciones del entorno. En Unity se puede hacer de tres maneras distintas:

- **Script:** se le puede pasar al agente directamente las posiciones y velocidades de los distintos objetos del entorno y del agente, en el script principal del agente y el entorno, que se ha comentado en el anterior apartado.
- **Raycast:** a través de rayos que proyecta el agente, que le sirven para detectar y distinguir los diferentes objetos en el entorno.
- **Imágenes:** mediante una cámara que se puede colocar dónde convenga, puede ser una vista general del entorno, o una vista en primera persona del agente. Y en cada step se le proporcionaría una imagen al agente, como observación.

En este trabajo se ha optado por la opción de Raycast, por ser una manera natural de captación de observaciones, debido a que es lo que está viendo el agente en cada momento. También se pueden combinar las dos primeras opciones, Raycast para la detección de los objetos del entorno, y por script para pasárselos indicadores, como podría ser la vida del agente.

El agente siempre tiene los siguientes scripts, en los que se encuentran todos los parámetros que se tienen que configurar para su entrenamiento:

- **Behaviour Parameters:** principalmente en este script se configuran los vectores de observación y de acciones. En el caso de los vectores de observación solo se configuran si la captación de observaciones se hace a través de un script. Este script es de la librería ML-Agents.
- **Script del Agente y el Entorno:** es dónde se encuentra la programación principal del agente y el entorno, que se ha comentado en el anterior apartado. Con parámetros que podrían ser la velocidad del agente, número de vidas, etc.

- **Ray Perception Sensor:** es el script que usa Unity para Raycast, y tiene un conjunto de parámetros, para poder configurar los rayos que proyecta el agente.

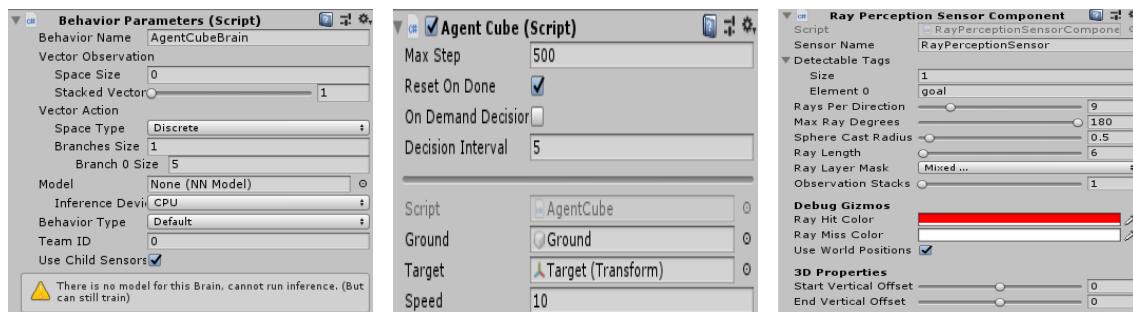


Imagen 12 Parámetros del Agente

En el anexo de Unity, apartado Parámetros del Agente, se pueden encontrar todos los parámetros de los diferentes scripts, explicados de manera detallada.

4.5 Entrenamiento del Agente

En Unity el algoritmo de aprendizaje por refuerzo que utiliza para entrenar al agente, es PPO. Para realizar el entrenamiento, se tienen que configurar previamente los parámetro de este algoritmo y de la red neuronal en un bloc de notas (con la extensión .yaml).

```
default:
  trainer: ppo
  batch_size: 256
  beta: 1.0e-2
  buffer_size: 2048
  epsilon: 0.2
  hidden_units: 256
  lambd: 0.95
  learning_rate: 1e-3
  learning_rate_schedule: linear
  max_steps: 5.0e6
  memory_size: 256
  normalize: false
  num_epoch: 3
  num_layers: 2
  time_horizon: 64
  sequence_length: 64
  summary_freq: 1000
  use_recurrent: false
  vis_encode_type: simple
  reward_signals:
    extrinsic:
      strength: 1.0
      gamma: 0.99
```

Después se ejecuta la siguiente línea de código en un terminal para empezar el entrenamiento:

```
mlagents-learn C:\config.yaml --run-id=AgentCube --time-scale=100 --train
```

El parámetro time-scale es un factor multiplicativo del transcurso del tiempo en la simulación, un mayor valor, significa que el tiempo transcurrirá más rápido. El valor 100 es el máximo por tema de físicas del entorno, debido a que más velocidad el rendimiento las colisiones, por ejemplo, no se detectarían correctamente.

Durante el entrenamiento, se puede visualizar gráficamente como los agentes van aprendiendo, y en el terminal se van indicando los steps que se han realizado, el tiempo transcurrido y las recompensas obtenidas.

5. Entornos

En este capítulo se explicarán los entornos que se han creado, para el entrenamiento de agentes en diversas situaciones y para probar los diferentes algoritmos de aprendizaje por refuerzo que se han mencionado.

5.1 Entorno 1

La finalidad del primer entorno ha sido básicamente probar diversas cosas. Que el agente aprendiera en Unity a través de la librería de ML-Agents, la comunicación entre el entorno exportado y Python, y que aprendiera con el primer algoritmo de este trabajo, que ha sido DQN. De esta manera se comprobaba que la base de este trabajo funcionaba correctamente.

En este entorno el agente (cubo negro/blanco) se encuentra en una plataforma, en la que se puede caer de ella, y su objetivo es tocar al cubo naranja.

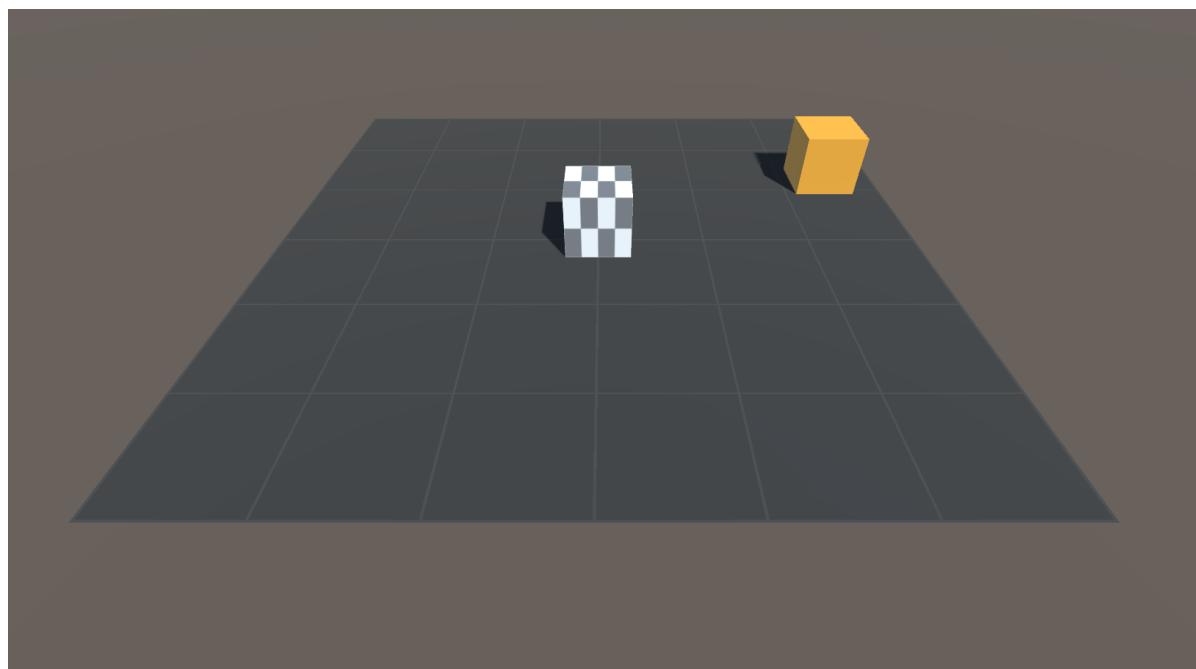


Imagen 13 Entorno 1

Especificaciones

- **Agente:** cubo negro/blanco.
- **Acciones:** ir hacia adelante, atrás, a derecha y a izquierda.
- **Elementos:** cubo naranja.
- **Posiciones iniciales:** el agente siempre se inicializa en el medio de la plataforma y el cubo naranja en una posición aleatoria por toda la plataforma.
- **Objetivo:** tocar el cubo naranja.
- **Recompensas:** +1 cuando el agente toca al cubo naranja y -1 cuando se cae de la plataforma.
- **Vector de observaciones (Raycast):** el agente es capaz de ver en todo momento, lo que tiene alrededor suyo con un ángulo de visión de 360º, con una longitud de rayos superior a la distancia entre el medio de la plataforma y el borde. El tamaño del vector de observaciones es de 57.

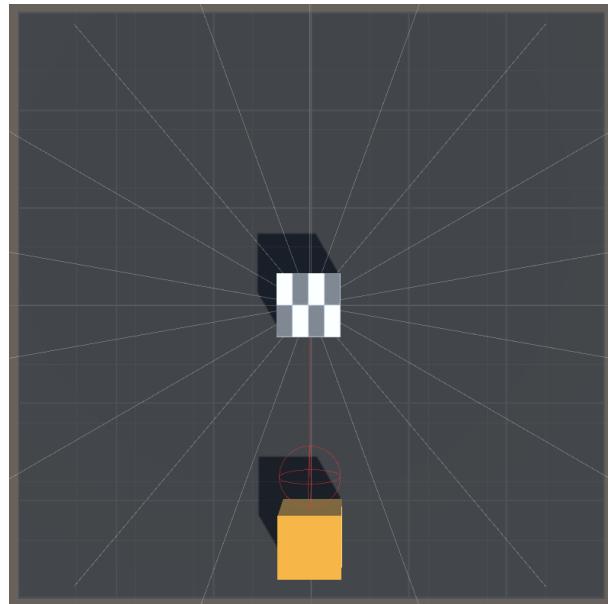


Imagen 14 Entorno 1 (Raycast)

5.2 Entorno 2

El segundo entorno es muy similar al primero, pero se le añade cierta dificultad, que consiste en que el agente (cubo azul) ahora tiene un campo de visión mucho menor, y por lo tanto tenga que moverse, no solo para tocar simplemente su objetivo (cubo naranja), sino también para buscar dónde está. En el primer entorno, el agente indiferentemente de la posición en la que estuviera, siempre detectaba al cubo naranja.

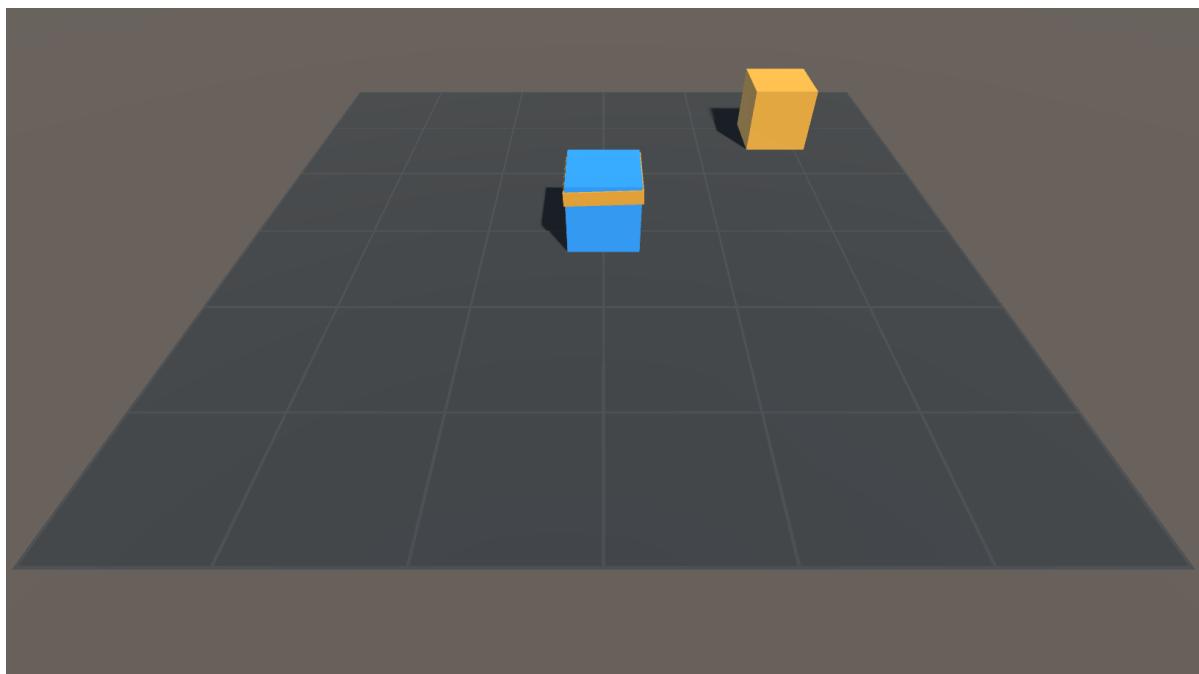


Imagen 15 Entorno 2

Especificaciones

- **Agente:** cubo azul.
- **Acciones:** ir hacia adelante o atrás y rotación a derecha o a izquierda.
- **Elementos:** cubo naranja.
- **Posiciones iniciales:** el agente siempre se inicializa en el medio de la plataforma y el cubo naranja en una posición aleatoria por toda la plataforma.
- **Objetivo:** tocar el cubo naranja.
- **Recompensas:** +1 cuando el agente toca al cubo naranja y -1 cuando se cae de la plataforma.
- **Vector de observaciones (Raycast):** el agente puede ver lo que tiene alrededor suyo con un ángulo de visión de 90º hacia adelante, con una longitud de rayos superior a la distancia entre el medio de la plataforma y el borde. El tamaño del vector de observaciones es de 39.

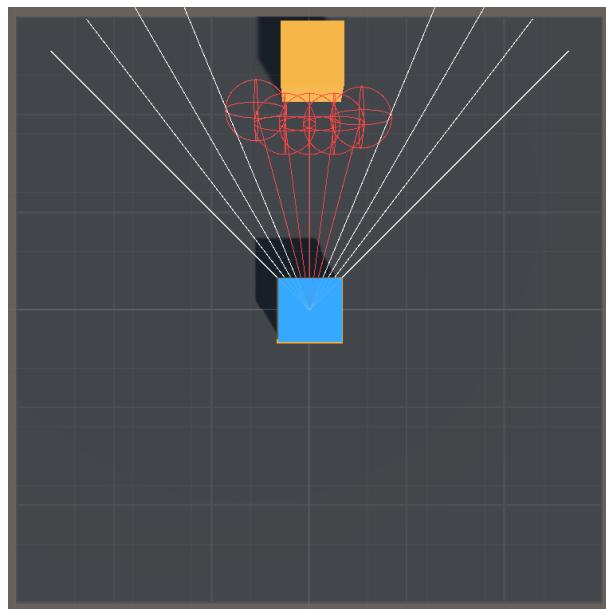


Imagen 16 Entorno 2 (Raycast)

5.3 Entorno 3

En este entorno el objetivo del agente es meter la pelota dentro de la portería. Hasta ahora el agente solo tenía que aprender una sola relación, tocar su objetivo. En cambio, en este entorno tiene que aprender más de una relación, tocar la pelota para después meterla en la portería.

El agente hace un primer lanzamiento, si la pelota va a dentro de la portería se finaliza el episodio, y si falla, la pelota rebotará en alguna de las paredes del campo y el agente podrá seguir intentándolo, hasta que lo logre, dentro del límite de máximos steps por episodio. Esto hace que la dificultad aumente considerablemente debido a la multitud de situaciones que se pueden llegar a dar.



Imagen 17 Entorno 3

Especificaciones

- **Agente:** cubo azul.
- **Acciones:** ir hacia adelante o atrás y rotación a derecha o a izquierda.
- **Elementos:** pelota y portería.
- **Posiciones iniciales:** la pelota se inicializa de manera aleatoria en todo el ancho del campo, pero siempre a la misma distancia en perpendicular respecto a la portería, y lo mismo con el agente, tal y como se puede observar en la imagen.
- **Objetivo:** meter la pelota dentro de la portería.
- **Recompensas:** +1 cuando el agente mete la pelota dentro de la portería.
- **Vector de observaciones (Raycast):** el agente puede ver lo que tiene alrededor suyo con un ángulo de visión de 120º hacia adelante, con una longitud de rayos igual que la distancia total del campo. Para saber la dirección en la que se está moviendo la pelota, se hace un stack de observaciones, de 5. El tamaño del vector de observaciones es de 420.



Imagen 18 Entorno 3 (Raycast)

5.4 Entorno 4

Este entorno tiene 2 agentes idénticos y lo que se ha pretendido es que cooperen entre ellos. El objetivo de ambos, es que uno de ellos, no importa cuál sea, entre en la zona verde. Para poder entrar en esta zona, uno de ellos debe permanecer en la zona naranja, para que la pared azul desaparezca y el otro agente pueda acceder a la zona verde.

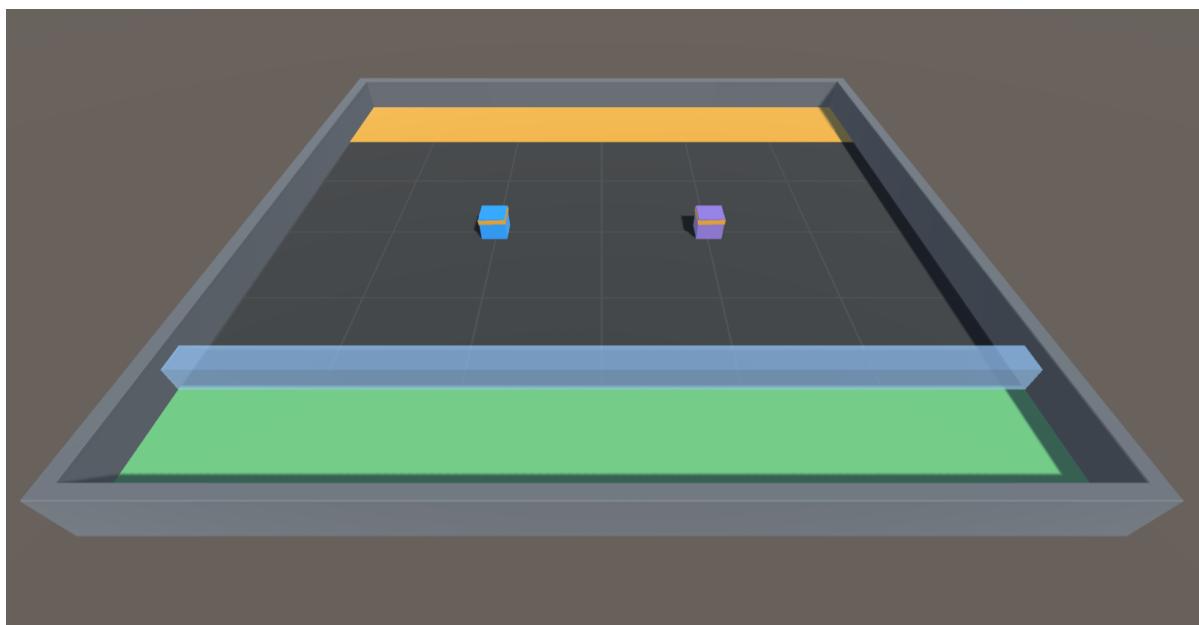


Imagen 19 Entorno 4

Especificaciones

- **Agentes:** cubo azul y púrpura.
- **Acciones:** ir hacia adelante o atrás y rotación a derecha o a izquierda.
- **Elementos:** zona naranja y verde y pared azul.
- **Posiciones iniciales:** los dos agentes se inicializan de manera aleatoria en cualquier posición entre la zona naranja y verde.
- **Objetivo:** que uno de los dos agentes, no importa cuál sea, entre en la zona verde. Mientras un agente está en la zona naranja, la pared azul desaparece, y de esta manera el otro agente puede acceder a la zona verde. Si el agente que está en la zona naranja, se sale de ella, la pared vuelve a aparecer.
- **Recompensas:** +1 cuando uno de los dos agentes entra en la zona verde.
- **Vector de observaciones (Raycast):** el agente puede ver lo que tiene alrededor suyo con un ángulo de visión de 180º hacia adelante, con una longitud de rayos igual que la distancia total del campo. Para saber la dirección en la que se está moviendo un agente respecto al otro, se hace un stack de observaciones, de 3. El tamaño del vector de observaciones es de 342.

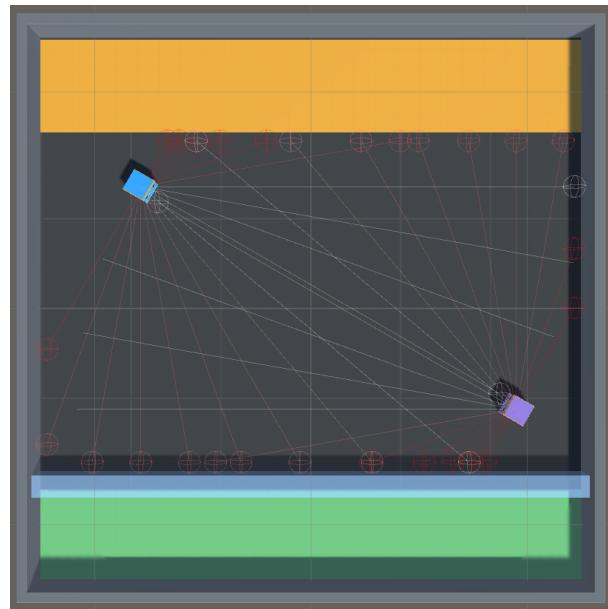


Imagen 20 Entorno 4 (Raycast)

5.5 Entorno 5

En el último entorno, se pretende que el agente aprenda una tarea más complicada, que implique relacionar un mayor número de cosas, en comparación a los entornos anteriores. También es un entorno muy estocástico, por lo tanto, el agente también tendrá que explorar y aprender una gran variedad de situaciones.

El agente tiene como objetivo entrar en la zona verde, para ello primero, tiene que colocar los dos bloques en cada zona de su respectivo color, luego la pared azul desaparecerá y podrá acceder a la zona verde.

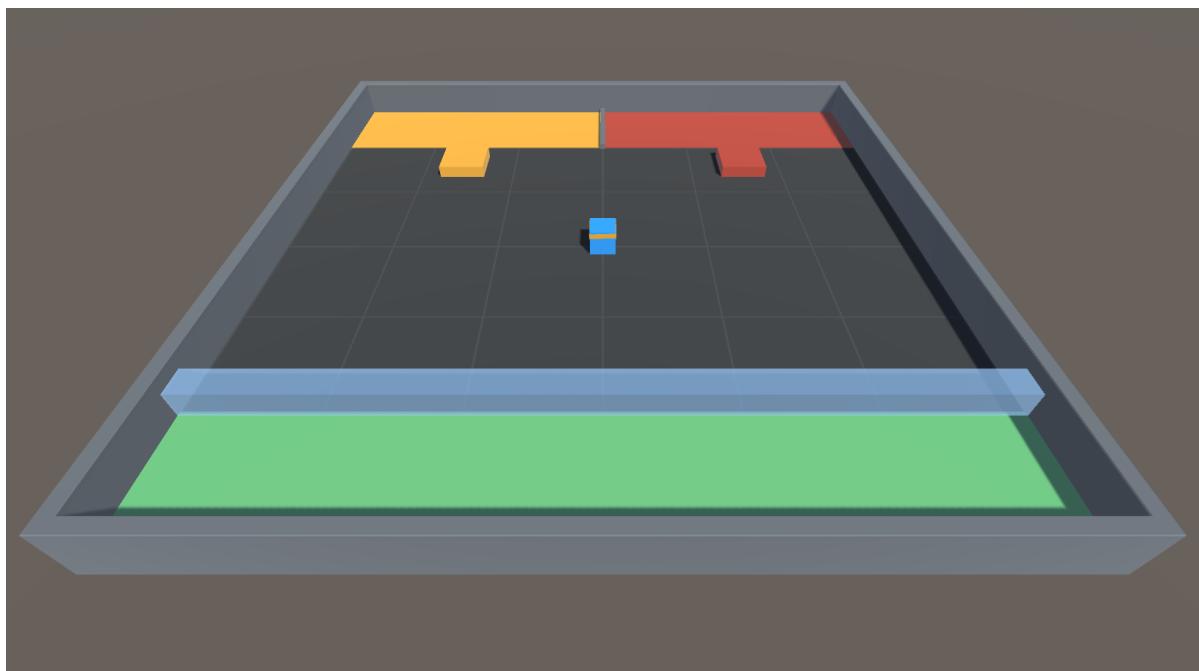


Imagen 21 Entorno 5

Especificaciones

- **Agente:** cubo azul.
- **Acciones:** ir hacia adelante o atrás y rotación a derecha o a izquierda.
- **Elementos:** zona naranja, roja y verde, bloque naranja y rojo y pared azul.
- **Posiciones iniciales:** los bloques y el agente se inicializan en una posición aleatoria entre las zonas naranja/roja y la zona verde.
- **Objetivo:** entrar en la zona verde. Primero no se puede acceder porque la pared azul transparente se lo impide. Para hacerla desaparecer, se tiene que colocar el bloque naranja en la zona naranja y el bloque rojo en la zona roja (no importa el orden en el que se haga). Una vez hecho esto, desaparece la pared azul transparente, y de esta manera el agente ya puede entrar en la zona verde.
- **Recompensas:** +0.2 cuando coloca 1 bloque, +0.6 cuando coloca 2 bloques, +1 cuando entra en la zona verde y -1 si coloca un bloque en la zona incorrecta, por ejemplo, colocar el bloque naranja en la zona roja.
- **Vector de observaciones (Raycast):** el agente puede ver lo que tiene alrededor suyo con un ángulo de visión de 180º hacia adelante, con una longitud de rayos igual que la distancia total del campo. Para saber la dirección en la que se están moviendo los bloques, se hace un stack de observaciones, de 3. El tamaño del vector de observaciones es de 399.

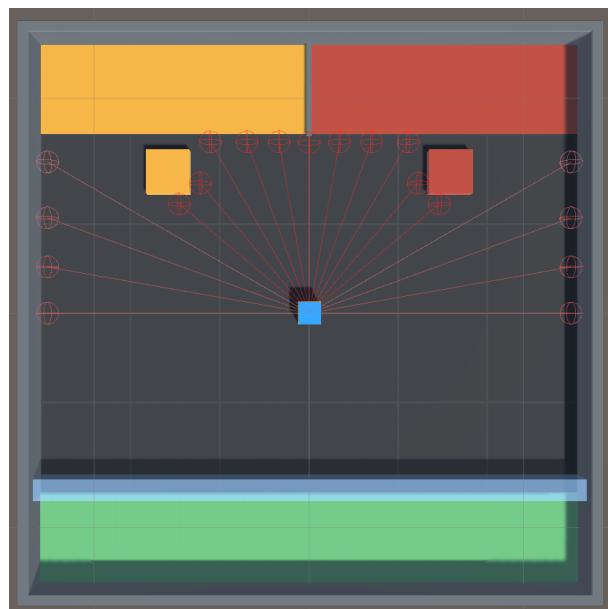


Imagen 22 Entorno 5 (Raycast)

Para que el agente, en los diversos entornos aprenda a realizar las respectivas tareas de manera más rápida, por cada step que pasa y sigue sin completar su objetivo, obtiene una recompensa negativa de -1/máximos steps por episodio. De esta manera, como el agente nunca va a lograr la recompensa total (+1) debido a qué ninguna tarea se completa con menos de un step, la recompensa siempre se podrá mejorar hasta llegar a un cierto límite, que estará marcado por las mecánicas del entorno.

6. Implementación

Una vez se tiene el entorno creado, desde Unity se exporta el entorno en un ejecutable, que luego se utilizará desde Python para realizar el entrenamiento.

6.1 Comunicación entre Python y Unity

A continuación, se explica el funcionamiento de la comunicación entre el entorno exportado de Unity y Python. En la siguiente imagen se puede observar el flujo de la información que se va intercambiando entre Python (dónde se entrena el agente con el entorno exportado de Unity), Gym y la API de Python de ML-Agents.



Imagen 23 Flujo de información entre Python y Unity

En Python, se realiza el entrenamiento del agente aplicando los diferentes algoritmos de aprendizaje por refuerzo, en los entornos que se han creado con Unity. Gym de ML-Agents, hace de puente entre el entorno exportado y la API de ML-Agents para la captación de información (ejecución de las acciones en el agente, estados, recompensas, etc.) del entorno exportado.

Gym de ML-Agents, sigue la misma estructura que la librería de Gym creada por Open AI [5], para la captación de información de los entornos.

6.2 Estructura del código

En este apartado se explica la estructura general del código de los diferentes algoritmos que se han aplicado. El código que se ha utilizado para la explicación es del algoritmo de PPO.

Inicialización

La primera parte del código, se encuentra la inicialización de todos los parámetros que se van a utilizar durante el entrenamiento, que son los hiperparámetros, el entorno, el modelo de la red neuronal y el buffer del agente dónde irá guardando las experiencias.

Hiperparámetros

En este apartado se configuran los hiperparámetros del algoritmo de aprendizaje por refuerzo. Entre los diferentes algoritmos, los hiperparámetros no siempre son los mismos.

```
# Hyperparameters
self.learning_rate = 0.0003
self.betas = (0.9, 0.999)
self.gamma = 0.99
self.eps_clip = 0.2
self.buffer_size = 2048
self.batch_size = 256
self.K_epochs = 3
self.max_steps = 50000

self.tau = 0.95
self.entropy_coef = 0.001

self.summary_freq = 1000
```

Entorno

Aquí es dónde se inicializa el entorno que se ha exportado de Unity.

```
# Environment
self.env_name = Environments/env5_M/Unity Environment"
channel = EngineConfigurationChannel()
self.env = UnityEnv(self.env_name, worker_id=0, use_visual=False,
                    side_channels=[channel], no_graphics = False, multiagent = True)
channel.set_configuration_parameters(time_scale = 100)
self.action_size, self.state_size = Utils.getActionStateSize(self.env)
self.n_agents = self.env.number_agents
print("Nº Agents: ",self.n_agents)
```

Como se ha comentado anteriormente, el parámetro `time_scale` de Unity es un factor multiplicativo del transcurso del tiempo, con Gym de ML-Agents predeterminado, no se puede acceder a este parámetro, y por lo tanto el entrenamiento se produciría a una velocidad de x1. Por este motivo, se han modificado los archivos "gym_unity" y "mlagents_envs" de la librería ML-Agents, para poder acceder al parámetro `time_scale`.

Con esta modificación, se pueden acceder a variables de Unity a través de Python utilizando Gym de ML-Agents, y mediante el canal de comunicación `EngineConfigurationChannel`, se puede acceder al parámetro `time_scale`.

También se pueden crear otros canales de comunicación y escoger otras variables que se quieran acceder.

Modelo

En esta parte del código se declara el modelo de la red neuronal que va utilizar el agente para su aprendizaje. La librería que se ha escogido en este trabajo para la creación de los modelos de las redes neuronales, es PyTorch [6]. Por el hecho de proporcionar una mayor accesibilidad a la hora de modificar cualquier cosa al respecto, en comparación a TensorFlow.

```
# Model
self.model = ActorCritic(self.state_size, self.action_size, seed = 0).to(device)
self.optimizer = optim.Adam(self.model.parameters(), lr=self.learning_rate)
self.MseLoss = nn.MSELoss()
```

La arquitectura de la red neuronal que se ha utilizado para este trabajo, es la siguiente; una primera capa con un tamaño igual al del vector de observaciones, cuatro capas ocultas con el tamaño de 256, 324, 512 y 128 neuronas, respectivamente, una capa de salida para el actor, con un tamaño igual al del vector de acciones y otra capa de salida para el critic con el tamaño de 1.

```
class ActorCritic(nn.Module):
    def __init__(self, state_size, action_size, seed):
        super(ActorCritic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.l1 = nn.Linear(state_size,256)
        self.bn1 = nn.BatchNorm1d(256)
        self.dp1 = nn.Dropout(0.2)
        self.l2 = nn.Linear(256,324)
        self.bn2 = nn.BatchNorm1d(324)
        self.l3 = nn.Linear(324,512)
        self.bn3 = nn.BatchNorm1d(512)
        self.l4 = nn.Linear(512,128)
        self.bn4 = nn.BatchNorm1d(128)
        self.actor = nn.Linear(128,action_size)
        self.critic = nn.Linear(128,1)

    def forward(self, x):
        x = self.dp1(F.relu(self.bn1(self.l1(x))))
        x = F.relu(self.bn2(self.l2(x)))
        x = F.relu(self.bn3(self.l3(x)))
        x = F.relu(self.bn4(self.l4(x)))
        policy = self.actor(x)
        value = self.critic(x)
        return policy, value
```

Buffer

El Buffer es la memoria del agente dónde irá guardando las experiencias (acciones, estados, recompensas, etc.) para su aprendizaje.

```
# Buffer memory
self.memory = []
for _ in range(self.n_agents):
    self.memory.append(Buffer())
```

Entrenamiento

Este es el bucle del entrenamiento. El agente va tomando acciones a partir de los estados del entorno, se van guardando las experiencias para ir actualizando el modelo de la red neuronal y de esta manera ir aprendiendo la tarea que se tiene que realizar. Durante el entrenamiento, se van mostrando valores como los steps que se han realizado, tiempo transcurrido, recompensas obtenidas, para poder hacer un seguimiento del entrenamiento. También se guardan valores, para mostrar los resultados a través de gráficas, una vez finalizado el entrenamiento.

```
def train(self):
    # Initial observation
    env_info = self.env.reset()
    state = env_info
    # Data
    self.data = Data(self.n_agents, self.summary_freq)
    # Training loop
    for _ in range(self.max_steps):
        action = []
        logprobs = []
        value = []
        # Action of agent
        for i in range(self.n_agents):
            a,b,c = self.act(state[i])
            action.append(a)
            logprobs.append(b)
            value.append(c)
        # Send the action to the environment
        next_state, reward, done, info = self.env.step(action)
        # Done
        done_ = []
        for i in range(self.n_agents):
            done_.append(1-done[i])
        # Agent step
        for i in range(self.n_agents):
            self.step(state[i], action[i], reward[i], next_state[i],
                      done_[i], logprobs[i], value[i], self.memory[i])
        # Update t_step
        self.t_step += 1
        # Next state
        state = next_state
        # Update the score
        self.data.update_score(reward, value, done, self.t_step)
        # Summary
        if self.t_step % self.summary_freq == 0:
            self.data.summary(self.t_step)
    # Save
    self.save()
```

Agente

Durante el entrenamiento, el agente va tomando acciones, guardando las experiencias y aprendiendo. Esto se hace a través de las siguientes funciones

Acciones

Esta es la función con la cual el agente toma una acción a partir del estado del entorno.

```
def act(self, state):
    state = torch.from_numpy(state).float().unsqueeze(0).to(device)
    # Get actions probabilities and value from ActorCritic model
    self.model.eval()
    with torch.no_grad():
        action_probs, value = self.model(state)
    self.model.train()
    prob = F.softmax(action_probs, -1)
    log_probs = F.log_softmax(action_probs, -1)
    # Get action and log of probabilities
    action = prob.multinomial(num_samples=1)
    log_probs = log_probs.gather(1, action)
    return action, log_probs, value
```

Experiencias

Por cada step que realice el agente, se guarda la experiencia de ese step, que vendría a ser, el estado del entorno, la acción que ha tomado el agente, la recompensa que ha obtenido, el próximo estado del entorno, si el episodio se ha terminado y otros parámetros que dependen del algoritmo de aprendizaje por refuerzo que se utilice.

Con esta función también se gestiona cuando se tiene que realizar el aprendizaje del agente.

```
def step(self, state, action, reward, next_state, done, logprobs, value, memory):
    # Update model when buffer_size is full
    if memory.len_() == (self.buffer_size/self.n_agents):
        self.learn()
        for i in range(self.n_agents):
            self.memory[i].reset()
    # Save experience in buffer memory
    memory.add(state, action, reward, next_state, done, logprobs, value)
```

Aprendizaje

Esta es la función que se utiliza para que el agente haga la actualización del modelo de la red neuronal, calculando la función de coste con las experiencias guardadas. Esta función varía dependiendo del algoritmo de aprendizaje por refuerzo que se utilice.

```

def learn(self):
    # Get Experiences
    states, actions, rewards, next_states, dones, logprobs_, values_ = self.getExp()
    returns_eval = self.compute_returns()
    # Optimize policy for K epochs:
    for _ in range(self.K_epochs):
        # List with all indices
        l = np.arange(self.buffer_size)
        l = list(l)
        x = self.buffer_size // self.batch_size
        for _ in range(x):
            # Take a random batch
            indices = random.sample(l, self.batch_size)
            old_logprobs = torch.empty(self.batch_size, 1)
            old_values = torch.empty(self.batch_size, 1)
            old_actions = torch.empty(self.batch_size)
            old_states = torch.empty(self.batch_size, self.state_size)
            old_next_states = torch.empty(self.batch_size, self.state_size)
            old_rewards = np.zeros(self.batch_size)
            returns = torch.empty(self.batch_size, 1)
            for i in range(len(indices)):
                old_logprobs[i] = logprobs_[indices[i]]
                old_values[i] = values_[indices[i]]
                old_actions[i] = actions[indices[i]]
                old_states[i] = states[indices[i]]
                old_next_states[i] = next_states[indices[i]]
                old_rewards[i] = rewards[indices[i]]
                returns[i] = returns_eval[indices[i]]
            # Remove indices to not repeat
            for i in indices:
                l.remove(i)
            # Evaluate
            logprobs, state_values, dist_entropy, _ =
                self.evaluate(old_states, old_next_states, old_actions,
                            rewards, dones, compute_gae = False)
            # Finding the ratio (pi_theta / pi_theta_old):
            ratios = torch.exp(logprobs - old_logprobs)
            # Finding Surrogate Loss:
            advantages = returns - old_values
            surr1 = ratios * advantages
            surr2 = torch.clamp(ratios, 1-self.eps_clip, 1+self.eps_clip)*advantages
            # LOSS = ACTOR_LOSS+CRITIC_DISCOUNT*CRITIC_LOSS-ENTROPY_BETA*ENTROPY
            loss = - torch.min(surr1,surr2)+self.value_loss_coeff*
                  self.MseLoss(state_values, returns)-
                  self.entropy_coeff*dist_entropy
            # Optimizer step
            self.optimizerStep(self.optimizer, loss.mean())

```

Otras funciones

En este apartado se comentan otras funciones, que también se utilizan para el aprendizaje del agente. Y se explican las funciones para guardar los resultados de un entrenamiento y para reanudarlo.

Funciones relacionadas con el algoritmo de aprendizaje

Cada algoritmo de aprendizaje por refuerzo es diferente, de modo, que cada uno tiene sus respectivas funciones, para el cálculo de ciertos valores y sobre la utilización de las experiencias del agente.

Guardar

Con esta función, se guarda el modelo de la red neuronal del agente que se ha estado entrenando, también los resultados en formato de gráficas del entrenamiento, con la utilización de las librerías de NumPy [7], Matplotlib [8] y Seaborn [9], también se guarda en formato de dataframe de la librería Pandas [10].

```
def save(self):  
    torch.save(self.model.state_dict(), 'Saved Models/model.pth')  
    self.data.results()
```

Cargar modelo

Esta función, se utiliza para cargar un modelo de una red neuronal, que se quiera seguir entrenando. De esta forma, se puede pausar el entrenamiento y seguir en otro momento.

```
def load_model(self, model):  
    self.model.load_state_dict(torch.load(model))
```

Una vez se ha visto la estructura principal del código, para entrenar un agente en un entorno determinado, a continuación, se explica cómo ejecutar el código y la información que se puede ver durante el entrenamiento.

6.3 Visualización del entrenamiento

Para empezar el entrenamiento del agente, se ejecutan las siguientes líneas de código en un terminal:

```
from Agent_PPO import PPO
agent = PPO()
agent.train()
```

Si se quiere reanudar un entrenamiento, con un modelo guardado previamente, se añade la función load_model en el código:

```
from Agent_PPO import PPO
agent = PPO()
agent.load_model("Saved Models/Env5_ppo_model.pth")
agent.train()
```

Una vez se han ejecutado las líneas anteriores, se obtiene la siguiente información sobre el entorno:

```
Action size: 5
State size: 399
Nº of Agents: 16
```

Dónde se indica el número de acciones que puede realizar el agente, el tamaño del vector de observaciones y el número de agentes.

Durante el entrenamiento, se va mostrando información sobre el desempeño del agente en el entorno, y de esta manera, se puede ver la progresión de su aprendizaje:

```
Step: 1000    Time Elapsed: 44 s    Mean Reward: 0.380    Std of Reward: 0.902
Step: 2000    Time Elapsed: 89 s    Mean Reward: 0.703    Std of Reward: 0.618
Step: 3000    Time Elapsed: 132 s   Mean Reward: 0.792    Std of Reward: 0.485
Step: 4000    Time Elapsed: 174 s   Mean Reward: 0.829    Std of Reward: 0.414
Step: 5000    Time Elapsed: 215 s   Mean Reward: 0.848    Std of Reward: 0.371
Step: 6000    Time Elapsed: 256 s   Mean Reward: 0.861    Std of Reward: 0.338
Step: 7000    Time Elapsed: 303 s   Mean Reward: 0.869    Std of Reward: 0.315
Step: 8000    Time Elapsed: 347 s   Mean Reward: 0.876    Std of Reward: 0.297
Step: 9000    Time Elapsed: 389 s   Mean Reward: 0.881    Std of Reward: 0.283
Step: 10000   Time Elapsed: 431 s   Mean Reward: 0.885    Std of Reward: 0.269
```

Imagen 24 Información del entrenamiento

Se indica los steps que se han realizado, el tiempo transcurrido, la recompensa media obtenida y su desviación estándar.

También se puede observar visualmente como los agentes van aprendiendo, tal y como se muestra en la siguiente imagen:

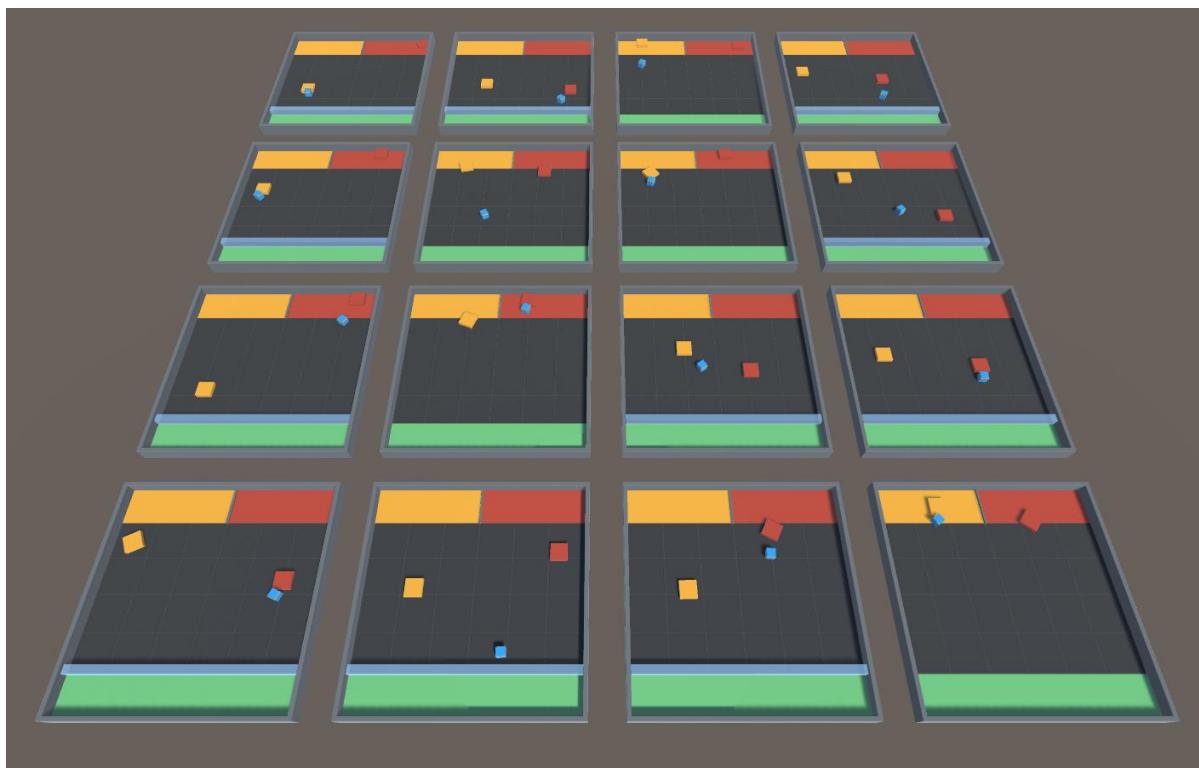


Imagen 25 Entrenamiento agentes

7. Resultados

En este capítulo se exponen los resultados obtenidos en los diferentes experimentos que se han realizado y que se explicaran a continuación.

7.1 Experimentos

Lo que se ha pretendido con los siguientes experimentos, es probar las implementaciones de los algoritmos de aprendizaje por refuerzo que se han expuesto en este trabajo.

A continuación, se explican las bases que se han utilizado en los experimentos.

Modelo de la red neuronal

En todos los experimentos se ha utilizado el mismo modelo de red neuronal, debido a que un punto fuerte del aprendizaje por refuerzo profundo, es la no dependencia, del ajuste empíricamente de los hiperparámetros de las redes neuronales, para la funcionalidad de los algoritmos.

Número de agentes

En los algoritmos que se realiza el aprendizaje con múltiples agentes, como es el caso de A3C, A2C y PPO, el número de agentes que se ha escogido para llevar acabo los diferentes experimentos, es de 16.

Cuantos más agentes, esto implica un mayor tiempo de ejecución por parte del algoritmo, por lo tanto, si se escoge un valor muy elevado de agentes, el tiempo de computación será muy grande, y si se escoge un número pequeño no se tendrá tanta exploración del entorno.

Por este motivo, se decide la elección de un valor intermedio, como es el caso de 16 agentes.

Hiperparámetros

Los hiperparámetros también se ha procurado que no cambien entre los diferentes experimentos. Se han escogido de tal manera, para que el entrenamiento se lo más estable posible. Se han probado empíricamente con una base teórica, y la configuración final que se ha elegido para los experimentos es la siguiente:

- **learning_rate = 0.0003**, es un valor pequeño para conseguir que, en cada actualización del modelo, no sea muy significativa y de esta manera la red neuronal, no realice un gran cambio en sus pesos en función del conjunto de

experiencias del agente, con el que se ha hecho la actualización del modelo. Esto sirve para que el agente, aprenda con el conjunto en general de las experiencias a lo largo del entrenamiento, y no inmediatamente con cada actualización, ya que esto significaría, que cada vez que se actualizará el modelo, habría un gran cambio en el comportamiento del agente, porque su aprendizaje iría en función solo de las ultimas experiencias, con las que ha hecho la actualización del modelo.

· **gamma = 0.99**, en el caso del factor de descuento para futuras recompensas, se elige un valor grande, para que el agente tome las decisiones pensando en la recompensa que pueda obtener en un futuro, en vez de esperar recompensas inmediatas.

· **entropía = 0.01** (DQN) y **0.001** (Policy Gradients), son diferentes debido a que se aplican de manera distinta en los dos tipos de algoritmos. En ambos casos se escoge un valor pequeño, para que la posibilidad en que el agente tome una acción aleatoria no sea muy elevada, pero sí que esté presente. De esta manera se evita que, el agente se quede estancado en algún mínimo o máximo local, al forzarlo a explorar nuevos estados con una acción aleatoria.

· **batch_size**, este hiperparámetro (número de experiencias que se van a utilizar para actualizar el modelo) sí que se ajusta en función del entorno, debido a que, dependiendo del entorno, interesa tener un valor pequeño o grande. En entornos dónde, el agente puede completar la tarea de manera rápida en términos de steps, no se necesita un batch_size muy grande, en cambio en entornos dónde se requiera de más pasos por parte del agente, sí se requiere un valor más elevado. La preferencia, es escoger un valor grande dentro de lo que cabe, para obtener una mayor estabilidad en el entrenamiento, porque de esta manera no se actualiza tan frecuentemente el modelo.

Como se puede apreciar, lo que se pretende es que, el agente durante el entrenamiento vaya aprendiendo de manera estable sin que cambie de manera repentina su comportamiento, y dejando un cierto grado de aleatoriedad en sus acciones para que siempre pueda seguir explorando.

En la siguiente tabla se muestra el batch_size que se ha utilizado en los diferentes entornos con los distintos algoritmos:

	Entorno 1	Entorno 2	Entorno 3	Entorno 4	Entorno 5
DQN	1024	1024	1024	-	-
Actor-Critic	64	256	512	-	-
A3C	64	256	512	128	-
A2C	256	256	256	128	-
PPO	256	256	256	256	256

Todos los hiperparámetros se pueden encontrar explicados de manera más detallada en el anexo de Hiperparámetros.

Exposición de los resultados

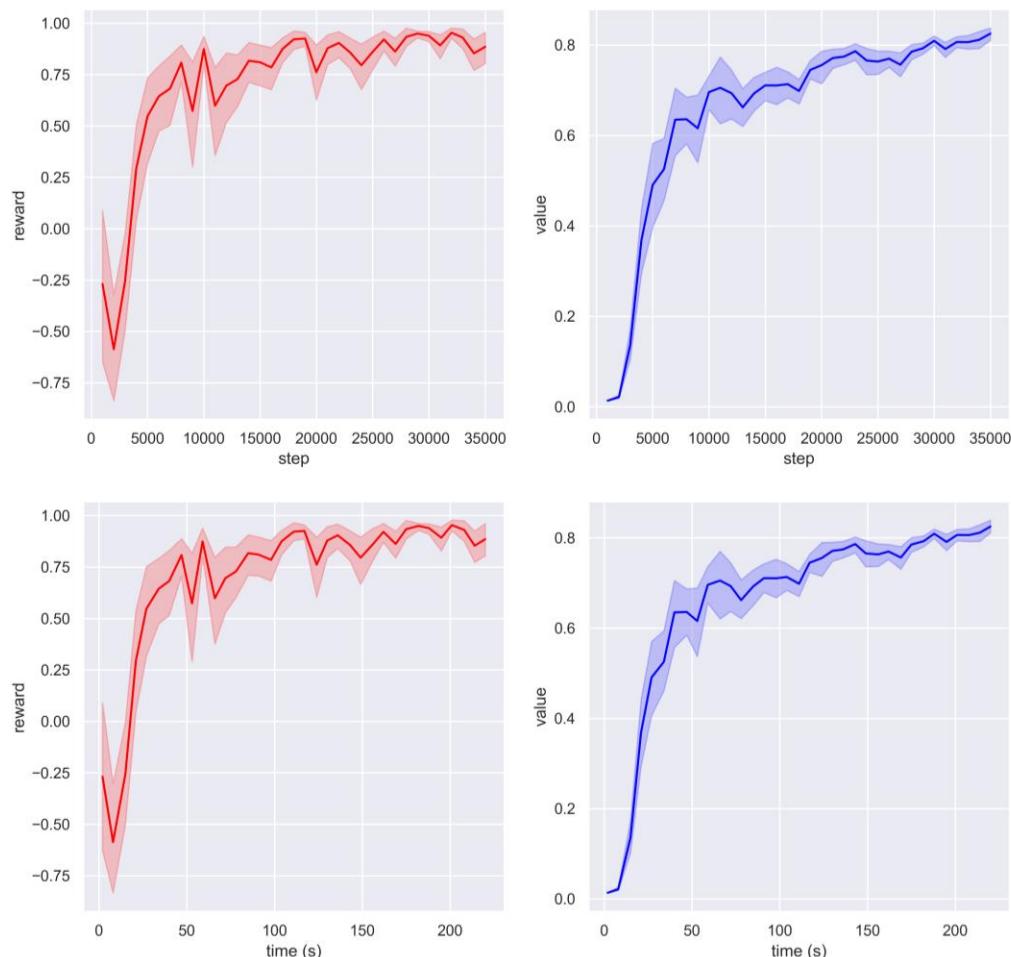
En cada entorno se ha entrenado al agente, con los diferentes algoritmos de aprendizaje por refuerzo que se han explicado en este trabajo, excepto del entorno 4 que, al haber 2 agentes, no se ha aplicado el algoritmo de DQN, porque se ha implementado con un solo agente. Se podría hacer en multiagente, pero como el objetivo de este trabajo es el algoritmo de PPO de la familia de Policy Gradients, no se le ha dedicado más tiempo a DQN. Y en el entorno 5, debido a su dificultad solo se aplica PPO.

Los resultados se presentan con cuatro gráficas; recompensa-step, recompensa-tiempo (de color rojo), value-step y value-tiempo (de color azul) con las implementaciones propias del trabajo.

Seguidamente se exponen los resultados obtenidos en los diferentes algoritmos.

Entorno 1

DQN

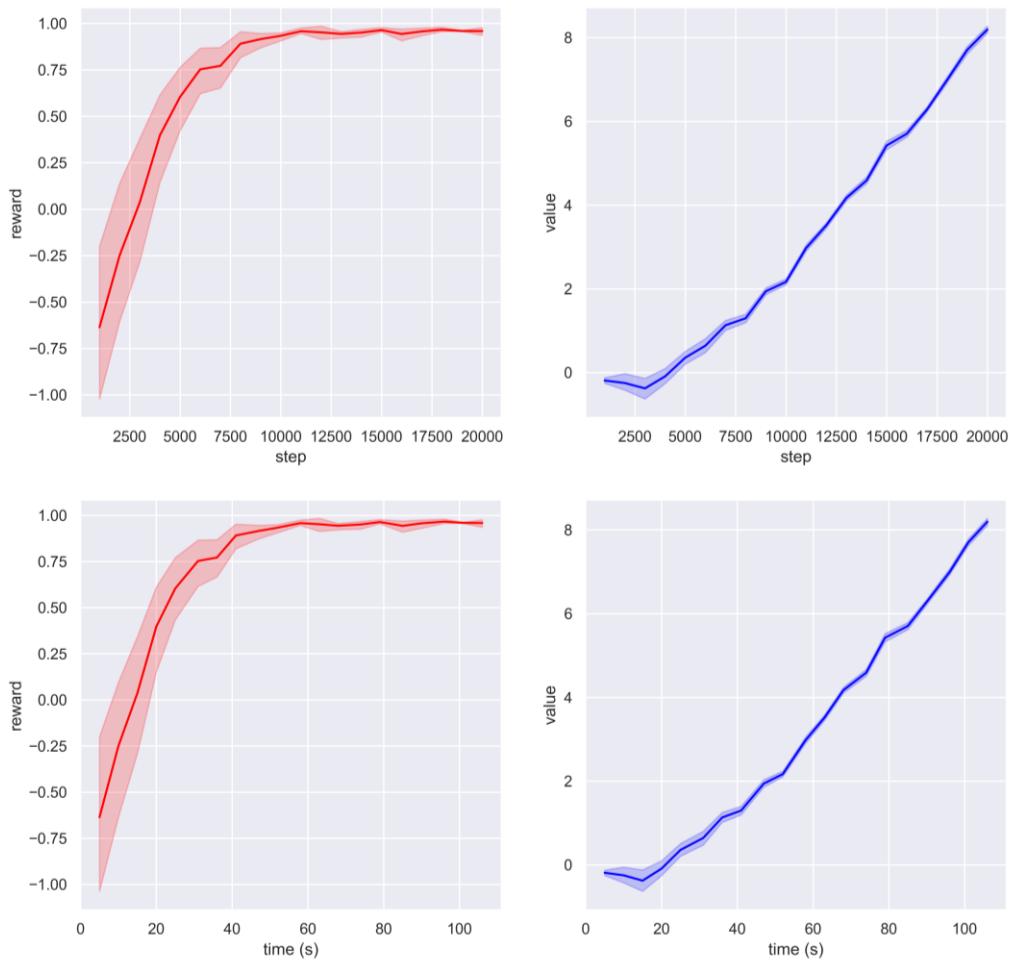


Gráfica 1 Entorno 1, DQN

Con el algoritmo de DQN, el agente aprende entre el step 5000 y 10000, que en tiempo viene a ser alrededor de los 50 segundos, después continúa aprendiendo, intentando mejorar la recompensa, aunque con una cierta inestabilidad.

En el caso de las gráficas del value, que viene a ser el valor que evalúa lo bueno que es para el agente estar en un estado, también va aumentando, y esto quiere decir que, el agente a medida que se va entrenando, sabe con más certeza qué acción tomar en cada estado.

Actor-Critic

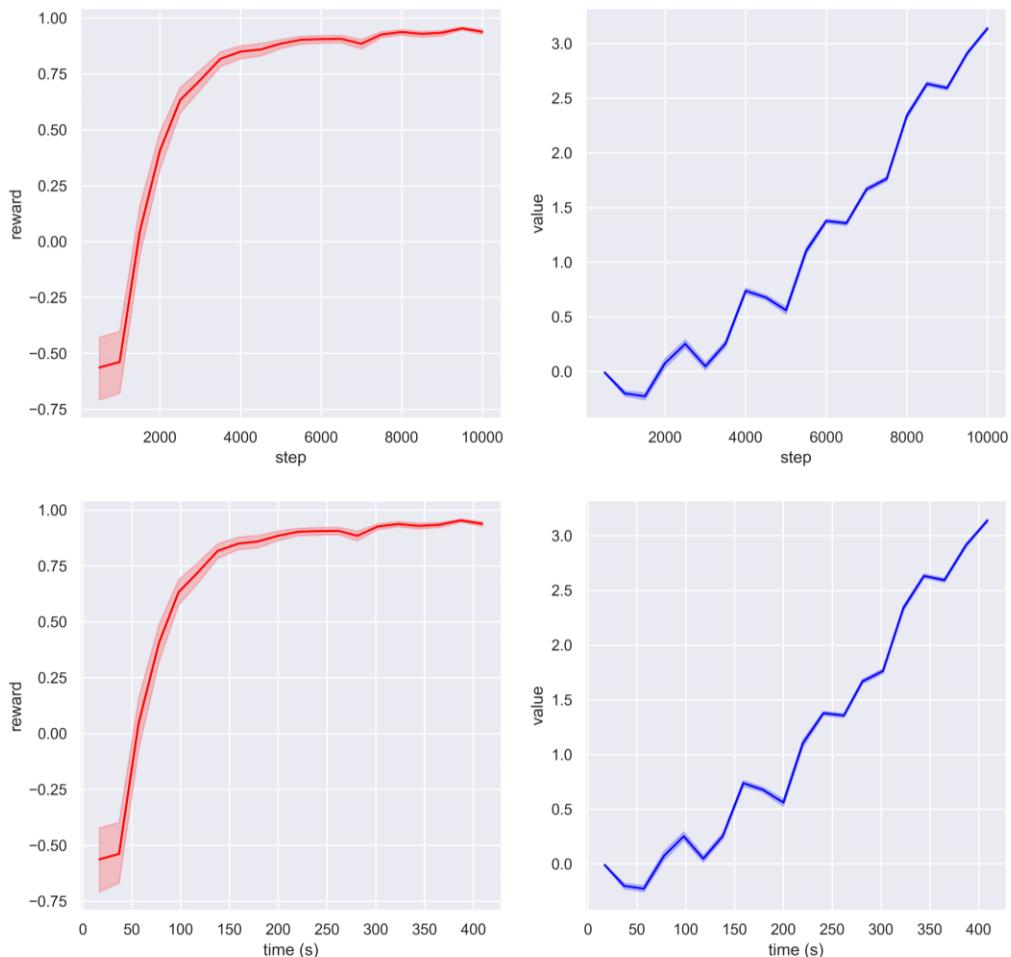


Gráfica 2 Entorno 1, Actor-Critic

Con Actor-Critic, el agente aprende cuando alcanza los 7500 steps con un tiempo de 40 segundos, y se estabiliza obteniendo la máxima recompensa.

Respecto a las gráficas del value, su valor va creciendo de manera lineal, a medida que va transcurriendo el entrenamiento.

A3C

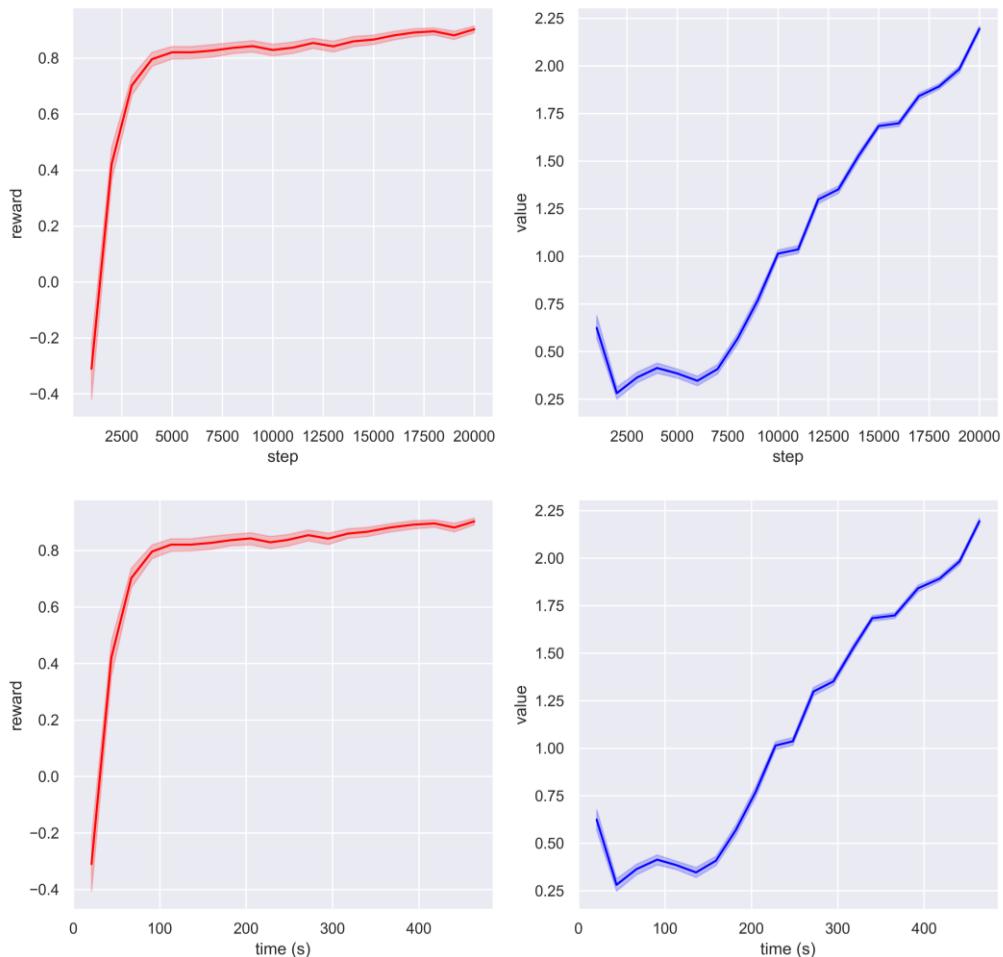


Gráfica 3 Entorno 1, A3C

En A3C, las gráficas de recompensa, se puede observar que son muy similares a las de Actor-Critic, aunque los agentes tardan ligeramente más en aprender. En este caso, lo hace en el step 4000 con un tiempo de 150 segundos.

Esto es debido a que la actualización del modelo a través de los distintos agentes, se hace de manera asíncrona, esto se puede apreciar en las gráficas del value. El valor va aumentando con tramos en los que también desciende, a causa de agentes que han estado en malos estados. Y esto conlleva a que el aprendizaje en este entorno, sea más lento.

A2C

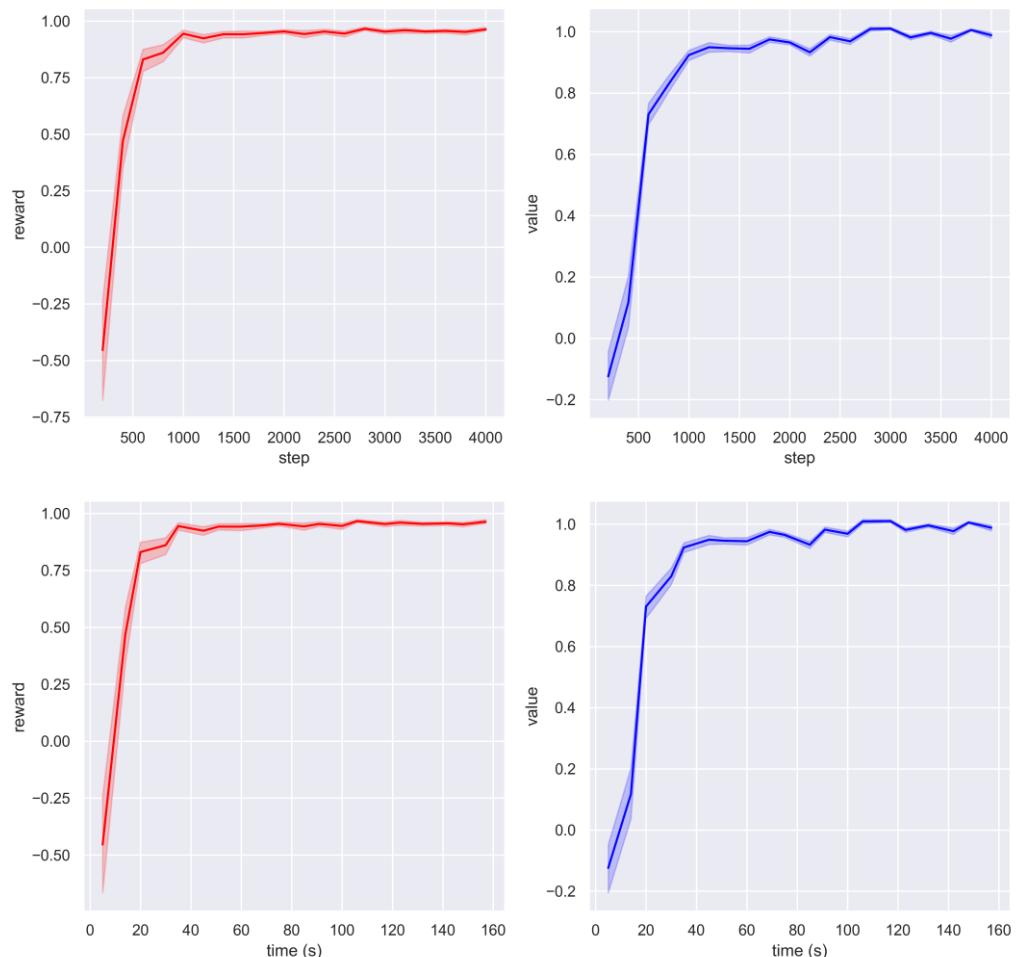


Gráfica 4 Entorno 1, A2C

Con A2C, el agente consigue aprender en torno a los 3000 steps, que vendrían a ser 100 segundos. Y después, sigue aprendiendo para obtener mejores recompensas de manera estable.

Respecto a las gráficas del value se puede observar, que al principio tiene un tramo con una tendencia a decrecer con valores pequeños, y en el momento en que el agente aprende, el valor del value empieza a aumentar de forma progresiva.

PPO



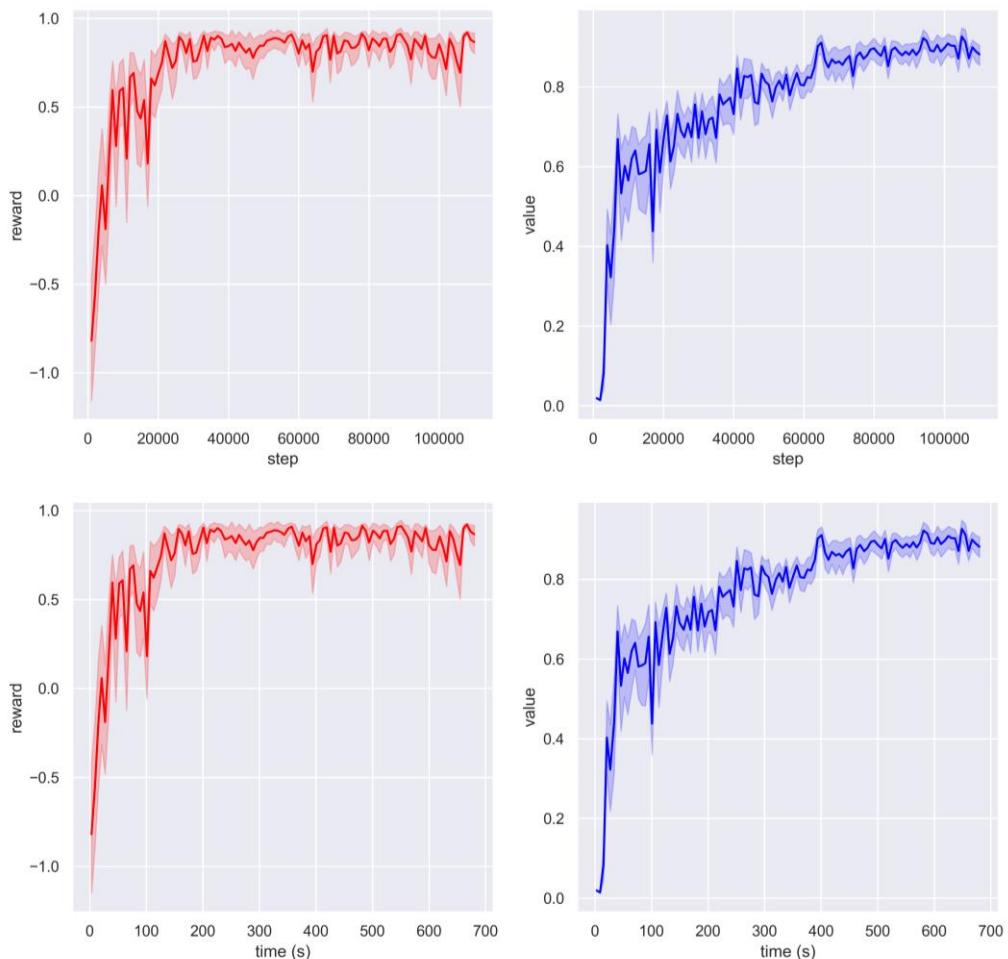
Gráfica 5 Entorno 1, PPO

Con el algoritmo de PPO, el agente aprende entre los 500 y 1000 steps, con un tiempo alrededor de 30 segundos, estabilizándose luego en la máxima recompensa.

En las gráficas del value, se aprecia que son distintas respecto a los anteriores algoritmos de Policy Gradients, y se termina estabilizando en el valor alrededor de 1.

Entorno 2

DQN

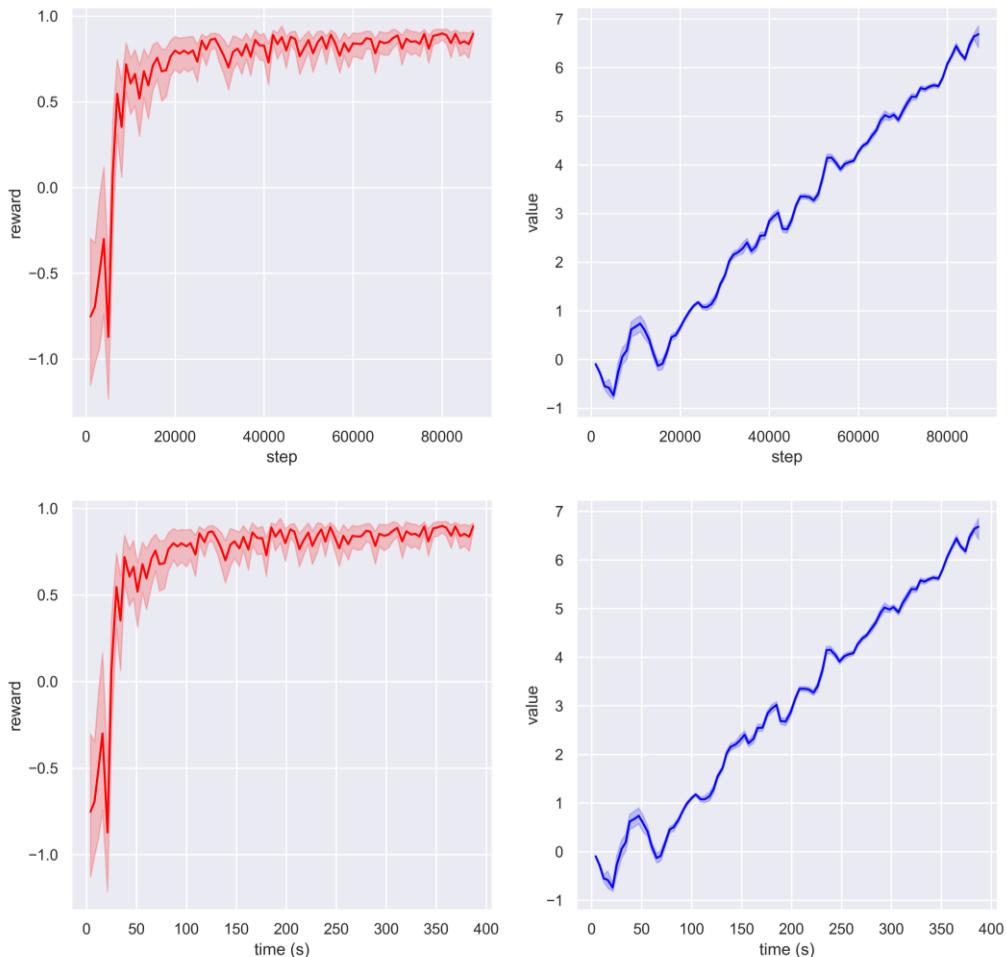


Gráfica 6 Entorno 2, DQN

El entorno 2 es un poco más complicado respecto al entorno 1, y con DQN el agente aprende entorno a los 30000 steps, que en tiempo vendría a ser alrededor de 200 segundos. Al igual que en el entorno 1, continúa aprendiendo, intentando mejorar la recompensa, aunque con una cierta inestabilidad.

Las gráficas del value, el valor va aumentando también con una cierta inestabilidad.

Actor-Critic

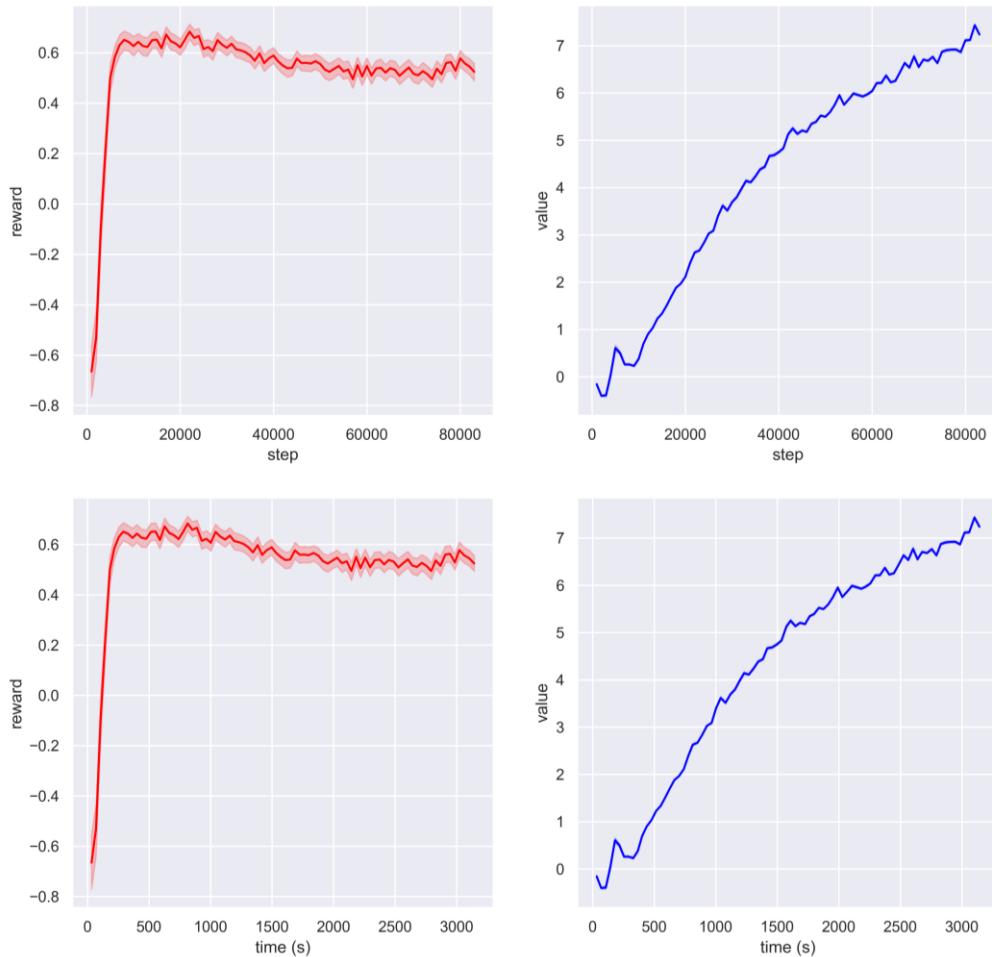


Gráfica 7 Entorno 2, Actor-Critic

Con el algoritmo de Actor-Critic, el agente aprende con 20000 steps con un tiempo alrededor de 100 segundos. Al ser un entorno más complicado que el primero, después el agente continúa aprendiendo para mejorar la recompensa, pero con una cierta inestabilidad.

Esta inestabilidad también se puede ver reflejada en las gráficas del value, mientras el valor va aumentando.

A3C

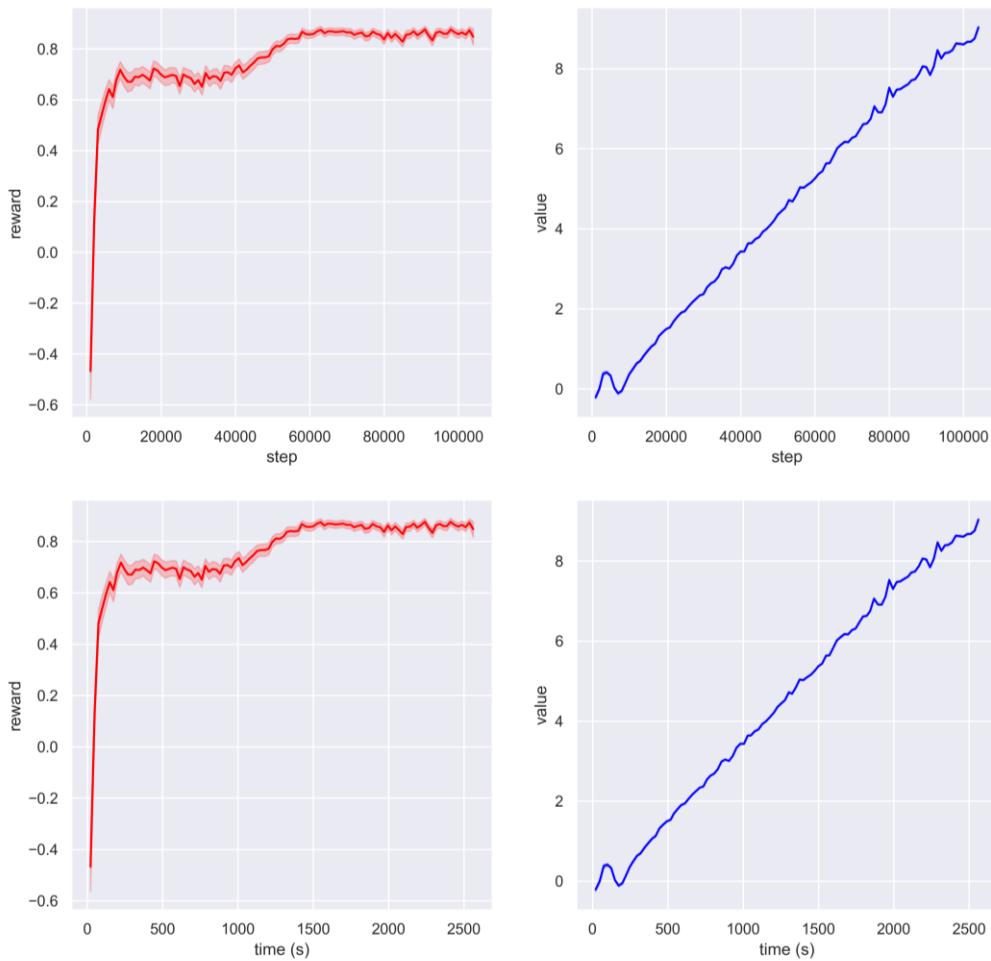


Gráfica 8 Entorno 2, A3C

Con A3C, el agente empieza a aprender a los 10000 steps con un tiempo alrededor de 250 segundos, aunque la recompensa no es tan buena como en Actor-Critic y también el aprendizaje es más lento.

En las gráficas del value, el valor va aumentando de manera progresiva.

A2C

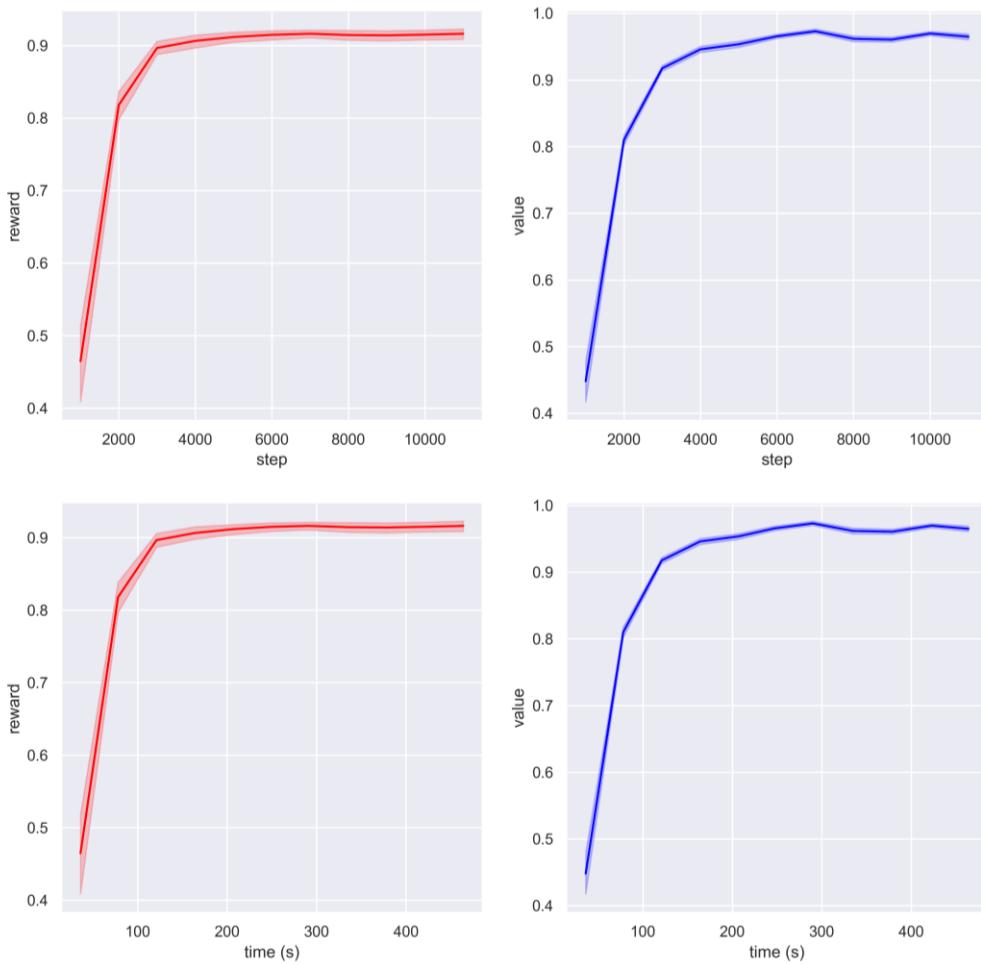


Gráfica 9 Entorno 2, A2C

Con el algoritmo de A2C, el agente aprende entorno a los 10000 steps con un tiempo de 250 segundos, y continúa aprendiendo para obtener mejores recompensas.

En las gráficas del value se puede apreciar como el valor va aumentando progresivamente.

PPO



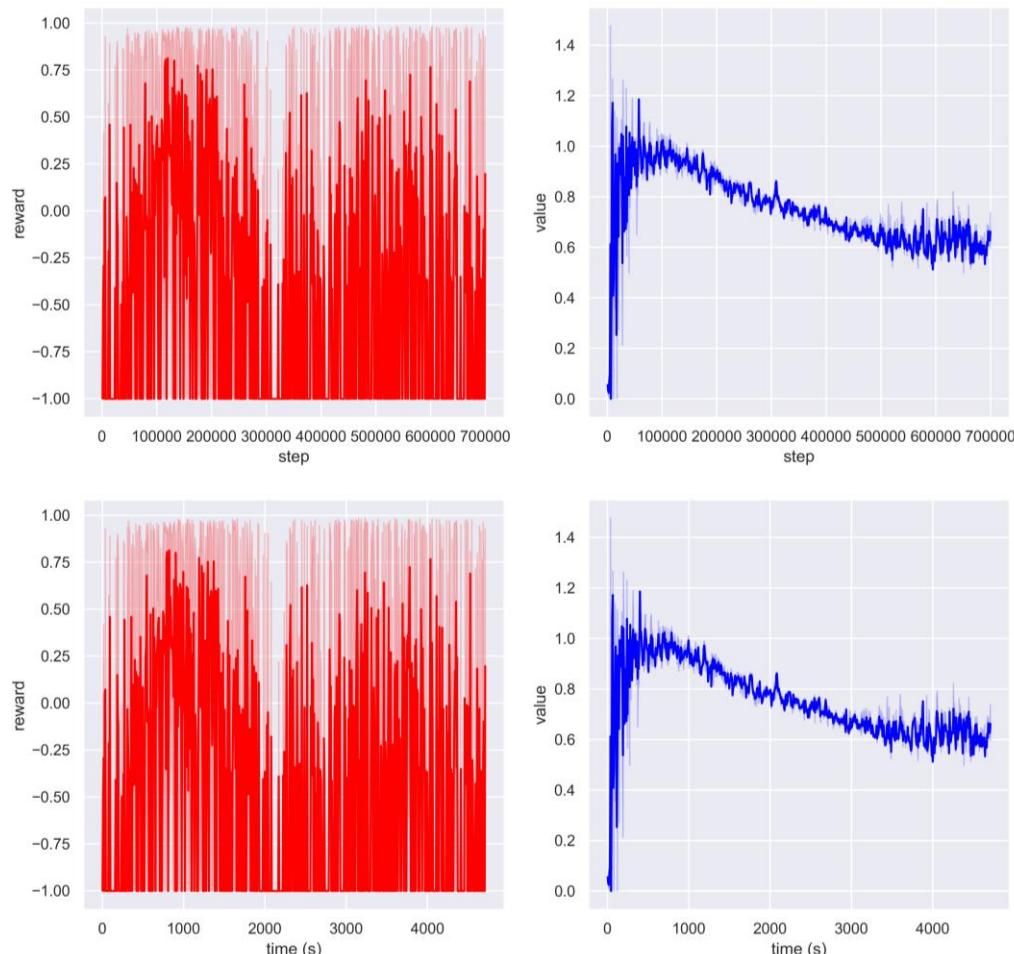
Gráfica 10 Entorno 2, PPO

Con PPO, el agente aprende entorno a los 30000 steps, que en tiempo son alrededor de 100 segundos, y después se estabiliza con la recompensa máxima.

En las gráficas del value, se puede observar también, como el valor va aumentando hasta estabilizarse en un valor cercano a 1.

Entorno 3

DQN



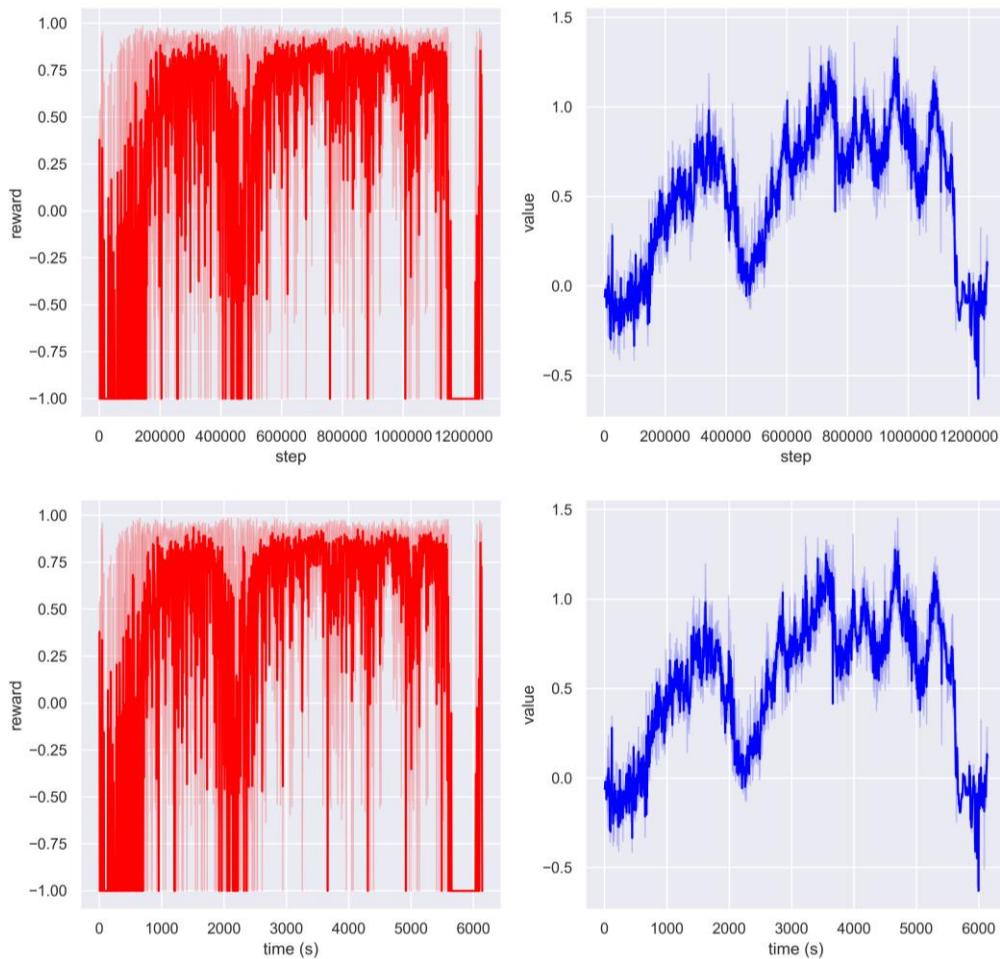
Gráfica 11 Entorno 3, DQN

El entorno 3 es muy estocástico debido a la inmensa cantidad de diferentes estados que se puede llegar a encontrar el agente, porque si el primer lanzamiento del agente no va a dentro de la portería, la pelota rebotará e irá en cualquier posición del campo.

Con un entrenamiento de alrededor de 700000 steps, que en este caso, son aproximadamente 5000 segundos, el agente no consigue aprender. Se puede ver reflejado en las gráficas del value, que al principio parece que está aprendiendo, aumentando el valor del value hasta llegar al alrededor de 1 y luego poco a poco va decreciendo.

La gran variedad en las recompensas es debido, a que a medida que va transcurriendo el entrenamiento, al ser un entorno muy estocástico, el agente cada vez se va encontrando con más estados y cada vez más diferentes uno respecto al otro, y no logra modelar un comportamiento para el agente dónde sea capaz de resolver la tarea en cualquier situación.

Actor-Critic



Gráfica 12 Entorno 3, Actor-Critic

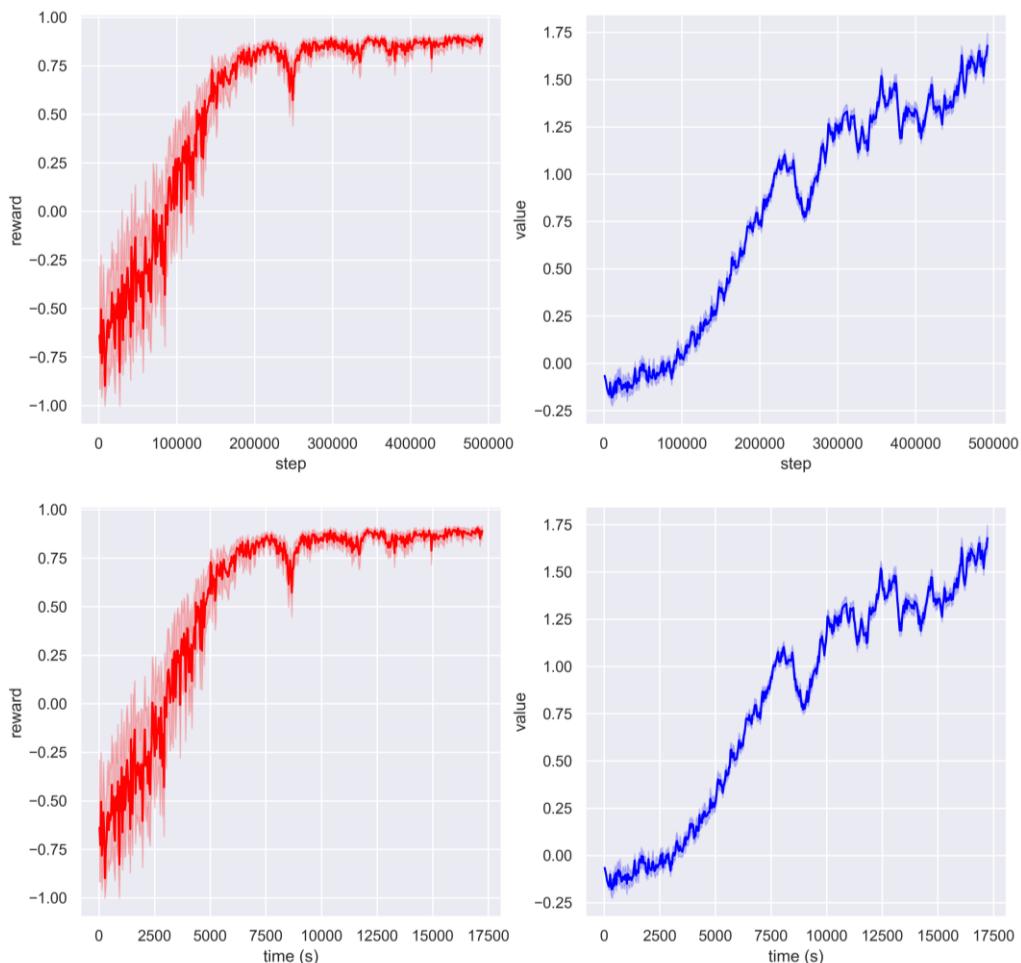
Con el algoritmo de Actor-Critic, el agente llega a aprender más o menos en el step 200000, con un tiempo alrededor de 1000 segundos, pero con mucha inestabilidad. Después intenta mejorar la recompensa, con tramos dónde llega incluso a desaprender, y como se puede apreciar, hay mucha variación en las recompensas.

En las gráficas del value, se puede apreciar justo este comportamiento, porque el valor no consigue aumentar de manera constante, sino que va teniendo tramos dónde aumenta, luego decrece y así sucesivamente.

Esto es debido a que el entorno es muy estocástico, y al haber solo un agente, no puede explorar tantos estados del entorno en comparación a los algoritmos

que utilizan más de un agente. En este caso, que solo hay uno, este va aprendiendo con los estados que se va encontrando, pero cuando aparecen nuevos, el agente no sabe qué hacer y son los tramos que se pueden observar en las gráficas del value, dónde se decrece.

A3C



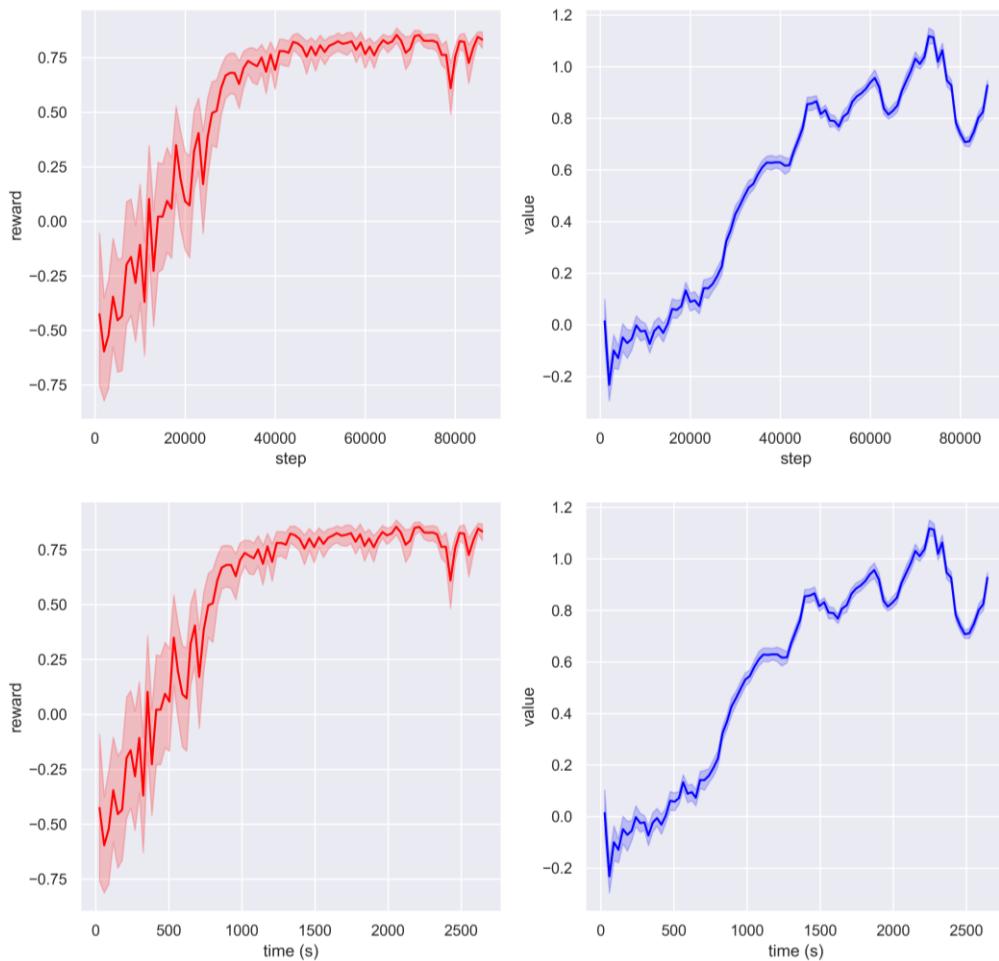
Gráfica 13 Entorno 3, A3C

Con A3C, el agente aprende entorno a los 200000 steps con un tiempo de 7500 segundos. Y luego intenta obtener mejores recompensas.

En las gráficas del value, se puede apreciar que el valor va aumentando progresivamente, con tramos en los que decrece, debido a agentes que han estado en malos estados.

En este caso, al ser un algoritmo con más de un agente, se pueden explorar más estados, y de esta manera obtener mejores recompensas dentro de una cierta estabilidad.

A2C

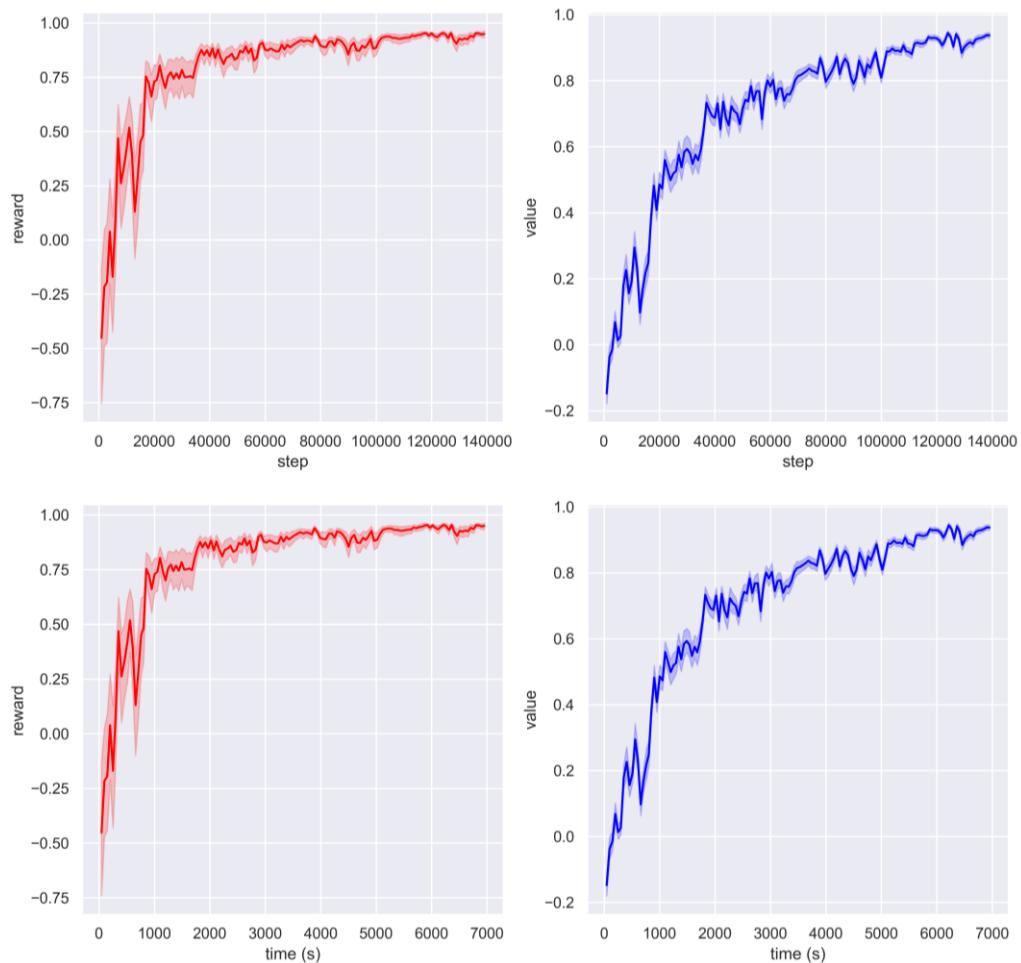


Gráfica 14 Entorno 3, A2C

En A2C, el agente aprende a los 30000 steps con un tiempo alrededor de los 1200 segundos y continúa aprendiendo para obtener mejores recompensas con una cierta inestabilidad.

Esta inestabilidad se puede apreciar en las gráficas del value, como el valor va aumentando de forma progresiva pero con tramos en los que decrece.

PPO



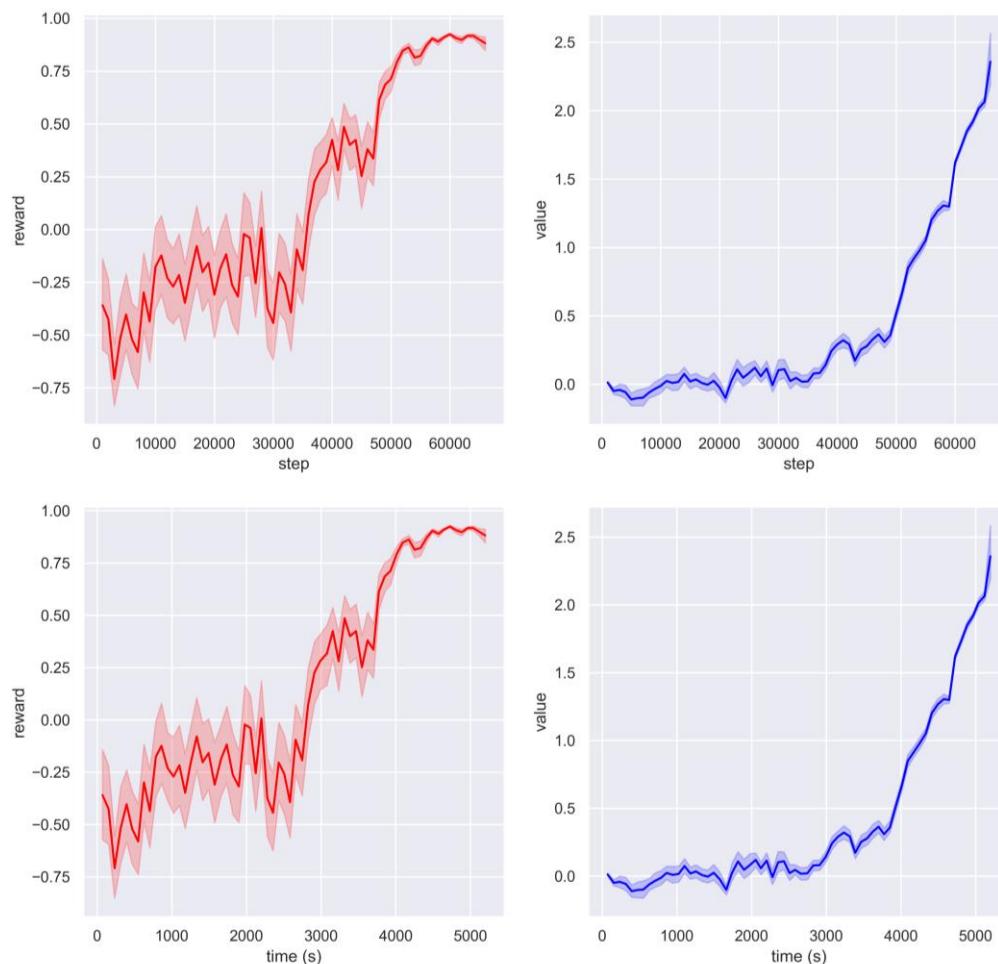
Gráfica 15 Entorno 3, PPO

Con el algoritmo de PPO, el agente aprende con 20000 steps que viene a ser alrededor de 1000 segundos, y después sigue mejorando las recompensas que va obteniendo.

En las gráficas del value, se puede observar que el valor va aumentando hasta estabilizarse alrededor de 1, aunque con una mayor inestabilidad en comparación a los anteriores entornos, debido a aleatoriedad del entorno.

Entorno 4

A3C

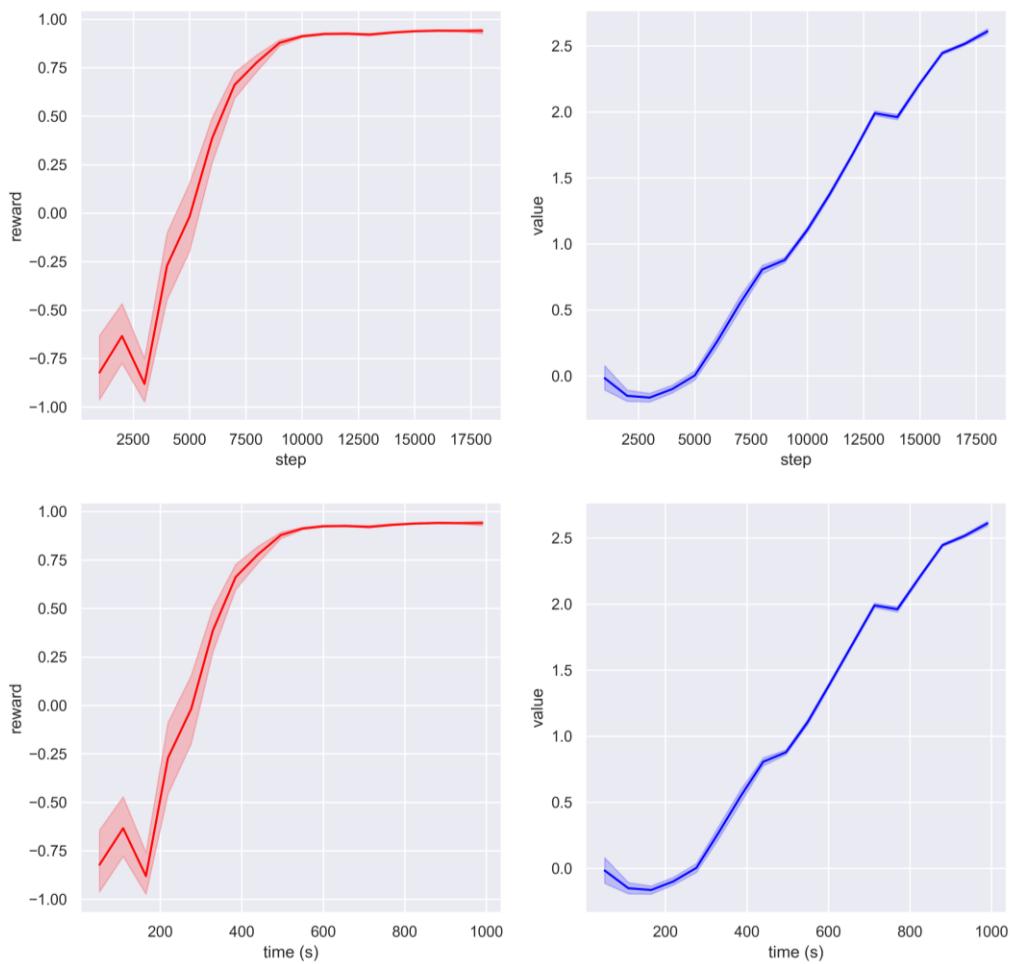


Gráfica 16 Entorno 4, A3C

Con el algoritmo de A3C y el entorno 4 con agentes cooperativos, los agentes consiguen aprender alrededor del step 50000, que vendrían a ser 4000 segundos. Durante el aprendizaje con bastante inestabilidad, al final consigue obtener la recompensa óptima.

Esto se puede apreciar en las gráficas del value, cómo no es hasta pasado buena parte del entrenamiento, que el valor no empieza aumentar de manera más acentuada.

A2C

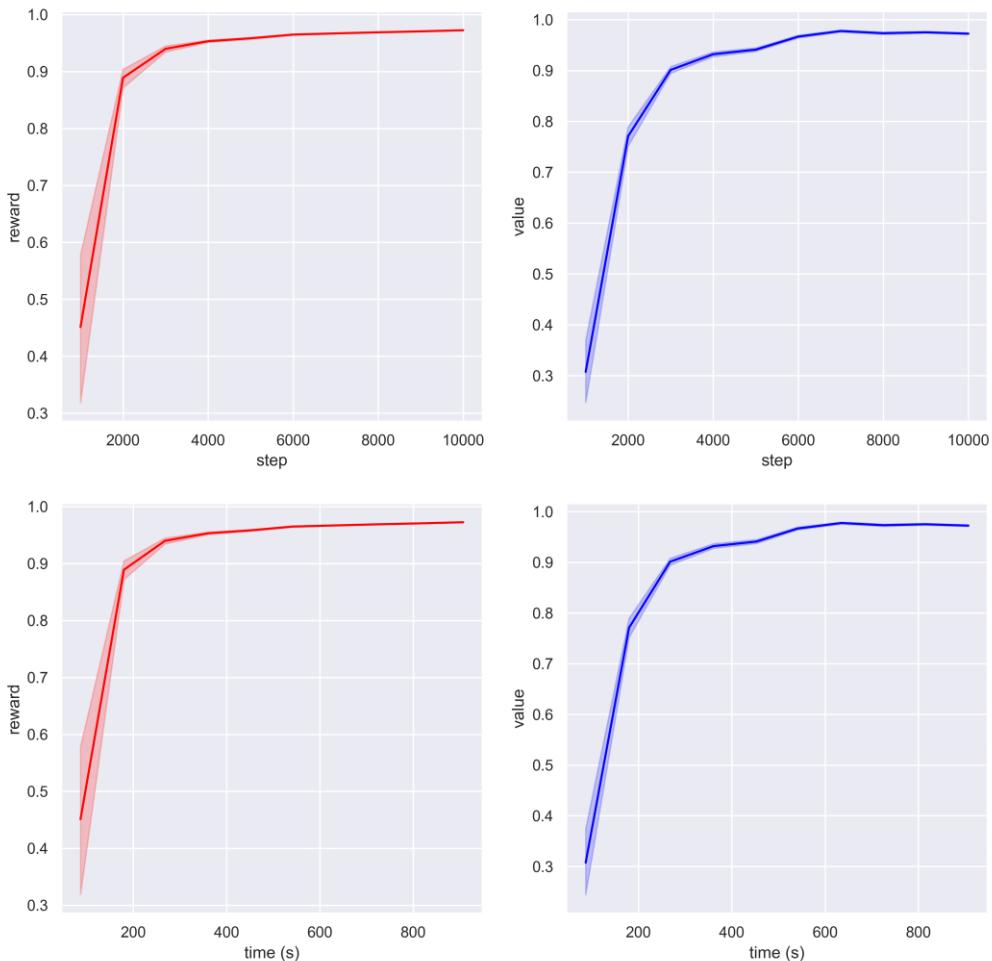


Gráfica 17 Entorno 4, A2C

En el caso de A2C, el agente aprende entorno a los 8000 steps con un tiempo de 500 segundos. Después se estabiliza con la máxima recompensa.

Con las gráficas del value, se puede observar que el valor aumenta de manera progresiva.

PPO



Gráfica 18 Entorno 4, PPO

Con PPO, el agente aprende con 2000 steps con un tiempo de 200 segundos y estabilizándose luego con la máxima recompensa.

En el caso de las gráficas del value, el valor aumenta hasta estabilizarse alrededor de 1.

Entorno 5

En este entorno, la tarea del agente es más complicada, porque se tiene que relacionar un mayor número de cosas, en comparación a los entornos anteriores. Al ser un entorno muy estocástico, el agente también tendrá que explorar y aprender una gran variedad de situaciones.

El agente tiene que aprender a relacionar que el bloque naranja tiene que ir a la zona naranja y el bloque rojo a la zona roja. Una vez realizado esto, darse cuenta de que la pared azul desaparece e ir a la zona verde para completar la tarea.

Los bloques y el agente se inicializan en una posición aleatoria. En la siguiente imagen, el rectángulo de color amarillo, representa la zona dónde el agente y los bloques pueden inicializarse.

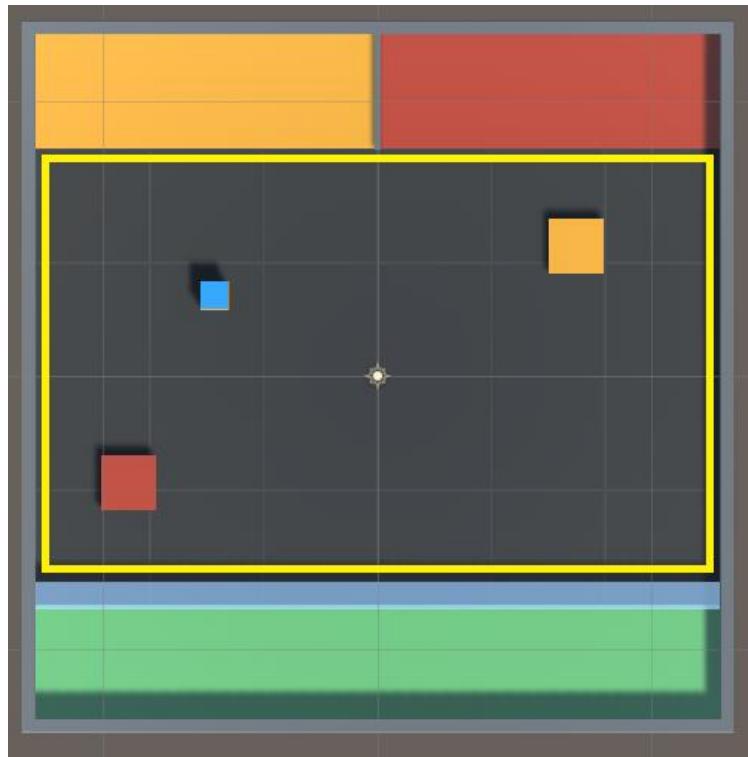
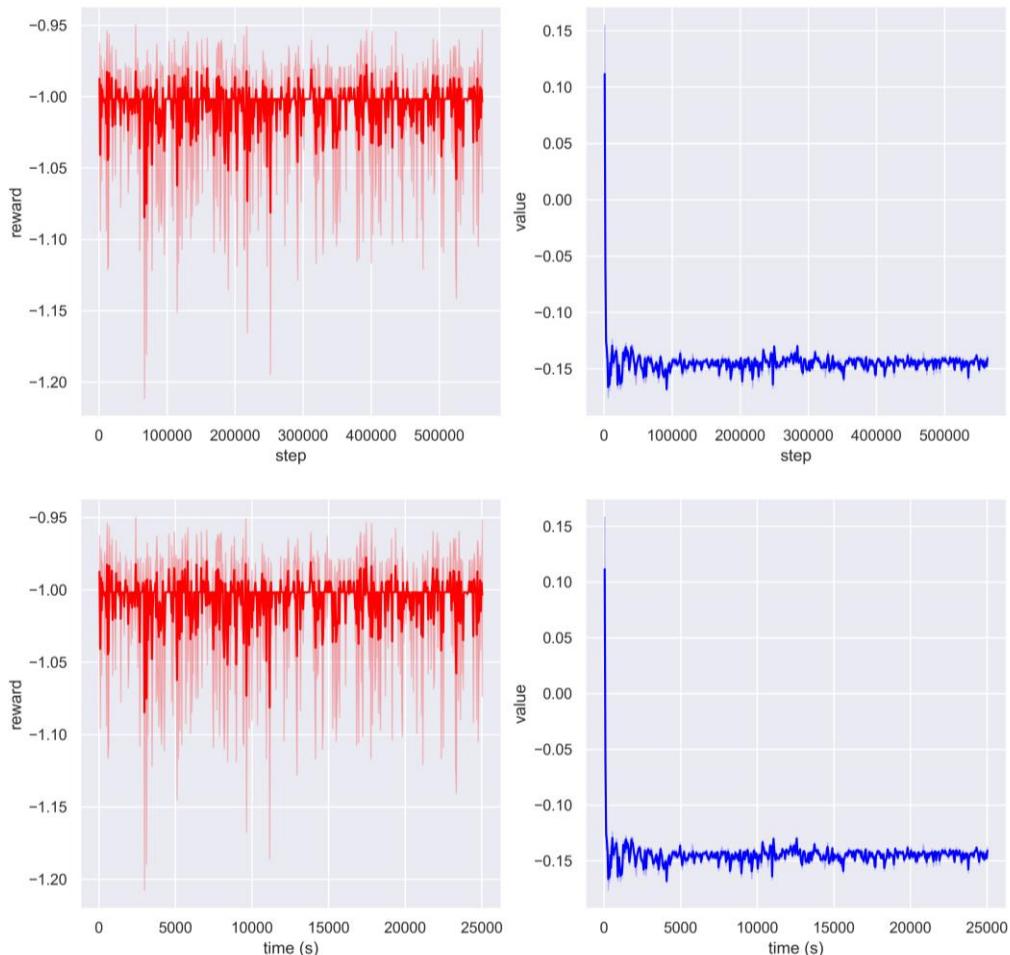


Imagen 26 Entorno 5, posiciones iniciales



Gráfica 19 Entorno 5, PPO

Como se puede observar en las gráficas, el agente es incapaz de aprender. En la gráfica de recompensas con un entrenamiento de más de 500000 steps, que vienen a ser 25000 segundos, la recompensa no incrementa. Y en las gráficas del value, se puede ver también que el valor tampoco aumenta. Esto es debido a que, al ser un entorno muy estocástico y tener que relacionar varias cosas, el agente le es imposible aprender dicha relación en un entorno en que cada inicialización, las posiciones de los objetos del entorno y de él mismo siempre son distintas.

Para solucionar este problema, se divide el entorno en distintos niveles para que el agente, primero aprenda la relación entre los distintos objetos del entorno con los cuales tiene que realizar la tarea y después a medida que se vaya cambiando de nivel, se vaya incrementando la dificultad, hasta llegar a la que se planteado en un inicio. Por lo tanto, el entorno se irá haciendo más estocástico en cada nivel.

Nivel 1

Con el primer nivel, se busca que el agente consiga aprender la relación de los objetos del entorno, que le llevan a completar la tarea. Esto se consigue, inicializando siempre las mismas posiciones de los objetos y del agente. Las posiciones del primer nivel se pueden apreciar en la siguiente imagen:

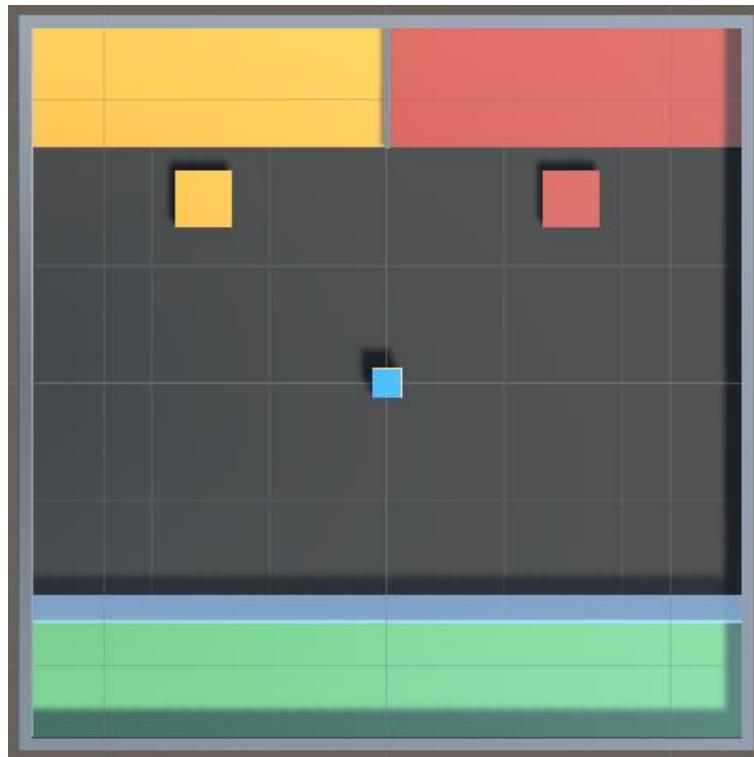
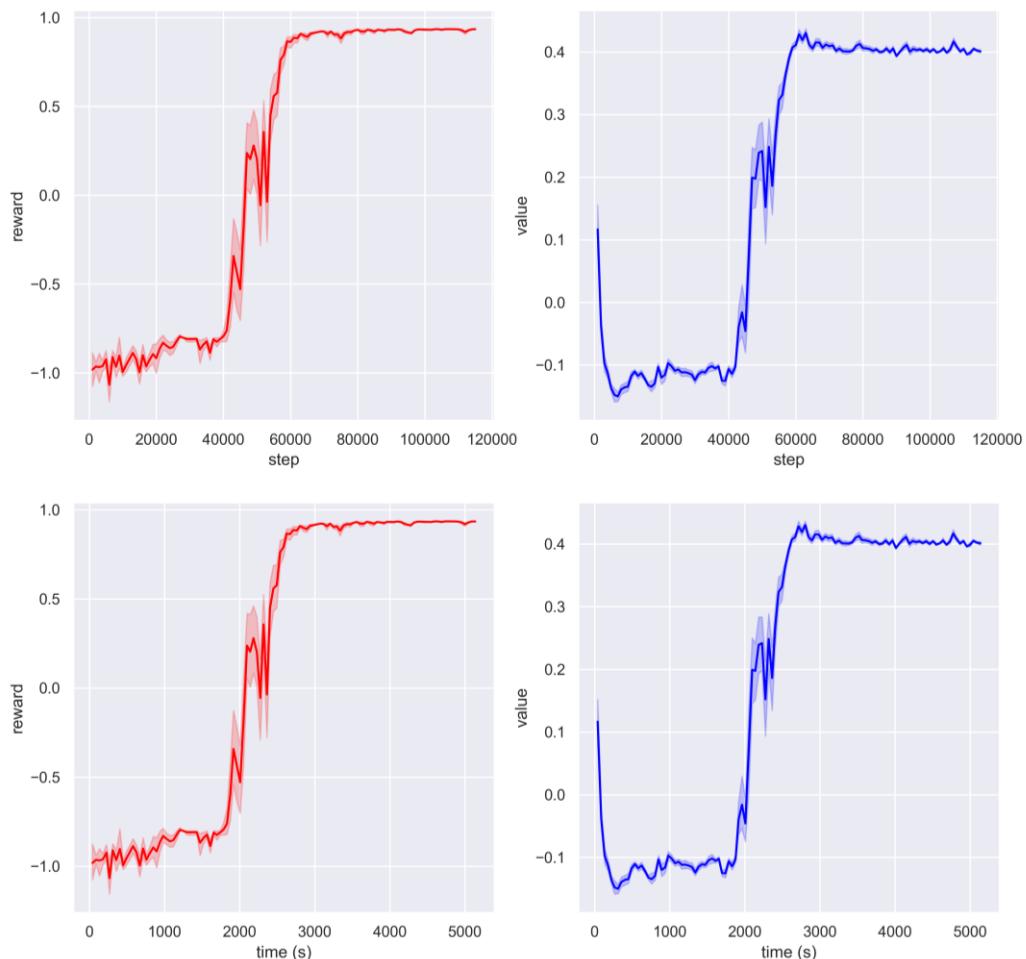


Imagen 27 Entorno 5 - Nivel 1, posiciones iniciales

De esta manera, el agente lo tiene bastante fácil para aprender a hacer la tarea, que es, colocar cada bloque en su respectiva zona y luego entrar en la zona verde.



Gráfica 20 Entorno 5 - Nivel 1, PPO

El agente consigue aprender a los 60000 steps con un tiempo de alrededor de 3000 segundos. Y se estabiliza en conseguir la máxima recompensa.

En las gráficas del value se puede apreciar, una primera parte donde el valor no aumenta debido a que el agente está explorando el entorno e intentando comprender qué es lo que tiene que hacer, y una vez que ha entendido la tarea, el valor del value aumenta hasta estabilizarse en un valor concreto.

Nivel 2

Una vez que el agente ha aprendido la relación entre los objetos del entorno, que le llevan a completar la tarea, se incrementa la dificultad añadiendo aleatoriedad en la inicialización de los bloques naranja y rojo. Las zonas en las que se pueden inicializar cada uno de los bloques, se pueden observar en la siguiente imagen (área del rectángulo de color naranja para el bloque naranja y área del rectángulo de color rojo para el bloque rojo):

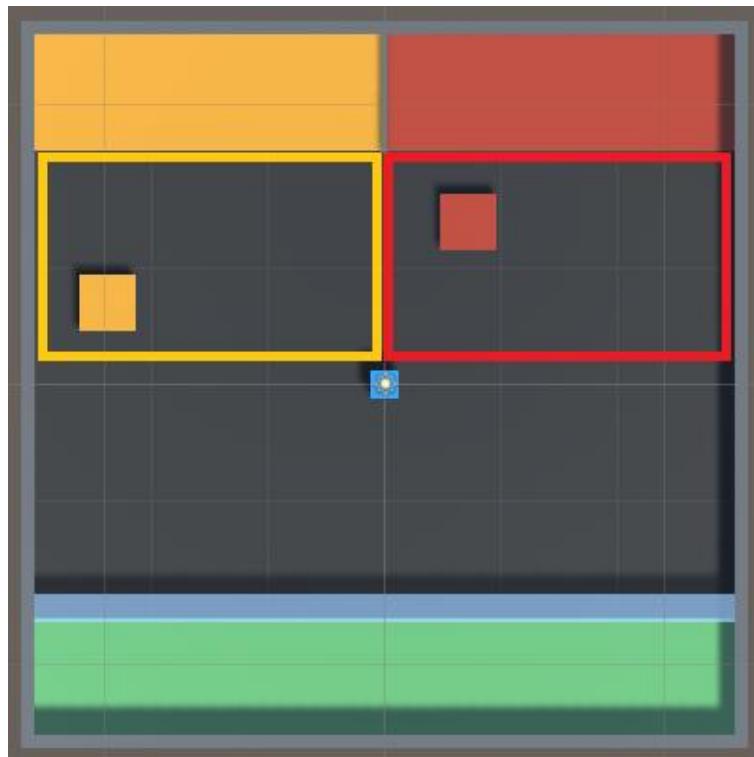
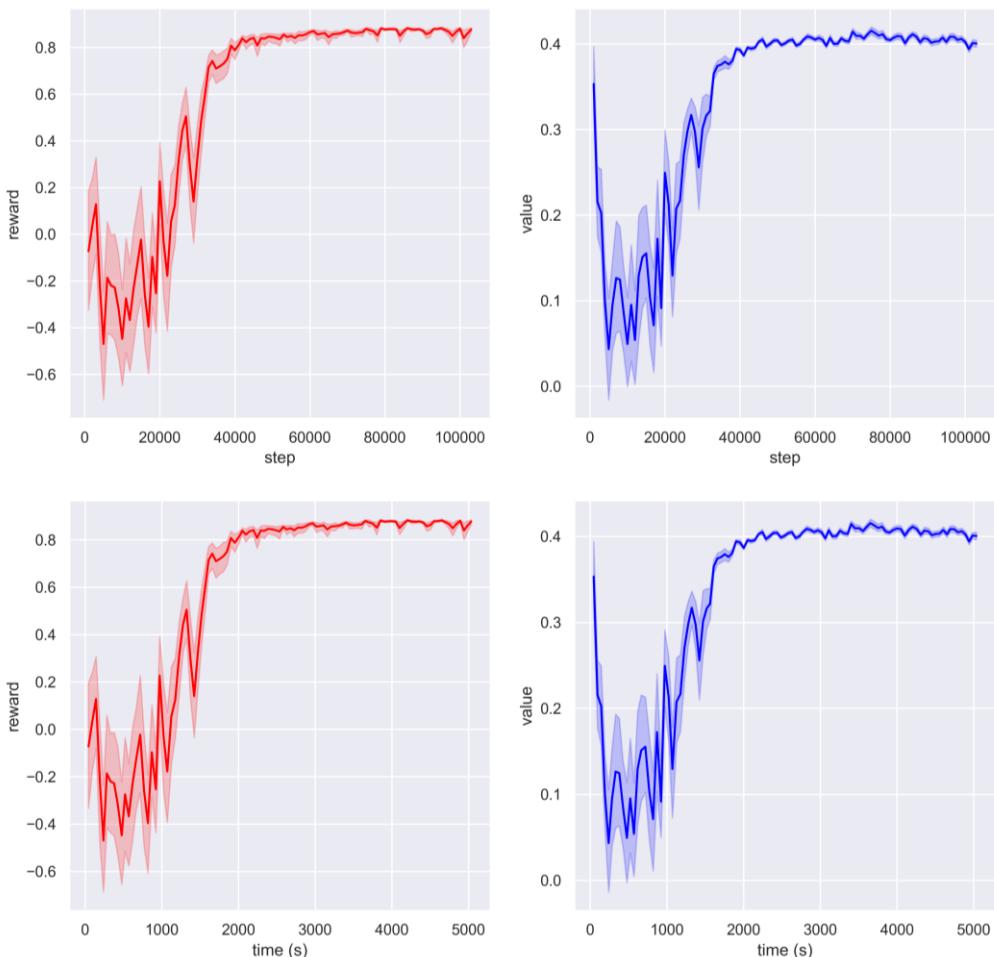


Imagen 28 Entorno 5 - Nivel 2, posiciones iniciales



Gráfica 21 Entorno 5 - Nivel 2, PPO

En este nivel, al principio el agente obtiene recompensas negativas, debido a que tiene que aprender como mover los objetos en distintas posiciones. A medida que va transcurriendo el entrenamiento, va aprendiendo cada vez mejor en como mover los bloques, hasta llegar a aprender a los 40000 steps con un tiempo de 2000 segundos.

En las gráficas del value, se puede observar que al principio el valor es un poco inestable y poco a poco va aumentando hasta estabilizarse en un valor.

Nivel 3

En el nivel 3, la zona en la que los bloques se pueden inicializar es más grande y de esta manera, se incrementa la dificultad del entorno sutilmente. Las zonas en las que se pueden inicializar cada uno de los bloques, se muestran en la siguiente imagen:

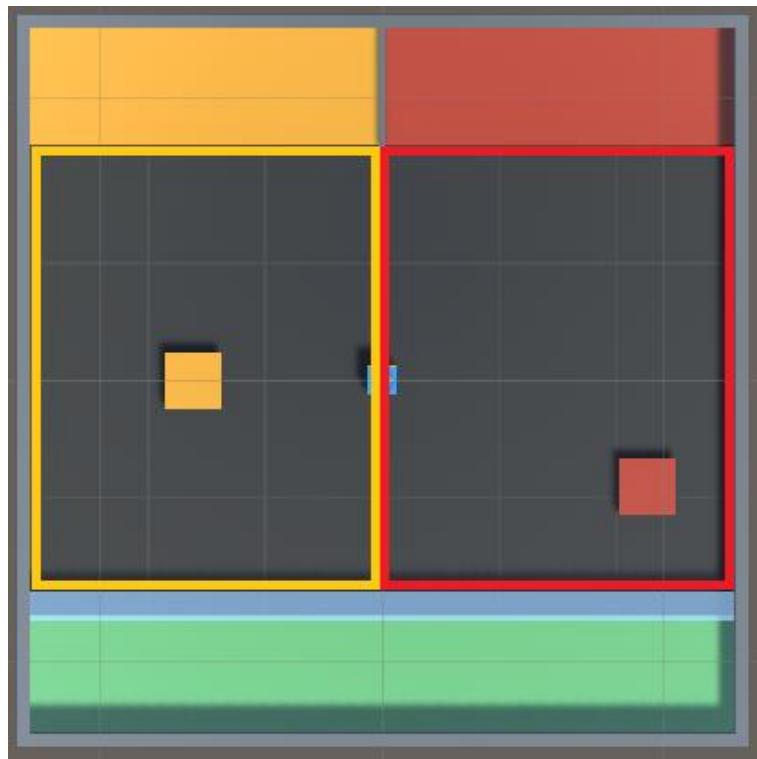
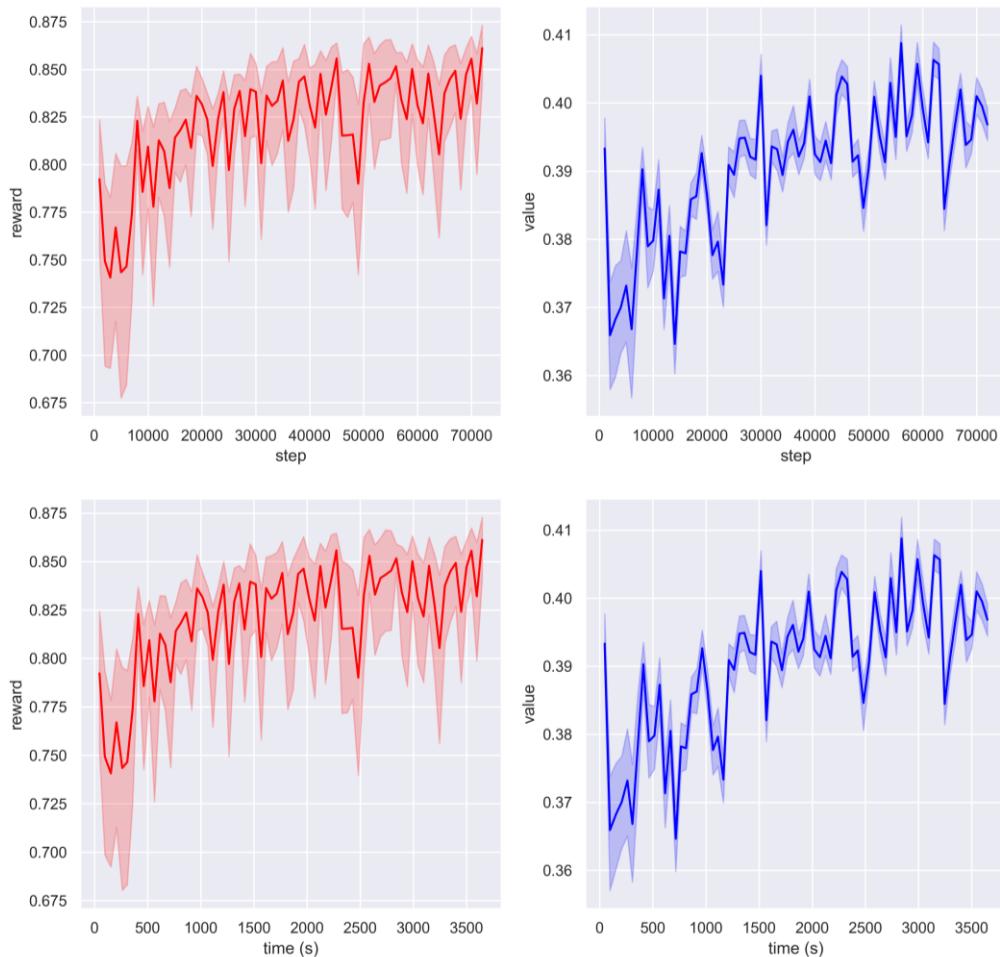


Imagen 29 Entorno 5 - Nivel 3, posiciones iniciales



Gráfica 22 Entorno 5 - Nivel 3, PPO

Al no haber mucha diferencia de dificultad en comparación al anterior nivel, el agente simplemente, aprende a mejorar en mover los bloques de manera más rápida, a la vez en que el rango de posiciones en las que pueden inicializarse, es más grande.

La variedad de las recompensas es debido, a que los bloques al inicializarse en un rango de posiciones más grande, estos pueden estar más alejados entre sí dependiendo de la inicialización, y esto conlleva a que el agente dependiendo de la posición de los bloques, pueda completar la tarea más o menos rápida.

En las gráficas del value, se puede apreciar justo lo que se ha comentado, en que el agente aprende a mejorar, aumentando el valor del value pero con una cierta inestabilidad debida, a que el tiempo en que tarda el agente en realizar la tarea, depende directamente de las posiciones iniciales de los bloques.

Nivel 4

Hasta ahora el agente siempre se inicializaba en una posición fija en el medio. La dificultad que se añade en este nivel respecto al anterior, es que el agente se inicialice en una posición aleatoria en el área marcada con el rectángulo azul, tal y como se puede observar en la siguiente imagen:

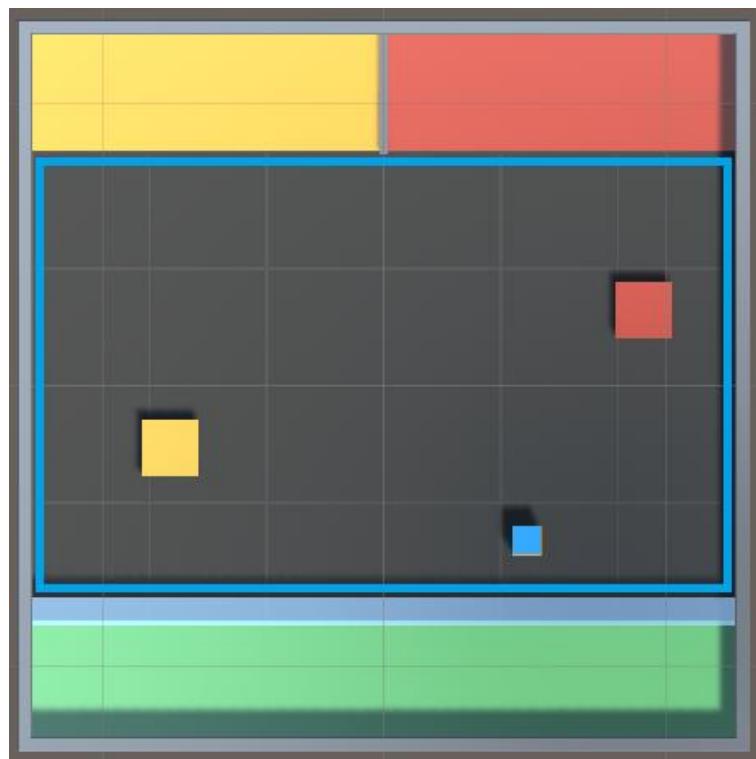
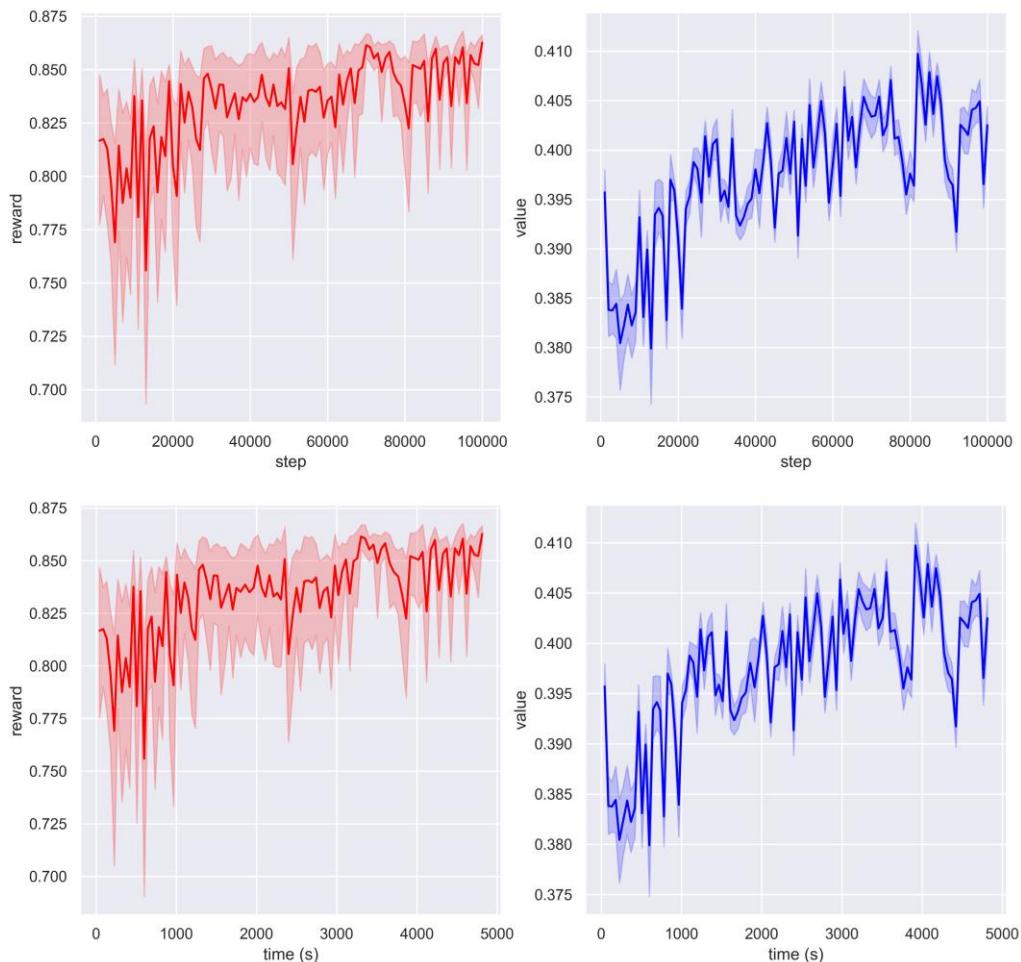


Imagen 30 Entorno 5 - Nivel 4, posiciones iniciales



Gráfica 23 Entorno 5 - Nivel 4, PPO

En el nivel 4, las recompensas no varian mucho respecto al anterior nivel y esto es positivo porque se le ha añadido una dificultad extra, con la inicialización de la posición del agente aleatoria, y sigue obteniendo las mismas recompensas.

Sigue habiendo variedad en las recompensas, debido a lo que se ha comentado antes, que el tiempo en que el agente tarda en completar la tarea depende de las posiciones iniciales de los objetos.

El value al igual que el anterior nivel, aumenta ligeramente con una cierta inestabilidad.

Nivel 5

En los anteriores niveles, cada bloque siempre se inicializa en un área justo delante de la zona, a dónde se tiene que colocar, y tiene cierta facilidad, porque en la mayoría de casos, el agente sólo tiene que empujar el bloque hacia adelante. En este nivel, se aumenta ligeramente el rango de inicialización (en ancho) de cada bloque, tal y como se puede apreciar en la siguiente imagen:

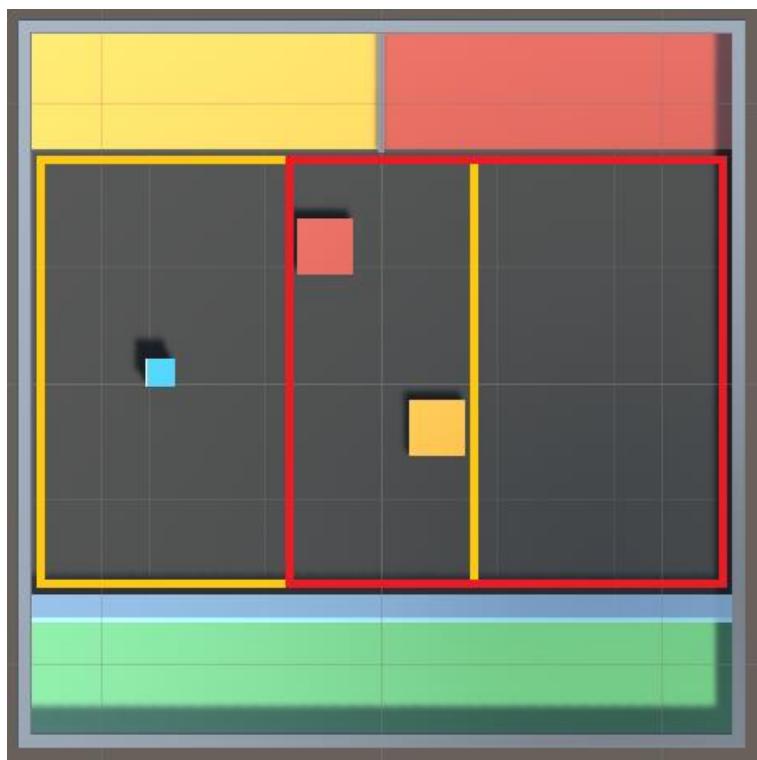
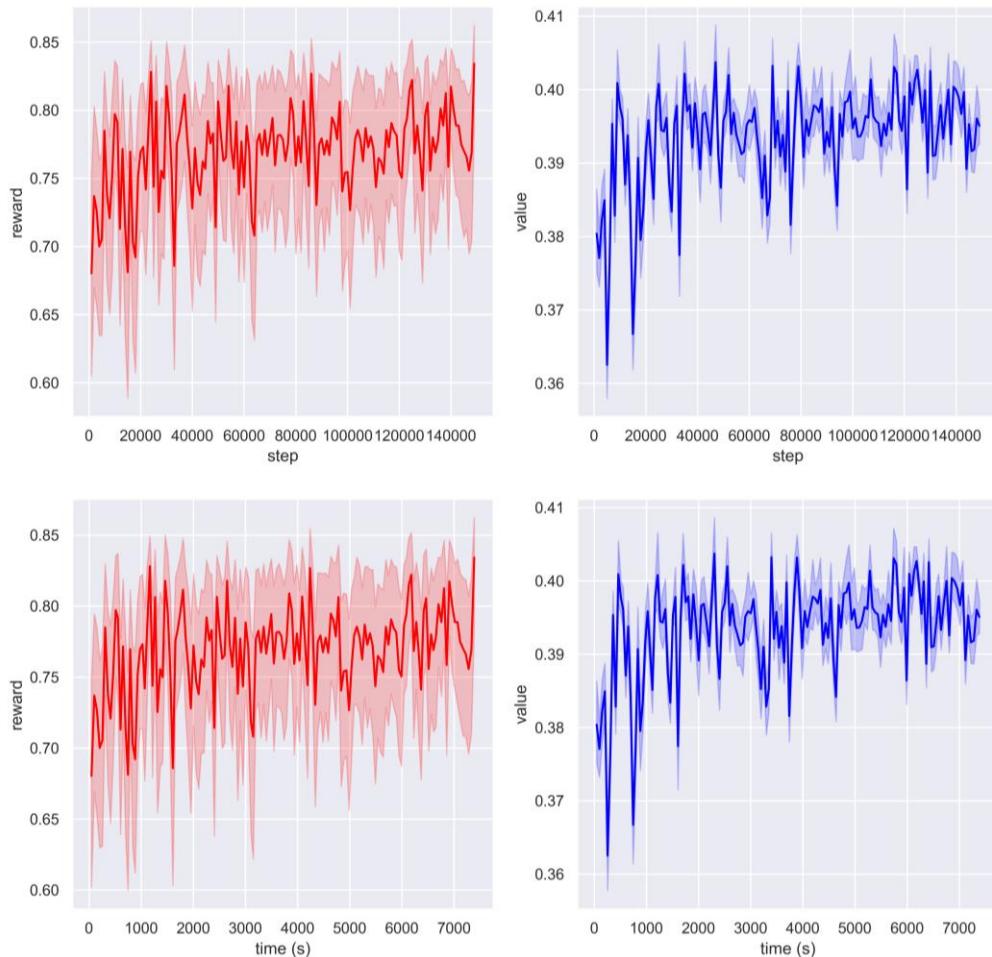


Imagen 31 Entorno 5 - Nivel 5, posiciones iniciales

De esta manera, la dificultad a la hora de mover cada bloque a su respectiva zona, aumenta considerablemente. Y el entorno también se vuelve más estocástico.



Gráfica 24 Entorno 5 - Nivel 5, PPO

En este nivel, se puede apreciar que se siguen manteniendo las mismas recompensas, a pesar de aumentar la dificultad del entorno, y al ser más estocástico, esto provoca que aumente la variedad de las recompensas, debido a lo que se lleva diciendo, acerca de las posiciones iniciales de los objetos.

Y en las gráficas del value se puede ver, que aumenta relativamente poco y se estabiliza dentro de una cierta inestabilidad, provocada por la aleatoriedad en la inicialización del entorno.

Nivel 6

En el último nivel, se aumenta el rango de inicialización, al máximo posible, de los bloques. El área de las posiciones iniciales de los bloques, se puede observar en la siguiente imagen (rectángulo amarillo):

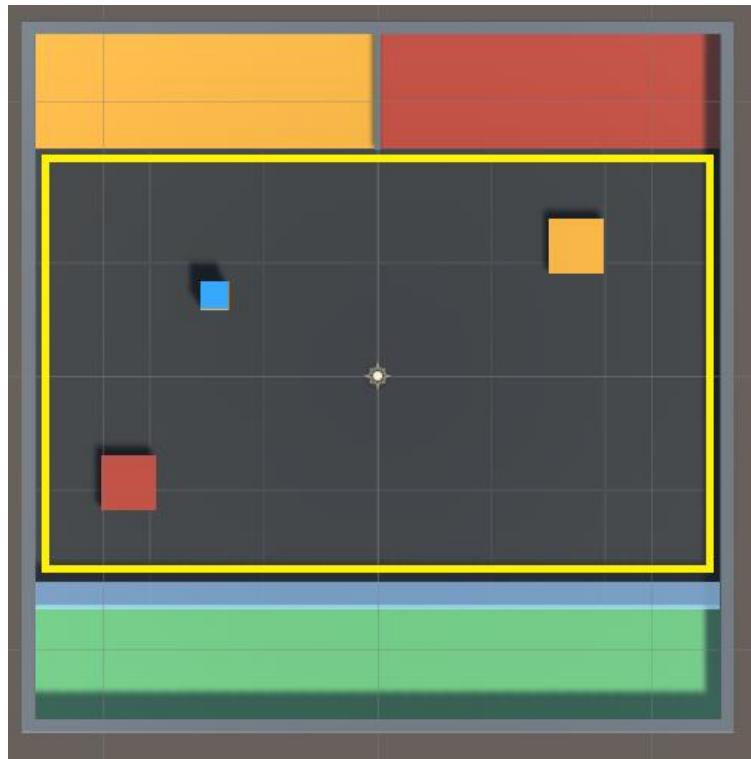
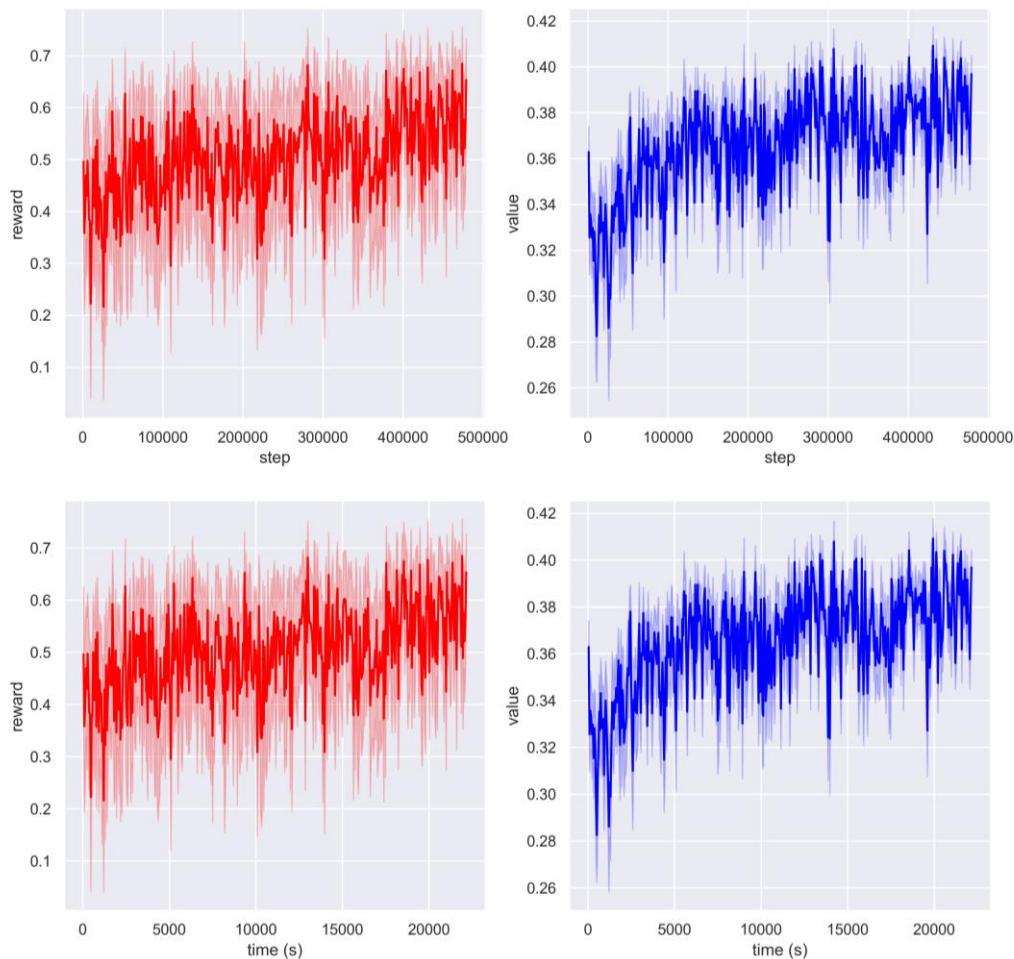


Imagen 32 Entorno 5 - Nivel 6, posiciones iniciales

De modo, que se llega al planteamiento inicial que se había hecho para este entorno.



Gráfica 25 Entorno 5 - Nivel 6, PPO

Como se puede observar, los valores máximos de las recompensas han bajado ligeramente, esto es porque el entorno es muy estocástico y la dificultad en cada episodio puede ser muy diferente, debido a que dependiendo de la posición inicial de los objetos, puede ser más fácil o más difícil moverlos a sus respectivas zonas. También se aprecia, que las recompensas tienen una tendencia en ir aumentando su valor poco a poco, de manera en que el agente sigue aprendiendo en mejorar en como mover los bloques, para completar la tarea más rápido y así mejorar las recompensas.

En las gráficas del value, al principio el valor aumenta de manera progresiva muy sútilmente, y luego se estabiliza en un valor dentro de una cierta inestabilidad.

En el anexo Código y Vídeos, se pueden encontrar los enlaces que dan acceso, a todo el código que se ha hecho para implementar los algoritmos de aprendizaje por refuerzo, que se han tratado en este trabajo, los códigos de los entornos que se han creado y los vídeos de agentes entrenados, con el algoritmo de PPO en todos los entornos.

8. Conclusiones

Al principio, se han estudiado los principales algoritmos de aprendizaje por refuerzo a nivel teórico, con el fin de poderlos implementar en la práctica. Después con el programa de Unity, se han diseñado los entornos desde cero para poder probar dichos algoritmos.

Con los diferentes experimentos que se han realizado, se ha podido ver que en general todos los algoritmos han conseguido que el agente aprenda, exceptuando en los algoritmos más sencillos como DQN y Actor-Critic, dónde en entornos muy estocásticos no han funcionado.

El algoritmo de PPO, es el que ha dado mejores resultados en todos los entornos, con una gran estabilidad en las recompensas obtenidas, en comparación a los otros algoritmos.

En entornos como el último, dónde el agente tiene que aprender a relacionar un mayor número de cosas, junto a la inmensa variedad de situaciones en las que se puede encontrar, se ha requerido de la modificación del entorno para ir aumentando su dificultad y de esta manera, serle más fácil el agente su aprendizaje. Debido a que, de primeras el agente es incapaz de aprender. Con esto se pretende llegar, en que, en entornos más complejos, la tarea principal del agente se tiene que dividir en subtareas más fáciles, y que el aprendizaje de todas ellas, le lleven al agente al aprendizaje de la tarea que tiene marcada como objetivo principal.

Los tiempos de ejecución en este tipo de algoritmos son lentos, el agente no aprende instantáneamente. Esto es debido, a que estos algoritmos se basan en hacer muchas iteraciones, en que se busca encontrar una configuración de los parámetros de la red neuronal, óptima. Con el paso de los años, se han ido mejorando las arquitecturas en el campo del hardware, para hacer que los cálculos se realicen cada vez más rápidos. Esto supone, que cada vez se puedan hacer entrenamientos con un menor tiempo y con tareas más complejas.

El aprendizaje por refuerzo aplicado al mundo real tiene un gran potencial, porque al tener la capacidad de aprender una gran cantidad de situaciones distintas, se ahorra tener que programar cada una de las posibles situaciones que pueden suceder. En el caso de un brazo robótico, dónde le dejan un objeto en una caja, lo tiene que coger y moverlo en otro lugar, con la programación clásica se tendría que dejar el objeto siempre exactamente en una posición concreta dentro de unas determinadas tolerancias. En cambio, con aprendizaje por refuerzo el brazo robot

sería capaz de coger el objeto, indistintamente de la posición en la que se dejara el objeto dentro de la caja. Debido a que se habría estado entrenando previamente, a coger y mover el objeto en multitud de situaciones diversas. Y si en la práctica, se encontrará con una situación que no se hubiera encontrado en el entrenamiento, también sería capaz de llegar a deducir como resolver esta nueva situación, en base a las que se había estado entrenando.

Para terminar, un camino posible a seguir en el ámbito del aprendizaje por refuerzo, sería en poder llegar a trasladarle de manera más directa el objetivo que tiene que realizar el agente. Al igual que hacen las personas, cuando una persona está enseñando a otra, y un ejemplo podría ser mover una caja de un lado a otro. La diferencia aquí, es que entre personas se puede trasladar directamente lo que se quiere, que es mover la caja. En cambio, en aprendizaje por refuerzo, el agente lo tiene que descubrir por sí mismo primero, descubrir que su objetivo es mover la caja, y luego perfeccionar el movimiento. Sería realmente interesante, poder llegar al segundo paso directamente, en el que el agente aprendiera solo a perfeccionar su objetivo, en vez de también descubrirlo.

9. Trabajo a futuro

Debido a que el tiempo del trabajo final de máster es limitado, no se han podido realizar diversas cosas, que a medida que se iba confeccionando el TFM también se querían hacer.

A continuación, se detallan las cosas que no se han podido realizar o bien se han quedado a medias, y se quieren hacer una vez finalizado el TFM.

Entorno con objetos dinámicos

Los objetos de los entornos que se han creado para este trabajo, son estáticos, aunque el agente los puede mover dependiendo del entorno.

Un próximo paso, sería incorporar objetos dinámicos, que en el caso del entorno 1, sería que el objetivo, que es el cubo naranja, en vez de estar quieto, estuviera en movimiento con trayectorias aleatorias y el agente tuviera que interceptarlo.

Entorno con parámetros extra

La captación de observaciones en este trabajo es mediante Raycast (proyección de rayos del agente para detectar los diferentes objetos del entorno).

La idea de este entorno, sería implementar algún parámetro, por ejemplo, la vida del agente o el cansancio, y que también formasen parte de las observaciones del agente. Estas observaciones en forma de parámetros se le pasarían al agente a través del script, por lo tanto, el agente tendría observaciones que le vendrían por parte de Raycast y por script.

Un ejemplo sería, que durante el transcurso de un episodio el agente se vaya cansando (parámetro que iría reduciendo la velocidad), y hubiera zonas en el entorno, dónde pudiera recuperarse.

Entorno con model based

Entorno estilo ajedrez que se pueda basar en un modelo, para poder implementar técnicas de aprendizaje por refuerzo con la combinación de métodos heurísticos o de inteligencia artificial para la búsqueda de soluciones óptimas durante el proceso de aprendizaje.

Entorno con 2 agentes compitiendo entre ellos

En el entorno 4, los agentes cooperan entre ellos y tienen el mismo objetivo.

Se empezó un entorno, pero en este caso, los 2 agentes son diferentes uno respecto al otro y con distintos objetivos.

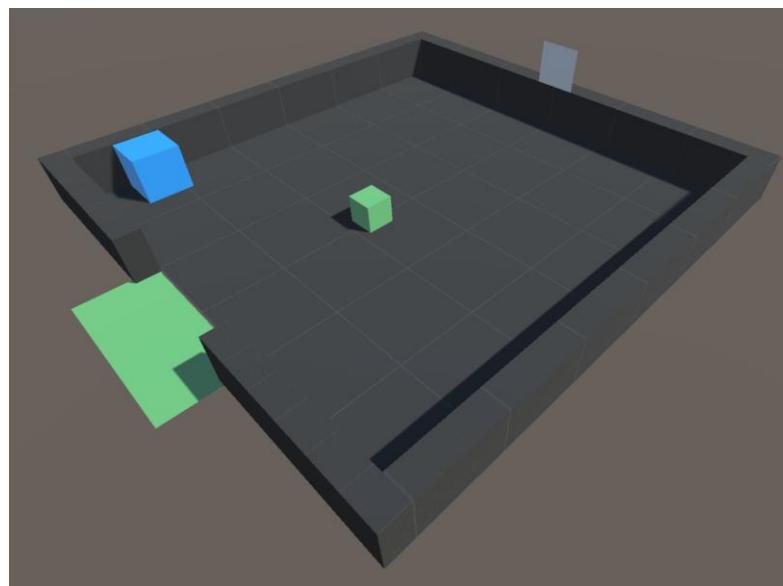


Imagen 33 Entorno con agentes compitiendo entre ellos

Hay 2 agentes, el cubo azul y el cubo verde. Se inicializan en posiciones aleatorias por todo el campo, el agente verde tiene como objetivo alcanzar la zona verde y el agente azul, por lo contrario, su objetivo es impedírselo.

La razón por la cual no se ha seguido adelante con este entorno, es porque la estructura del código con la que se han implementado los algoritmos, como se ha comentado, es la siguiente:



Imagen 34 Flujo de información entre Python y Unity

El problema es, que con Gym de ML-Agents, no deja entrenar 2 agentes diferentes, refiriéndose a que las acciones, las observaciones y la manera de obtener las recompensas, pueden ser diferentes entre ambos agentes. Y por lo tanto la estructura tiene que ser de la siguiente manera:

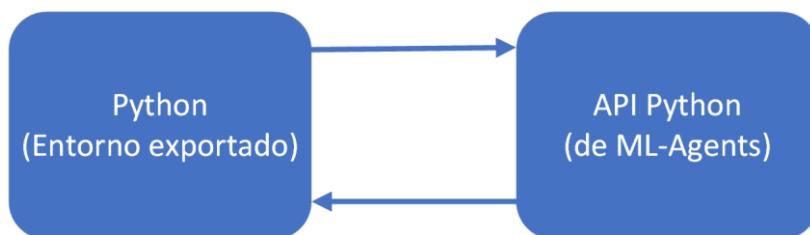


Imagen 35 Flujo de información entre Python y Unity (2)

Este cambio se empezó hacer, pero como el trabajo ya estaba en un punto bastante avanzado, requería cambiar la estructura base de como se había realizado y por motivos de tiempo, se ha seguido con la estructura original, utilizando Gym de ML-Agents.

Captación de observaciones a través de imágenes

Como se ha comentado anteriormente, el agente también puede captar las observaciones en forma de imágenes. Esto implica modificar el modelo de la red neuronal y algunos cambios más en el código de cada algoritmo.

Se empezó a probar en el entorno 1, y el punto en el que estaba el trabajo, se hizo con el algoritmo de Actor-Critic. Pero el agente no aprendía y por motivos de tiempo, no se le ha dedicado más tiempo en hacer funcionar esta implementación.

Acciones en un espacio continuo

El espacio de acciones del agente en los diferentes entornos de este trabajo es discreto. Hacer que el espacio de acciones del agente sea en continuo implica también modificar la red neuronal y algunos cambios más en el código de cada algoritmo.

Partiendo de la idea del entorno 3, de fútbol, se creó un entorno dónde el agente hace de portero, sólo puede moverse hacia la derecha o hacia la izquierda, se le va lanzando la pelota hacia la portería (con una dirección aleatoria y diferente cada vez) y tiene que aprender a parar los lanzamientos.

En el punto en el que estaba el trabajo se implementó con el algoritmo de Actor-Critic, pero el agente no aprendía y por motivos de tiempo, no se le ha dedicado más tiempo en hacer funcionar esta implementación.



Imagen 36 Entorno con el espacio de acciones en continuo

10. Bibliografía

Páginas web

- GitHub. Consulta de códigos de algoritmos.

<https://github.com/>

- Blog de OpenAI sobre aprendizaje por refuerzo y algoritmos de Policy Gradients.

<https://spinningup.openai.com/>

- Blog sobre algoritmos de Policy Gradients.

<https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html>

- Blogs de Unity.

<https://blogs.unity3d.com/>

- Blogs de inteligencia artificial y sobre aprendizaje por refuerzo.

<https://medium.com/>

<https://towardsdatascience.com/artificial-intelligence/home>

<https://becominghuman.ai/>

- Documentación de Unity.

<https://docs.unity3d.com/>

- Foros de programación.

<https://es.stackoverflow.com/>

<https://stats.stackexchange.com/>

- Foros de Unity.

<https://forum.unity.com/>

<https://answers.unity.com/index.html>

- Foro de PyTorch.

<https://discuss.pytorch.org/>

- Foro de Reddit sobre aprendizaje por refuerzo

<https://www.reddit.com/r/reinforcementlearning/>

Referencias

[Sutton & Barto, 2018] Reinforcement Learning: An Introduction, Second Edition, MIT Press, 2018.

<https://mitpress.mit.edu/books/reinforcement-learning-second-edition>

[OpenAI, 2018] Open AI. Kinds of RL Algorithms.

https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

[Mnih et al, 2013] DQN. Playing Atari with Deep Reinforcement Learning, Mnih et al, 2013.

<https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>

[Konda & Tsitsiklis, 2002] Actor Critic algorithms, Konda & Tsitsiklis, 2002.

<https://papers.nips.cc/paper/1786-actor-critic-algorithms.pdf>

[Schulman et al, 2016] High-dimensional continuous control using generalized advantage estimation, Schulman et al, 2016.

<https://arxiv.org/pdf/1506.02438.pdf>

[Mnih et al, 2016] A3C. Asynchronous Advantage Actor-Critic, Mnih et al, 2016.

<https://arxiv.org/abs/1602.01783>

[OpenAI, 2017] A2C. OpenAI Baselines: ACKTR & A2C.

<https://openai.com/blog/baselines-acktr-a2c/>

[Schulman et al, 2017] PPO. Proximal Policy Optimization, Schulman et al, 2017

<https://arxiv.org/abs/1707.06347>

[1] Unity.

<https://unity.com/>

[2] ML-Agents de Unity.

<https://unity3d.com/es/machine-learning>

[3] Librería TensorFlow. Biblioteca de aprendizaje automático.

<https://www.tensorflow.org/>

[4] ML-Agents GitHub.

<https://github.com/Unity-Technologies/ml-agents>

[5] Librería Gym de Open AI.

<https://gym.openai.com/>

[6] Librería PyTorch. Biblioteca de aprendizaje automático.

<https://pytorch.org/>

[7] Librería NumPy. Biblioteca de funciones matemáticas.

<https://numpy.org/>

[8] Librería Matplotlib. Generación de gráficos a partir de datos.

<https://matplotlib.org/>

[9] Librería Seaborn. Generación de gráficos a partir de datos.

<https://seaborn.pydata.org/>

[10] Librería Pandas. Extensión de NumPy.

<https://pandas.pydata.org/>

Imágenes

· Imagen 1 Brazo robot simulado

<https://www.youtube.com/watch?v=jwSbzNHGfIM&t=80s>

· Imagen 2 Brazo robot real

<https://www.youtube.com/watch?v=jwSbzNHGfIM&t=41s>

- Imagen 3 Esquema general del aprendizaje por refuerzo

https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

- Imagen 4 Clasificación algoritmos de aprendizaje por refuerzo

<https://medium.com/@SmartLabAI/reinforcement-learning-algorithms-an-intuitive-overview-904e2dff5bbc>

- Imagen 5 Esquema del algoritmo de DQN

<https://greentec.github.io/reinforcement-learning-third-en/>

- Imagen 6 Esquema del algoritmo de Actor-Critic

https://www.youtube.com/watch?v=w_3mmm0P0j8=175s

- Imagen 7 Esquema del algoritmo de A3C

<https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html#a3c>

- Imagen 8 Esquema del algoritmo de A2C

<https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html#a2c>

Anexos

Pseudocódigos de Algoritmos

En este anexo se pueden encontrar los pseudocódigos de los diferentes algoritmos de aprendizaje por refuerzo, que se han utilizado en este trabajo.

DQN

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
    Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
    for  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
        Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
    end for
end for

```

Actor-Critic

1. Initialize s, θ, w at random; sample $a \sim \pi_\theta(a|s)$.
2. For $t = 1 \dots T$:
 1. Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$;
 2. Then sample the next action $a' \sim \pi_\theta(a'|s')$;
 3. Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a) \nabla_\theta \ln \pi_\theta(a|s)$;
 4. Compute the correction (TD error) for action-value at time t:

$$\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$$
and use it to update the parameters of action-value function:

$$w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$$
 5. Update $a \leftarrow a'$ and $s \leftarrow s'$.

A3C

1. We have global parameters, θ and w ; similar thread-specific parameters, θ' and w' .
2. Initialize the time step $t = 1$
3. While $T \leq T_{\text{MAX}}$:
 1. Reset gradient: $d\theta = 0$ and $dw = 0$.
 2. Synchronize thread-specific parameters with global ones: $\theta' = \theta$ and $w' = w$.
 3. $t_{\text{start}} = t$ and sample a starting state s_t .
 4. While ($s_t \neq \text{TERMINAL}$) and $t - t_{\text{start}} \leq t_{\text{max}}$:
 1. Pick the action $A_t \sim \pi_{\theta'}(A_t | S_t)$ and receive a new reward R_t and a new state s_{t+1} .
 2. Update $t = t + 1$ and $T = T + 1$
 5. Initialize the variable that holds the return estimation

$$R = \begin{cases} 0 & \text{if } s_t \text{ is TERMINAL} \\ V_{w'}(s_t) & \text{otherwise} \end{cases}$$
 6. For $i = t - 1, \dots, t_{\text{start}}$:
 1. $R \leftarrow \gamma R + R_i$; here R is a MC measure of G_i .
 2. Accumulate gradients w.r.t. θ' : $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi_{\theta'}(a_i | s_i) (R - V_{w'}(s_i))$;
Accumulate gradients w.r.t. w' : $dw \leftarrow dw + 2(R - V_{w'}(s_i)) \nabla_{w'} (R - V_{w'}(s_i))$.
 7. Update asynchronously θ using $d\theta$, and w using dw .

A2C

Initialize timestep counter $N = 0$ and network weights θ, θ_v
 Instantiate set e of n_e environments
repeat
for $t = 1$ to t_{max} **do**
 Sample a_t from $\pi(a_t | s_t; \theta)$
 Calculate v_t from $V(s_t; \theta_v)$
 parallel for $i = 1$ to n_e **do**
 Perform action $a_{t,i}$ in environment e_i
 Observe new state $s_{t+1,i}$ and reward $r_{t+1,i}$
 end parallel for
end for

$$R_{t_{\text{max}}+1} = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_{t_{\text{max}}+1}; \theta) & \text{for non-terminal } s_t \end{cases}$$

for $t = t_{\text{max}}$ down to 1 **do**
 $R_t = r_t + \gamma R_{t+1}$
end for

$$d\theta = \frac{1}{n_e \cdot t_{\text{max}}} \sum_{i=1}^{n_e} \sum_{t=1}^{t_{\text{max}}} (R_{t,i} - v_{t,i}) \nabla_{\theta} \log \pi(a_{t,i} | s_{t,i}; \theta) + \beta \nabla_{\theta} H(\pi(s_{e,t}; \theta))$$

$$d\theta_v = \frac{1}{n_e \cdot t_{\text{max}}} \sum_{i=1}^{n_e} \sum_{t=1}^{t_{\text{max}}} \nabla_{\theta_v} (R_{t,i} - V(s_{t,i}; \theta_v))^2$$

 Update θ using $d\theta$ and θ_v using $d\theta_v$.
 $N \leftarrow N + n_e \cdot t_{\text{max}}$
until $N \geq N_{\text{max}}$

PPO

```
for iteration=1, 2, ... do
    for actor=1, 2, ..., N do
        Run policy  $\pi_{\theta_{\text{old}}}$  in environment for  $T$  timesteps
        Compute advantage estimates  $\hat{A}_1, \dots, \hat{A}_T$ 
    end for
    Optimize surrogate  $L$  wrt  $\theta$ , with  $K$  epochs and minibatch size  $M \leq NT$ 
     $\theta_{\text{old}} \leftarrow \theta$ 
end for
```

Unity

Script del Agente y el Entorno

En este apartado se detalla con más profundidad a nivel de código, el diseño de la programación del agente y el entorno. Para una mayor comprensión, se explicará con el código del Entorno 1, que es el más sencillo.

Código del script principal del agente y el entorno:

- **Start:** inicialización de las variables (dependiendo del entorno pueden variar).

```
void Start()
{
    areaBounds = ground.GetComponent<Collider>().bounds;
}
```

La variable `areaBounds`, en el caso de multiagente sirve para saber la posición del área del agente, de esta manera se sabe la posición que le corresponde a cada agente en su determinada área.

- **AgentReset:** reinicialización del agente y los objetos del entorno (posiciones y parámetros).

```
public override void AgentReset()
{
    // Agent: reset position
    this.transform.position = ground.transform.position + new Vector3(0, 0.5f, 0);
    // Target: new random position
    var randomPosX = Random.Range(-areaBounds.extents.x, areaBounds.extents.x);
    var randomPosZ = Random.Range(-areaBounds.extents.z, areaBounds.extents.z);
    var randomSpawnPos = ground.transform.position + new Vector3(randomPosX, 0.5f,
        randomPosZ);
    Target.position = randomSpawnPos;
}
```

Son las posiciones que toma el agente y los diferentes objetos del entorno al principio de cada episodio y también se resetean los parámetros.

- **MoveAgent:** todas las posibles acciones que puede tomar el agente.

```
public void MoveAgent(float[] act)
{
    var action = Mathf.FloorToInt(act[0]);
    switch (action)
    {
        case 1:
            this.transform.position = transform.position +
                new Vector3(0, 0, speed * Time.deltaTime);
            break;
        case 2:
            this.transform.position = transform.position -
                new Vector3(0, 0, speed * Time.deltaTime);
            break;
        case 3:
            this.transform.position = transform.position +
                new Vector3(speed * Time.deltaTime, 0, 0);
            break;
        case 4:
            this.transform.position = transform.position -
                new Vector3(speed * Time.deltaTime, 0, 0);
            break;
    }
}
```

Son los movimientos que puede realizar el agente, como por ejemplo ir hacia adelante, atrás, rotar, etc.

- **AgentAction:** es la acción que toma el agente en cada step.

```
public override void AgentAction(float[] vectorAction)
{
    // We punish slightly to encourage hurrying up
    AddReward(-1f / agentParameters.maxStep);
    // Move the agent using the action.
    MoveAgent(vectorAction);
    // Fall off platform
    if (this.transform.position.y < 0.45f)
    {
        AddReward(-1f);
        AgentReset();
        Done();
    }
}
```

Como esta función se ejecuta en cada step, se añade una recompensa negativa por cada step que transcurre y el agente aún no ha terminado la tarea que tiene que realizar. También se comprueba dependiendo del entorno, si se cumple con alguna condición en qué el agente ha realizado alguna acción errónea o si ha completado la tarea y por lo tanto se tenga que finalizar el episodio antes de los

máximos steps. En este caso, si el agente se cae de la plataforma, es una acción errónea o si toca el cubo naranja, se completa la tarea. En ambos casos se finaliza el episodio de inmediato.

- **Heuristic:** son las entradas de un usuario para mover el agente directamente.

```
public override float[] Heuristic()
{
    if (Input.GetKey(KeyCode.W))
    {
        return new float[] { 1 };
    }
    if (Input.GetKey(KeyCode.S))
    {
        return new float[] { 2 };
    }
    if (Input.GetKey(KeyCode.D))
    {
        return new float[] { 3 };
    }
    if (Input.GetKey(KeyCode.A))
    {
        return new float[] { 4 };
    }
    return new float[] { 0 };
}
```

Con esta función se puede mover el agente directamente mediante teclado o ratón, a través de las posibles acciones que puede tomar el agente. Sirve para testear el entorno y comprobar que todo funciona correctamente, antes de entrenar el agente. En este caso se tiene la combinación de teclas W, S, A, D para mover al agente hacia adelante, atrás, izquierda y derecha respectivamente.

- **Tarea completada:** es cuando el agente completa su tarea.

```
void OnCollisionEnter(Collision col)
{
    // Touched target
    if (col.gameObject.CompareTag("target"))
    {
        ScoredAGoal();
    }
}

public void ScoredAGoal()
{
    AddReward(1f);
    AgentReset();
    Done();
}
```

La tarea se completa cuando el agente toca el objeto target (cubo naranja). Esto se detecta a través de una colisión entre el agente y el target, cuando esto sucede se llama a la función ScoredAGoal para recompensar al agente positivamente por haber completado la tarea, se finaliza el episodio y se comienza uno nuevo.

Parámetros del Agente

En esta sección se explicarán todos los parámetros con más detalle, que se tienen que configurar en el agente para su entrenamiento en Unity.

Vector de observaciones y de acciones

Son los parámetros que hacen referencia al vector de observaciones y de acciones del agente.

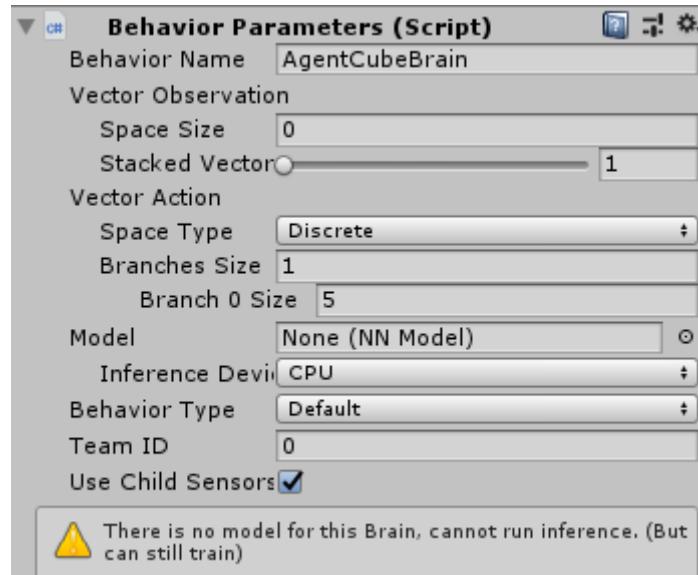


Imagen 37 Parámetros del vector de observaciones y de acciones

Vector Observation

- **Space Size:** es el tamaño del vector de observaciones del agente.
- **Stacked Vectors:** es el número de observaciones previas que se guardan y se utilizan colectivamente para la toma de decisiones del agente. El tamaño del vector de observaciones es: Space Size x Stacked Vector.

Vector Action

- **Space Type:** corresponde a si el vector de acciones es un solo entero (Discreto) o una serie de valores floats (Continuo).
- **Space Size (Continuo):** tamaño del vector de acciones en un espacio de acciones continuo.
- **Branches (Discreto):** es un array de enteros, que corresponde a múltiples acciones discretas concurrentes. Y cada Branch tiene un número de posibles acciones.
- **Model:** el modelo de la red neuronal utilizado para la inferencia (obtenido después del entrenamiento).
- **Inference Device:** elección de utilizar CPU o GPU para ejecutar la inferencia del modelo.
- **Behaviour Type:** corresponde si las acciones del agente se toman con la función Heuristic (explicado en el anterior apartado), por inferencia cuando se tiene un modelo entrenado, o con Default para el entrenamiento.

Generales

Son parámetros generales del agente y el entorno.

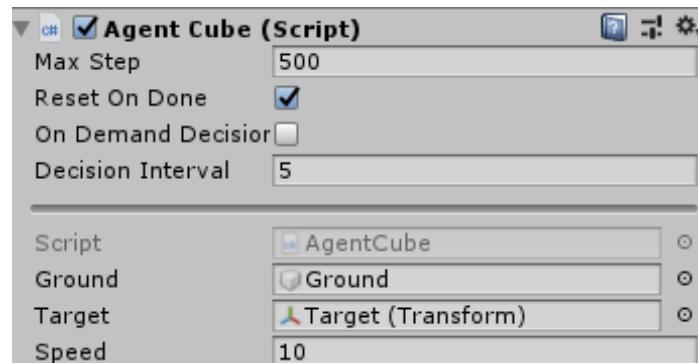


Imagen 38 Parámetros generales del Agente

- **Max Step:** es el número máximo de steps del agente en un episodio. Una vez se alcance este número, si la casilla Reset On Done está marcada, el agente se reinicializará.

- **Reset On Done:** hace referencia a si se debe llamar o no a la función AgentReset, cuando el agente llegue a los Max Step, cuando ha completado la tarea o ha hecho alguna acción errónea que implique finalizar el episodio.
- **Parámetros del agente y el entorno:** son parámetros propios del agente y el entorno que se ha diseñado y pueden variar dependiendo del entorno. Pueden ser, por ejemplo, la velocidad del agente, posiciones de los objetos, etc.

Raycast

Parámetros del sensor Raycast, para la captación de observaciones del entorno por parte del agente.

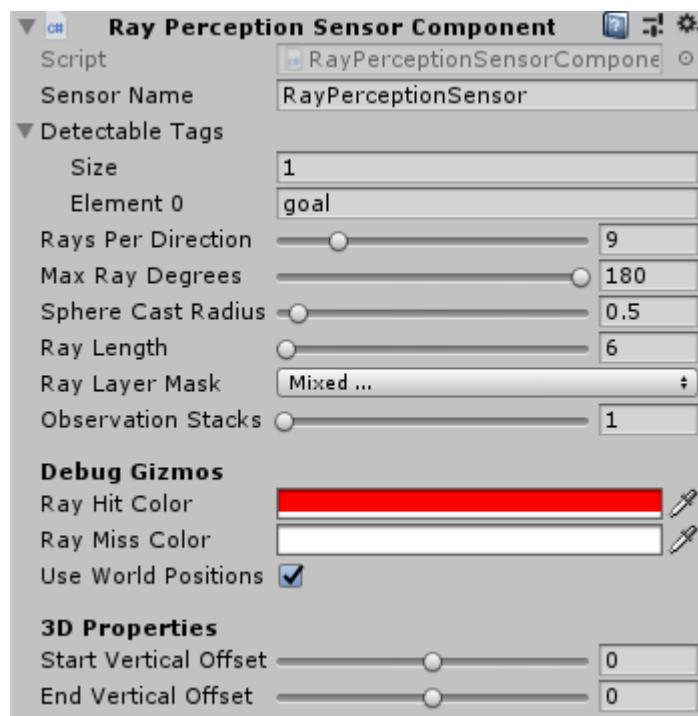


Imagen 39 Parámetros de Raycast

- **Detectable Tags:** es una lista de los nombres correspondientes a los distintos tipos de objetos que se encuentran en el entorno. Y el agente tiene que ser capaz de poder distinguirlos.
- **Rays per Direction:** son el número de rayos que se proyectan. Siempre se proyecta un rayo hacia adelante, y el número de rayos que se proyectan, son hacia la izquierda y hacia la derecha.
- **Max Ray Degrees:** es el ángulo (en grados) de los rayos más externos. 90 grados representa a la izquierda y a la derecha del agente.

- **Sphere Cast Radius:** es el tamaño de la esfera que se proyecta al final del rayo, que sirve para detectar un objeto en el entorno. Si se establece en 0, se usarán rayos en lugar de esferas. Los rayos pueden ser más eficientes, especialmente en escenas complejas.
- **Ray Length:** es la longitud de los distintos rayos.
- **Ray Layer Mask:** son las capas de los objetos del entorno, que la proyección de rayos o esferas detectaran. Si un objeto no está dentro de una de estas capas, los rayos no lo detectaran.
- **Observation Stacks:** es el número de observaciones previas que se guardan y se utilizan colectivamente para la toma de decisiones del agente.
- **Start Vertical Offset:** es el offset vertical del punto de inicio del rayo.
- **End Vertical Offset:** es el offset vertical del punto final del rayo.

Para calcular el tamaño total del vector observaciones con el sensor Raycast, se utiliza la siguiente formula:

$$(Observation Stacks) * (1 + 2 * Rays per Direction) * (N.º Detectable Tags + 2)$$

Hiperparámetros

En este anexo, se explican los hiperparámetros con más profundidad, de los diversos algoritmos de aprendizaje por refuerzo que se han utilizado en este trabajo.

Generales

En este apartado se encuentran los hiperparámetros generales que aparecen en todos los algoritmos.

- **max_steps:** número total de pasos (recopilación de observaciones y acciones tomadas) que se deben realizar en el entorno antes de finalizar el proceso de entrenamiento. Es decir, la cantidad máxima de acciones que puede realizar el agente para completar la tarea asignada.

Rango típico: 5e5 - 1e7

- **learning_rate:** tasa de aprendizaje inicial para el descenso de gradientes. Corresponde a la fuerza de cada paso de actualización de descenso de gradientes. Si el entrenamiento es inestable y la recompensa no aumenta de manera constante, se debería reducir este valor.

Rango típico: 1e-5 - 1e-3

- **batch_size:** número de experiencias utilizadas en cada iteración para el descenso de gradientes. Este valor tiene que ser varias veces más pequeño que el buffer_size.

Rango típico: 32 - 512

- **buffer_size:** número total de experiencias para recopilar antes de actualizar el modelo. Tiene que ser varias veces mayor que el batch_size.

Rango típico: 2048 - 10240

- **gamma:** factor de descuento para futuras recompensas provenientes del entorno. Se puede pensar, en qué tan lejos en el futuro el agente debería preocuparse por las posibles recompensas. En situaciones en las que el agente debería estar actuando en el presente para prepararse para recompensas en el futuro distante, este valor debería ser grande. En los casos en que las recompensas son más inmediatas, puede ser más pequeño.

Rango típico: 0.8 – 0.995

DQN

Hiperparámetros que solo se encuentran en el algoritmo de DQN.

- **epsilon:** parámetro relacionado con la exploración de la agente. Se empieza con un valor igual a 1, donde las acciones del agente son más aleatorias (etapa de exploración) y se va decreciendo hasta llegar a un valor cercano a 0, donde las acciones, se escogen a partir de la policy (etapa de explotación).

Rango típico: 0.01 - 0.05 (valor final de epsilon)

- **update_every:** es la frecuencia (número de pasos) con la que se irá actualizando el modelo.

Rango típico: 16 - 256

Policy Gradients

Hiperparámetros que solo están en los algoritmos de Policy Gradients.

- **entropy_coef:** fuerza de la regularización de la entropía, lo que hace que la policy sea “más aleatoria”. Esto asegura que los agentes exploren adecuadamente el espacio de acción durante el entrenamiento. Aumentar este parámetro implica que los agentes tomen más acciones aleatorias. Se debe ajustar de modo que la entropía disminuya lentamente junto con aumentos en la recompensa. Si la entropía cae demasiado rápido, se tiene que aumentar este valor. Si disminuye muy lentamente, se tiene que disminuir.

Rango típico: 1e-4 - 1e-2

- **lambda:** parámetro utilizado para calcular la Generalized Advantage Estimate (GAE). Esto puede considerarse como cuánto depende el agente de su estimación del valor actual, al calcular una estimación del valor actualizado. Los valores bajos corresponden a confiar más en la estimación del valor actual, y los valores altos corresponden a confiar más en las recompensas reales recibidas en el entorno.

Rango típico: 0.9 – 0.95

PPO

Hiperparámetros que pertenecen al algoritmo de PPO solamente.

- **epsilon:** influye en la rapidez en que la policy puede evolucionar durante el entrenamiento. Corresponde al umbral aceptable de divergencia entre las policies antiguas y nuevas durante la actualización del descenso de gradiente.

Valores pequeños dará como resultado actualizaciones más estables, pero también ralentizará entrenamiento.

Rango típico: 0.1 – 0.3

· **num_epoch:** número de pasadas a través del buffer de experiencia al realizar la optimización del descenso de gradiente.

Rango típico: 3 – 10

Código y vídeos

En este anexo se proporcionan los enlaces que dan acceso, a todos los archivos que se han hecho para implementar todos los algoritmos de este trabajo, los scripts de los entornos simulados que se han creado y los vídeos de los agentes realizando las tareas de cada entorno, entrenados previamente con el algoritmo de PPO.

En cualquiera de los dos enlaces, se puede encontrar el material que se acaba de mencionar:

- **Drive:**

https://drive.google.com/drive/folders/12Rtv YY_sTOByK-Qli4VRmLR2FLXm1CZ?usp=sharing

- **Dropbox:**

<https://www.dropbox.com/sh/as36iok0cveuh0p/AAAAJyoKMQeu8e-3YDI2wAOna?dl=0>

Las carpetas que hay en cada enlace son las mismas, y son las siguientes:

- **Environments:** en esta carpeta están los scripts de los entornos que se han creado.
- **RL Algorithms:** toda la parte de código que se ha realizado, para la implementación de los algoritmos de aprendizaje por refuerzo.
- **Videos:** visualización de un agente en cada entorno, haciendo la tarea varias veces en distintas situaciones.