Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Aprendizaje por refuerzo en sistemas multiagente mediante MARLÖ

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Martínez Sanchis, Genís

Tutor: Julian Inglada, Vicente J.

Director experimental: Palanca Cámara, Javier

Curso 2020-2021

Resum

En aquest treball de fi de grau es realitzarà un estudi basat en l'anàlisi de l'aplicació d'algoritmes d'aprenentatge per reforç per a entorns mono-agent sobre entorns multi-agent basats en la plataforma MARLÖ. Tot això amb l'objectiu de comparar l'eficàcia i eficiència d'aquests algorismes en entorns per als quals no han estat dissenyats. Per això serà necessari tant el disseny i creació d'entorns personalitzats com la modificació de les implementacions dels algoritmes per adaptar-les a aquests entorns.

Paraules clau: Sistemes multi-agent, intel·ligència artificial, aprenentatge per reforç, videojocs

Resumen

En este trabajo de fin de grado se realizará un estudio basado en el análisis de la aplicación de algoritmos de aprendizaje por refuerzo para entornos mono-agente sobre entornos multi-agente basados en la plataforma MARLÖ. Todo esto con el objetivo de comparar la eficacia y eficiencia de dichos algoritmos en entornos para los cuales no han sido diseñados. Para esto será necesario tanto el diseño y creación de entornos personalizados como la modificación de las implementaciones de los algoritmos para adaptarlas a dichos entornos.

Palabras clave: Sistemas multi-agente, inteligencia artificial, aprendizaje por refuerzo, videojuegos

Abstract

In this final degree thesis, we perform a study based on the analysis of the application of reinforcement learning algorithms designed for single-agent environments on multi-agent environments based on the MARLÖ platform. All this in order to compare the effectiveness and efficiency of these algorithms in environments for which they have not been designed. This will require both the design and creation of custom environments and the modification of the algorithm implementations to adapt them to those environments.

Key words: Multi-agent systems, artificial intelligence, reinforcement learning, video games

Índice general

Índice general v

Índice de figuras vii

Índice de tablas vii

Índice de algoritmos viii

1 Introducción 1
- 1.1 Motivación . . . 1
- 1.2 Objetivos . . . 2
- 1.3 Estructura de la memoria . . . 2

2 Estado del arte 3
- 2.1 Aprendizaje por refuerzo . . . 3
- 2.2 DQN . . . 6
- 2.3 Policy Gradients . . . 7
- 2.4 Estado actual . . . 9

3 Solución propuesta 11
- 3.1 MARLÖ . . . 11
- 3.2 DDQN . . . 13
- 3.3 TRPO . . . 15
- 3.4 MADDPG . . . 17
- 3.5 Vista general . . . 18

4 Experimentación y Resultados 19
- 4.1 Experimentación con el algoritmo DDQN . . . 21
  - 4.1.1 Entorno 3x3 con DDQN . . . 21
  - 4.1.2 Entorno 5x5 con DDQN . . . 24
- 4.2 Experimentación con el algoritmo TRPO . . . 28
  - 4.2.1 Entorno 3x3 con TRPO . . . 28
  - 4.2.2 Entorno 5x5 con TRPO . . . 30
- 4.3 Experimentación con el algoritmo MADDPG . . . 33
- 4.4 Análisis de resultados . . . 35

5 Conclusiones 37
- 5.1 Trabajo futuro . . . 38
- 5.2 Relación con los estudios cursados . . . 38
- 5.3 Agradecimientos . . . 39

Bibliografía 41

Índice de figuras

2.1 Representación de la interacción agente-entorno 4
2.2 Agentes en el entorno de Hide and Seek de OpenAI 10

3.1 Ejemplo de un entorno de MalmÖ donde el agente tiene que conseguir llegar a la meta lo más rápido posible y sin caerse a la lava. 11
3.2 Modelo de la red utilizada en los algoritmos 14

4.1 Representación de Stag Hunt en Minecraft en un recinto de 3x3 20
4.2 Representación de Stag Hunt en Minecraft en un recinto de 5x5 21
4.3 Recompensas en el entorno 3x3 usando DDQN 22
4.4 Finales de los episodios en el entorno 3x3 usando DDQN 23
4.5 Acciones de los agentes en el entorno 3x3 DDQN 24
4.6 Recompensas en el entorno 5x5 usando DDQN 25
4.7 Acciones de los agentes en el entorno 5x5 DDQN 25
4.8 Finales de los episodios en el entorno 5x5 usando DDQN 26
4.9 Finales de los episodios en el entorno 5x5 usando DDQN con parámetros modificados 27
4.10 Recompensas en el entorno 5x5 usando DDQN con parámetros modificados 27
4.11 Evolución de la recompensa media de los agentes en el entorno 3x3 usando TRPO 28
4.12 Evolución de los finales de los episodios en el entorno 3x3 usando TRPO . 29
4.13 Distribución de las acciones de los agentes en el entorno 3x3 TRPO 30
4.14 Evolución de la recompensa media de los agentes en el entorno 5x5 usando TRPO 31
4.15 Evolución de los finales de los episodios en el entorno 5x5 usando TRPO . 31
4.16 Distribución de las acciones de los agentes en el entorno 5x5 TRPO 32
4.17 Finales de los episodios en el entorno 5x5 usando TRPO parámetros modificados 32
4.18 Recompensa media en el entorno 5x5 usando TRPO con parámetros modificados 33
4.19 Finales de los episodios en el entorno 3x3 usando MADDPG 34
4.20 Recompensa media de los agentes en el entorno 3x3 usando MADDGP . . 34

Índice de tablas

4.1 Representación tabular de Stag Hunt. 19

Índice de algoritmos

2.1 Algoritmo Q-learning . . . 6
2.2 Deep Q-learning con *Replay buffer* . . . 7

3.1 Double Q-learning . . . 14
3.2 TRPO . . . 15
3.3 Implementación de TRPO . . . 16
3.4 Algoritmo MADDPG . . . 17

CAPÍTULO 1
Introducción

Los avances en en el aprendizaje automático han supuesto la capacidad de encontrar soluciones a problemas cada vez más complicados. La primera demostración de este potencial se obtiene en 2015 con la victoria de la IA de Google, Alpha Go, contra un jugador profesional de Go y posteriormente contra el 18 veces campeón del mundo del mismo juego. Este solo fue el comienzo de la gran cantidad de avances que habría por llegar. En 2017 DeepMind presentó a AlphaZero, capaz de jugar tanto a Go como ajedrez y shogi y vencer a jugadores profesionales de estos juegos. En 2019 AlphaStar fue la primera IA capaz de jugar a un nivel profesional el famoso videojuego y eSport StarCraft II. Tampoco se debe olvidar el trabajo de OpenAI con su librería "gym" que ofrece la capacidad a cualquier persona de entrenar algoritmos que jueguen y dominen un gran catálogo de juegos de la Atari 2600.

Pero algo que tienen estos avances en común es el uso en menor o mayor medida de técnicas de aprendizaje por refuerzo. El aprendizaje por refuerzo es una técnica de aprendizaje automático no supervisado basada en cómo los seres vivos aprenden. Esta técnica se basa en que los agentes recibirán recompensas por cada una de sus acciones, estas recompensas serán positivas si cumplen el objetivo o negativas en caso contrario. De esta manera los agentes pasarán de comportarse aleatoriamente a priorizar las acciones que más recompensas les ofrecen y por ende serán capaces de cumplir los objetivos. Esta técnica es interesante porque no necesita de una gran colección de datos para poder aprender sino que los datos se generan según los agentes aprenden.

Aunque mientras que el aprendizaje por refuerzo con un solo agente está bastante avanzado y ofrece buenos resultados, en entornos multi-agente, donde los objetivos requieren de la cooperación o la competición entre agentes, surgen problemas con los algoritmos actuales debido a no estar diseñados para tener en cuenta a otro agente capaz de aprender como un parámetro más del aprendizaje.

1.1 Motivación

La motivación de este proyecto viene dada por una parte por la necesidad de profundizar en uno de los campos más actuales de los agentes inteligentes como es el muti-agent learning en entornos que permiten comportamientos cooperativos o competitivos entre los agentes. Este campo presenta varios problemas en los algoritmos de aprendizaje que normalmente funcionaban perfectamente en entornos mono-agente. Por lo que la experimentación y análisis de diferentes algoritmos sometidos a estos entornos puede ayudar al entendimiento de como estos problemas pueden afectar al proceso de aprendizaje.

Otra motivación es el uso de dichos algoritmos en el área de los videojuegos. Para ello emplearemos MARLÖ, que es una modificación de la plataforma MalmÖ, plataforma que tiene como base el famoso videojuego Minecraft. Dicha modificación se origina en una competición en 2018 en la que los participantes debían diseñar algoritmos con el objetivo de conseguir que los agentes guiados por estos algoritmos superaran diferentes pruebas de carácter cooperativo y competitivo. Esto, junto a todos los logros que se ha conseguido en el campo de agentes inteligentes aprendiendo a jugar a videojuegos, convierten a MarlÖ en una plataforma que puede ofrecer grandes oportunidades de experimentación en entornos multi-agente.

Por otra parte otro impulso de este proyecto es de carácter personal y viene dado por mi deseo de expandir mis conocimientos en el área de la inteligencia artificial y al mismo tiempo poner a prueba los conocimientos que he aprendido en mis años en el Grado de Ingeniería Informática. El hecho de elegir el Aprendizaje por Refuerzo como método de aprendizaje para los agentes está influenciada por el crecimiento y los éxitos que ha tenido dichos métodos en los últimos años impulsados por organizaciones como OpenAI y Google. Esto puede ayudarme a obtener experiencia en herramientas y métodos que pueden ser incluidos en el estado del arte de la informática.

1.2 Objetivos

Con este proyecto se pretende ampliar la información y comprensión actual sobre el entrenamiento de agentes inteligentes en entornos multi-agente utilizando aprendizaje por refuerzo. Esto se cumplirá observando y analizando los resultados obtenidos al realizar experimentos con algoritmos destinados a aprendizaje mono-agente en entorno multi-agente de la plataforma MARLÖ. Los objetivos que se plantean en este proyecto son los siguientes:

- Investigación, comprensión y selección de los algoritmos que se utilizarán en el proceso de experimentación.
- La modificación de las implementaciones de los algoritmos seleccionados para que adecuen a las necesidades de la plataforma MARLÖ.
- Diseño de entornos multi-agente personalizados para su uso en los experimentos.
- La realización de los experimentos necesarios y el posterior análisis de los resultados.

1.3 Estructura de la memoria

Esta memoria se compondrá de un total de 5 capítulos. En el primer capítulo se presenta la introducción, motivación y objetivos. En el segundo capítulo se encuentra el estado del arte en el que se dará una explicación sobre fundamentos del aprendizaje por refuerzo y se expondrá su estado actual. El tercer capítulo corresponde a la solución propuesta donde se explicará el funcionamiento y las razones de la elección tanto de la plataforma MARLÖ como de los algoritmos elegidos para la experimentación. El cuarto capítulo es el de experimentación y resultados, en este capítulo se expondrán los diferentes experimentos realizados y se analizarán los resultados obtenidos. Finalmente, encontraremos las conclusiones en el quinto capítulo donde se sintetizará toda la información obtenida en el proyecto.

CAPÍTULO 2
Estado del arte

En este capítulo se establecerán y explicarán los conceptos necesarios para el debido entendimiento de este proyecto. Se hablará de qué es y cómo funciona el aprendizaje por refuerzo y se presentarán diferentes métodos que serán usados en el proyecto.

2.1 Aprendizaje por refuerzo

El nombre de *reinfocement learning* (RL) o aprendizaje por refuerzo viene dado por el término reforzamiento o *reinforcement*, proveniente de la psicología[1]. Este se basa en el procedimiento en el cual un organismo altera su conducta después de verse afectado por un estímulo nervioso tanto positivo como negativo para el organismo. Un ejemplo de este procedimiento es cómo un perro puede cambiar su conducta y aprender a realizar "trucos" cuando se le suministra un estímulo positivo en forma de galleta. Como podrán ver a continuación el RL tiene mucho parecido con este procedimiento.

Una definición formal de RL sería: una aproximación a la automatización del aprendizaje dirigido por objetivos y a la toma de decisiones[1].

Antes de continuar se procederá a definir diferentes términos:

- Agente: Componente software capaz de actuar rigurosamente con el fin de realizar tareas en nombre de su usuario. Específicamente en el RL es una entidad capaz de reaccionar a su entorno, actuar de forma correcta con el fin de obtener la mayor recompensa y aprender de su experiencia[2].
- Entorno: La representación del problema a ser resuelto. Puede ser real, por ejemplo un dron que tiene que aprender a volar a una posición y tiene sensores de altitud, velocidad y GPS, o puede ser simulado como un juego de blackjack o un videojuego.
- Señal de recompensa: Valor numérico que es devuelto en cada paso que se realiza en el entorno y determina cual es el objetivo a cumplir. El agente se encargará de maximizar esa señal eligiendo las acciones correctas[1].

El aprendizaje por refuerzo se diferencia de otras aproximaciones por su énfasis en que el aprendizaje se produzca a consecuencia de la interacción de los agentes con el entorno. A diferencia de los métodos de aprendizaje supervisado que se entrenan utilizando corpus de muestras etiquetadas, el RL genera sus propias muestras con la interacción entre el agente y el entorno.

Esta interacción se repite continuamente, con el agente eligiendo una acción a realizar y el entorno reaccionando a esa acción y presentado una nueva situación al agente.

Además el entorno otorgará al agente un valor numérico o recompensa por cada una de sus acciones el cual el agente intentará maximizar. Más específicamente las interacciones se realizarán en una secuencia de pasos discretos $t = 1,2,3\ldots$ en cada paso t el agente recibirá una representación del entorno en ese paso, a la que se llamará estado s_t, usando esa representación el agente decidirá realizar una acción a, un paso después el agente recibirá la recompensa r_t correspondiente a la acción realizada y la nueva representación del entorno. En la figura 2.1 se puede observar una representación del proceso explicado anteriormente.

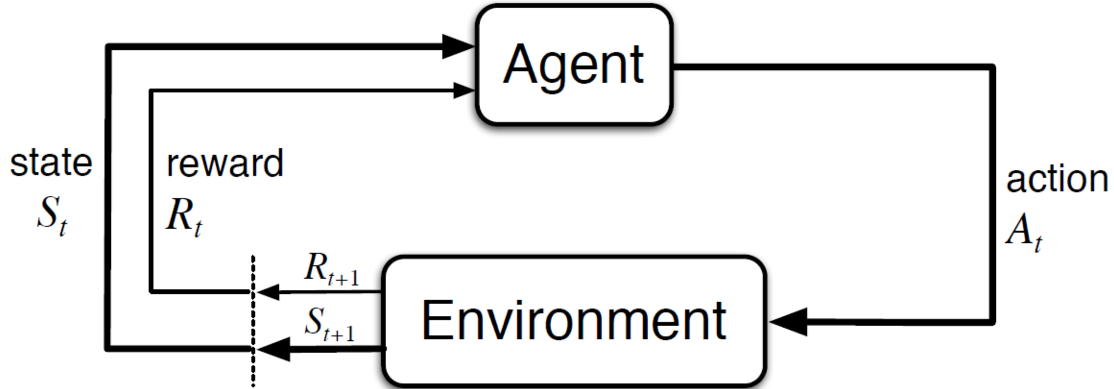


Figura 2.1: Representación de la interacción agente-entorno

El objetivo de los agentes es maximizar la suma de recompensas que le otorga el entorno, para ello utilizarán lo que se suele llamar una política π. Formalmente una política es un mapeado de los estados del entorno a probabilidades de seleccionar cada una de las acciones posibles. Por lo tanto $\pi(a|s)$ es la probabilidad de la acción a en el estado s bajo la política π. Los diferentes métodos de aprendizaje por refuerzo especificarán cómo cambiará la política según las experiencias del agente.

Hay que destacar que la recompensa que reciben en cada estado representa la recompensa momentánea del entorno, pero para que el agente pueda decidir con una mayor eficiencia sus acciones debería ser capaz de computar la recompensa a largo lazo del episodio. Por eso se considera el valor de retorno R_t en el momento t como:

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T \\ &= \sum_{i=0}^{T-t-1} \gamma^i r_{t+i+1} \end{aligned} \tag{2.1}$$

Donde la recompensa es la suma ponderada de todas las siguientes recompensas hasta el final del episodio, siendo γ la ponderación dado $0 <= y < 1$. Si el agente toma valores de y cercanos a 1 dará mucha importancia a las recompensas futuras mientras que si es cercano a 0 se centrará en las recompensas más inmediatas.

El siguiente paso es definir la *value function* que nos dice como de bueno es un estado s. Por lo tanto la *value function* $V(s)$ se define como el valor de retorno esperado de un estado s y de los posteriores a él aplicando la política π:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \tag{2.2}$$

Otra opción es la función estado-acción $Q(s,a)$ como el valor del par estado s acción a siguiendo la política π:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right] \tag{2.3}$$

Estas ecuaciones se pueden expandir a unas ecuaciones más completas:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma\, \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right], \text{ para todo } s \in S \end{aligned} \tag{2.4}$$

Una explicación más general sería que la *value function* se puede definir como el valor de todos los estados alcanzables desde el estado dado s hasta el fin del episodio, ponderándolos con tanto la probabilidad de obtener ese estado a partir de un estado s y una acción a como la probabilidad de dicha acción a dado el estado s siguiendo la política π.

Las tres ecuaciones anteriores pertenecen a las ecuaciones de Bellman, lo más importante de estas ecuaciones es que podemos calcular el valor de un estado a partir del valor de otros estados y abre la puerta a aproximaciones iterativas como la programación dinámica. La contrapartida de estos métodos es que necesitan un modelo completo y preciso del entorno, cosa que no es posible en algunos casos y en entornos que tengan una gran cantidad de acciones y estados se necesitará una gran cantidad de recursos computacionales.

La solución es el uso de otros métodos que intentan aproximar la ecuación de Bellman pero sin depender del modelo del entorno.

Los métodos de Monte Carlo aprenden las funciones de valor y políticas óptimas utilizando la experiencia del agente. Esto les permite aprender un comportamiento óptimo a través de la interacción con el entorno sin necesidad de un modelo de las dinámicas del entorno[1]. Estos métodos, en vez de usar el modelo para calcular el valor de cada estado, generan episodios repetidamente y almacenan la media de las recompensas para cada estado o para cada estado-acción.

Uno de los problemas de Monte Carlo es que necesita que todos los estados sean visitados una cierta cantidad de veces para poder asegurar que se obtiene el resultado óptimo.

Hay dos soluciones a este problema. La primera solución es seleccionar aleatoriamente el estado inicial de cada episodio, pero el elegir el estado inicial no es una opción en algunos entornos por lo que hay otro método, el agente seguirá su política pero con una probabilidad de ϵ podrá realizar una acción al azar en vez de la que su política le indicaría. Estos dos métodos se encargan de lo que en RL se llama exploración, el proceso en el que el agente intenta obtener más información del entorno para poder mejorar su política y no quedar atascado en un máximo local.

El equilibrio entre la exploración y la explotación es algo muy importante en el RL. Queremos que el agente explore para que pueda encontrar cambios en el entorno pero no tanto que perjudique a su rendimiento [1].

Una distinción que se puede realizar en los métodos de Monte Carlo es que están divididos en *on-policy* y *off-policy*. Los métodos *on-policy* utilizan su propia política π para

la evaluación y la exploración, mientras que los métodos *off-policy* utilizan una política distinta para generar los episodios.

Los métodos de *Temporal Diference* o TD son la unión de los métodos de Monte Carlo y programación dinámica por el hecho de que también utilizan la experiencia del agente para aprender pero con la diferencia de que no necesitan esperar al final de cada episodio para actualizar su política, sino que pueden realizar la actualización a cada paso pudiendo así converger más rápidamente.

Uno de los primeros algoritmos más famosos del aprendizaje por refuerzo pertenece a estos métodos. Es el denominado Q-learning [21] definido por:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{2.5}$$

Como se puede observar el algoritmo pretende aprender la función estado-acción, Q, intentado aproximar directamente la función óptima q^*, sin tener en cuenta su propia la política. Esta decisión provoca que la convergencia sea mucho más rápida[1]. El papel de la política en este algoritmo es decidir que pares de acción-estado son visitados.

A continuación se puede observar el algoritmo completo:

Algorithm 2.1 Algoritmo Q-learning

Inicializar los parámetros: factor de aprendizaje α y factor de exploración ϵ
Inicializar $Q(s, a)$ para $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, aleatoriamente excepto $Q(terminal,) = 0$
for cada episodio **do**
  Inicializar S
  for cada paso en el episodio **do**
    Elegir A a partir de S utilizando una política derivada de Q(e.g. ϵ-greedy)
    Reaizar la accion A y observar la recompensa R y el nuevo estado S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A) \right] S \leftarrow S'$
  end for
end for

2.2 DQN

Las soluciones anteriormente mencionadas están dentro del grupo de soluciones tabulares. Este grupo, como su nombre indica, basan su aprendizaje en tablas donde se almacenan las funciones de valor o las funciones de acción-estado de todos los posibles estados o pares acción-estado. Esta tabla se modifica en cada iteración del algoritmo, aproximando poco a poco la política definida por la tabla a la política óptima del entorno.

El problema de estos métodos es simple, cuanto más complicados sean los entornos y más acciones puedan realizar los agentes mayor sera el tamaño de dichas tablas.

En 2013 se desarrolló el primer algoritmo que consiguió de una manera satisfactoria aprender políticas directamente a través de un input sensorial de gran dimensionalidad usando aprendizaje por refuerzo[3]. Más específicamente dicho input eran píxeles de los fotogramas que se obtenían en diferentes juegos de Atari 2600, y el modelo que se utilizó para el aprendizaje estaba constituido por una red neuronal convolucional y una variante del algoritmo de Q-learning.

Esta fue la primera muestra de *Deep Reinforcement Learning* y de *Deep Q Network*. Nombres que se les da a las técnicas de RL cuando se utiliza una red neuronal como modelo en vez de los métodos tabulares.

Más específicamente la política en DQN la define la red neuronal que transformará los *inputs* (e.g. representación del estado) en los valores de las diferentes acciones que puede tomar el agente. Desde un punto de vista matemático la red neuronal estará parametrizada por β y esta se entrenará con el objetivo de minimizar la perdida utilizando la ecuación de Bellman:

$$\mathcal{L}(\beta) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a'|\beta') - Q(s, a|\beta)\right)^2\right] \tag{2.6}$$

Además las DQN necesitan utilizar un mecanismo denominado *replay buffer* para evitar inestabilidades debidas a la correlación que se produce entre muestras [5][4]. El *replay buffer* se utilizará para almacenar las experiencias y posteriormente, cada N pasos, se extraerán aleatoriamente experiencias con las que entrenar la red. La extracción aleatoria de las muestras elimina la correlación entre ellas lo que permite entrenar redes más estables.

El funcionamiento de este algoritmo se puede ver en el siguiente pseudocódigo:

Algorithm 2.2 Deep Q-learning con *Replay buffer*

Inicializar el *replay buffer* D a la capacidad N
Inicializar la función acción-valor Q con pesos aleatorios
for episodio = 1, M **do**
  Inicializar la secuencia $s1 = \{x_1\}$
  for $t = 1, T$ **do**
    Con probabilidad ϵ seleccionar una acción aleatoria a_t
    de lo contrario seleccionar $a_t = Q^*(s_t, a; \theta)$
    Ejecutar la acción a_t y obtener la recompensa r_t y la imagen x_{t+1}
    Definir $s_{t+1} = s_t, a_t, x_{t+1}$
    Guardar la transición (s_t, a_t, r_t, s_{t+1}) en D
    Obtener una muestra aleatoria de transiciones (s_j, a_j, r_j, s_{j+1}) del buffer D
    Definir $y_j = r_j$ solo si θ_{j+1} es terminal
    de lo contrario $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta)$
    Realizar un paso de descenso por gradiente en $(y_j - Q(s_j, aj; \theta))^2$
  end for
end for

El *Deep Reinforcement Learning* y el DQN solucionan el problema de la dimensionalidad que se producía en los métodos tabulares. Esto abre las puertas a nuevas aplicaciones del RL, sobretodo en el uso de entornos que utilizan fotogramas como *inputs*, como los videojuegos o la robótica.

2.3 Policy Gradients

Los *policy gradients* son otro tipo de métodos que se diferencian de todos los métodos anteriores que aprendían los valores de sus acciones y seguidamente decidían según los valores estimados.

Estos métodos aprenden una política parametrizada que puede seleccionar acciones sin la necesidad de las *action-value functions*. Una *value function* puede seguir siendo utilizada para el aprendizaje pero no en la toma de decisiones[1].

El aprendizaje se conseguirá actualizando los parámetros θ (e.g. una red neuronal) de una política a través del gradiente de un valor escalar $J(\theta)$ que mida el rendimiento de la política respecto a los parámetros de la política. Definiremos el valor de rendimiento como el valor del estado inicial del episodio siguiendo la política parametrizada por θ:

$$J(\theta) \doteq v_{\pi_\theta}(s_0) \tag{2.7}$$

El objetivo de estos métodos será maximizar el rendimiento, por lo que su aprendizaje aproximará el ascenso por gradiente de J:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta t)} \tag{2.8}$$

El problema es que el valor de rendimiento depende de la selección de acciones y de la distribución de estados en las cuales esas acciones son seleccionadas. El efecto de la política en la selección de acciones no es un problema pero el efecto de la política en la distribución de los estados es normalmente desconocida. Afortunadamente el teorema de *policy gradient* provee una expresión analítica que no incluye la distribución de los estados[1]:

$$\begin{aligned} \nabla J(\theta) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \\ &= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\theta) \right] \end{aligned} \tag{2.9}$$

Donde S_t representa solamente al estado actual. Si se realiza el mismo procedimiento con las acciones y se introduce A_t se obtiene la siguiente derivación:

$$\begin{aligned} \nabla J(\theta) &= \mathbb{E}_\pi \left[\sum_a \pi(a|S_t,\theta) q_\pi(S_t,a) \frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)} \right] \\ &= \mathbb{E}_\pi \left[q_\pi(S_t,A_t) \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \right] \\ &= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \right] \end{aligned} \tag{2.10}$$

Donde G_t es la recompensa. La expresión final define una cantidad que puede ser computada en cada paso cuya expectación es igual al gradiente. Si se utiliza esta expresión podemos definir como se realiza el ascenso por gradiente:

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \tag{2.11}$$

Cada incremento es proporcional al producto de la recompensa y un vector, el gradiente de la probabilidad de elegir la acción elegida dividido por la probabilidad de elegir la acción. El vector resultante es la dirección en el espacio de los parámetros que mayor aumento supone a la probabilidad de repetir la acción A_t en futuras visitas al estado S_t.

Esto supone que los parámetros se moverán en las direcciones que favorezcan a conseguir la mayor recompensa.

El desarrollo de estas ecuaciones anteriores son la base del primer algoritmo basado en *policy gradients*, REINFORCE[7].

2.4 Estado actual

El uso de videojuegos o simulaciones como entorno para la experimentación con agentes inteligentes es algo de lo más frecuente gracias a la gran cantidad de entornos don diferentes dinámicas que pueden poner a prueba a los agentes, como ViZDoom[14] basada en el famoso juego Doom. Pero el mayor referente respecto a entornos basados en videojuegos es la librería de python gym creada por OpenAI[10] que incluye tanto simulaciones como videojuegos de la consola Atari 2600. Esta librería ha sido usada en múltiples ocasiones para el testeo de algoritmos como el antes mencionado DQN o el sofisticado Rainbow[13], estos algoritmos han llegado incluso a superar con creces las capacidades humanas en estos juegos[12].

El aprendizaje por refuerzo no se ha centrado solo en juegos antiguos sino que también ha habido implementaciones en juegos modernos de ámbito profesional. SC2LE (StarCraft II Learning Environment)[15] es una plataforma desarrollada sobre el juego de Starcraft II pensada para que los agentes aprendan solo con la información que un humano tendría. Y ha sido en Starcraft II donde se ha conseguido entrenar a un agente con un nivel de habilidad que iguala a los mejores jugadores del mundo[16]. Pero la mayor hazaña hasta la fecha pertenece a OpenAI por vencer en 2019 al mejor jugador del mundo del videojuego Dota 2[17]. Todos estos logros no hacen más que reforzar el potencial del RL.

Pero uno de los problemas más actuales del RL son los sistemas multi-agente (MAS). Los MAS, gracias a la interacción entre agentes, presentan diferentes restos a los métodos RL ordinarios el más problemático es la falta de convergencia. Entrenar dos agentes concurrentemente tiene un gran impacto en la manera en la que estos aprenden. En un entorno mono-agente el agente se adapta a las dinámicas de dicho entorno, pero en un entorno multi-agente no solo debe adaptarse al entorno sino a otros agentes que estarán intentando adaptarse también. Esto provoca que las políticas de estos agentes tengan dificultades para estabilizarse.

Existen diferentes algoritmos que intentan solucionar los problemas de los MAS, algunos ejemplos pueden ser DRUQN, DLCQN o MADDPG que intentan evitar la no convergencia y DRQN que se centra en las observaciones parciales.

También se han conseguido grandes hitos en los problemas MAS, el más reciente es el caso de OpenAI que experimentó con agentes jugando al "pilla pilla" en un entorno multi-agente en 3D específicamente diseñado para ese experimento[18].

El objetivo en este experimento era observar las estrategias que desarrollaban los agentes, cómo las adaptaban a las de sus contrincantes y cómo se veía afectada la convergencia en un entorno tan complejo como ese.

Respecto a plataformas para el entrenamiento de sistemas multi-agente, existen diferentes opciones, una de ellas es Multi-Agent Particle Environment[19] dedicada a entrenar agentes que controlan partículas en un entorno 2D y utilizado en el algoritmo MADDPG, o MAgent[20] también centrada en entornos basados en partículas pero con miles de agentes en cada entorno.

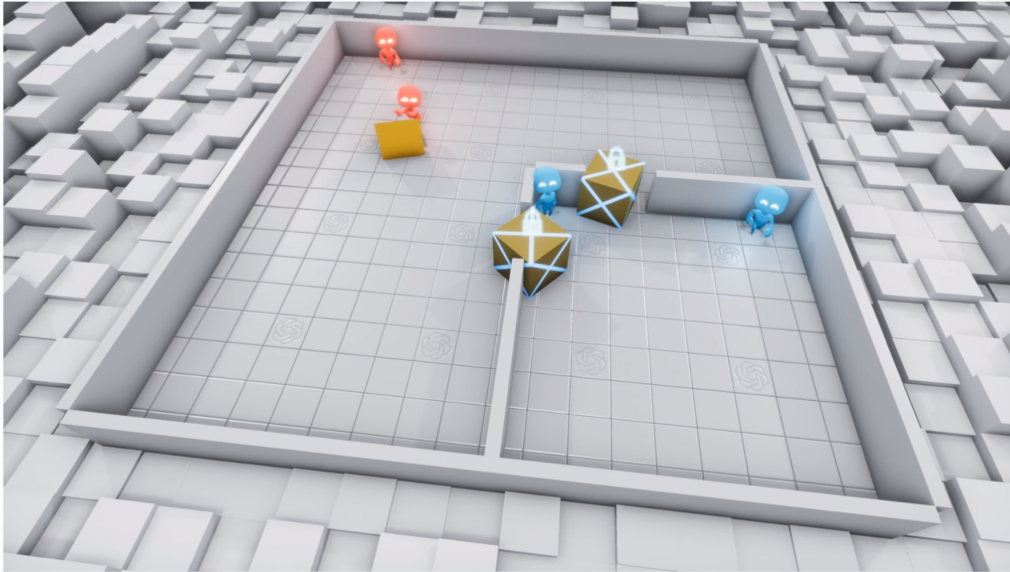


Figura 2.2: Agentes en el entorno de Hide and Seek de OpenAI

El Multi-Agent Reinforcement Learning in MalmÖ (MARLÖ) fue una competición desarrollada por CrowdAI en 2018 donde se utilizó una versión modificada de la plataforma MalmÖ para permitir el entrenamiento en entornos cooperativos y competitivos [9].

El proyecto MalmÖ ofrece una plataforma que utiliza como base el videojuego Minecraft y aspira a ofrecer un entorno que pueda soportar la experimentación en diversas áreas del aprendizaje automático[8].

Las posibilidades que ofrece MARLÖ son muy interesantes. Por un lado existe la posibilidad de poder editar los entornos de entrenamiento a placer y crear nuevos. Además se pueden utilizar observaciones más corrientes para un ser humano como son los fotogramas de la vista en primera persona que obtienen los agentes en Minecraft.

Esto hace de MARLÖ una plataforma en la que puede ser interesante experimentar y analizar con algoritmos de aprendizaje por refuerzo.

CAPÍTULO 3
Solución propuesta

Este capítulo se centrará en la explicación del funcionamiento tanto de la plataforma MARLÖ como de los algoritmos elegidos para la experimentación. Se darán las razones por las que se han elegido dichos algoritmos y la plataforma MARLÖ para la realización del proyecto. Además se comentarán las modificaciones que se tuvieron que realizar en las implementaciones de los algoritmos. Todo el código de las modificaciones mencionadas se encuentra disponible en (`https://github.com/SrGnis/tfgrepo`) para su disposición y consulta.

3.1 MARLÖ

Como se comentó anteriormente MARLÖ es una modificación de MalmÖ, una plataforma para el desarrollo de aprendizaje por refuerzo en el videojuego Minecraft.

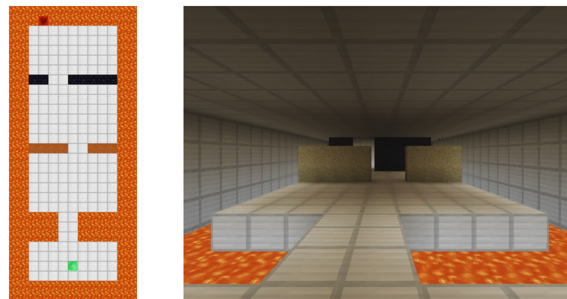


Figura 3.1: Ejemplo de un entorno de MalmÖ donde el agente tiene que conseguir llegar a la meta lo más rápido posible y sin caerse a la lava.

La elección de MarlÖ y por ende MalmÖ viene condicionada por las características que ofrece[8]:

- Esta soportada la interacción humano-agente y multi-agente.
- El entorno es complejo, diverso e interactivo gracias a la estructura del juego Minecraft.
- El entorno es dinámico y abierto, la plataforma permite una variedad de entornos y misiones infinitas.
- Nuevas tareas pueden ser creadas y modificadas a placer pudiendo adaptarse a las necesidades de cada experimento.

- La capacidad de utilizar observaciones basadas en fotogramas y no en información interna del juego.

Estas características hacen de MARLÖ una plataforma en la que se pueden realizar experimentos personalizados. Permitiendo analizar el efecto del aprendizaje multi-agente con observaciones complejas en algoritmos básicos de RL.

Al profundizar en el funcionamiento y empleo de MARLÖ podemos encontrar que MARLÖ esta compuesto de dos elementos: las instancias de Minecraft, en las cuales los agentes interaccionarán y un conjunto de librerías de python que se conectarán con estas instancias y se encargaran de la transmisión de datos entre los agentes y los entornos, que por un lado pueden ser las observaciones y recompensas desde el entorno a los agentes y por otro las acciones a realizar por cada uno de los agentes.

Un experimento básico de varios agentes en MARLÖ comenzaría con la ejecución de tantas instancias de Minecraft como agentes se necesite, especificando los puertos en los que estas instancias se comunicarán con los agentes. Seguidamente se procedería a lanzar un código en python que utilizando la librería de MARLÖ se encargará de la ejecución de los agentes.

```
import marlo
client_pool = [('127.0.0.1', 10000),('127.0.0.1', 10001)]
join_tokens = marlo.make('MarLo-MazeRunner-v0',
                         params={
                           "client_pool": client_pool,
                           "agent_names" :
                             [
                               "MarLo-Agent-0",
                               "MarLo-Agent-1"
                             ]
                         })
assert len(join_tokens) == 2
```

Listing 3.1: Inicialización de MARLÖ

La primera parte del código 3.1 se encargará de inicializar el entorno deseado especificando el nombre de este y los diferentes parámetros que se quieran incluir. Los parámetros pueden cambiar diferentes factores como la eficiencia de la plataforma, por ejemplo reduciendo o aumentado el tiempo entre pasos o priorizando la renderización en segundo plano.

```
@marlo.threaded
def run_agent(join_token):
    env = marlo.init(join_token)
    observation = env.reset()
    done = False
    count = 0
    while not done:
        _action = env.action_space.sample()
        obs, reward, done, info = env.step(_action)
    env.close()
```

Listing 3.2: Ejecución de los agentes

La segunda parte del código 3.2 ejecutará los agentes de forma concurrente. En primer lugar se iniciará y reseteará el entorno y posteriormente se seleccionarán y enviarán las

acciones de los agentes al entorno y se recibirán las observaciones, recompensas, una indicación sobre si el entorno ha terminado así como información adicional.

El código anterior ejecutará agentes con acciones aleatorias. Para que exista un proceso de aprendizaje se debe aplicar un algoritmo que procese la información recibida del entorno y seleccione las acciones que considere más eficaces.

La edición de los entornos se producirá a través de la modificación de ficheros XML. En estos ficheros se podrá modificar la estructura del entorno, especificando la posición y tipo de cada uno de los bloques que componen dicho entorno.

Por ejemplo:

```
<DrawBlock type="gold_block" x="1" y="3" z="1"/>
```

colocaría un bloque de oro en la posición (1, 3, 1). El encadenamiento de diferentes de estas lineas puede generar estructuras complejas como por ejemplo laberintos.

También se pueden cambiar los objetivos de los entornos, especificando la recompensa al realizar una acción especifica, por ejemplo si queremos que un agente reciba una recompensa al tocar un tipo de bloque en concreto usaríamos:

```
<Block behaviour="onceOnly" reward="1" type="bone_block"/>
```

De esta manera se pueden generar los entornos que uno necesite y modificarlos a placer.

3.2 DDQN

El primer algoritmo seleccionado para el proyecto es una variación del antes mencionado DQN llamada DDQN o Doble DQN. Esta variación soluciona uno de los grandes problemas que tiene el algoritmo DQN. El problema radica en que se utilizan los mismos valores para la selección como para la evaluación de una acción. Esto provoca que sea más probable seleccionar valores sobrestimados[22].

Este problema se resuelve separando la selección y la evaluación en dos redes, la red primaria Q_θ y la red objetivo $Q_{\theta'}$. La red primaria se encargará de la evaluación y la red objetivo de la selección:

$$Q^*(s_t, a_t) \approx r_t + \gamma Q(s_{t+1} \text{argmax}_{a'} Q'(s_{t+1}, a')) \tag{3.1}$$

Se minimizará el error cuadrático medio entre las dos redes y lentamente los parámetros de Q_θ se irán copiando en $Q_{\theta'}$, esto se consigue utilizando el promedio de Polyak [23]:

$$\theta \leftarrow \tau * \theta + (1 - \tau) * \theta' \tag{3.2}$$

La elección de este algoritmo se debe básicamente a dos factores.

El primero de ellos es la incapacidad de utilizar un método tabular junto a las observaciones basadas en fotogramas que aporta MARLÖ. Debido a que los requisitos de memoria en los métodos tabulares para tener todos los posibles fotogramas en una tabla sería inmenso e inviable.

El segundo es la popularidad de este algoritmo, dentro de la familia del *Deep Reinforcement Learnign* existen una gran variedad de algoritmos derivados del DQN que mejoran el rendimiento y evitan problemas. Pero ya que el objetivo de este proyecto es observar el efecto de los entornos multi-agente en algoritmos de RL se ha elegido el algoritmo más

popular y con los refinamientos necesarios para que el proceso de aprendizaje solo se vea afectado por los problemas derivados del entorno.

Una representación del algoritmo del DDQN puede verse en el siguiente pseudocódigo:

Algorithm 3.1 Double Q-learning

Inicializar la red primaria Q_θ, red objetivo $Q_{\theta'}$y buffer de experiencias D
for cada iteración **do**
  for cada paso en el entorno **do**
    Observar el estado s_t y seleccionar $a_T \sim \pi(a_t, s_t)$
    Ejecutar a_T y observar el siguiente estado s_{t+1} y la recompensa $r_t = R(s_t, a_t)$
    Guardar (s_t, a_t, r_t, s_{t+1}) en el buffer de experiencias D
  end for
  for cada paso de actualización **do**
    Obtener muestras $e_t = (s_t, a_t, r_t, s_{t+1}) \sim D$
    Computar el valor objetivo de Q:
      $Q^*(s_t, a_t) \approx r_t + \gamma Q(s_{t+1} \text{argmax}_{a'} Q'(s_{t+1}, a'))$
    Realizar un paso de descenso por gradiente en $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Actualizar los parámetros de la red objetivo:
      $\theta \leftarrow \tau * \theta + (1 - \tau) * \theta'$
  end for
end for

Respecto a la implementación de este algoritmo se utilizará una implementación de DDQN utilizada en 2019 para la experimentación en entornos mono-agente de MalmÖ [11], la cual será modificada para que sea posible utilizarla en los entornos multi-agente de MARLÖ.

Las modificaciones se basarán en adaptar las observaciones obtenidas por MARLÖ al formato requerido por el algoritmo y convertir tanto la fase de ejecución del entorno como la fase de aprendizaje de los agentes en una fase que se pueda realizar de forma concurrente. Obteniendo así dos agentes que aprenderán al mismo tiempo utilizando una DDQN distinta para cada uno.

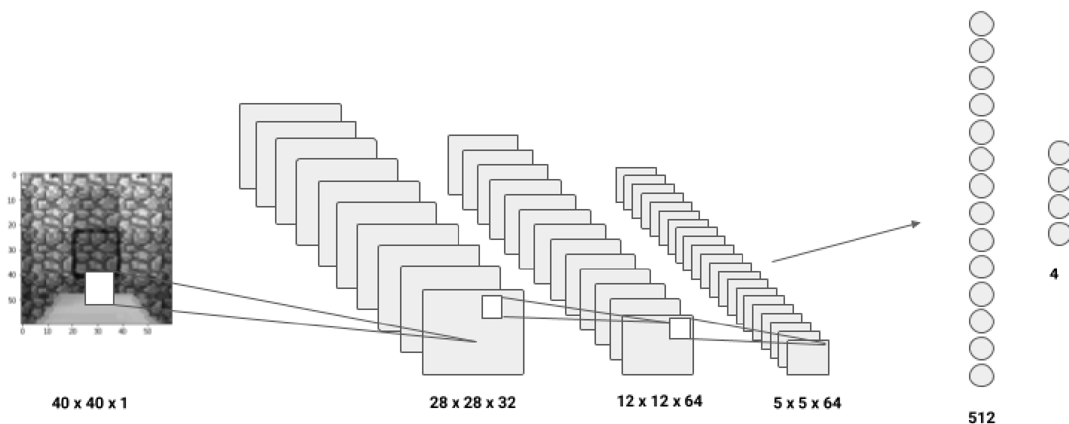


Figura 3.2: Modelo de la red utilizada en los algoritmos

La red neuronal de los agentes 3.2 a la cual se le suministrará un fotograma en escala de grises estará compuesta primeramente de una red convolucional de tres capas seguida

de una capa oculta *feedforward* de 512 nodos y finalmente la capa de salida. La salida de esta red será el valor esperado de tomar cada una de las acciones posibles en el fotograma suministrado.

3.3 TRPO

El segundo algoritmo elegido para la la experimentación es el algoritmo TRPO o Trust Region Policy Optimization.

TRPO o Trust Region Policy Optimization es un algoritmo de *reinforcement learning* basado en *policy gradient* que ha demostrado un buen rendimiento en varias tareas como jugar a juegos de Atari con imágenes como input y en la simulación de modelos robóticos caminando y a nado[6].

Pero en contraposición de los típicos algoritmos ”Natural Policy Gradient”, éste evita dar pasos demasiado grandes en el gradiente, evitando así acabar en una política peor a la inicial. Para decidir cómo de grandes son los pasos se calcula la distancia entre las políticas utilizando la Divergencia KL[24], una medida de la distancia entre dos distribuciones probabilísticas. De esta manera, el algoritmo TRPO consigue garantizar un aumento monótono del valor de la política.

Uno de los problemas de TRPO, es que al ser métodos ”on policy” no se pueden reutilizar las experiencias generadas en siguientes iteraciones del algoritmo, teniendo que generar nuevas experiencias cada vez que se quiere realizar una nueva iteración. Una solución para esto es utilizar múltiples entornos paralelos para generar la mayor cantidad de experiencias en el menor tiempo posible.

Algorithm 3.2 TRPO

for cada iteración **do**
  Ejecutar la política durante T pasos o N episodios
  Estimar la función de ventaja en todos los pasos
  for $t = 1, T$ **do**
    maximizar $\sum_{n=1}^{N} \frac{\pi_\theta(a_n|s_n)}{\pi_{\theta_{old}}(a_n|s_n)} A_n$
    s.t. $\overline{KL}_{\pi_{\theta_{old}}}(\pi_\theta) < \delta$
  end for
end for

La idea básica del algoritmo TRPO se puede ver en el algoritmo 3.2, el cual no es más que un *policy gradient* que controla cómo de grandes son sus pasos en el gradiente. La función de ventaja anteriormente mencionada en el algoritmo es un método que se utiliza para reducir la varianza en el entrenamiento:

$$A(s,a) = Q(s,a) - V(s,a) \tag{3.3}$$

Algorithm 3.3 Implementación de TRPO

1: Inicializar los parámetros de la política θ_0, y de la función de valor ϕ_0
2: Hiperparametros: limite de la divergencia-KL δ, coeficiente de *backtraking* α, numero máximo de pasos de *backtraking* K
3: **for** $k = 0,1,2,...$ **do**
4: Obtener un conjunto de trayectorias $D_k = \{\tau_i\}$ utilizando la política $\pi_k = \pi(\theta_k)$
5: Computar el conjunto de recompensas R_t
6: Computar la estimación de la ventaja, A_t
7: Estimar el gradiente de la política como

$$g_k = \frac{1}{D_k} \sum_{\tau \in D_k} \sum_{t=0}^{T} \nabla_\theta log \pi_\theta(a_t|s_t)|_{\theta_k} A_t$$

8: Usar el algoritmo del gradiente conjugado para computar

$$x_k = H_k^{-1} g_k$$

9: Actualizar la política con

$$\theta_{k+1} = \theta_k + \alpha^j \sqrt{\frac{2\delta}{x_k^T H_k x_k}} x_k$$

donde $j \in \{0, 1, 2, ..., K\}$ es el mínimo valor que mejora la perdida y cumple con la restricción de la divergencia-KL
10: Ajustar la función de valor usando el error cuadrático medio:

$$\phi_{k+1} = argmin_\phi \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0} (V_\phi(s_t) - R_t)^2$$

11: **end for**

Al utilizar en la experimentación un algoritmo como el DDQN, se puede obtener información interesante de la comparación que se puede realizar con el TRPO, el cual es un algoritmo algo más avanzado y perteneciente al campo de los *policy gradients*.

La elección de TRPO viene dada a la necesidad de elegir un algoritmo que corresponda a un término medio en los *policy gradients*, no tan simple como REINFORCE pero tampoco un algoritmo demasiado complejo como pueda ser PPO[25].

La implementación que se ha utilizado en este caso ha sido diseñada para ser utilizada en el entorno de OpenAi Gym, más específicamente en el entorno basado en el videojuego Pong, el cual también ofrece como observaciones fotogramas.

Dicha implementación se ha modificado para poder utilizar los entornos de MARLÖ y sus observaciones. En este caso solo la fase de ejecución del entorno ha tenido que ser modificada para permitir la concurrencia. Esto es debido a que el algoritmo se puede dividir en dos fases: (1) Obtener experiencia generando una muestra de X episodios y (2) utilizar la experiencia anterior para entrenar una sola red neuronal que controlará a los dos agentes.

El hecho de que esta implementación utilice una sola red neuronal para los dos agentes, al contrario de como ocurre en el DDQN, añade más variedad a la comparación de estos algoritmos.

Respecto al modelo de red neuronal que se empleará en el TRPO, ésta será la misma que en el DDQN 3.2, es decir, una red convolucional de 3 capas seguida de una capa oculta feedforward.

3.4 MADDPG

Por último, también se realizará una experimentación con el algoritmo MADDPG. Este es un algoritmo perteneciente a la familia de los *policy gradients*, con un diseño enfocado en entornos multi-agente, especialmente para el entorno Multi-Agent Particle Environment basado en partículas que se mueven en un entorno 2D y que deben conseguir diferentes objetivos cooperativos y competitivos.

Este algoritmo se basa en tres restricciones auto impuestas: **(1)** Las políticas aprendidas solo pueden utilizar información local en el momento de ejecución (pero pueden utilizar información extra en el entrenamiento con el objetivo de agilizarlo), **(2)** no asumen unas dinámicas del entorno diferenciables y **(3)** no se asume ningún método de comunicación entre los agentes.

Esto genera un algoritmo de propósito general que puede funcionar en entornos competitivos y cooperativos. Para alcanzar estos objetivos se usa un framework de entrenamiento centralizado, ejecución distribuida e inferencia de las políticas de otros agentes[19].

El pseudo-código correspondiente a este algoritmo es el siguiente:

Algorithm 3.4 Algoritmo MADDPG

for cada episodio **do**
  Inicializar un proceso $\mathcal{N}$ para la exploración
  Obtener el estado inicial x
  for cada paso del episodio **do**
    Para cada agente i seleccionar la accion $a_i = \mu_{\theta i}(o_i) + \mathcal{N}_t$
    Ejecutar las acciones $a = (a_1, ..., a_N)$ y observar la recompensa r y el nuevo estado x'
    Almacenar (x, a, r, x') en el replay buffer D
    $x \leftarrow x'$
    for agente $i = 1$ a N **do**
      Obtener S muestras aleatorias (x^j, a^j, r^j, x'^j) de D
      Definir $y^j = r_i^j + \gamma Q i^{\mu'}(x'^j, a'_1, ..., a'_N)|_{a'_k = \mu'_k(oj^k)}$
      Actualizar el critico minimizando la perdida:
        $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\mu}(x^j, a_1^j, ..., a_N^j)\right)^2$
      Actualizar el actor utilizando el gradiente
        $\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\mu_i(o_i^j)\nabla_{a_i} Q_i^{\mu}(x^j, a_1^j, ..., ai, ..., a_N^j)|_{a_i = \mu_i(o_i^j)}$
    end for
    Actualizar los parámetros de la red objetivo para cada agente i:
      $\theta'_i \leftarrow \tau\theta_i + (1 - \tau) * \theta'_i$
  end for
end for

La elección de este algoritmo está basada en el hecho de querer comparar los algoritmos mono-agente anteriores con un algoritmo especialmente diseñado para entornos multi-agente. Esto nos puede ayudar a observar las diferencias que se producen en el aprendizaje entre estos algoritmos.

La implementación de este algoritmo también ha tenido que ser modificada como en los algoritmos anteriores. Se debe recalcar que esta implementación del algoritmo está especialmente diseñada para funcionar con el entorno con el que se testeó Multi-Agent Particle Environment. En este caso los agentes decidirán sus acciones de forma secuencial para posteriormente de forma concurrente ejecutar dichas acciones. Esto se debe a las peculiaridades de la implementación del algoritmo.

Por otra parte también se han tenido que realizar modificaciones en el propio MARLÖ para permitir que las observaciones suministradas por la plataforma fueran las posiciones de los agentes y no los fotogramas. Esto se debe a que MADDPG solo acepta observaciones unidimensionales y los fotogramas no pueden ser procesados.

3.5 Vista general

Por último, recopilando todo lo expuesto en las secciones anteriores, se procede a sintetizar toda la información en un resumen para proporcionar una visión general del procedimiento que se va a seguir.

La plataforma seleccionada para la experimentación es MARLÖ. Con esta plataforma se crearán diferentes entornos multi-agente personalizados. En estos entornos los agentes deberán aprender a cumplir diferentes objetivos tanto de carácter cooperativo como competitivo. El aprendizaje de estos agentes estará dirigido por los algoritmos seleccionados. Este proceso de aprendizaje será analizado para la obtención de conclusiones.

Los algoritmos seleccionados son:

- **DDQN**: Mono-agente, Deep Reinforcement Learning, una red neuronal por agente.
- **TRPO**: Mono-agetne, Policy Gradient, una red para los dos agentes.
- **MADDPG**: Multi-agetne, Policy Gradiet, una red neuronal por agente pero con información compartida.

Las implementaciones de los algoritmos han tenido que sufrir modificaciones tanto para permitir que los entornos de MARLÖ funcionen correctamente como para adaptarlos a los requisitos de dichas implementaciones.

CAPÍTULO 4
Experimentación y Resultados

El entorno para experimentos en el que nos hemos querido centrar en este trabajo es el juego conocido en teoría de juegos como "Stag Hunt" o caza del ciervo en español. Es un juego cooperativo descrito por Jean-Jacques Rousseau, que se basa en la idea de que dos cazadores han encontrado un gran ciervo y un conejo. Solo si los dos cazadores trabajan juntos podrán cazar el ciervo y obtendrán un gran recompensa para los dos, pero también pueden decidir no ir tras el ciervo y cazar al conejo, que será una recompensa menor, puesto que únicamente es necesario un cazador para conseguirla.

Se debe tener en cuenta además que si uno caza al conejo y el otro al ciervo, éste último no se llevará ninguna recompensa y que si los dos quieren cazar al conejo la recompensa será aun menor ya que tendrán que compartirla.

En la tabla 4.1 se puede observar una representación tabular del juego:

	Ciervo	Conejo
Ciervo	4,4	0,2
Conejo	2,0	1,1

Tabla 4.1: Representación tabular de Stag Hunt.

Lo que hace interesante a este juego es que la mejor estrategia es que los dos cacen al ciervo pero es mucho más fácil que decidan no cooperar y cacen al conejo.

Respecto a la implementación de este juego en MARLÖ, se coloca a dos agentes y un animal en un espacio cerrado, por lo tanto para cazar el animal tienen que rodearlo físicamente. También se habrá colocado en una parte de ese espacio un bloque que simulará al conejo. En el momento de que uno de los agentes toque dicho bloque la partida finalizará dando la recompensa a ese agente.

Como se puede intuir, en esta situación los dos agentes no pueden cazar al conejo ya que en el momento que uno se adelante al otro el juego termina.

Una representación básica de Stag Hunt en Minecraft estaría compuesta por un recinto cerrado por vallas con un ancho y largo de 3 bloques y en cada uno de sus vértices se colocarían a los agentes o los animales.

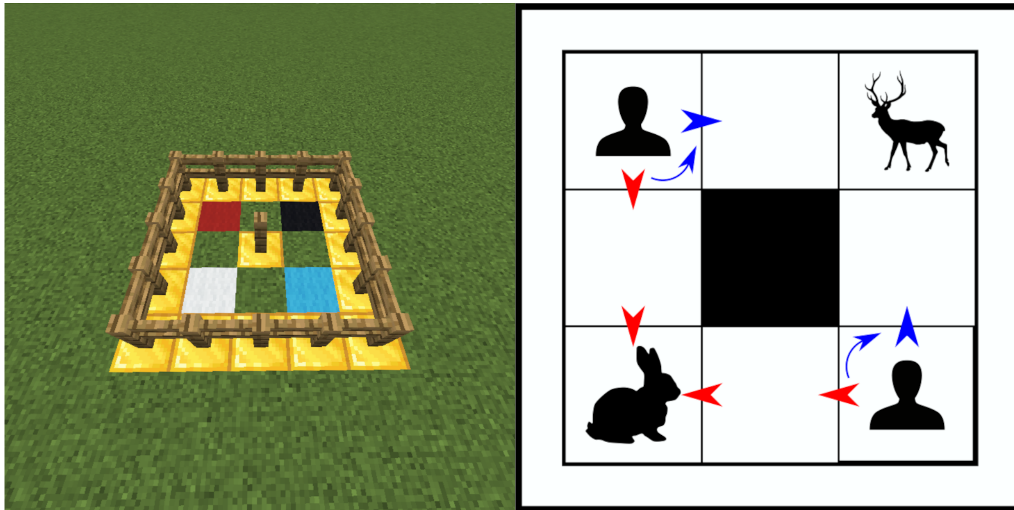


Figura 4.1: Representación de Stag Hunt en Minecraft en un recinto de 3x3

En la figura 4.1 los bloques blanco y negro representan al conejo y el ciervo y los bloques azul y blanco a los agentes. Los agentes estarían colocados mirando hacia el conejo para que así las acciones necesarias para cazar al conejo y al ciervo fueran las mismas. De este modo, la solución para cazar al conejo sería dar dos pasos hacia adelante, mientras que para cazar al ciervo se ha de realizar un giro a la derecha seguido de un paso hacia delante, siempre y cuando el otro agente esté en la posición correcta para encerrar al ciervo.

Las recompensas que los agentes pueden obtener son cuatro diferentes:

- Cazar al ciervo, se traducirá como una recompensa de 1 punto para los dos agentes.
- Cazar al conejo, se traducirá como una recompensa de 0.2 puntos para el agente que lo haya conseguido.
- Realizar una acción, se traducirá como una penalización de -0.02 puntos. Esto provoca que los agentes aprendan a cumplir sus objetivos lo más rápido posible ya que así evitan obtener penalizaciones.
- Alcanzar el número máximo de acciones realizadas en un episodio; se traducirá en un penalización de -0.2.

Por último, las acciones que los agentes podrán realizar estarán reducidas a solo girar tanto a la izquierda como a la derecha y dar un paso hacia delante o hacia atrás.

El número de pasos óptimo para conseguir cazar a uno de los dos animales sería igual a 2, caminar hacia delante dos veces para cazar al conejo o girarse y caminar hacia el ciervo.

La anterior representación, así como la definición de las dinámicas del entorno, se especificarán en el mismo fichero XML. Un ejemplo de la configuración de la recompensas del entorno se puede ver en el siguiente fragmento de texto:

```
<RewardForSendingCommand reward="-0.02"/>
<RewardForTouchingBlockType>
  <Block behaviour="onceOnly" reward="0.2" type="bone_block"/>
</RewardForTouchingBlockType>
<RewardForMissionEnd>
  <Reward description="command_quota_reached" reward="-0.2" />
</RewardForMissionEnd>
<RewardForCatchingMob>
    <Mob type="Cow" reward="1" distribution="Agent0:1 Agent1:1" oneshot="
        true" global="true"/>
</RewardForCatchingMob>
```

Listing 4.1: Ejemplo de definición de las dinámicas del entorno en MARLÖ

Como segundo entorno en el que se realizarán los experimentos, se ha escalado el entorno 4.1 de un recinto con unas dimensiones 3x3 a un recinto de dimensiones 5x5. Esta escala hace que el número de pasos necesarios para que los agentes cacen al conejo o al ciervo sea de 4. Aunque solo sean dos pasos más esto reduce significativamente las probabilidades de que los agentes consigan cazar al ciervo de forma aleatoria, como se podrá observar en los próximos experimentos.

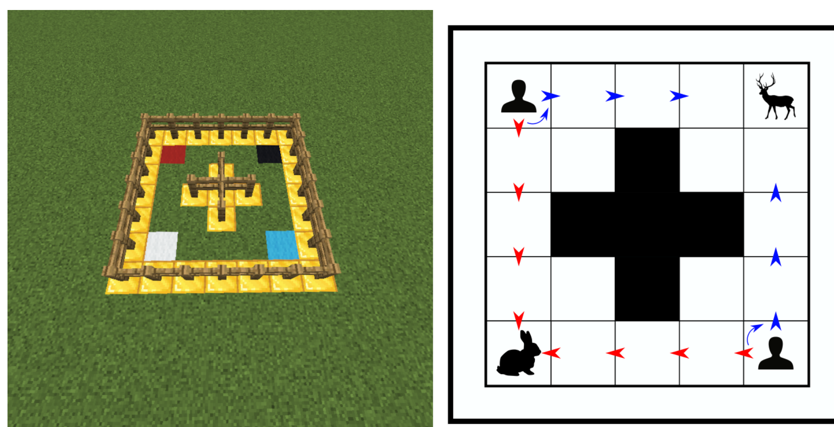


Figura 4.2: Representación de Stag Hunt en Minecraft en un recinto de 5x5

4.1 Experimentación con el algoritmo DDQN

El proceso de experimentación comenzará con el algoritmo DDQN. Este se utilizará en los dos entornos anteriormente mencionados y se obtendrán los resultados del proceso de aprendizaje para su posterior análisis.

4.1.1. Entorno 3x3 con DDQN

Los experimentos con el algoritmo DDQN empezaron utilizando un entorno de MARLÖ de dimensiones 3x3 que se puede observar en la figura 4.1. Se realizaron un total de 10000 episodios, cada uno de ellos con una duración máxima de 10 pasos. Se ha elegido 10 como el número máximo de pasos con la intención de que los agentes tengan suficiente holgura en los episodios para poder explorar. Siendo 2 el número óptimo de pasos para completar el episodio, 10 es un numero que ofrece a los agentes esa holgura anteriormente comentada sin extender demasiado los episodios.

Los 10000 episodios de entrenamiento se dividen en tres partes:

- Los 2000 primeros se utilizarán para llenar parcialmente el buffer de experiencias. En estos episodios los agentes realizarán acciones de forma totalmente aleatoria sin seguir ninguna política. Esto ofrecerá una gran exploración con la que los agentes pueden empezar a entrenarse.
- En los 7000 episodios siguientes el valor de épsilon, que regula la exploración indicando la probabilidad de tomar una acción aleatoria, se irá reduciendo desde el valor de 1.0 en el episodio 2000 hasta el valor 0.1 en el episodio 9000. Con esto se espera reducir la exploración y fomentar la explotación en los episodios más avanzados donde el agente ya tendrá una DDQN más eficiente que al principio.
- En los últimos mil episodios el valor de la épsilon se mantendrá constante. Aquí se busca la estabilización de las DDQN al ofrecerles la posibilidad de explotar al máximo sus políticas y por tanto refinarlas.

La ejecución de este experimento tiene una duración aproximada de alrededor de 15 horas. La ejecución de 10 episodios de 10 pasos tendría una duración de 1 minuto. Hay que tener en cuenta que según los agentes vayan aprendiendo los episodios se irán reduciendo en duración ya que los agentes no necesitarán los 10 pasos para terminarlos.

Una vez terminada la ejecución el dato que nos puede ilustrar con mayor facilidad si los agentes han aprendido a cooperar o no es observar la evolución de la suma de las recompensas de cada episodio.

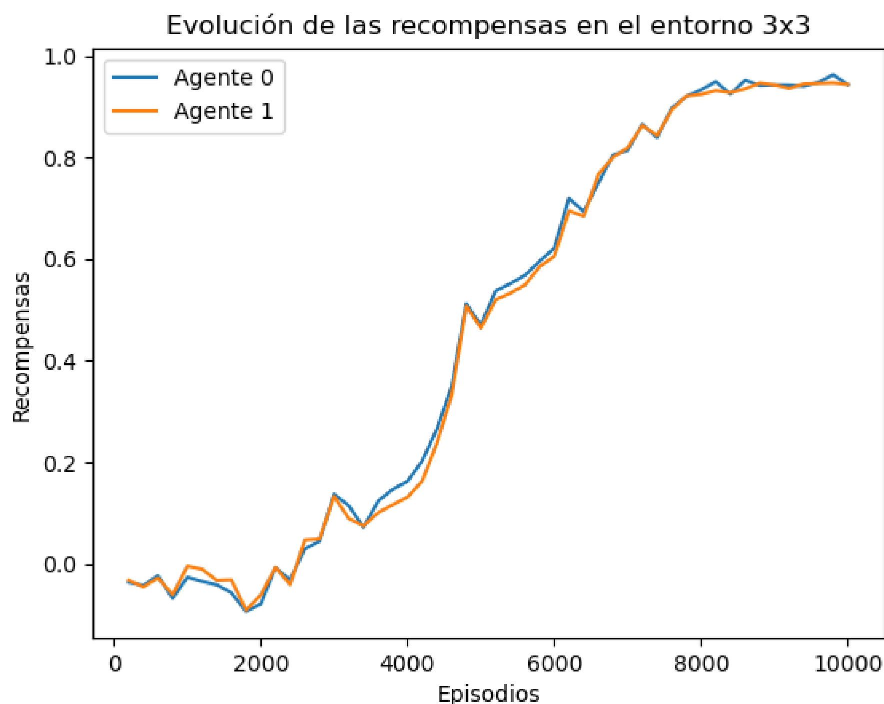


Figura 4.3: Recompensas en el entorno 3x3 usando DDQN

Observando la figura 4.3 se puede ver como en los primeros 2000 pasos tienen recompensas muy bajas debido a que se eligen las acciones de forma aleatoria. Pero una vez que el entrenamiento comienza se puede ver como los agentes empiezan a aprender y las recompensas que van obteniendo son cada vez mayores hasta alrededor de los 8000

episodios donde el aprendizaje se estabiliza prácticamente en el valor máximo de este entorno.

Con estos datos se puede confirmar que los agentes han conseguido aprender a colaborar para cazar al ciervo.

Aun así, analizando los demás datos obtenidos podemos ver la evolución de este proceso de aprendizaje. Uno de estos datos es la evolución de la distribución de los finales de los episodios. Pudiendo mostrar cómo las estrategias de los agentes han ido cambiado según aprendían.

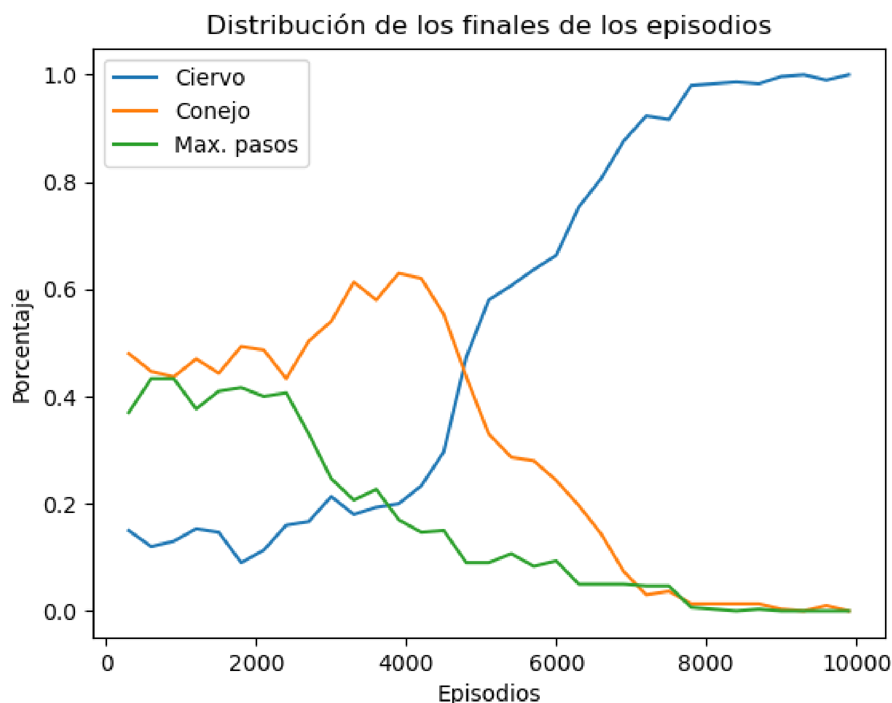


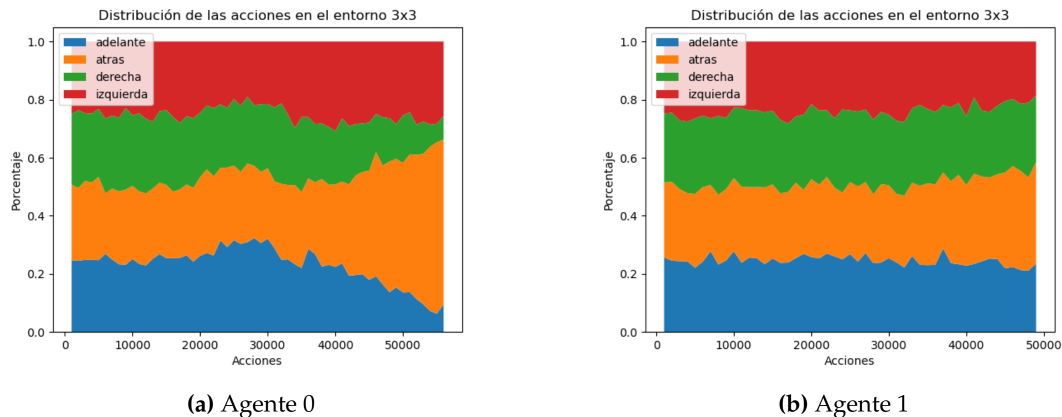
Figura 4.4: Finales de los episodios en el entorno 3x3 usando DDQN

Lo primero que podemos ver en la figura 4.4, centrándonos específicamente en los 2000 primeros episodios, es la diferencia entre las distribuciones de los finales. Hay que tener en cuenta que tanto para cazar al conejo como al ciervo solo se necesitan 2 pasos, pero cazar al conejo es el doble de común que cazar al ciervo. Esta diferencia se debe a que para cazar al conejo solo se necesita a un agente mientras que para cazar al ciervo los dos agentes deben de colocarse en posición al mismo tiempo actuando de forma aleatoria.

Centrándonos en las estrategias desarrolladas por los agentes se puede ver que lo primero que aprenden es a cazar al conejo, representándose como el aumento del porcentaje entre los episodios 2000 y 4000, pero poco después los agentes empiezan a aprender a cooperar para cazar al ciervo y terminan por converger en dicha estrategia.

Por último, si visualizamos la evolución de la distribución del tipo de acciones que los agentes deciden realizar, podremos saber más a fondo cómo llevan acabo los agentes sus estrategias.

Si observamos la gráfica correspondiente al Agente 0 en la figura 4.5, se puede ver que en las primeras 20000 acciones, las cuales pertenecen a la fase de exploración inicial, se eligen de manera aleatoria ya que todas tienen prácticamente la misma distribución. Sin embargo, una vez que el aprendizaje comienza ciertas acciones empiezan a priorizarse más que otras, como es el caso de las acciones de movimiento atrás e izquierda.



(a) Agente 0

(b) Agente 1

Figura 4.5: Acciones de los agentes en el entorno 3x3 DDQN

Sabiendo que los pasos necesarios para que los agentes cacen al ciervo son girarse y avanzar hacia el ciervo, podemos deducir que la estrategia desarrollada por el agente 0 para alcanzar al ciervo es girar a la izquierda y caminar hacia atrás.

Pero si se observa la gráfica del Agente 1 no se puede observar la periodización de acciones que podemos ver en el Agente 0. Esto puede llevar a pensar que este agente no ha aprendido ninguna estrategia pero los resultados observados en las figuras 4.3 y 4.4 nos demuestran que la cooperación existe entre los dos agentes.

Si echamos un vistazo a las trazas de los episodios podemos ver que el Agente 1 sigue dos estrategias, la primera es girar a la derecha e ir hacia atrás y la segunda girar a la izquierda e ir hacia adelante.

```
Agent0 accion:izq reward:-0.02  done: False
Agent1 accion:der reward:-0.02  done: False
Agent0 accion:atr reward:-0.02 done: False
INFO:marlo.base_env_builder:Mission ended: caught_the_Chicken
Agent1 accion:atr reward:0.98  done: True
Agent0 accion:atr reward:1.0  done: True

Agent0 accion:izq reward:-0.02  done: False
Agent1 accion:izq reward:-0.02  done: False
Agent0 accion:atr reward:-0.02 done: False
INFO:marlo.base_env_builder:Mission ended: caught_the_Chicken
Agent1 accion:ade reward:0.98  done: True
Agent0 accion:atr reward:1.0  done: True
```

Estas dos estrategias son equivalentes ya que requieren del mismo número de pasos. El hecho de que un agente elija una u otra viene dada por el azar.

4.1.2. Entorno 5x5 con DDQN

Tras los resultados del entorno 3x3 se procede a la realización de otro experimento con el entorno 5x5. En este experimento se utilizaron los mismos parámetros y el mismo número de episodios distribuidos de la misma manera. El único cambio es la substitución del entorno 3x3 por el entorno 5x5 en el cual el máximo de pasos por episodio es 30.

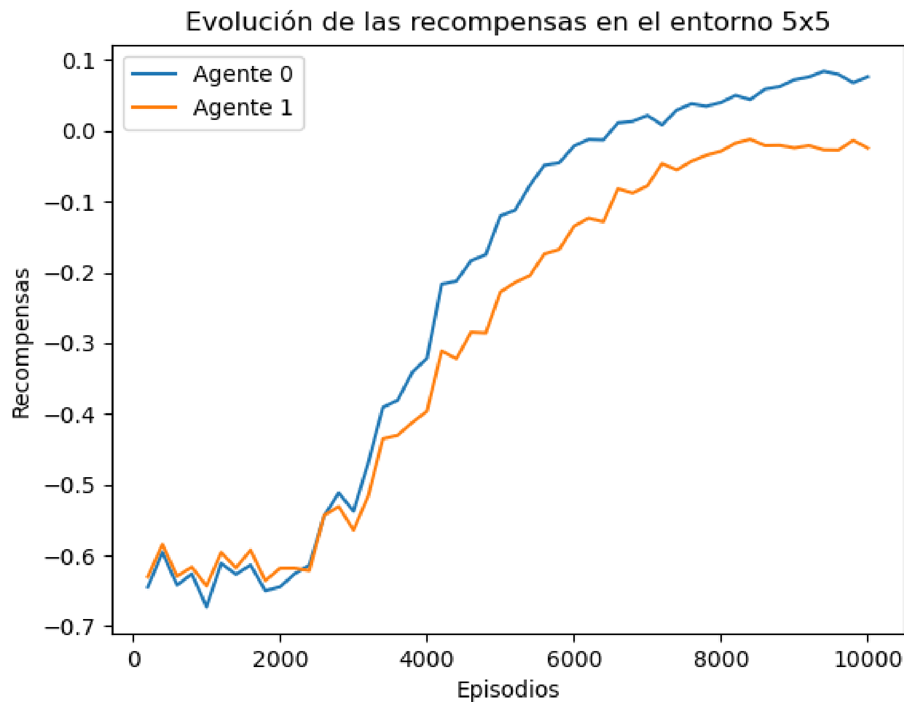
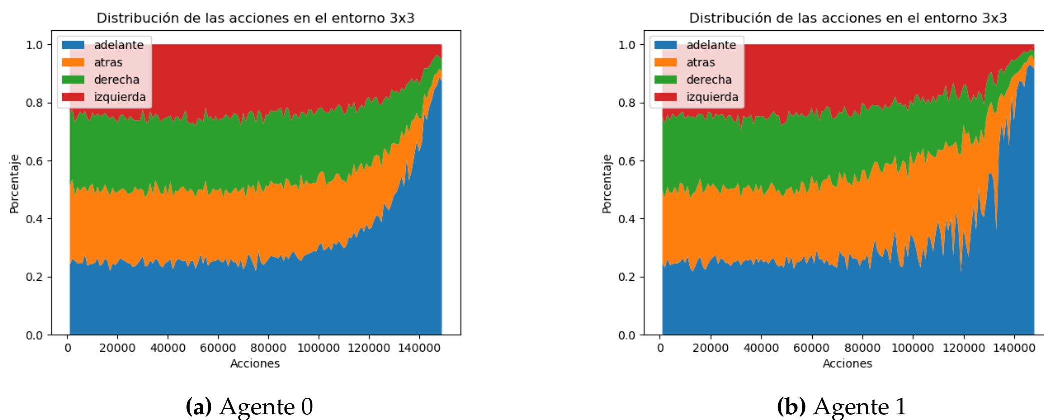


Figura 4.6: Recompensas en el entorno 5x5 usando DDQN

Como se puede observar en la figura 4.6 en este entorno los agentes no han sido capaces de aprender a cooperar y sus estrategias han convergido en intentar atrapar al conejo.

La diferencia entre las recompensas entre los dos agentes es debido a que el Agente 0 es el primero que realiza la acción dándole ventaja sobre el Agente 1.

Por último, se debe tener en cuenta que el hecho por el que las recompensas han tenido tal crecimiento viene dado porque los agentes han aprendido a terminar los episodios con menos pasos, por lo que no obtienen tantas penalizaciones.



(a) Agente 0 **(b)** Agente 1

Figura 4.7: Acciones de los agentes en el entorno 5x5 DDQN

En la figura 4.7 también se puede observar cómo los agentes, tras empezar con acciones aleatorias, desarrollan la estrategia de solo moverse hacia adelante siendo ésta la manera más óptima de cazar al conejo.

Si se analizan los datos de la distribución de los finales de los episodios 4.8 y nos centramos en los episodios que terminan cazando al ciervo podemos ver que el porcentaje de dichos finales es muy inferior a los que finalizan cazando al conejo. Al haber tan pocos episodios que hayan finalizado con la caza del ciervo se dificulta el aprendizaje de los agentes respecto a las estrategias cooperativas.

El uso de un mayor número de episodios de exploración seguida de una reducción más lenta del parámetro épsilon puede que genere muestras con suficiente exploración para que los agentes aprendan las estrategias cooperativas.

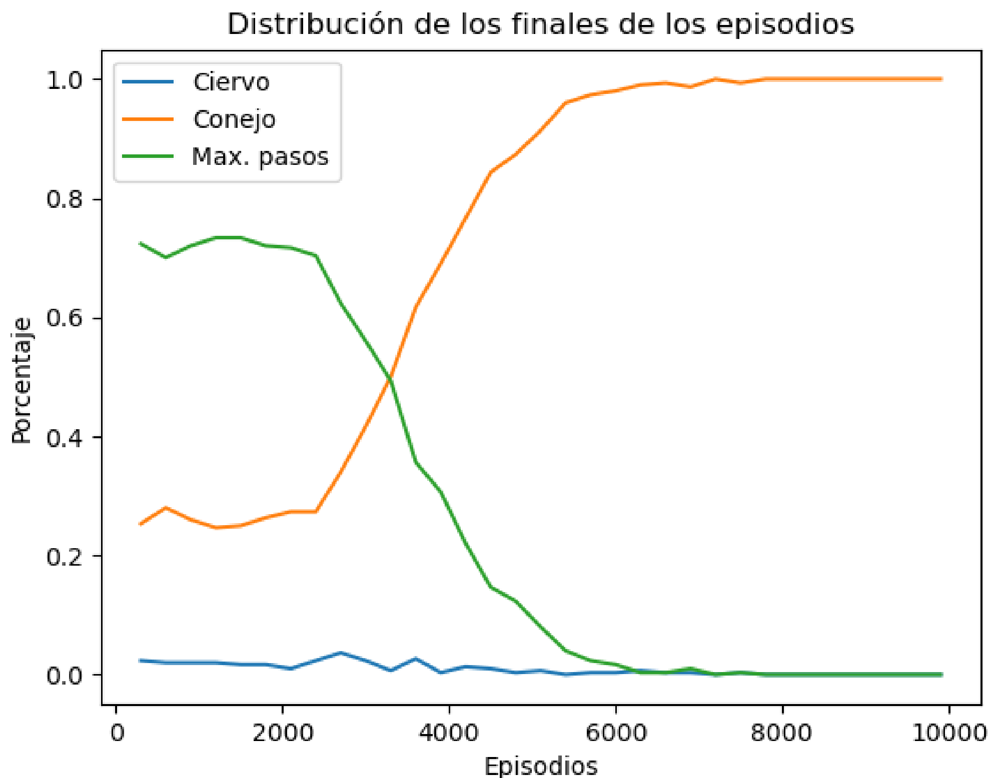


Figura 4.8: Finales de los episodios en el entorno 5x5 usando DDQN

Por último, con el fin de esclarecer si los problemas encontrados en el entorno 5x5 pueden ser solucionados con la modificación de los parámetros de entrenamiento, se volvió a repetir el experimento en el entorno 5x5. Esta vez se adaptaron los parámetros relacionados con la exploración con el fin otorgar al algoritmo una fase de exploración más extensa.

Este experimento abarcó un total de 20000 episodios. Los 5000 primeros se utilizaron para obtener experiencia previa y llenar el *replay buffer* con acciones aleatorias. Posteriormente, se utilizaron 10000 episodios para la fase de exploración-explotación donde ϵ descendió periódicamente desde el valor 1 a 0.1. Finalmente se utilizaron 5000 episodios al final del entrenamiento para que el algoritmo se estabilizara.

Pese a las modificaciones en los parámetros los resultados obtenidos (figuras 4.9 y 4.10) fueron similares al experimento anterior. Los agentes sólo consiguieron aprender a cazar al conejo y los finales en los que se consiguió capturar al ciervo fueron insuficientes para el desarrollo de las estrategias cooperativas.

Por otra parte en 4.10 se puede ver como, pese a que el Agente 1 es el primero en aprender a cazar al conejo, cuando el Agente 0 consigue aprender también cómo capturar al conejo éste obtiene ventaja al ser el primero en realizar las acciones.

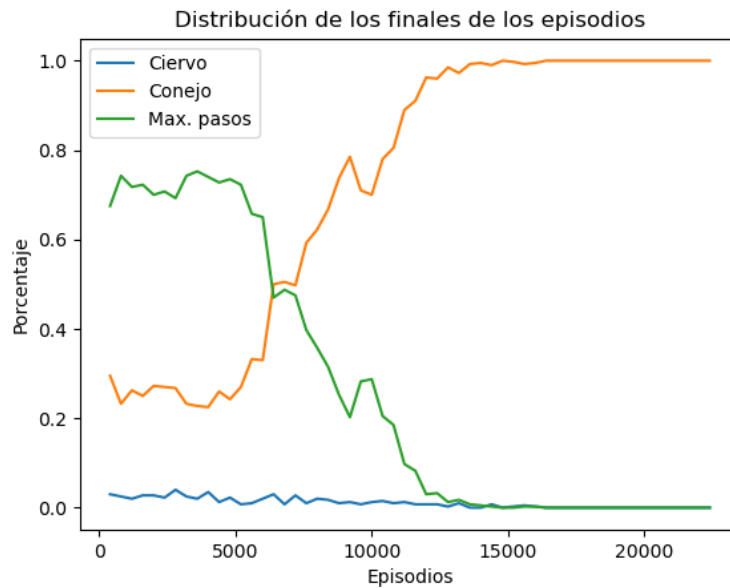


Figura 4.9: Finales de los episodios en el entorno 5x5 usando DDQN con parámetros modificados

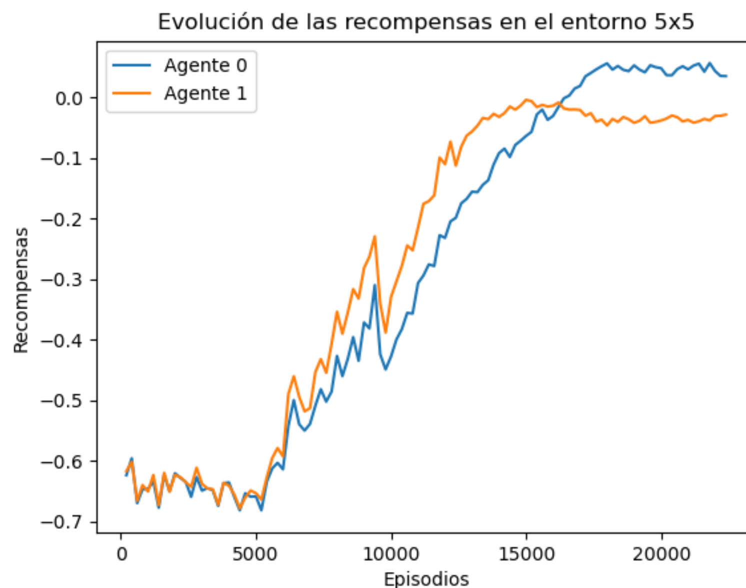


Figura 4.10: Recompensas en el entorno 5x5 usando DDQN con parámetros modificados

4.2 Experimentación con el algoritmo TRPO

Una vez finalizados los experimentos con el algoritmo DDQN nos centramos en la realización de los experimentos con el algoritmo TRPO. En estos experimentos se mantendrá en la medida de lo posible los mismos parámetros que en los ejecutados con el DDQN para así facilitar la posterior comparación entre los dos algoritmos.

4.2.1. Entorno 3x3 con TRPO

Los parámetros no pueden ser exactamente los mismos ya que, como se ha comentado anteriormente, el algoritmo TRPO no utiliza un *replay buffer*, al contrario de como lo utiliza el algoritmo DDQN. Esto implica que no se necesita ejecutar episodios para rellenar el buffer antes de empezar el aprendizaje. Por lo que en el algoritmo TRPO se empezará el entrenamiento desde el principio.

En este caso se realizarán algo más de 11000 episodios divididos en fases donde se obtendrá una muestra de 50 episodios, entre cada una de estas habrá una fase de entrenamiento. Por cada una de estas fases el valor de la épsilon se reducirá en 0.005, o lo que viene ser equivalente; cada 50 episodios el valor de épsilon se reducirá en 0.005 siendo el valor inicial de ésta 0.90 y el valor final 0. Por lo tanto el proceso de exploración durará un total de 9000 episodios o 180 fases de entrenamiento.

Las fases de muestreo tienen una duración aproximada de 140 segundos en los episodios iniciales, este valor se reduce hacia los 100 segundos en las últimas fases. Todo el experimento ha tenido una duración aproximada de 12 horas en el transcurso de 220 fases de entrenamiento.

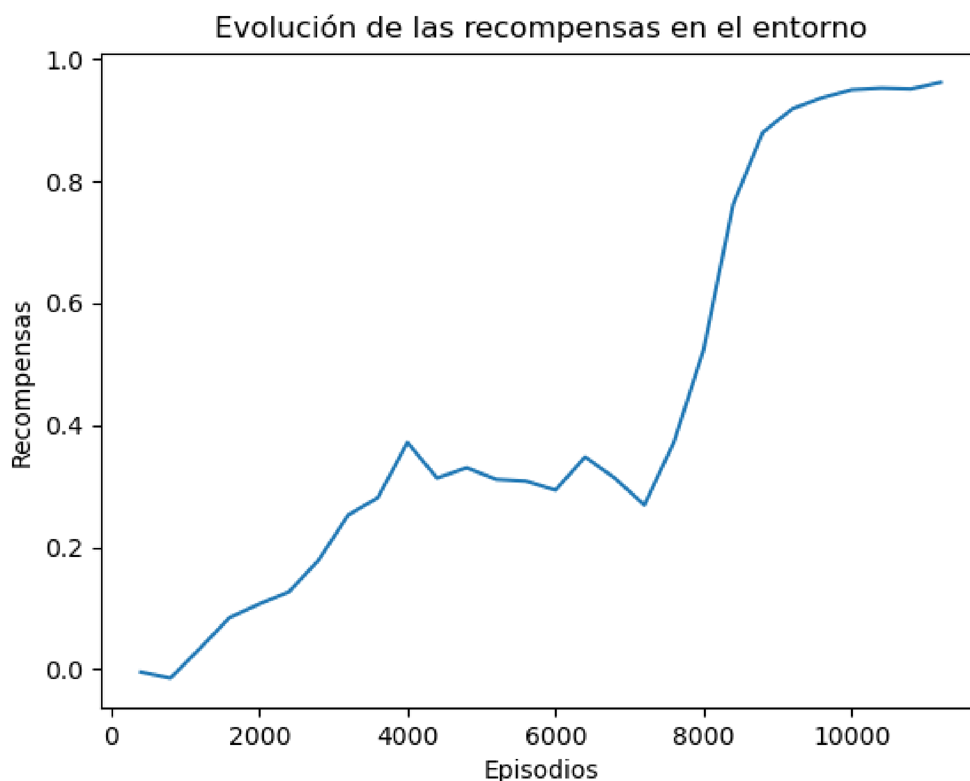


Figura 4.11: Evolución de la recompensa media de los agentes en el entorno 3x3 usando TRPO

Como se puede ver en la gráfica 4.11 el aprendizaje de los agentes empieza desde el principio, en los primeros 4000 episodios las recompensas de los episodios aumenta de forma lineal hasta un valor cercano al 0.3. Durante otros 4000 episodios los agentes mantienen el valor de sus recompensas en este nivel para posteriormente, cerca de los 8000 episodios, recibir en un gran incremento de las recompensas que termina por acercarse al valor máximo que se puede obtener.

La información que nos ofrece esta gráfica nos permite afirmar que los agentes han aprendido a cooperar y cazar al ciervo. También podemos ver las primeras diferencias con el algoritmo DDQN, mientras que en el DDQN se puede observar un aumento constante de las recompensas en el TRPO se puede ver el efecto de la *Trust Region* que genera un aprendizaje más pausado y más estable.

Si visualizamos la distribución de los finales de los episodios podemos obtener información sobre como ha evolucionado el proceso de aprendizaje y sobretodo nos dará un mayor entendimiento de lo sucedido en las primeras fases del aprendizaje.

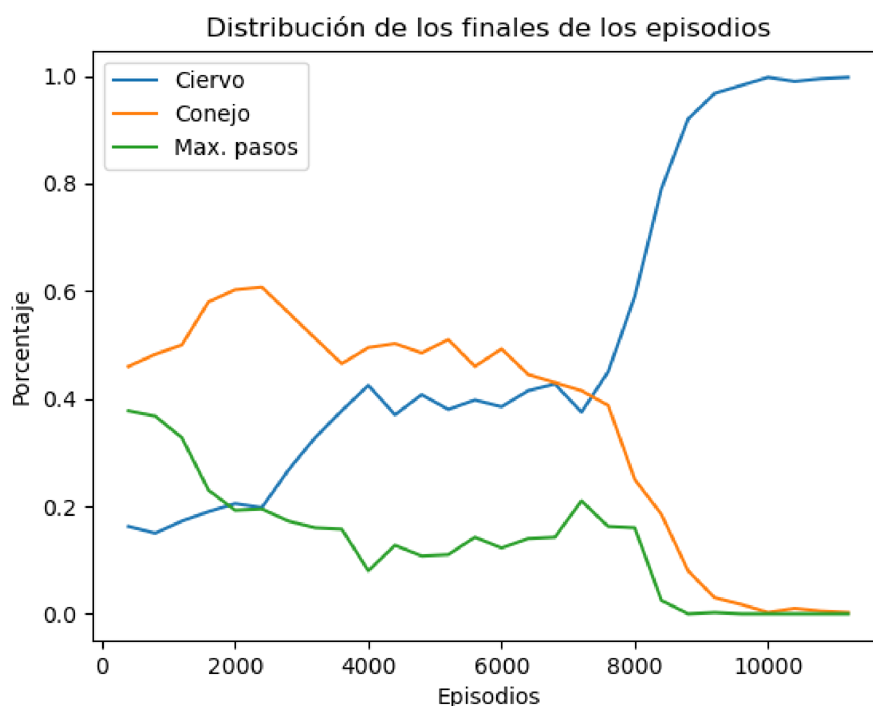
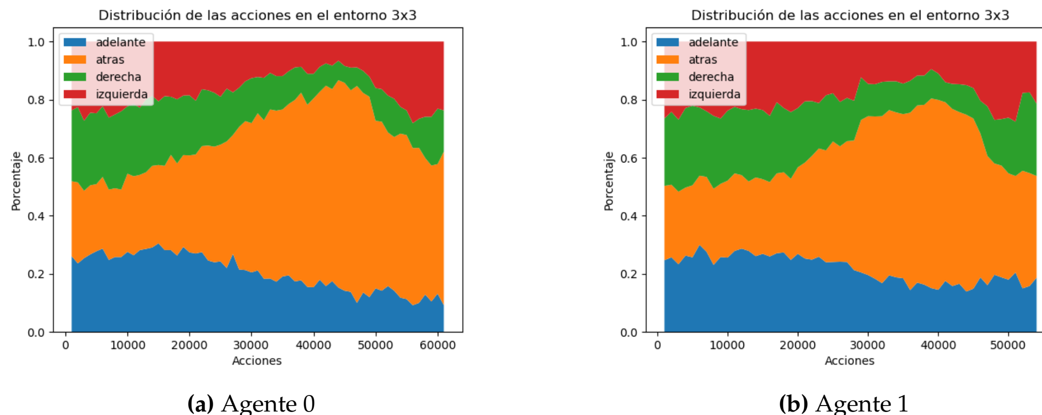


Figura 4.12: Evolución de los finales de los episodios en el entorno 3x3 usando TRPO

En la figura 4.12 se pueden observar varias cosas: lo primero de todo es que la distribución de los finales en los primeros cientos de episodios es la misma que en el caso del algoritmo DDQN, pero poco después, los episodios que terminan por alcanzar el máximo de pasos se reducen considerablemente mientras que los finales por la caza del conejo aumentan. Posteriormente las finalizaciones por caza del ciervo también aumentan manteniéndose a la par con las capturas del conejo.

Aquí se puede ver que los dos algoritmos primero aprendieron a capturar al conejo pero la transición entre la estrategia de cazar al conejo y la estrategia de cazar al ciervo es mucho más abrupta en el DDQN mientras que en el TRPO se realiza lentamente con la intención de no dar pasos demasiado grades en el gradiente y perjudicar al aprendizaje. Cerca del episodio 8000 es cuando la estrategia de cazar al ciervo toma más prioridad y rápidamente desplaza al conejo.



(a) Agente 0

(b) Agente 1

Figura 4.13: Distribución de las acciones de los agentes en el entorno 3x3 TRPO

Estas figuras ofrecen información interesante, se puede ver como los agentes han desarrollado estrategias parecidas a la del Agente0 en el entorno 3x3 4.5, girando y después moviéndose hacia atrás en dirección al ciervo. Pero hacia el final del entrenamiento se puede ver como la preferencia por ir hacia atrás disminuye, sin embargo en 4.11 y 4.12 las recompensas y el número de capturas de ciervos no disminuyen.

La explicación a este fenómeno es que alrededor de los 8000 episodios, cuando la estrategia de cazar al ciervo supera a la de cazar al conejo, la política de los agentes también aumenta la probabilidad de que los agentes elijan ir hacia atrás, pero una vez que la política se estabiliza la estrategia de girar e ir hacia adelante empieza a ser aprendida.

No olvidemos que el objetivo del algoritmo TRPO es obtener una función de recompensa lo más aproximada a la función óptima. Dicha función óptima dará la misma probabilidad de seguir la estrategia de ir hacia atrás como la de ir hacia adelante.

4.2.2. Entorno 5x5 con TRPO

Al igual que con los experimentos de DDQN tras la obtención de los primeros resultados se procedió a la realización de otro experimento utilizando el entorno 5x5.

Este experimento sigue los mismos principios que su contra parte en el algoritmo DDQN: en este tampoco se alterarán los parámetros utilizados, manteniéndose igual en número de episodios (10000) y la reducción del valor de épsilon (0.005). Con esto se intenta observar el efecto de aumentar la complejidad del entorno en los algoritmos.

Este experimento tomó aproximadamente unas 16 horas, durando una media de 300 segundos cada muestra de 50 episodios.

Como se puede ver en la figura 4.14, el resultado es muy parecido a lo observado con el algoritmo DDQN (figura 4.6). Los agentes se estabilizan en en un valor de recompensa bajo, lo cual solo puede significar que los agentes han aprendido a cazar al conejo pero no a cazar al ciervo.

Las figuras 4.15 y 4.16 confirman esta teoría. Podemos ver que los finales donde se caza al conejo componen prácticamente la totalidad de todos los episodios desde los 5000 episodios y los agentes priorizan la acción de caminar hacia adelante que les lleva a cazar al conejo. También se puede observar que los episodios en los que se consigue cazar al ciervo son muy escasos durante el principio del experimento cuando la exploración es más alta. Estos bajos valores en los finales de los episodios en los que se consigue cazar al

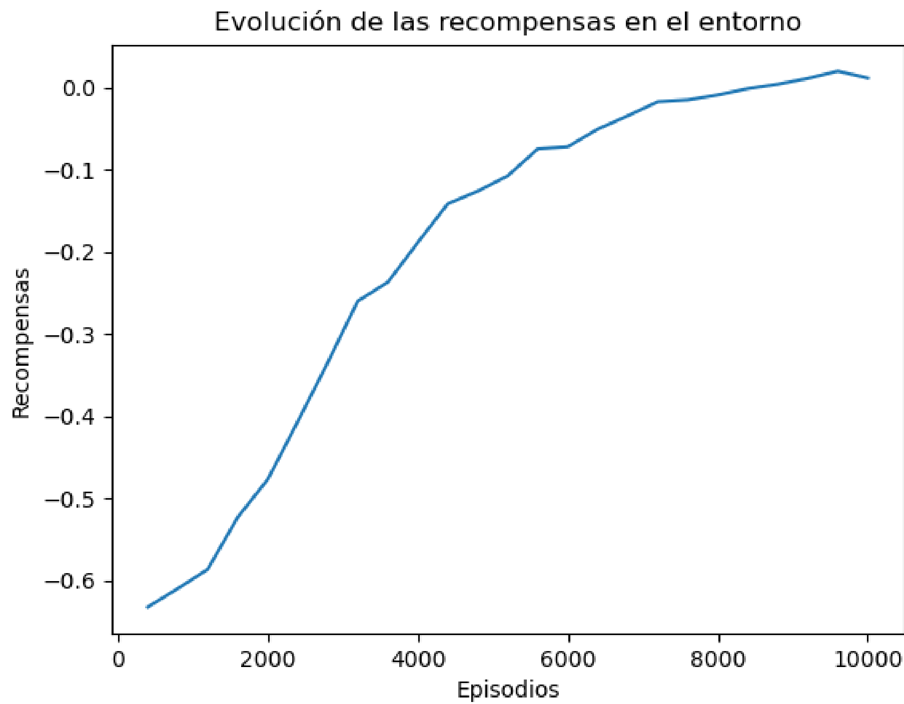


Figura 4.14: Evolución de la recompensa media de los agentes en el entorno 5x5 usando TRPO

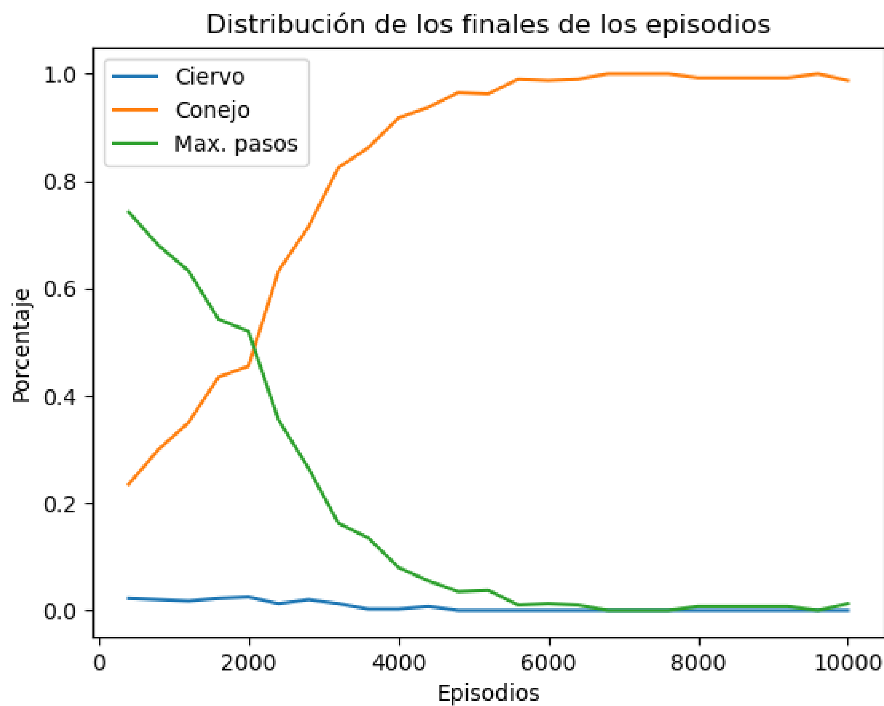
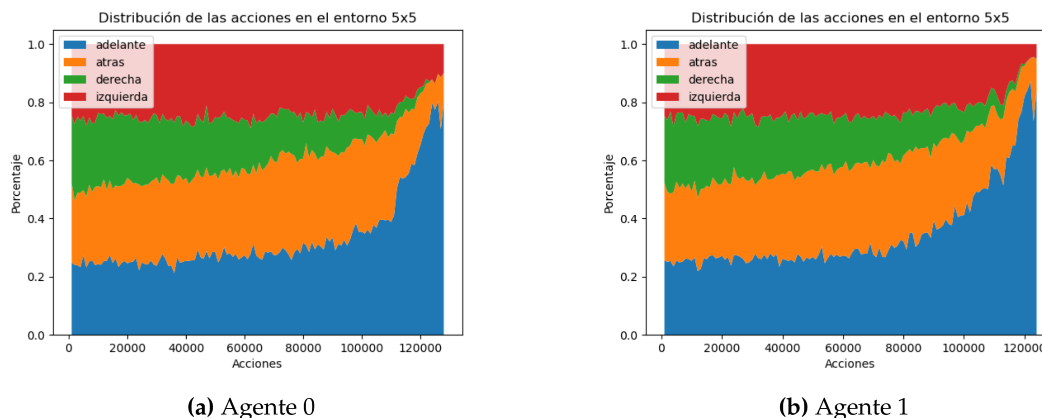


Figura 4.15: Evolución de los finales de los episodios en el entorno 5x5 usando TRPO

ciervo son la causa por la que los agentes no consiguen desarrollar la estrategia de cazar al ciervo.

Del mismo modo que se realizaron los experimentos del algoritmo DDQN, tras observar que los resultados de los experimentos en el entorno 5x5 no concluían en compor-



(a) Agente 0 **(b)** Agente 1

Figura 4.16: Distribución de las acciones de los agentes en el entorno 5x5 TRPO

tamientos cooperativos, se volvió a realizar el mismo experimento pero modificando los parámetros que controlan la exploración para poder saber si este algoritmo es capaz de obtener diferentes resultados.

El parámetro modificado en el algoritmo fue específicamente la velocidad de reducción de ϵ. En esta ejecución ϵ reducirá su valor en 0.002 cada paso de entrenamiento, o lo que viene ser equivalente, cada 50 episodios.

Del mismo modo que en el caso del DDQN, los resultados no difieren demasiado de lo observado en el anterior experimento. Los agentes terminan aprendiendo únicamente a cazar al conejo.

Por otra parte se puede observar que el aprendizaje es más lento que en el experimento anterior. Se observa una tendencia lineal en la evolución de las recompensas. Esto puede estar provocado por la lentitud de la reducción de ϵ, ya que los episodios que finalizan por alcanzar el máximo número de pasos está ligado al valor de ϵ.

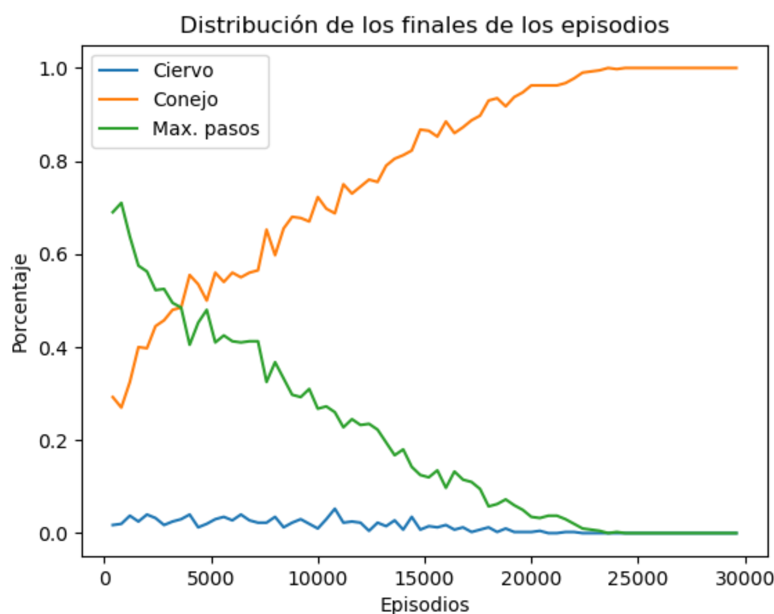


Figura 4.17: Finales de los episodios en el entorno 5x5 usando TRPO parámetros modificados

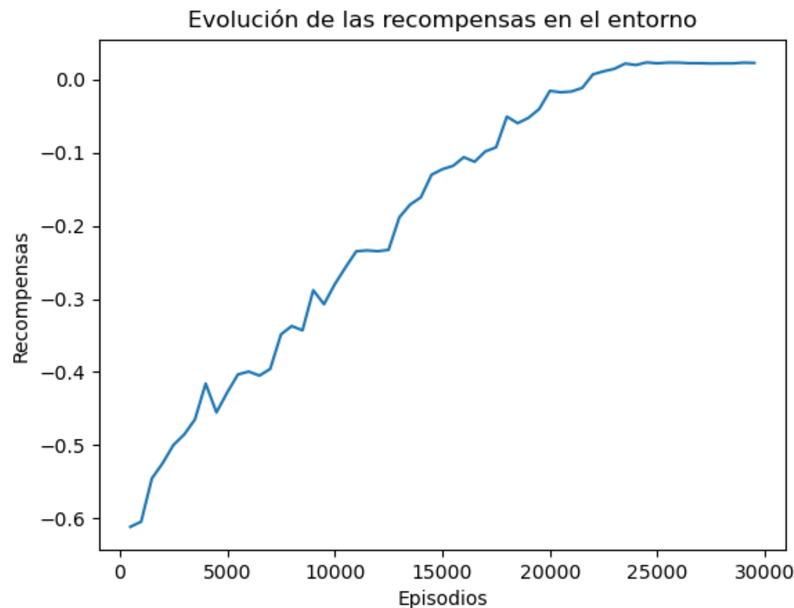


Figura 4.18: Recompensa media en el entorno 5x5 usando TRPO con parámetros modificados

4.3 Experimentación con el algoritmo MADDPG

Por último se emprendió la experimentación con el algoritmo MADDPG. Este algoritmo, al estar especializado en entornos multi-agente podría ser una buena comparativa con los dos anteriores algoritmos mono-agente.

En este caso los parámetros correspondientes a la exploración son controlados directamente por el algoritmo. Por lo que una vez realizadas las modificaciones necesarias para la integración de la plataforma MARLÖ y el correcto funcionamiento del algoritmo se procedió a la ejecución del experimento.

Una vez que la ejecución concluyó y los resultados obtenidos fueron analizados y se pudo observar que estos no eran los esperados para un algoritmo diseñado para entornos multi-agente.

En las figuras 4.19 y 4.20 se puede observar que los agentes solo consiguieron aprender a cazar al conejo pese a estar utilizando el entorno 3x3 donde los algoritmos anteriores no tuvieron problemas en aprender a cazar al ciervo.

Tras obtener estos resultados se repitieron varias ejecuciones en el mismo entorno, obteniendo en todas ellas los mismos resultados.

A partir de aquí se plantearon dos posibles explicaciones para este comportamiento:

- El algoritmo prioriza las estrategias competitivas sobre las cooperativas a causa de la capacidad de los agentes de inferir las políticas de los demás.
- Un fallo en la modificación del algoritmo o en el algoritmo en sí provoca que los agentes no consigan aprender a cazar al ciervo.

La primera explicación es solo una suposición que necesitaría de una investigación más a fondo del algoritmo para poder probar su veracidad. Esta suposición se basa en que el mecanismo que utiliza el algoritmo para inferir las políticas de los demás agentes provoque que se priorice la estrategia competitiva frente a la cooperativa. Además pese a

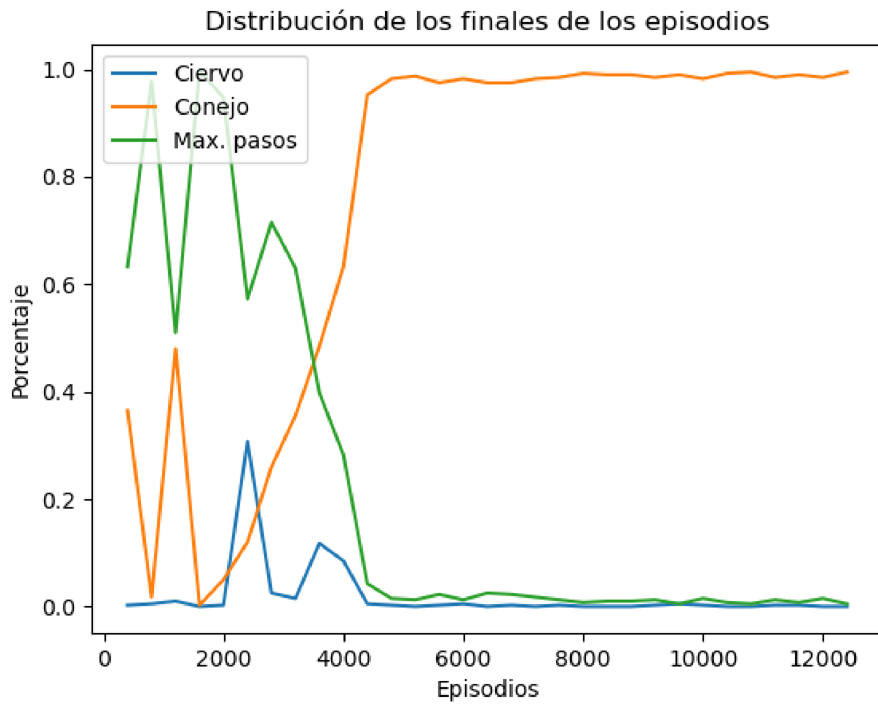


Figura 4.19: Finales de los episodios en el entorno 3x3 usando MADDPG

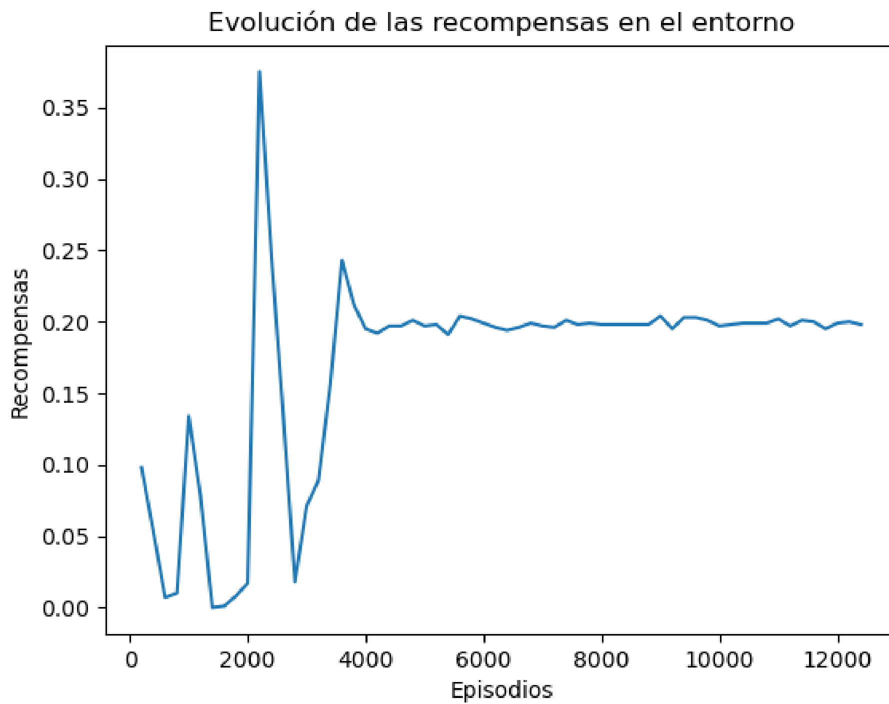


Figura 4.20: Recompensa media de los agentes en el entorno 3x3 usando MADDGP

que el algoritmo ha tenido éxito en entornos con agentes con roles cooperativos o competitivos no se ha probado en entornos donde los agentes no tuvieran roles definidos, como es este entorno donde los agentes pueden tanto cooperar como competir.

La segunda explicación es la que más se ha explorado. Tras los experimentos, las modificaciones realizadas al algoritmo y el propio algoritmo en sí fueron revisados en busca de algún fallo que provocara dicho comportamiento pero nada fue encontrado.

Posteriormente se planteó que fuera un problema de la exploración y el algoritmo no tuviera las suficientes experiencias para aprender a cazar al ciervo. Hay que destacar que el entorno utilizado por MADDPG en su desarrollo utiliza un enfoque diferente en las recompensas que el que se utiliza en los entornos de MARLÖ. Las recompensas en Multi-Agent Particle Environment siempre dan información sobre el objetivo a cumplir por el agente, e.g. distancia a un punto o a otro agente, mientras que las recompensas que obtienen los agentes en MARLÖ solo darán información sobre el objetivo cuando consigan cumplir uno de ellos. Esto puede provocar que la exploración diseñada en MADDPG no sea la correcta para los entornos de MARLÖ. No obstante, debido a que la exploración está controlada internamente por el algoritmo, no se pudieron realizar modificaciones que afectaran a este proceso.

Pese a que la comparación de resultados con un algoritmo como MADDPG puede ser beneficiosa para las conclusiones de este trabajo, estos no pueden ser utilizados. Esto es debido a que no se conoce una explicación clara que los justifique y es necesaria una investigación más exhaustiva para poder obtenerla.

4.4 Análisis de resultados

Tras analizar los resultados de los experimentos realizados podemos llegar a ciertas conclusiones respecto a cómo se comportan los algoritmos mono-agente en entornos multi-agente.

En los experimentos realizados en el entorno 3x3, tanto en el algoritmo DDQN como en el TRPO, se puede observar que aunque los agentes desarrollaran estrategias competitivas en las primeras fases del entrenamiento, no tuvieron problemas al aprender las estrategias cooperativas y eligieron priorizar estas y no las estrategias competitivas.

En cambio en los entornos 5x5 los dos algoritmos se vieron incapaces de aprender ninguna estrategia cooperativa. Resultando en el aprendizaje de estrategias orientadas a cazar al conejo.

Los datos obtenidos en las gráficas 4.4 y 4.12 para el entorno 3x3 y las gráficas 4.8 y 4.15 para el entorno 5x5 nos ayudan a deducir una de las razones de estos resultados.

Si comparamos estas gráficas podemos ver como en las correspondientes a los entornos 3x3 la cantidad de finales en los que se consigue cazar al ciervo es mucho mayor a la cantidad en las gráficas del entorno 5x5. Esto provoca que los algoritmos en los entornos 5x5 no tengan suficientes muestras de finales en los que se caza al ciervo por lo que no consiguen aprender las estrategias correspondientes.

Se ha de recalcar que la diferencia entre el numero mínimo de pasos para conseguir cazar al ciervo en los dos entornos pasa de ser 2 pasos en el entorno 3x3 a 4 en el entorno 5x5 pero la probabilidad de conseguir el final de forma aleatoria en pasa de ser un 13 % a un 1.9 %.

Esta gran diferencia es debida al requerimiento de que los dos agentes se encuentren en el lugar especifico al mismo tiempo para poder cazar al ciervo. Este requerimiento de cooperación hace que cuanto más difícil le sea a un agente llegar al lugar especifico

para cazar al ciervo, la probabilidad de que los dos agentes se posicionen correctamente disminuirá de forma exponencial.

Esto se puede ver si desarrollamos las ecuaciones que nos indican la probabilidad de que los agentes obtengan los los finales cooperativo y competitivo en el numero mínimo de pasos:

$$\prod_{t=1}^{m} p_t \tag{4.1}$$

$$\prod_{t=1}^{m} p_{0_t} \prod_{t=1}^{m} p_{1_t} \tag{4.2}$$

Siendo p la probabilidad de que los agentes realicen la acción correcta en el momento t. Sabiendo que la probabilidad de las acciones en un comportamiento aleatorio son las mismas, la probabilidad de coger al conejo seria $\frac{1}{4}^{m}$ y la probabilidad de cazar al ciervo $\frac{1}{4}^{2m}$

Estos datos sugieren que en los entornos multi-agente un aumento de la complejidad del entorno puede ser un factor crítico en los resultados del aprendizaje. Sobretodo en entornos que requieran la coordinación de varios agentes para obtener una recompensa.

Por otra parte no se han encontrado diferencia significativas entre el eficiencia de los algoritmos DDQN y TRPO. Pese a que estos dos algoritmos presentan un comportamiento diferente en el proceso de aprendizaje. Como se puede observar en la figuras 4.5 y 4.13, los dos algoritmos consiguen llegar a estrategias similares con la misma cantidad de episodios.

CAPÍTULO 5
Conclusiones

La primera fase de este proyecto se ha centrado en la investigación de los algoritmos existentes en el campo del aprendizaje por refuerzo. De la gran variedad de algoritmos existentes en este campo se ha elegido tres de ellos para la realización de los experimentos. Dichos algoritmos han sido seleccionados teniendo en cuenta los diferentes métodos que utilizan en su proceso de aprendizaje, con la intención de obtener una colección de algoritmos variada que beneficie al proceso de experimentación.

La siguiente fase del proyecto se centró en la modificación de las implementaciones de los algoritmos para adecuarlos al entorno de MARLÖ. Las dos primeras implementaciones, DDQN y TRPO, fueron modificadas satisfactoriamente. Respecto a la tercera modificación correspondiente al algoritmo MADDPG, pese a tener un funcionamiento correcto, podemos concluir que los resultados obtenidos en la experimentación no fueron los esperados. Una posible explicación es que puede deberse a un fallo en la implementación que no ha sido detectado. Sin embargo en el presente trabajo se plantea también el razonamiento por el que podría no ser un algoritmo adecuado para este entorno.

Las modificaciones realizadas en las implementaciones se centraron sobre todo en dos ámbitos. El primero fue la adaptación de las observaciones provistas por los entornos de MARLÖ a las utilizadas por los algoritmos. El segundo, se centró en permitir que los agentes se ejecutaran concurrentemente en los diferentes algoritmos. Para cada uno de estos algoritmos fue necesario una modificación distinta debido a las peculiaridades de cada uno.

Uno de los mayores problemas en el proceso de modificación fue el propio MARLÖ que también necesitó de diferentes refinamientos para poder obtener ejecuciones eficientes y minimizar los errores.

Antes de realizar la experimentación se tuvo que diseñar unos entornos personalizados para dicha causa. Los entornos adaptaron el problema de teoría de juegos 'Stag Hunt' a los entornos de MARLÖ. Se generaron dos entornos, el primero más sencillo compuesto por un recito de 3x3 bloques y el segundo con un recinto de 5x5 bloques. Con estos entornos se quiso analizar la eficacia y eficiencia de los algoritmos y como afecta un aumento de la dimensionalidad en los entornos multi-agente al proceso de aprendizaje.

Por último, se realizaron los experimentos con los algoritmos utilizando los dos entornos anteriormente diseñados. De estos experimentos se obtuvieron diferentes datos sobre el proceso de aprendizaje de cada algoritmo que posteriormente fueron analizados.

Tras el análisis de los datos se alcanzaron diferentes conclusiones referentes al aprendizaje de dichos algoritmos en entornos multi-agente.

En primer lugar los resultados obtenidos por MADDPG no fueron los esperados, siendo estos peores que los algoritmos mono-agente. Como se comentó anteriormente, esto

puede deberse a un error en la modificación o a una incompatibilidad con los entornos utilizados.

Los resultados obtenidos por los otros dos algoritmos fueron fructíferos y suministraron información interesante respecto al proceso de aprendizaje. Las conclusiones extraídas de estos datos indican que los algoritmos mono-agente utilizados no tuvieron problemas en aprender a cooperar en entornos multi-agente siempre que obtengan suficiente información de la exploración. Además también se pudo observar como la dimensionalidad de los entornos tiene un efecto crítico en la experiencia obtenida en la exploración. Esto hace necesario el uso de métodos que mejoren la eficacia de la exploración en entornos multi-agente complejos.

5.1 Trabajo futuro

Este trabajo presenta diferentes caminos que pueden seguirse y que enriquecerían los resultados obtenidos hasta el momento en este proyecto.

El primero sería ahondar más en la implementación del algoritmo MADDPG para así poder averiguar cual ha sido el problema que ha generado los resultados obtenidos y si es posible modificar la implementación para obtener resultados que aporten más información sobre la eficacia de este algoritmo en los entornos desarrollados.

Otra opción complementaria a esta sería la experimentación con otros algoritmos tanto multi-agente como mono-agente y el diseño de más entornos en MARLÖ con diferentes dinámicas.

Por último, el estudio de los efectos de diferentes métodos que ayuden a la exploración, como pueden ser los *replay buffer* con prioridad, en los entornos diseñados y que pueden ofrecer información de cómo sobrepasar los problemas obtenidos por el aumento de dimensionalidad.

5.2 Relación con los estudios cursados

Para la realización de este trabajo han sido necesarios los conocimientos de varias de las asignaturas cursadas en el grado de Ingeniería Informática. Dentro de estas entran tanto asignaturas obligatorias como asignaturas pertenecientes a la rama de Computación. A continuación se podrá encontrar una lista que expondrá cuales son dichas asignaturas y como han beneficiado al proyecto:

- Estadística (11539), ha facilitado la comprensión, tratamiento y análisis de los resultados.

- Concurrencia y sistemas distribuidos(11562), las modificaciones para permitir la concurrencia de los agentes se han visto altamente beneficiadas de los conocimientos obtenidos en esta asignatura.

- Agentes inteligentes (11587), Aprendizaje automático (11594)y Técnicas, entornos y aplicaciones de inteligencia artificial (11592), este conjunto de asignaturas proporcionó conocimientos respecto a la Inteligencia Artificial y a las Redes Neuronales.

- Agentes inteligentes (11587), proporcionó conocimientos sobre los sistemas multi-agente.

- Gestión de proyectos (11554), proporcionó los conocimientos necesarios para la gestión de este trabajo.

Además de estos conocimientos, la realización de este trabajo ha necesitado de la adquisición complementaria de conocimientos en el campo del aprendizaje por refuerzo.

Por último se listarán las competencias transversales que mas relevancia han tenido en el desarrollo de este proyecto:

- CT1 - Comprensión e integración: En el proyecto se ha requerido el aprendizaje, comprensión y uso de nuevas tecnologías.
- CT2 – Aplicación y pensamiento práctico: En el proyecto ha sido necesaria la recopilación de información en diferentes ámbitos para poder tomar decisiones.
- CT3 – Análisis y resolución de problemas: Para la realización del proyecto ha sido necesario el planear y seguir una serie de pasos de forma ordenada.
- CT9 - Pensamiento crítico: Durante el proyecto han surgido situaciones inesperadas que han llevado a deducir que se hayan podido producir fallos en el desarrollo.
- CT11 – Aprendizaje permanente: Como se ha mencionado anteriormente este proyecto ha necesitado la obtención de nuevos conocimientos, especialmente del aprendizaje por refuerzo.
- CT12 - Planificación y gestión del tiempo: Para el correcto desarrollo de este trabajo ha sido necesaria la planificación y gestión del tiempo.

5.3 Agradecimientos

Por último me gustaría agradecer a todas aquellas personas que han colaborado en este TFG de una manera u otra. Ante todo, dar las gracias a mis tutores Vicente J. Julian y Javi Palanca por toda la ayuda que me han prestado en la realización del TFG. También doy las gracias al Instituto Valenciano de Investigación en Inteligencia Artificial (VRAIN) por concederme la beca de formación que ha hecho este trabajo posible.

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan V GPU used for this research.

Por otra parte, en un ámbito más personal, también quiero agradecer el apoyo que me han brindado mi padre y mi madre durante el transcurso de este proyecto. Por último doy las gracias a todos mis amigos por estar siempre ahí para animarme, darme fuerzas para seguir adelante y sacarme una sonrisa.

Bibliografía

[1] Richard Sutton. y Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[2] Hyacinth S. Nwana. Software Agents: An Overview. *en Knowledge Engineering Review*, vol. 11, No 3, pp. 205-244, 1996.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra y Martin Riedmiller Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[4] John N. Tsitsiklis y Benjamin Van Roy Analysis of temporal-difference learning with function approximation. *Automatic Control, IEEE Transactions* , vol. 42, no. 5, pp.674–690, 1997.

[5] Biao Luo, Yin Yang y Derong Liu, Adaptive Q-learning for databased optimal output regulation with experience replay. *IEEE Transactions on Cybernetics*, vol. 48, no. 12, pp. 3337-3348, Dic. 2018.

[6] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan y Pieter Abbeel. Trust region policy optimization. *arXiv preprint arXiv:1502.05477*, 2015.

[7] Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, vol. 8, no. 3, pp. 229-256, 1992.

[8] Matthew Johnson, Katja Hofmann, Tm Hutton, David Bignell y Katja Hofmann (2016) The Malmo Platform for Artificial Intelligence Experimentation. en *International joint conference on artificial intelligence (IJCAI)*.

[9] Diego Perez-Liebana, Katja Hofmann, Sharada Prasanna Mohanty, Noburu Kuno, Andre Kramer, Sam Devlin, Raluca D Gaina, y Daniel Ionita. The multi-agent reinforcement learning in Malmö (MARLÖ) competition. *arXiv preprint arXiv:1901.08129* , 2019.

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

[11] Clément Romac, Vincent Béraud. Deep Recurrent Q-Learning vs Deep Q-Learning on a simple Partially Observable Markov Decision Process with Minecraft. *arXiv preprint arXiv:1903.04311* , 2019.

[12] Mnih, V., Kavukcuoglu, K., Silver, D. et al. Human-level control through deep reinforcement learning. *en Nature*, vol. 518, pp. 529–533, 2015.

[13] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar y David Silver (2018) Rainbow: Combining Improvements in Deep Reinforcement Learning. en *AAAI Conference on Artificial Intelligence (AAAI)* .

[14] MichałKempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek y Wojciech Jaśkowski (2016) ViZDoom: A Doom-based AI research platform for visual reinforcement learning. *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1-8, Santorini.

[15] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. Sasha Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser, et al. StarCraft II: A New Challenge for Reinforcement Learning. *arXiv preprint arXiv:1708.04782* , 2017.

[16] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michael Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *en Nature*, vol. 575, pp. 350–354, 2019.

[17] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *arXiv preprint arXiv:1912.06680* , 2019.

[18] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew y Igor Mordatch. Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528* , 2019.

[19] Ryan Lowe, YI WU, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel y Igor Mordatch (2017) Multi-agent actor-critic for mixed cooperative-competitive environments. en *the Annual Conference on Neural Information Processing Systems (NIPS)* .

[20] Lianmin Zheng, Jiacheng Yang, Han Cai, Weinan Zhang, Jun Wang y Yong Yu. Magent: A many-agent reinforcement learning platform for artificial collective intelligence. *arXiv preprint arXiv:1712.00600* , 2017.

[21] Christopher JCH Watkins y Peter Dayan, . Q-learning. *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[22] Hado Van Hasselt, Arthur Guez, y David Silver. Deep Reinforcement Learning with Double Q-learning. en *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[23] B.T. Polyak. Some methods of speeding up the convergence of iteration methods. en *USSR Computational Mathematics and Mathematical Physics*, vol. 8, pp. 1-17, 1964.

[24] Joy A. Thomas y Thomas M. Cover *Elements of Information Theory*. Wiley, Nueva York, 1991.

[25] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford y Oleg Klimov. Proximal policy optimization algorithms. en *arXiv preprint arXiv:1707.06347* 2017.