



viu

**Universidad
Internacional
de Valencia**

Aprendizaje por Refuerzo Profundo Multiagente

Titulación:
Máster Universitario en
Inteligencia Artificial

Curso académico
2023 – 2024

Alumno/a: Soto Santos,
Pedro Luis

D.N.I: 53912470X

Director/a de TFM: Dr.
César Guzmán

Convocatoria:
Segunda

Agradecimientos

En primer lugar, me gustaría agradecer a todos y cada uno de los profesores que se han cruzado en mi camino, ya que cada uno ha dejado una huella en mi formación, tanto académica como personal.

También quiero agradecer profundamente a mi familia, especialmente a mis padres y hermana, por brindarme su amor incondicional y apoyo constante. A mis amigos y compañeros, les agradezco su empatía y su invaluable compañía y ayuda a lo largo de este viaje.

Finalmente, me gustaría recordar cada obstáculo y desafío que he enfrentado. Gracias a ellos, he aprendido que con esfuerzo y perseverancia no solo se pueden superar las adversidades, sino también alcanzar metas que alguna vez parecieron inalcanzables.

A todos ellos, gracias.

*"Aquel que tiene un porqué para vivir,
puede soportar casi cualquier cómo"*

Friedrich Nietzsche

Índice general

Índice de figuras	II
Resumen	1
Abstract	2
1. Introducción	3
2. Objetivos	4
2.1. Objetivos Específicos	4
3. Estado del Arte	5
3.1. Marco Teórico	6
3.1.1. Aprendizaje por Refuerzo	6
3.1.2. Aprendizaje por Refuerzo Multiagente	11
4. Caso de Estudio	15
4.1. Entorno Lunar	15
4.2. Simulación	16
4.3. Algoritmos	20
4.3.1. Double Dueling DQL	22
4.3.2. Multi-Agent PPO	24
5. Experimentación y Pruebas	29
6. Conclusiones y Trabajo Futuro	41
Lista de Acrónimos	44
Bibliografía	45



Índice de figuras

3.1. Interacción Agente-Entorno	6
3.2. Pseudocódigo Deep Q-Learning	8
3.3. Pseudocódigo Proximal Policy Optimization - Clip	11
3.4. Paradigmas de entrenamiento multiagente	13
4.1. Diferentes tipos de obstáculos en el entorno lunar	15
4.2. Captura del entorno renderizado por <i>pygame</i>	18
4.3. Pseudocódigo del algoritmo Double DQL	22
4.4. Fórmula para el cálculo de los Q valores en el algoritmo Dueling DQL	23
4.5. Comparación de las arquitecturas de las redes neuronales DQN y Dueling DQN	23
4.6. Pseudocódigo del algoritmo MAPPO	25
5.1. Evolución de la pérdida o <i>loss</i> DDDQL	30
5.2. Evolución de la pérdida o <i>loss</i> MAPPO	31
5.3. Evolución de la recompensa media por acción DDDQL	31
5.4. Evolución de la recompensa media por acción MAPPO	32
5.5. Evolución del número de episodios finalizados entre puntos de control DDDQL .	32
5.6. Evolución del número de episodios finalizados entre puntos de control MAPPO	33
5.7. Evolución del número de acciones máximas utilizadas en un solo episodio entre puntos de control DDDQL	33
5.8. Evolución del número de acciones máximas utilizadas en un solo episodio entre puntos de control MAPPO	34
5.9. Primer entorno prueba uno	35
5.10. Primer entorno prueba dos	36
5.11. Segundo entorno prueba uno	36
5.12. Segundo entorno prueba dos	37
5.13. Tercer entorno prueba uno	37
5.14. Tercer entorno prueba dos	38
5.15. Cuarto entorno prueba uno	38
5.16. Cuarto entorno prueba dos	39



Resumen

Este Trabajo de Fin de Máster presenta un sistema avanzado de planificación de rutas multiagente basado en algoritmos de *Deep Reinforcement Learning* o Aprendizaje por Refuerzo Profundo (DRL), diseñado como una solución alternativa frente a las limitaciones de los métodos centralizados tradicionales. El objetivo de este sistema es optimizar la navegación de múltiples *lunar rovers* en un entorno lunar altamente dinámico y complejo, donde serán utilizados para la recolección y transporte de minerales necesarios para la construcción de la primera Estación Internacional de Investigación Lunar en el año 2030.

Para lograr este objetivo, se han implementado y entrenado dos algoritmos clave, *Double Dueling Deep Q-Learning* (DDDQL) y *Multi-Agent Proximal Policy Optimization* (MAPPO). Estos algoritmos están orientados a maximizar la eficiencia en la planificación de rutas de los *lunar rovers*, minimizando el consumo energético y reduciendo el riesgo de colisiones. La investigación incluye el diseño y desarrollo de un simulador de entornos lunares que incorpora configuraciones aleatorias de obstáculos y objetivos para diversificar las condiciones durante la fase de entrenamiento, así como configuraciones predefinidas para la evaluación de los modelos entrenados. Los resultados obtenidos demuestran el considerable potencial de los algoritmos desarrollados para enfrentar los desafíos presentes en el entorno lunar, a la vez que destacan áreas de mejora que podrían abordarse en futuras investigaciones.

Palabras clave: Aprendizaje por refuerzo profundo multiagente, planificación de rutas multiagente, DDDQL, MAPPO, astromóviles lunares

Abstract

This Master Thesis presents an advanced multi-agent route planning system based on *Deep Reinforcement Learning* o Aprendizaje por Refuerzo Profundo (DRL) algorithms, designed as an alternative solution to the limitations of traditional centralized methods. The goal of this system is to optimize the navigation of multiple lunar rovers in a highly dynamic and complex lunar environment, where they will be used for the collection and transportation of minerals needed for the construction of the first International Lunar Research Station in 2030.

To achieve this goal, two key algorithms, *Double Dueling Deep Q-Learning* (DDDQL) and *Multi-Agent Proximal Policy Optimization* (MAPPO), have been implemented and trained. These algorithms are aimed at maximizing the route planning efficiency of the lunar rovers, minimizing energy consumption and reducing the risk of collisions. The research includes the design and development of a lunar environment simulator that incorporates randomized configurations of obstacles and targets to diversify the conditions during the training phase, as well as predefined configurations for the evaluation of the trained models. The results obtained demonstrate the considerable potential of the developed algorithms to address the challenges present in the lunar environment, while highlighting areas for improvement that could be addressed in future research.

Keywords: Multi-agent deep reinforcement learning, multi-agent pathplanning, DDDQL, MAPPO, lunar rovers

Introducción

1

En el año 2030, se busca establecer en la Luna una Estación Internacional de Investigación Lunar, para la que se necesitará una gran cantidad de material de construcción. Parte de estos materiales serán obtenidos a través de la utilización de múltiples astromóviles lunares, como los *lunar rovers*, equipados con las herramientas necesarias para la recolección en diversas minas de los materiales requeridos [Lu et al. \(2024\)](#).

El objetivo principal de este Trabajo de Fin de Máster es investigar alternativas a los métodos tradicionales de planificación de rutas, como la planificación centralizada desde la Tierra, que han demostrado ser inefficientes para el control de múltiples *lunar rovers*. Estos métodos carecen de la capacidad de adaptación necesaria para satisfacer las necesidades dinámicas que experimentan los múltiples *lunar rovers* operando hacia diferentes objetivos, enfrentando diversos obstáculos en el camino y buscando rutas óptimas con el menor consumo energético posible.

Estos desafíos son especialmente críticos en entornos tan remotos como la Luna, donde la escasez de recursos y la limitada logística requieren maximizar la eficiencia energética para prolongar la duración operativa de los *lunar rovers* a lo largo de la vida útil de sus baterías. Además, es crucial minimizar el riesgo de averías que puedan dejar inutilizado cualquier *lunar rover*, dado el alto costo de estos vehículos y las dificultades asociadas con las reparaciones en tales entornos.

Objetivos

2

El objetivo principal de este Trabajo de Fin de Máster es desarrollar una solución de planificación de rutas multiagente basada en algoritmos de *Deep Reinforcement Learning* o Aprendizaje por Refuerzo Profundo (DRL), permitiendo a los agentes, en nuestro caso los *lunar rovers*, encontrar y adaptarse de manera autónoma a los desafíos de los entornos lunares complejos e inciertos.

2.1. Objetivos Específicos

1. Revisar el estado del arte actual sobre los algoritmos de *Multi-Agent Reinforcement Learning* o Aprendizaje por Refuerzo Multiagente (MARL) y *Multi-Agent Deep Reinforcement Learning* o Aprendizaje por Refuerzo Profundo Multiagente (MADRL) más utilizados en problemas de planificación de rutas multiagente. Esto incluirá un análisis de los algoritmos base y sus adaptaciones para la resolución de problemas multiagente, como el que se busca resolver.
2. Diseñar y desarrollar la simulación del entorno lunar reproduciendo adecuadamente las condiciones del problema a completar.
3. Seleccionar y desarrollar los algoritmos más destacados para la planificación de rutas multiagente, en este caso, para múltiples *lunar rovers* en un mismo entorno lunar.
4. Realizar el entrenamiento con los distintos algoritmos desarrollados haciendo uso del simulador. Los entrenamientos serán supervisados para optimizar sus rendimientos mediante el ajuste de hiperparámetros y demás opciones configurables en el entorno lunar simulado.
5. Evaluar mediante experimentación práctica, dentro de simulaciones configuradas con entornos de prueba, la eficacia y rendimiento de los modelos entrenados.

Estado del Arte

3

Tradicionalmente, los problemas de planificación de rutas se han abordado mediante métodos analíticos o algoritmos de búsqueda deterministas como A*, diseñados para entornos estáticos y completamente conocidos con el objetivo de recrear de manera perfecta el entorno disponible. Sin embargo, esta aproximación idealizada no refleja la complejidad de los entornos reales, donde la información no está completamente disponible, sino que se obtiene conforme el agente va explorando el entorno a través de sus sensores. Además, puede ocurrir que los entornos contengan obstáculos dinámicos que cambian con el paso del tiempo, aumentando aún más la complejidad del entorno.

Recientemente, las investigaciones en el campo de los robots móviles autónomos se han centrado en la aplicación y combinación de redes neuronales profundas y aprendizaje por reforzamiento, conocido como *Deep Reinforcement Learning* o Aprendizaje por Refuerzo Profundo (DRL), para la resolución de problemas de planificación de rutas, como se discute en el estudio [Singh et al. \(2023\)](#), obteniendo resultados bastante prometedores. Entre los algoritmos de la familia DRL investigados, destacan *Proximal Policy Optimization* (PPO) y *Deep Q-Learning* (DQL), utilizados tanto para la planificación de rutas de extremo a extremo para un único agente [Yu et al. \(2021\)](#) [Kulathunga \(2022\)](#), como para la planificación de rutas considerando la evasión de colisiones en entornos con obstáculos dinámicos [Zhu et al. \(2022\)](#) [Thanh Ha y Vinh \(2024\)](#).

En el contexto de la planificación de rutas multiagente, donde varios agentes buscan su propia ruta evitando los obstáculos estáticos del propio entorno, surge el desafío adicional de que cada agente debe evitar a los demás agentes en movimiento, convirtiéndose estos en obstáculos dinámicos del entorno. Anteriormente, para la evitación recíproca de colisiones entre múltiples agentes se ha utilizado el algoritmo *Optimal Reciprocal Collision Avoidance* (ORCA) [van den Berg et al. \(2011\)](#). Sin embargo, una limitación de ORCA es que se centra exclusivamente en la evasión recíproca de colisiones entre agentes y necesita ser complementado con otros métodos de planificación de rutas, cuya combinación puede llegar a ser compleja y no siempre es eficiente.

Con el creciente interés en la investigación de los algoritmos de DRL como métodos para la resolución de problemas de planificación de rutas, también se ha profundizado en el desarrollo de variantes adaptadas para problemas de *Multi-Agent Path Planning* o Planificación de Rutas Multiagente (MAPP) [Chung et al. \(2023\)](#). En este contexto, destacan variantes de los algoritmos

PPO y DQL, que no solo actúan como planificadores de rutas multiagente Park et al. (2022) Yang et al. (2020), sino que también logran la evitación de colisiones entre los agentes en movimiento dentro del entorno dinámico Long et al. (2018) Ma et al. (2021). Estas variantes de los algoritmos PPO y DQL permiten resolver de manera conjunta tanto la planificación de rutas multiagente como la evasión de colisiones entre agentes, eliminando la necesidad de combinar múltiples métodos para resolver los diferentes aspectos del problema.

A continuación, se expondrá el marco teórico de los algoritmos mencionados anteriormente para tener una contextualización completa antes de profundizar en el desarrollo de estos. Se comenzará introduciendo los fundamentos del *Reinforcement Learning* o Aprendizaje por Refuerzo (RL), se explicará su evolución hacia el *Deep Reinforcement Learning* o Aprendizaje por Refuerzo Profundo (DRL) con la incorporación de redes neuronales profundas, y se analizará su aplicación en problemas multiagente, dando lugar al *Multi-Agent Reinforcement Learning* o Aprendizaje por Refuerzo Multiagente (MARL) y *Multi-Agent Deep Reinforcement Learning* o Aprendizaje por Refuerzo Profundo Multiagente (MADRL).

3.1. Marco Teórico

3.1.1. Aprendizaje por Refuerzo

El *Reinforcement Learning* o Aprendizaje por Refuerzo (RL) es una rama del aprendizaje automático donde un agente aprende a tomar decisiones interactuando con el entorno. A través de esta interacción, el agente recibe recompensas o castigos según las acciones que toma y los estados a los que llega. El objetivo final es maximizar la recompensa acumulada a lo largo del tiempo. Esta rama de la inteligencia artificial está inspirada, en cierta forma, en el aprendizaje humano, en el cual aprendemos a través de la interacción y la observación de las consecuencias de nuestras acciones. Mediante un proceso de ensayo y error, acumulamos experiencias que nos permiten mejorar nuestro comportamiento en entornos conocidos y adaptarnos mejor a nuevos entornos desconocidos.

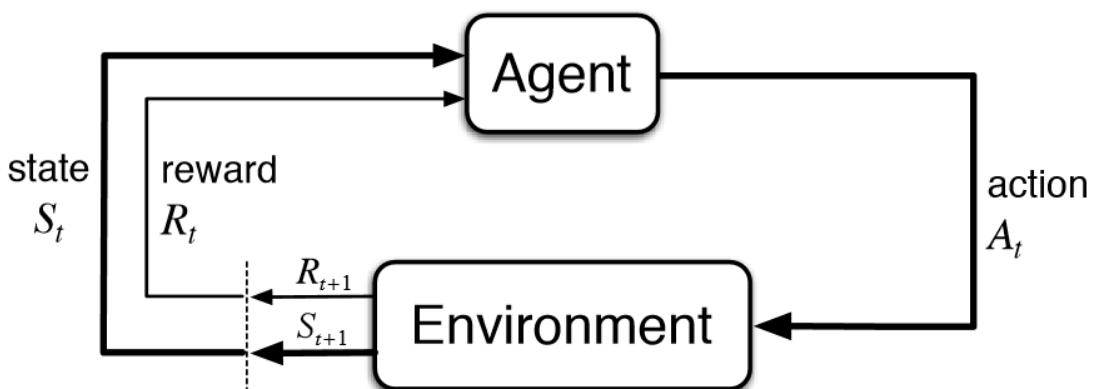


Figura 3.1: Interacción Agente-Entorno.

Uno de los principales problemas del aprendizaje por refuerzo era que, hasta hace poco tiempo, los algoritmos requerían la elaboración manual de variables para representar el entorno y las posibilidades del agente. Esto exigía un amplio conocimiento del dominio del problema a resolver, limitando el rendimiento de los algoritmos, ya que dependía en gran medida de la calidad de la representación de las características del problema. Por lo tanto, era inviable utilizar estos sistemas en problemas con estados demasiado complejos como para capturar todas sus características.

Sin embargo, en el año 2013, gracias al gran avance en la rama del *Deep Learning* o Aprendizaje Profundo (DL), la empresa *DeepMind* publicó el primer artículo en el que combinaba ambas ramas [Mnih et al. \(2013\)](#). Esta combinación entre aprendizaje por refuerzo y aprendizaje profundo, llamada *Deep Reinforcement Learning* o Aprendizaje por Refuerzo Profundo (DRL), permite que los agentes puedan aprender a partir de entradas de datos no estructurados, como por ejemplo, imágenes de un videojuego, eliminando la necesidad de diseñar manualmente la estructura de datos necesaria para representar las características del problema. Esto ha ampliado la aplicabilidad de estos algoritmos a problemas más complejos.

En este Trabajo de Fin de Máster, nos centraremos en dos algoritmos de tipo *model-free*, es decir, aquellos en los que la simulación no conoce un modelo para predecir los siguientes estados en el entorno. Este tipo de algoritmos se ajusta mejor al problema que queremos resolver, ya que el entorno lunar es muy complejo y no existe un modelo para predecir su evolución. Normalmente, este tipo de algoritmos se aplica a entornos con comportamientos no deterministas, lo que impide conocer el modelo, por lo que el agente busca una estrategia que depende únicamente del estado actual en el que se encuentra e incluso de algún estado anterior.

Antes de entrar en detalle con la definición de los algoritmos, es importante aclarar que un problema de aprendizaje por refuerzo se define dentro del marco matemático de un *Márkov Decision Process* o Proceso de Decisión de Márkov (MDP). En un MDP se representan los posibles estados en los que se puede encontrar el agente, las acciones que puede tomar en cada estado, una función de transición, que define la probabilidad de que el agente pase de un estado a otro tras tomar una acción, y la recompensa inmediata recibida tras pasar de un estado a otro debido a una acción específica. Todo esto lleva a la existencia de una política o estrategia óptima, que corresponde a un mapeo entre el espacio de estados y acciones, a partir del cual se puede calcular la mejor acción basada en la recompensa acumulada, convirtiéndose en una cadena de Markov donde las transiciones entre estados se determinan completamente por la política y la matriz de transición.

En el caso del aprendizaje por refuerzo, se utilizan MDPs en los que las probabilidades de transición e incluso las recompensas obtenidas son desconocidas. Esto es lo que se va aprendiendo con la experiencia obtenida por el agente durante el entrenamiento mediante pares de estado-acción, donde se guarda la experiencia del agente al estar en un estado, realizar una acción y acabar en otro estado, e incluso obtener cierta recompensa.

3.1.1.1. Deep Q-Learning

El algoritmo *Q-Learning* es un algoritmo *off-policy*, es decir, no necesita seguir una política específica, por lo que se puede usar experiencia adquirida con políticas distintas para encontrar la política óptima. Este algoritmo se centra en entrenar una función acción-valor, llamada función Q, que determina el valor de estar en un estado concreto y realizar una acción específica en dicho estado, en otras palabras, es un algoritmo *value-based*. Para determinar esta función Q se debe establecer una tabla Q donde cada celda corresponde al par de valores estado-acción, funcionando como la memoria del algoritmo que se irá actualizando con las recompensas obtenidas de la exploración del agente durante el entrenamiento.

El problema de este algoritmo es que se vuelve inabordable cuando el MDP que lo representa contiene un espacio de estados-acciones demasiado grande como para ser representado por tablas, en otras palabras, este algoritmo no es escalable. Por ello, con la llegada del aprendizaje por refuerzo profundo surge el algoritmo *Deep Q-Learning* (DQL) Mnih et al. (2013), en el que se reemplaza la tabla Q por una red neuronal, también conocida como *Deep Q-Network* (DQN), para aproximar los diferentes valores Q para cada acción posible dado un estado, la cual se irá entrenando para obtener cada vez mejores resultados. Esta red neuronal usa como entrada el estado actual en el que se encuentra el agente, normalmente representado mediante una o varias imágenes, e incluso también se puede usar como entrada algún estado anterior para poder capturar información temporal. La entrada es procesada por capas densas o capas convolucionales, en el caso de que se utilicen imágenes, para extraer las características necesarias, y las capas finales serán capas densas que calcularán los valores Q para cada una de las acciones posibles.

Algorithm 1 Deep Q-learning with Experience Replay

```

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

```

Figura 3.2: Pseudocódigo del algoritmo Deep Q-Learning.

Como podemos ver en el pseudocódigo anterior, el algoritmo de entrenamiento de *Deep Q-Learning* consta principalmente de dos fases:

- **Muestreo:** Mediante una selección de acción ϵ -greedy, es decir, elegimos con una probabilidad ϵ una acción al azar en vez de la mejor acción, para mantener un equilibrio entre explotación y exploración. Este enfoque permite que el agente explore nuevas acciones que pueden llevar a mejores recompensas a largo plazo. Realizamos la acción seleccionada y observamos la recompensa y el nuevo estado para almacenarlo como una experiencia en forma de tupla (estado, acción, recompensa, siguiente estado).
- **Entrenamiento:** Seleccionamos de manera aleatoria un pequeño lote de tuplas de experiencias pasadas y utilizamos estas para actualizar la red neuronal. El objetivo es minimizar la diferencia entre el valor Q actual, valor estimado por la red neuronal para una acción específica en un estado, y el valor Q objetivo, valor correcto que se quiere que la red aprenda, calculado usando las recompensas inmediatas y los valores Q futuros, normalmente estimados por una *target network*, utilizando un factor de descuento para dar mayor importancia a las recompensas más cercanas.

Además, para ayudar a estabilizar el entrenamiento, se utilizan técnicas como *Experience Replay* o repetición de experiencias. Durante el muestreo, se usa un buffer para guardar las tuplas de las experiencias obtenidas por el agente durante el entrenamiento (estado, acción, recompensa, siguiente estado), para ser reutilizadas posteriormente durante el entrenamiento, evitando que estas experiencias sean descartadas tras ser utilizadas para actualizar la red neuronal.

Otra técnica importante es la llamada *Double DQNs*, con la que se hace uso de una red objetivo o *target network*. Esta es una copia de la DQN principal que se sincroniza periódicamente y se utiliza para calcular los valores objetivos durante el entrenamiento. Al usar una red separada para estimar los valores objetivos, se estabiliza el entrenamiento al proporcionar una referencia más estable, lo que reduce las oscilaciones y la divergencia en las actualizaciones de la red principal.

3.1.1.2. Proximal Policy Optimization

El algoritmo *Proximal Policy Optimization* (PPO) [Schulman et al. \(2017\)](#), al contrario que el algoritmo DQL, es un algoritmo *on-policy*. Esto significa que evalúa y mejora la misma política que se está utilizando para interactuar con el entorno y generar una secuencia de transiciones, anteriormente llamadas experiencias (estado, acción, recompensa, siguiente estado), solo que ahora son almacenadas en orden de obtención. Durante el entrenamiento, se recopila una trayectoria, que es el conjunto de transiciones generadas al interactuar con el entorno utilizando la política actual, asegurando que las actualizaciones de la política dependen únicamente de los datos obtenidos con el uso de esa misma política.

Además, PPO es un algoritmo *policy-based*, lo que significa que su objetivo es aprender directamente sobre la política, a diferencia de los algoritmos *value-based* como DQL, que buscan aprender una función Q. Esto permite su aplicación en problemas con espacios de acciones

muy grandes, siendo incluso compatibles con espacios de acciones continuas, mientras que los algoritmos *value-based* son más comúnmente usados en espacios de acciones discretas.

PPO hace uso de una red neuronal que representa la política, con la que, dado un estado devuelve una distribución de probabilidades sobre las acciones posibles en ese estado. Sobre esta distribución se realiza un muestreo para seleccionar la acción a aplicar, introduciendo así el factor de exploración. El objetivo principal de PPO es mejorar la política actualizando los parámetros de la red neuronal de manera que se incremente la probabilidad de seleccionar acciones que conduzcan a mayores recompensas. Esto se logra mediante el cálculo del ascenso del gradiente basado en la probabilidad de la acción seleccionada y la recompensa obtenida, buscando así maximizar el retorno esperado, por ello este algoritmo forma parte de la familia de algoritmos *policy gradient*.

Centrándonos en el algoritmo PPO, este puede considerarse una evolución natural de los algoritmos de la familia *policy gradient*, ya que utiliza el ascenso del gradiente, pero añadiendo la mejora de controlar las actualizaciones de la política para que no sean demasiado agresivas. Para lograr esto, PPO emplea un mecanismo de recorte mediante el uso de una función objetivo recortada llamada *Clipped Surrogate Objective*. Esta nueva función objetivo utiliza un rango $[1-\epsilon, 1+\epsilon]$, controlado por el hiperparámetro ajustable ϵ , para recortar el ratio de cambio de la nueva política respecto a la anterior. Este enfoque evita que el proceso de aprendizaje se aleje demasiado de la política anterior, estabilizando el entrenamiento y asegurando que las actualizaciones sean graduales y más seguras.

Este algoritmo pertenece también a la rama del DRL, ya que utiliza redes neuronales para su funcionamiento. Más concretamente, PPO es un algoritmo de tipo Actor-Critic, en el que se emplean dos redes neuronales: el Actor y el Critic. Estas redes interactúan continuamente durante el entrenamiento, siendo el Actor el encargado de optimizar la política del agente y el Critic el encargado de evaluar el valor de los estados, para mejorar el rendimiento del agente en el entorno:

- **Policy Network:** Esta red neuronal corresponde al Actor y predice la distribución de probabilidades de las acciones dada la observación de un agente. Mejorar esta red es el objetivo principal del algoritmo, ya que representa la política que el agente seguirá en la interacción con el entorno durante el entrenamiento.
- **Value Function Network:** Esta red neuronal corresponde al Critic y predice la función de valor para un estado dado, que representa cuán bueno es estar en ese estado particular considerando las recompensas futuras esperadas. La función de valor se utiliza para calcular los estimadores de ventaja o *advantage estimates*, los cuales representan la ventaja de tomar una acción en comparación con el valor promedio esperado del estado. Durante el entrenamiento, estos estimadores se utilizan en la función objetivo para reducir la varianza en las actualizaciones de la política, lo que contribuye a un entrenamiento más estable y eficiente.

Algorithm 1 PPO-Clip

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

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

typically via stochastic gradient ascent with Adam.

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

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

typically via some gradient descent algorithm.

- 8: **end for**

Figura 3.3: Pseudocódigo Proximal Policy Optimization - Clip.

Al igual que en el algoritmo DQL, en este pseudocódigo también se pueden diferenciar principalmente dos fases:

- **Muestreo:** Se recopilan un conjunto de transiciones (estado, acción, recompensa, nuevo estado) mediante la ejecución de la política actual en el entorno, para ello se debe determinar el número de transiciones a recopilar. Con las transiciones obtenidas y la función de valor actual se calculan los estimadores de ventaja o *advantage estimates*.
- **Entrenamiento:** Una vez obtenidos los estimadores de ventaja, se actualiza la *policy network* maximizando la función objetivo mediante ascenso del gradiente estocástico. Posteriormente, se ajusta la *value function network*, con el propósito de mejorar la estimación de la función de valor actual.

3.1.2. Aprendizaje por Refuerzo Multiagente

Cuando en un problema de aprendizaje por refuerzo hay múltiples agentes, estamos frente a un problema de *Multi-Agent Reinforcement Learning* o Aprendizaje por Refuerzo Multiagente (MARL). En este tipo de problemas, cada agente toma decisiones e interactúa en un entorno compartido con otros agentes, donde las acciones que realiza un agente influyen en el entorno, afectando potencialmente las observaciones y recompensas de los demás agentes. En otras palabras, las decisiones de cada agente pueden impactar directa o indirectamente a los otros agentes.

Dentro de los problemas MARL, principalmente podemos encontrar tres tipos:

- **Problemas cooperativos:** Todos los agentes tienen un objetivo común y colaboran para alcanzarlo.
- **Problemas competitivos:** Los agentes compiten entre sí, cada uno buscando su propio beneficio.
- **Problemas mixtos:** Combinan elementos de cooperación y competitividad.

Para resolver problemas dentro del campo del MARL, surgen tres métodos principales según la manera en la que se toman las decisiones de los distintos agentes:

- **Control Centralizado:** La toma de decisiones para todos los agentes depende de un controlador central. Este controlador recibe las observaciones de todos los agentes y, basándose en esta información global del entorno, decide la mejor acción para cada agente o la acción que maximice la recompensa global. Esta alternativa es útil cuando la coordinación entre los agentes es esencial, pero se vuelve poco escalable a medida que aumenta el número de agentes. En la Figura 3.4 se muestra el esquema del paradigma *Centralized Training and Centralized Execution* (CTCE), un ejemplo de control totalmente centralizado durante el entrenamiento y la ejecución.
- **Control Descentralizado:** Cada agente actúa como su propio controlador y toma decisiones basadas en su observación local del entorno. A diferencia del control centralizado, este puede tener dificultades en la coordinación de múltiples agentes, ya que cada uno tiene su propia observación y no conoce el estado de los demás. Sin embargo, este problema podría resolverse con mecanismos de comunicación entre agentes cercanos si fuese necesario. La ventaja de esta alternativa es su alta escalabilidad a un mayor número de agentes y la reducción del tiempo de ejecución, ya que cada agente maneja menos información y la ejecución individual es más ligera que en los controladores centralizados. En la Figura 3.4 se puede apreciar un ejemplo de control descentralizado con el esquema del método *Decentralized Training and Decentralized Execution* (DTDE).
- **Control Híbrido:** Este método combina características de los controles centralizado y descentralizado. Un ejemplo de este tipo es el esquema del *Centralized Training and Decentralized Execution* (CTDE) mostrado en la Figura 3.4, que aprovecha las ventajas del control centralizado durante la fase de entrenamiento, utilizando la información global del entorno o la información obtenida por todos los agentes, y aplica un control descentralizado, durante la ejecución, en cada agente basándose únicamente en la información local disponible para cada uno [Saifullah et al. \(2024\)](#).

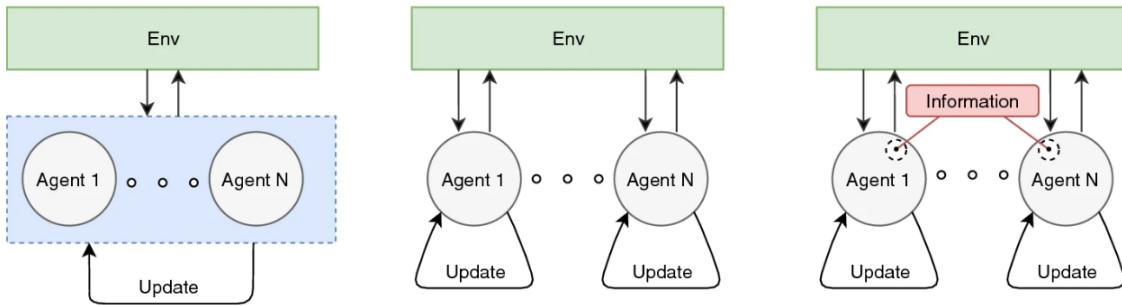


Figura 3.4: En esta imagen se pueden apreciar los tres paradigmas de entrenamiento y ejecución en entornos multiagente. A la izquierda, se representa el paradigma de CTCE, donde todos los agentes siguen una única política centralizada. En el centro, el paradigma DTDE muestra cómo cada agente desarrolla y sigue su propia política descentralizada basada en su observación local del entorno. A la derecha, el paradigma CTDE describe un enfoque de entrenamiento centralizado, donde los agentes utilizan información global del entorno proporcionada por los demás agentes durante el entrenamiento, pero luego cada agente ejecuta su propia política basada únicamente en información local.

Además de decidir el tipo de toma de decisiones que se desea utilizar, también se pueden implementar diversas soluciones según la secuencia en la que los agentes realizarán sus acciones. A continuación, se describen los principales tipos:

- **Simultánea:** Los agentes toman decisiones y ejecutan acciones al mismo tiempo. Esta metodología puede llevar a conflictos y colisiones si no se mantiene una correcta coordinación entre los agentes.
- **Cíclica o Por Turnos:** Los agentes toman decisiones y realizan acciones uno tras otro en un orden predefinido, repitiendo el ciclo continuamente.
- **Condicional:** Los agentes toman decisiones sobre las acciones a realizar basándose en condiciones específicas del entorno o en el estado de otros agentes.
- **Asíncrona:** Cada agente toma decisiones y ejecuta acciones de manera independiente y no coordinada en el tiempo con los demás agentes. Al igual que con la ejecución simultánea, esta opción puede generar caos y conflictos entre los agentes.
- **Prioritaria:** Cada agente tiene asignada una prioridad que determina el orden de ejecución, de modo que los agentes con mayor prioridad ejecutan sus acciones antes que aquellos con menor prioridad.

Esto proporciona una amplia libertad de elección, permitiendo combinar diferentes métodos de toma de decisiones con diversos tipos de secuencias de acciones, lo que resulta en una gran variedad de algoritmos para resolver este tipo de problemas multiagente. Sin embargo,

es crucial realizar un estudio exhaustivo del problema que queremos abordar antes de elegir qué tipo de algoritmo desarrollar, para seleccionar la combinación más adecuada según la naturaleza del problema.

Es importante destacar que, al igual que con el aprendizaje por refuerzo, el avance significativo en el campo del aprendizaje profundo ha permitido combinar algoritmos de MARL con el uso de redes neuronales profundas, dando origen a una nueva área conocida como *Multi-Agent Deep Reinforcement Learning* o Aprendizaje por Refuerzo Profundo Multiagente (MADRL) [Stanford \(2016\)](#). Dentro de esta nueva rama han surgido variantes de los algoritmos ya mencionados anteriormente. Por ejemplo, los algoritmos *Multi-Agent Proximal Policy Optimization* (MAPPO) e *Independent Proximal Policy Optimization* (IPPO) han sido diseñados para resolver problemas colaborativos entre múltiples agentes con una recompensa compartida [Yu et al. \(2022\)](#). Además, se han explorado combinaciones como el uso del algoritmo PPO con una arquitectura de toma de decisiones híbrida como CTDE [Chen et al. \(2023\)](#), así como la aplicación de la variante *Rainbow* [Hessel et al. \(2017\)](#) del algoritmo DQL en problemas multiagente con un control centralizado en la toma de decisiones y una secuencia de acciones cíclica para los agentes [Lu et al. \(2024\)](#).

Gracias a los avances en el marco teórico del aprendizaje por refuerzo y del aprendizaje profundo, ahora somos capaces de resolver problemas complejos que antes eran inabordables debido a la poca eficiencia de los algoritmos existentes. Sin embargo, estas ramas de la inteligencia artificial continúan evolucionando rápidamente, lo que requiere una investigación constante en busca de mejoras y nuevas alternativas. Una vez introducido el estado del arte y el marco teórico, nos centraremos en el caso de estudio, brevemente introducido en los objetivos, describiendo detalladamente el problema a resolver, explicando el desarrollo de la simulación e introduciendo los algoritmos que serán desarrollados para resolver el problema en cuestión.

Caso de Estudio

4

4.1. Entorno Lunar

En este caso de estudio, nos enfrentamos al desafío de operar en un entorno lunar en el que encontramos varios *lunar rovers* diseñados y equipados con herramientas específicas para la extracción de un material designado. La operación consiste en que cada *lunar rover* se debe dirigir a la mina correspondiente para extraer el material asignado, y luego tiene que transportar el material obtenido a una mezcladora central, que sirve como punto de recolección y procesamiento de los distintos materiales recolectados por todos los agentes.

Cada *lunar rover* deberá planificar su ruta paso a paso desde su posición inicial hasta la mina asignada, y posteriormente hasta la mezcladora, con el objetivo de minimizar el consumo de energía y extender al máximo posible la vida operativa de su batería. Dentro del entorno lunar, los *lunar rovers* se encontrarán con diversos obstáculos, desde obstáculos pequeños que pueden sobrepasar con un mayor gasto de energía, hasta obstáculos grandes difícilmente superables que podrían causar daños graves dejando al *lunar rover* inutilizable, algo muy costoso en un entorno tan remoto. Además, la presencia de otros agentes en el mismo entorno lunar añade un nivel adicional de complejidad, ya que estos agentes también representan un obstáculo para los demás, surgiendo un riesgo crítico de colisión que podría dañar a varios *lunar rovers* simultáneamente.

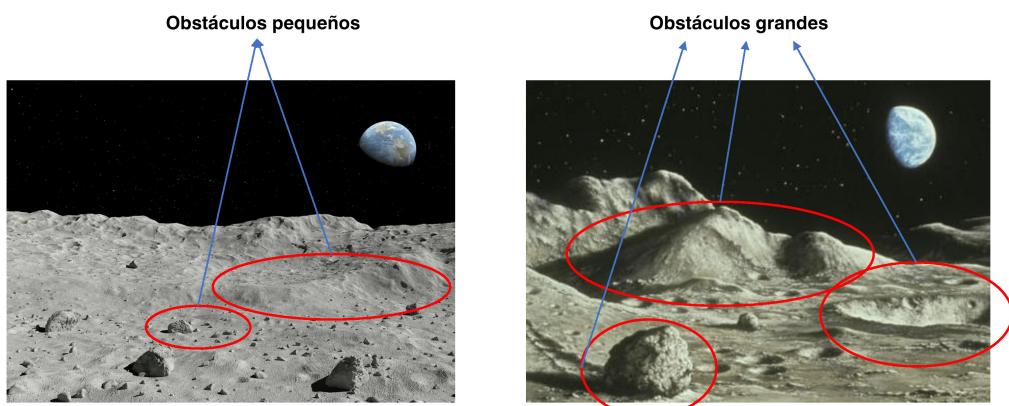


Figura 4.1: Diferentes tipos de obstáculos en el entorno lunar.

Cabe destacar que la planificación de la ruta no puede ser calculada de antemano debido a la complejidad del entorno lunar y la limitada capacidad de los sensores de los *lunar rovers*, que no proporcionan una visión completa del terreno ni de todos los obstáculos presentes en este, sino que mantiene un rango de visión que les permite observar tan solo parte del terreno que les rodea. Sin embargo, se podría asumir que los *lunar rovers* tendrán conocimiento previo de la ubicación exacta de su mina asociada y de la mezcladora, ya que es realista suponer que estos puntos estratégicos son conocidos de antemano. Aun así, la decisión de si el agente tiene el conocimiento a priori de estas ubicaciones principales o si debe descubrirlas mediante la exploración del terreno es un aspecto que será desarrollado para que pueda ser configurable a la hora de crear una simulación del entorno lunar.

4.2. Simulación

Para el desarrollo de la simulación se ha utilizado la biblioteca *Gymnasium Foundation* y *OpenAI* (2023), originalmente desarrollada por *OpenAI* y actualmente mantenida por la *Fundación Farama*, una organización sin ánimo de lucro dedicada al desarrollo y mantenimiento de herramientas de aprendizaje por refuerzo de código abierto. Esta biblioteca ofrece una robusta API para la creación de entornos de simulación en el ámbito del aprendizaje por refuerzo, incluyendo numerosas implementaciones de entornos ya desarrollados, como una extensa colección de juegos de *Atari*. Además de las implementaciones preexistentes, *Gymnasium* permite la creación de entornos personalizados. En nuestro caso, hemos optado por esta alternativa debido a la falta de entornos predefinidos que se asemejen al entorno lunar específico de nuestro problema.

Para la creación de un entorno personalizado en *Gymnasium*, es fundamental mantener una estructura específica que asegure la funcionalidad de la API de la biblioteca. Esta estructura consta de una clase principal que debe heredar de la clase abstracta *gymnasium.Env* proporcionada por la biblioteca. Luego, dentro de esta clase se deben implementar 4 métodos principales que controlarán la simulación:

- ***__init__***: Este es un método especial que se llama automáticamente al crear un objeto a partir de una clase. En nuestro caso, este método se utiliza para inicializar los atributos que representan las características del entorno, como el tamaño del mapa del entorno y el número de *lunar rovers* que contendrá la simulación.
- ***reset***: El método *reset* debe ser llamado siempre antes de iniciar un nuevo episodio para realizar la configuración inicial del entorno. En nuestro caso, este método incluye la generación aleatoria de obstáculos y la ubicación variable de los *lunar rovers*, las distintas minas y la mezcladora, añadiendo así factores estocásticos de aleatoriedad e incertidumbre para representar adecuadamente las condiciones reales que enfrentarán los *lunar rovers* en la Luna.

- **step:** Este método contiene la mayor parte de la lógica del entorno, donde dada una acción, se ejecuta sobre el entorno y se devuelve el nuevo estado calculado. El cálculo del nuevo estado tiene en cuenta toda la lógica del entorno, incluyendo las restricciones del problema, las recompensas obtenidas por la acción, y si la acción ha llevado a un estado final donde se ha resuelto el problema o terminado el juego. En nuestro caso, este método ha sido adaptado para entornos multiagente, permitiendo recibir una lista de acciones, cada una asociada a un *lunar rover*, para que cada agente pueda realizar su acción de manera independiente en el entorno. Para ello, se ha creado una clase que representa a cada *lunar rover* de manera independiente al entorno y a los demás agentes, la cual incluye su propio método *step* en el que se calcula el nuevo estado y las recompensas obtenidas por el agente al realizar una acción.

- **render:** Finalmente, el método *render*, aunque opcional, es un método altamente recomendable utilizado para renderizar el entorno. En nuestro caso, se ha utilizado la biblioteca *pygame* [Dudfield et al. \(2000\)](#), enfocada en el desarrollo de videojuegos, para crear una visualización del entorno que se actualiza con cada movimiento realizado. Esto facilita enormemente la fase de experimentación, permitiendo analizar visualmente la evolución de los algoritmos durante y después del entrenamiento, y haciendo más sencillo identificar puntos de mejora en las acciones elegidas por los algoritmos.

El estado del problema se representará mediante una matriz de tamaño $n \times n$ que modelará el entorno lunar, donde n es un parámetro configurable previamente a la creación del entorno. Como se mencionó anteriormente, la ubicación de los obstáculos, agentes, minas y la mezcladora se determinará de manera aleatoria para mantener la complejidad del entorno lunar y poder utilizar una amplia variedad de entorno durante el entrenamiento. La lógica asociada a la distribución de los obstáculos garantizará que haya más obstáculos pequeños que grandes, apareciendo ambos en proporción directa al tamaño de la matriz.

El número de *lunar rovers* y, con ello, de minas asociadas, también será un parámetro configurable antes de la creación del entorno. Adicionalmente, se deberá especificar el rango de visión, en número de casillas, que cada *lunar rover* tendrá dentro del entorno lunar. Este rango de visión afecta directamente la observación del agente, ya que determinará cuántos obstáculos y elementos podrá ver a su alrededor en cada momento. Una mayor visión permitirá al *lunar rover* obtener más información del entorno, facilitando la toma de decisiones de movimientos al evitar o enfrentar obstáculos.

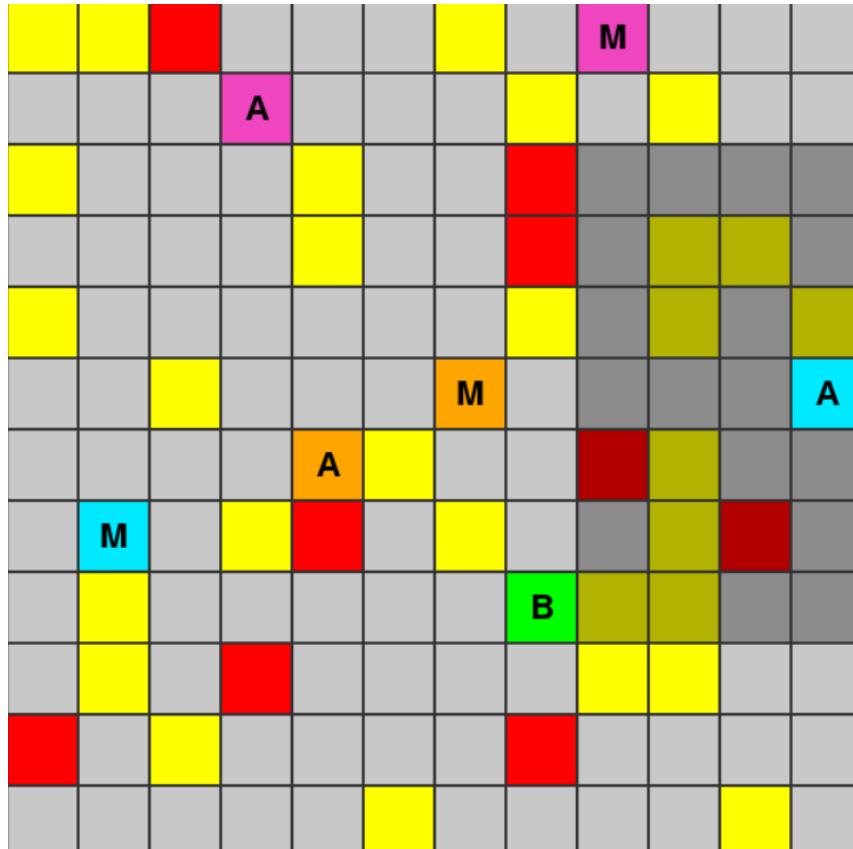


Figura 4.2: Captura del entorno renderizado por pygame donde los espacios vacíos están representados por casillas grises, la mezcladora o blender está simbolizada por una casilla verde con una 'B', los obstáculos pequeños se encuentran en las casillas amarillas y los obstáculos grandes en las casillas rojas. Cada lunar rover se denota con una letra 'A', de agente, y se distinguen por un color específico que a la vez se utiliza para representar la mina correspondiente al lunar rover marcada con una 'M' en una casilla del mismo color. Las casillas oscurecidas alrededor del agente azul muestran el rango de visión de este, estando configurado un rango de visión de 3 casillas para este ejemplo.

Cada *lunar rover* tiene la capacidad de moverse en cuatro direcciones: arriba, abajo, izquierda y derecha, siempre y cuando la dirección elegida no lleve al agente a una posición fuera del mapa, o también puede no realizar movimiento alguno y mantenerse en su misma posición. Cada vez que un *lunar rover* realiza un movimiento, el entorno se actualiza y se calcula la recompensa obtenida por el agente según la nueva situación en la que se encuentra. Esta recompensa puede ser tanto negativa, representando una situación desfavorable para el *lunar rover*, como positiva, indicando una situación beneficiosa. A continuación, se detallan las distintas situaciones en las que un *lunar rover* obtiene una recompensa y su valor asociado:

Situación	Recompensa
Esperar	- 2
Moverse	- 1
Sobrepasar Obstáculo pequeño	- 3
Sobrepasar Obstáculo grande	- 25
Chocar con otro <i>lunar rover</i>	- 50
Descubrir la posición de su mina asociada	50
Descubrir la posición de la mezcladora	50
Llegar a la mina por primera vez	100
Llegar a la mezcladora tras minar	700

Las recompensas negativas están asociadas al gasto de energía que conlleva cada acción para el *lunar rover*, es obvio que si el *lunar rover* decide esperar en su posición, el gasto de energía es menor en comparación con superar algún obstáculo, pero también se espera que el agente explore el entorno, por lo que la recompensa por mantenerse quieto es algo menor que la de moverse a un espacio seguro, reforzando de esta manera la exploración segura. Además, se ha otorgado una gran importancia a evitar los choques entre *lunar rovers*, imponiendo una recompensa negativa elevada en caso de colisión. Esto se debe a que una colisión entre múltiples agentes puede dejar a estos inutilizados, lo cual es una situación que debe evitarse a toda costa. Por otro lado, las recompensas positivas están ligadas a los objetivos del agente, siendo el primer objetivo el de llegar a su mina correspondiente para obtener el material y tras ello trasladarlo a la mezcladora, aunque también se otorga gran importancia a la exploración otorgando una recompensa positiva a los agentes cuando descubren las posiciones relevantes del problema.

Como se mencionó anteriormente, este problema ofrece dos alternativas en cuanto al tratamiento de la posición de las minas y la mezcladora por parte de los *lunar rovers*. Una posibilidad es que los *lunar rovers* conozcan de antemano las ubicaciones de estos elementos, mientras que la otra es que las descubran cuando se hagan visibles dentro de su rango de visión mediante la observación del entorno. Finalmente, se ha decidido incluir ambas opciones, permitiendo que durante la configuración inicial del entorno se pueda elegir entre estas dos alternativas. Esto dota al entorno de una mayor configurabilidad, lo que nos permitirá utilizar y comparar estas dos opciones durante la fase de experimentación. Durante la fase de entrenamiento se hará uso de la opción en la que los *lunar rovers* no saben de antemano las ubicaciones de los objetivos, lo que llevará a una mayor exploración durante los entrenamientos, dotando de experiencias y trayectorias más diversas. Una vez completados los entrenamientos se realizarán pruebas con ambas configuraciones, comprobando si mejora o no la eficiencia de los agentes cuando estos conocen las posiciones de los objetivos a priori.

Es importante mencionar que, dado que la biblioteca *gymnasium* está diseñada para la implementación de entornos con un único agente, no es compatible con una secuencia de ejecución asíncrona para todos los agentes, la cual no depende de ningún tipo de coordinación entre ellos. Por ello, se ha optado por una secuencia de ejecución cíclica, en la que se debe

suponer que en el entorno lunar habrá algún tipo de coordinación entre todos los *lunar rovers* para respetar los turnos de ejecución de cada uno.

Por otro lado, se ha desarrollado un control descentralizado, en el que cada agente tomará sus propias decisiones basándose en su observación local del entorno, según el rango de visión asignado e información previamente obtenida en observaciones anteriores. Así, aunque se utilice una secuencia de ejecución cíclica, la toma de decisiones sigue siendo totalmente individual e independiente para cada *lunar rover*. Esto respeta las condiciones del problema original, donde cada *lunar rover* debe operar de manera autónoma sin un control centralizado.

Por último, cabe destacar que, de manera complementaria, se ha desarrollado una interfaz gráfica con la biblioteca *tkinter* [Lumholt et al. \(2000\)](#) que muestra información detallada sobre la simulación, facilitando así una mejor comprensión del estado de la misma. Mientras que el renderizado del mapa solo muestra la posición de los objetos, en esta interfaz gráfica, se puede incluir tanto información general sobre la simulación como datos específicos de cada *lunar rover*, como, por ejemplo, la recompensa acumulada por cada uno. El desarrollo de esta interfaz se ha diseñado de forma simplificada, con la idea de poder ser modificada fácilmente durante la fase de experimentación según el tipo de pruebas que se deseen realizar y la información útil que se quiera mostrar. Esto permitirá ajustar la interfaz para que sea más adecuada a las necesidades de análisis y comprensión en cada situación específica.

4.3. Algoritmos

Una vez comprendidas las características específicas del problema que enfrentamos y analizados los algoritmos más utilizados en el estado del arte para tareas similares y con elementos comunes a nuestro caso, se ha llevado a cabo la selección y adaptación de los algoritmos que, en teoría, podrían ofrecer el mejor rendimiento en función de nuestros requisitos.

Antes de abordar los dos algoritmos desarrollados, es importante destacar que el proceso de desarrollo no ha sido lineal, sino más bien un proceso iterativo e incremental. En este proceso, los algoritmos se han desarrollado progresivamente, comenzando con aspectos básicos y avanzando hacia los más complejos. A lo largo de este desarrollo, se han probado y evaluado diferentes componentes para determinar si su inclusión mejoraba o no el rendimiento, lo que ha llevado a la incorporación de algunos elementos no previstos inicialmente y al descarte de otros que, en un principio, se consideraban esenciales.

Este ciclo continuo de desarrollo y pruebas no solo ha permitido mejorar el rendimiento de los propios algoritmos, sino que también ha revelado aspectos de la simulación que debían ser añadidos para agilizar los entrenamientos y que no se habían considerado previamente. El primero de estos aspectos fue la inclusión de un mapa de visitas en el que los *lunar rovers* llevan el registro de las veces que han visitado las posiciones que se encuentran dentro de su observación. Esta necesidad surgió durante las primeras pruebas de entrenamiento, en las

cuales muchos agentes, en busca de encontrar los puntos estratégicos por primera vez, quedaban atrapados en ciclos de exploración sobre zonas ya exploradas previamente. Gracias a este mapa de visitas los modelos son capaces de identificar cuando el agente se encuentra en áreas ya exploradas y dirigirlo hacia zonas sin visitar o menormente exploradas. Este ajuste no solo mejora significativamente el rendimiento del entrenamiento para el problema que abordamos, sino que también representa un aspecto realista que podría implementarse en los *lunar rovers* reales y que sería útil para otro tipo de tareas desarrolladas por estos en el entorno lunar.

También se identificó la necesidad de añadir pequeñas recompensas intermedias que guíen al agente hacia sus objetivos. Inicialmente, la variedad de recompensas positivas era muy limitada, ya que solo se otorgaban cuando las posiciones objetivo eran descubiertas o alcanzadas, provocando que durante las primeras fases del entrenamiento, en las que los movimientos son mayoritariamente aleatorios, los algoritmos recibiesen recompensas negativas de forma constante sin apenas obtener recompensas positivas, debido a la dificultad de alcanzar el objetivo por azar o con políticas primarias. Esto, sumado a una limitación agregada en el número de pasos permitidos por episodio para evitar el sobreentrenamiento en situaciones específicas, hacía que mayoritariamente se concluyeran los episodios sin cumplir los objetivos, resultando en un aprendizaje prácticamente nulo.

Para resolver esto, se añadió una pequeña recompensa que el agente va obteniendo conforme se va acercando a su objetivo, siempre que este se encuentre dentro del rango de visión del agente. En este caso, si el agente aún no ha extraído mineral, se considera la distancia a su propia mina, mientras que si ya ha extraído el mineral y lo está transportando, se tiene en cuenta la distancia a la mezcladora. Esta recompensa se basa en el potencial atractivo que podemos encontrar dentro de un campo de potencial artificial [Yao et al. \(2020\)](#), sirviendo como incentivo a mantener la dirección correcta, ya que esta pequeña recompensa aumenta a medida que el agente reduce la distancia al objetivo.

Adicionalmente, se decidió normalizar la observación de cada *lunar rover* de manera que cada uno solo distinga su propia mina, mientras que todas las demás minas y *lunar rovers* son considerados indistinguibles entre ellos. Esta normalización permite utilizar conjuntamente las experiencias recolectadas por todos los agentes utilizados durante el entrenamiento, aumentando la variedad de experiencias y ampliando así la capacidad de generalización de la red, permitiendo su uso en simulaciones con características diferentes a las utilizadas durante el entrenamiento, como en mapas de distintos tamaños o con un número diferente de agentes.

Una vez explicadas las mejoras aplicadas a la simulación que surgieron durante el desarrollo y prueba de los algoritmos, se puede revelar que los algoritmos seleccionados fueron *Double Dueling Deep Q-Learning* (DDDQL) [Huang et al. \(2018\)](#) y *Multi-Agent Proximal Policy Optimization* (MAPPO) [Yu et al. \(2022\)](#). Ambos algoritmos fueron elegidos por su capacidad de abordar problemas multiagente desde enfoques diferentes, lo que permitirá realizar una comparación enfrentada de sus rendimientos.

La implementación de ambos algoritmos ha sido realizada desde cero, sin utilizar bibliotecas con implementaciones predefinidas, dada la necesidad de adaptar estas implementaciones a las características específicas del problema a resolver. Se ha optado por el uso de la biblioteca *numpy* Oliphant (2006) para mejorar la eficiencia en el procesamiento y la manipulación de los datos numéricos, mientras que se han utilizado las bibliotecas de *tensorflow* Team (2015) y *keras* Chollet (2015) para la construcción y optimización de las redes neuronales, habilitando además su ejecución en GPU.

4.3.1. Double Dueling DQL

El algoritmo *Double Dueling Deep Q-Learning* (DDDQL) es una evolución sobre el algoritmo base DQL, explicado anteriormente en la sección del estado del arte 3.1.1.1, en la que se mejora su rendimiento mediante la incorporación de dos elementos clave que se reflejan en su propio nombre:

- **Double:** Esta técnica aborda uno de los principales problemas del DQL estándar, que es la sobreestimación de los valores de Q, provocada por utilizar la misma red neuronal tanto para seleccionar la acción como para evaluar su valor. El nuevo algoritmo DDDQL resuelve este problema separando la selección y la evaluación de la acción en dos redes neuronales diferentes. Primero, la red neuronal principal selecciona la acción que considera mejor, para que posteriormente, la red objetivo evalúe el valor de la acción seleccionada. La red objetivo es una copia de la red principal que se actualiza periódicamente, según una frecuencia configurada por un hiperparámetro, lo que mejora la estabilidad y precisión del entrenamiento, evitando cambios abruptos en la política.

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

```

Initialize primary network  $Q_\theta$ , target network  $Q_{\theta'}$ , replay buffer  $\mathcal{D}$ ,  $\tau \ll 1$ 
for each iteration do
    for each environment step do
        Observe state  $s_t$  and select  $a_t \sim \pi(a_t, s_t)$ 
        Execute  $a_t$  and observe next state  $s_{t+1}$  and reward  $r_t = R(s_t, a_t)$ 
        Store  $(s_t, a_t, r_t, s_{t+1})$  in replay buffer  $\mathcal{D}$ 
    for each update step do
        sample  $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$ 
        Compute target Q value:
            
$$Q^*(s_t, a_t) \approx r_t + \gamma \max_{a'} Q_{\theta'}(s_{t+1}, a')$$

        Perform gradient descent step on  $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$ 
        Update target network parameters:
            
$$\theta' \leftarrow \tau * \theta + (1 - \tau) * \theta'$$


```

Figura 4.3: Pseudocódigo del algoritmo Double DQL, en el que se muestra cómo se usa la red objetivo en la ecuación de Bellman para calcular el valor objetivo de Q, en contraste con la ecuación utilizada en el algoritmo DQL básico 3.2.

- **Dueling:** El segundo componente clave es la arquitectura Dueling, que modifica la red neuronal al dividir la predicción de los valores Q en dos partes: el valor del estado actual y la ventaja de cada acción en ese estado. Esta arquitectura permite que la red neuronal aprenda de manera más eficaz cuál es la importancia relativa de cada acción en función del estado actual. En lugar de calcular directamente los valores Q para cada acción como en el algoritmo base, la red calcula primero el valor del estado en el que se encuentra el agente y luego determina cuánto mejor es cada acción en comparación con las demás en dicho estado. Estos dos valores se combinan para calcular los valores Q finales utilizando la siguiente fórmula:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha))$$

Figura 4.4: Fórmula para el cálculo de los Q valores en el algoritmo Dueling DQL.

En la siguiente imagen se compara la arquitectura de la red neuronal utilizada en el algoritmo DQL básico con la evolución utilizada en el algoritmo Dueling DQL. Se puede observar que ambas redes comparten la misma base en la que se procesa la entrada, diferenciándose únicamente en la salida, en la que la red de la arquitectura Dueling DQN se divide en dos ramas, una para calcular el valor del estado y otra para calcular la ventaja de cada acción. Finalmente, estos dos cálculos se combinan para producir los valores Q finales utilizando la fórmula vista en la imagen anterior.

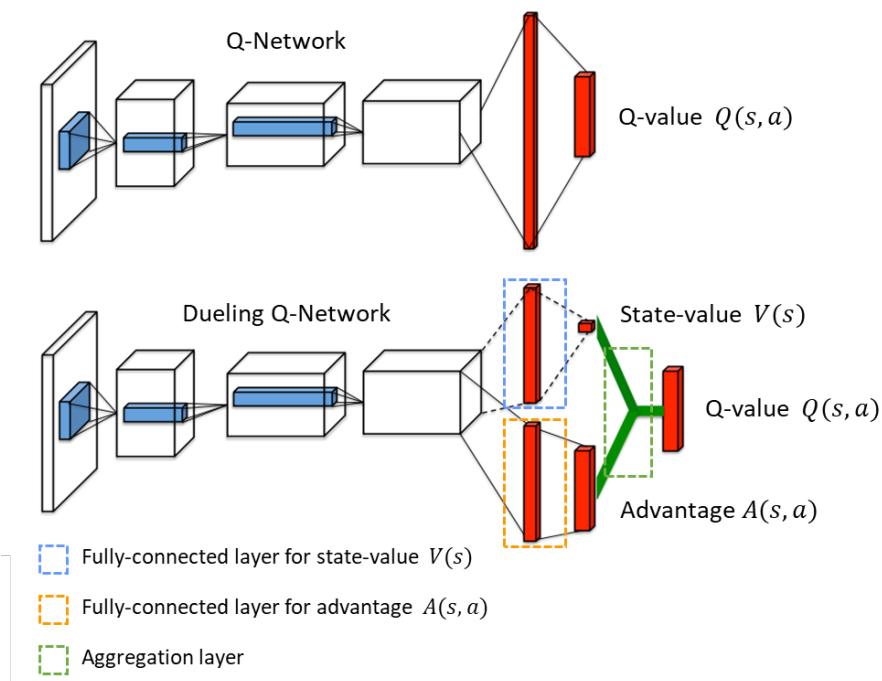


Figura 4.5: Comparación de las arquitecturas de redes neuronales utilizadas en el algoritmo base DQL (arriba) y su evolución Dueling DQL (abajo).

Esta mejora permite a la red neuronal discernir de manera más efectiva cuál es la acción más adecuada en función de todas las demás acciones disponibles y el valor del estado actual, añadiendo una capa adicional de información que la red DQN del algoritmo DQL básico no considera.

Aunque el algoritmo DQL y sus evoluciones Double DQL y Dueling DQL fueron diseñadas para problemas con un único agente, la implementación desarrollada ha sido modificada para adaptarse a nuestro problema. En nuestro caso de estudio, los agentes son homogéneos, es decir, comparten habilidades, mismos movimientos y rango de visión, y objetivos, minar y llevar el mineral a la mezcladora, aunque partiendo de posiciones iniciales diferentes y con distintas posiciones objetivo. Es por esto que es posible utilizar una única política para todos ellos, eliminando la necesidad de entrenar una política específica para cada agente.

Además, dado que DQL es un algoritmo *off-policy*, que no sigue una política específica y puede utilizar experiencias adquiridas con políticas distintas para encontrar la política óptima, se ha adoptado un paradigma de *Centralized Training and Decentralized Execution* (CTDE) en el que, durante el entrenamiento, se utiliza un *Experience Replay Buffer* centralizado que almacena las experiencias recolectadas por todos los agentes. En cada paso del entrenamiento, se muestrea un conjunto aleatorio de estas experiencias guardadas para actualizar la red, proporcionando una capa ligera de centralización durante el entrenamiento, pero manteniendo la ejecución descentralizada, donde cada agente utiliza únicamente su propia observación para tomar decisiones.

Finalmente, se ha añadido un decaimiento de épsilon o *epsilon decay* en la estrategia de exploración ϵ -greedy, es decir, en la probabilidad de seleccionar una acción aleatoria dentro de las disponibles en vez de seleccionar la acción con mejor Q valor. El problema es que, durante las primeras pruebas de entrenamiento, se observó que utilizar un valor fijo de épsilon complicaba el equilibrio entre exploración y explotación, ya que dependía en gran medida del valor asignado. Para abordar esto, se implementó un épsilon dinámico que comienza con un valor máximo y disminuye exponencialmente a medida que avanza el entrenamiento, hasta llegar a un épsilon mínimo, siendo estos valores de decaimiento y valor máximo y mínimo de épsilon hiperparámetros ajustados antes del entrenamiento.

4.3.2. Multi-Agent PPO

El algoritmo *Multi-Agent Proximal Policy Optimization* (MAPPO) es una variante del algoritmo PPO adaptada para abordar problemas multiagente cooperativos. En estos problemas, los agentes colaboran entre sí para alcanzar un objetivo común, es decir, existe una recompensa global obtenida entre todos los agentes.

Algorithm 1 Recurrent-MAPPO

```

Initialize  $\theta$ , the parameters for policy  $\pi$  and  $\phi$ , the parameters for critic  $V$ , using Orthogonal
initialization (Hu et al., 2020)
Set learning rate  $\alpha$ 
while  $step \leq step_{max}$  do
    set data buffer  $D = \{\}$ 
    for  $i = 1$  to  $batch\_size$  do
         $\tau = []$  empty list
        initialize  $h_{0,\pi}^{(1)}, \dots, h_{0,\pi}^{(n)}$  actor RNN states
        initialize  $h_{0,V}^{(1)}, \dots, h_{0,V}^{(n)}$  critic RNN states
        for  $t = 1$  to  $T$  do
            for all agents  $a$  do
                 $p_t^{(a)}, h_{t,\pi}^{(a)} = \pi(o_t^{(a)}, h_{t-1,\pi}^{(a)}; \theta)$ 
                 $u_t^{(a)} \sim p_t^{(a)}$ 
                 $v_t^{(a)}, h_{t,V}^{(a)} = V(s_t^{(a)}, h_{t-1,V}^{(a)}; \phi)$ 
            end for
            Execute actions  $u_t$ , observe  $r_t, s_{t+1}, o_{t+1}$ 
             $\tau += [s_t, o_t, h_{t,\pi}, h_{t,V}, u_t, r_t, s_{t+1}, o_{t+1}]$ 
        end for
        Compute advantage estimate  $\hat{A}$  via GAE on  $\tau$ , using PopArt
        Compute reward-to-go  $\hat{R}$  on  $\tau$  and normalize with PopArt
        Split trajectory  $\tau$  into chunks of length L
        for  $l = 0, 1, \dots, T/L$  do
             $D = D \cup (\tau[l : l + T], \hat{A}[l : l + L], \hat{R}[l : l + L])$ 
        end for
    end for
    for mini-batch  $k = 1, \dots, K$  do
         $b \leftarrow$  random mini-batch from  $D$  with all agent data
        for each data chunk  $c$  in the mini-batch  $b$  do
            update RNN hidden states for  $\pi$  and  $V$  from first hidden state in data chunk
        end for
    end for
    Adam update  $\theta$  on  $L(\theta)$  with data  $b$ 
    Adam update  $\phi$  on  $L(\phi)$  with data  $b$ 
end while

```

Figura 4.6: Pseudocódigo del algoritmo MAPPO propuesto en su artículo inicial Yu et al. (2022).

En nuestro caso, el problema que abordamos no es del todo cooperativo, ya que las acciones de los agentes conllevan una recompensa individual y dependen únicamente de sus propias acciones anteriores, es decir, no se ven afectadas por las acciones de los demás agentes. Por ello, la implementación que se ha desarrollado, aunque basada en la forma en que el algoritmo MAPPO maneja la existencia de múltiples agentes en el entorno, incluye ciertos cambios necesarios para que el algoritmo se ajuste a nuestro problema.

El algoritmo MAPPO, al depender las acciones de los agentes de todas las demás acciones tomadas por el mismo y por los demás agentes, mantiene un único buffer en el que se guarda la secuencia completa de acciones realizadas con la política actual, para utilizar esta secuencia en la actualización de la política durante el entrenamiento.

Sin embargo, dado que nuestro problema es individual desde el punto de vista de cada

lunar rover, se ha decidido implementar un buffer para cada agente, en el que se guardará la secuencia ordenada de acciones realizadas por cada uno de ellos. Con estos buffers individuales, podemos realizar los cálculos necesarios de manera independiente para cada agente, manteniendo la coherencia en la secuencia de acciones, para, posteriormente, utilizar estos cálculos individuales de manera conjunta para actualizar el modelo, obteniendo una nueva política a seguir. Este proceso se repite con la nueva política, recolectando nuevas secuencias de acciones para todos los agentes y continuando el ciclo de entrenamiento.

El algoritmo implementado, al igual que el algoritmo base PPO, hace uso de una estructura *Actor-Critic*, como ya se explicó anteriormente en el marco teórico 3.1.1.2. En nuestro caso, siguiendo el paradigma *Centralized Training and Decentralized Execution* (CTDE) se utiliza una única red Critic para calcular el valor del estado, la cual toma como entrada el mapa completo del entorno, actuando como parte centralizada durante el entrenamiento. Para la parte de ejecución descentralizada, se emplea una red Actor que representa la política a seguir por los agentes. Esta red toma como entrada la observación individual del *lunar rover* que debe actuar, siguiendo una arquitectura inicial bastante similar a la red utilizada en la implementación del algoritmo DDDQL, y devuelve una distribución de probabilidades entre todas las acciones.

Al igual que en el algoritmo DDDQL, hemos utilizado una única red Actor y una única red Critic para todos los agentes, dada su naturaleza homogénea. No obstante, el algoritmo también podría ser adaptado para manejar agentes no homogéneos, que contarían con sus propias redes individuales *Actor-Critic*.

Dado que el algoritmo PPO y sus variantes son algoritmos *on-policy*, el ciclo de entrenamiento es algo diferente al de los algoritmos *off-policy*, en los que al contener un único buffer en el que se mezclan experiencias de todo tipo de políticas, se pueden realizar actualizaciones continuas de la red en cada paso realizado en el entorno. Sin embargo, en un algoritmo *on-policy*, la frecuencia de actualización de las redes debe ser controlada, ya que antes de actualizarse, es necesario recopilar una cantidad suficiente de experiencias obtenidas con la política actual, para que la actualización de la red esté sustentada en una variedad suficiente de experiencias, asegurando así que la actualización sea relevante. Tras la actualización de la red y la obtención de una nueva política a seguir, se debe limpiar el buffer para volver a completarlo con nuevas experiencias recopiladas usando la nueva política.

La pérdida o *loss* de la red Actor se calcula en función de las ventajas o *advantages* de las acciones tomadas. Estas ventajas se calculan siguiendo la fórmula de la *Generalized Advantage Estimation* o Estimación Generalizada de Ventajas (GAE), que utiliza el valor del estado calculado por la red Critic y las recompensas obtenidas. Además, se incluye el mecanismo de *clipping*, comúnmente utilizado en el algoritmo base PPO, que actúa como limitador en el cálculo de la pérdida del Actor, en función de un hiperparámetro que define el ratio de *clip*, para evitar actualizaciones bruscas en la política y mantener la estabilidad durante el entrenamiento.

La salida de la red Actor, en forma de una distribución de probabilidades sobre todas las

acciones disponibles, se utiliza para seleccionar la acción a realizar en base a las probabilidades asignadas a cada una, lo que permite mantener el factor de exploración durante todo el proceso de entrenamiento. Sin embargo, una vez entrenado el algoritmo, en la fase de inferencia, se utiliza esta red Actor para seleccionar la acción con mayor probabilidad, ya que está sería la acción más óptima según la red entrenada.

Por otro lado, la pérdida de la red Critic, utilizada para su actualización, se calcula como la diferencia entre el valor predicho para un estado por la red y sus recompensas descontadas. Dado que ya se tiene la secuencia completa de acciones y recompensas, se realiza el cálculo del valor real que se puede obtener desde un estado, en función de las recompensas siguientes, aplicando un factor de descuento ajustado como hiperparámetro para ir reduciendo la importancia de las recompensas futuras. Esto permite determinar qué tan buenas son las predicciones de la red sobre las recompensas que se pueden obtener desde un estado, en comparación con las recompensas que realmente se obtuvieron desde ese mismo estado.

Dada la naturaleza de la red Critic, la cual sirve como parte centralizada del entrenamiento al utilizar el estado completo del entorno, es importante destacar que esta red no es utilizada durante la fase de inferencia, manteniendo así una ejecución descentralizada para cada agente. Esto es posible debido a que la red Critic solo se emplea durante el entrenamiento para calcular los valores de los estados, necesarios para el cálculo de las ventajas y, con ello, la pérdida del Actor, por lo que una vez realizado el entrenamiento y obtenida una red Actor con una política óptima, la red Critic es totalmente innecesaria.

Para ambos algoritmos, se han implementado elementos comunes destinados a evitar ciertos conflictos derivados de las características específicas del problema. La primera y más importante implementación ha sido el cálculo de una máscara para evitar seleccionar acciones inválidas, ya que, cuando un agente se encuentra en el borde del mapa, no debería poder realizar una acción que lo saque de los límites. Dado que las dimensiones de las salidas de las redes neuronales no pueden modificarse dinámicamente una vez creadas, se han incorporado máscaras que deshabilitan estas acciones inválidas durante los procesos en los que cabía la posibilidad de que se tuviesen en cuenta acciones inválidas dentro del entrenamiento y de la inferencia de ambos algoritmos. Estas máscaras están personalizadas en base a la naturaleza de cada algoritmo asegurando que las acciones inválidas se manejen adecuadamente en ambos. En el algoritmo DDDQL el valor Q de estas acciones se reduce a menos infinito, mientras que en el algoritmo MAPPO se reduce a cero la probabilidad de seleccionar dichas acciones. Esto garantiza que ambos algoritmos siempre seleccionen y usen las acciones válidas para cada situación manteniendo coherencia con la simulación en todo momento.

Además, con el objetivo de mejorar la eficiencia en el procesamiento de los datos, se ha normalizado la entrada de las redes neuronales para ambos algoritmos en un rango de 0 a 1. También se ha aplicado una ligera regularización Ridge, o L2, que se ha ido reduciendo gradualmente durante el entrenamiento, permitiendo mantener controlada la magnitud de los pesos de las conexiones para evitar grandes variaciones en las pérdidas y actualizaciones

inestables. Asimismo, se ha implementado una reducción dinámica del *learning rate*, que disminuye progresivamente desde un valor máximo hasta uno mínimo según un factor configurado mediante hiperparámetros. Esta reducción se activa cuando la métrica de pérdida no mejora tras un número determinado de iteraciones, evitando así que el entrenamiento se estanque.

Todo el código desarrollado para la creación de la simulación, así como la implementación y el uso de ambos algoritmos, ha sido publicado en un repositorio público de *GitHub* (<https://github.com/plss12/Lunar-Rover-MADRL>) con el objetivo de que cualquier persona interesada pueda acceder fácilmente y revisar el proyecto completo.

Experimentación y Pruebas

5

Los algoritmos desarrollados han sido entrenados en entornos de tamaño 12x12, cada uno con 3 agentes que disponían de un rango de visión de 3 casillas a su alrededor. Una vez completado el entrenamiento, los modelos resultantes se pueden aplicar a entornos con características variables, como diferentes tamaños de entorno y un número distinto de agentes.

Durante el proceso de entrenamiento, se establecieron puntos de control tras un número determinado de actualizaciones. Estos puntos de control permitieron evaluar en tiempo real la evolución del aprendizaje y determinar si era necesario ajustar los hiperparámetros de los algoritmos o modificar aspectos del entorno, como la distribución de recompensas, con el fin de mejorar el entrenamiento o incluso reiniciarlo desde cero.

Además, estos puntos de control funcionaron como puntos de guardado, donde se almacenaban tanto los modelos como las métricas de entrenamiento. Esto facilita retomar el proceso de entrenamiento desde cualquier punto y mantener un registro completo de la evolución del aprendizaje, permitiendo su análisis detallado mediante gráficas de las métricas registradas una vez finalizado este.

Ambos algoritmos fueron sometidos a múltiples entrenamientos de prueba, gracias a los cuales se fueron ajustando poco a poco los hiperparámetros en función de la evolución de los entrenamientos previos. Este proceso ha concluido en un entrenamiento completo, para cada algoritmo, en el que se ha podido configurar la combinación de hiperparámetros que mejor ha evolucionado entre todos los entrenamientos de prueba. A continuación, se presentan en una tabla los hiperparámetros seleccionados para cada algoritmo:

Hiperparámetros MAPPO	Valor
Learning Rate Máximo	1e-3
Learning Rate Mínimo	5e-5
Gamma	0.95
Lamda	0.95
Clip	0.2
Coeficiente de entropía	0.01
Frecuencia de entrenamiento	500

Hiperparámetros DDDQL	Valor
Learning Rate Máximo	1e-2
Learning Rate Mínimo	5e-5
Épsilon Máximo	1
Épsilon Mínimo	0.4
Gamma	0.95
Buffer Size	50000
Batch Size	64
Warmup Steps	5000
Frecuencia de Actualización de la Red Objetivo	500

Durante los entrenamientos, se registraron métricas que permiten visualizar la evolución de la pérdida o *loss*, la recompensa media por acción, el número de episodios completados entre puntos de control, y el número de pasos máximo utilizado para completar un único episodio entre cada punto de control. Las gráficas que muestran la evolución de estos parámetros, y con ello la evolución de los entrenamientos, se presentan a continuación:

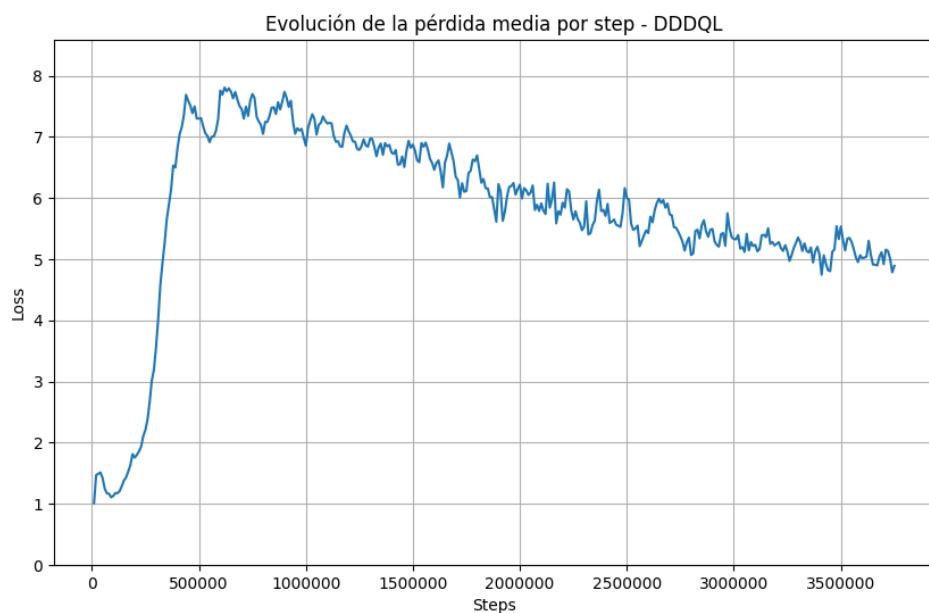


Figura 5.1: Gráfica con la evolución de la pérdida o *loss* durante el entrenamiento del algoritmo DDDQL.

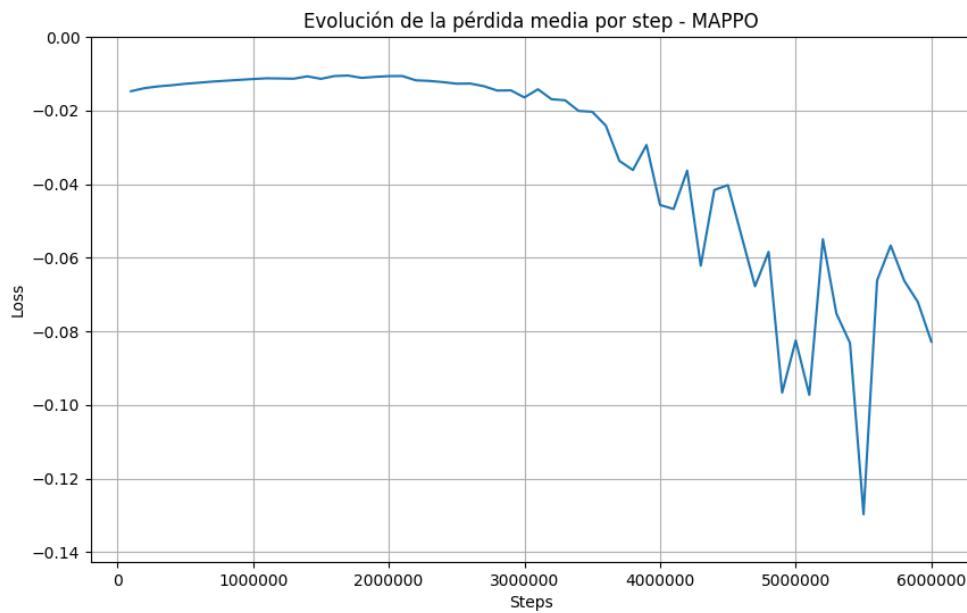


Figura 5.2: Gráfica con la evolución de la pérdida o *loss* para el modelo Actor durante el entrenamiento del algoritmo MAPPO.

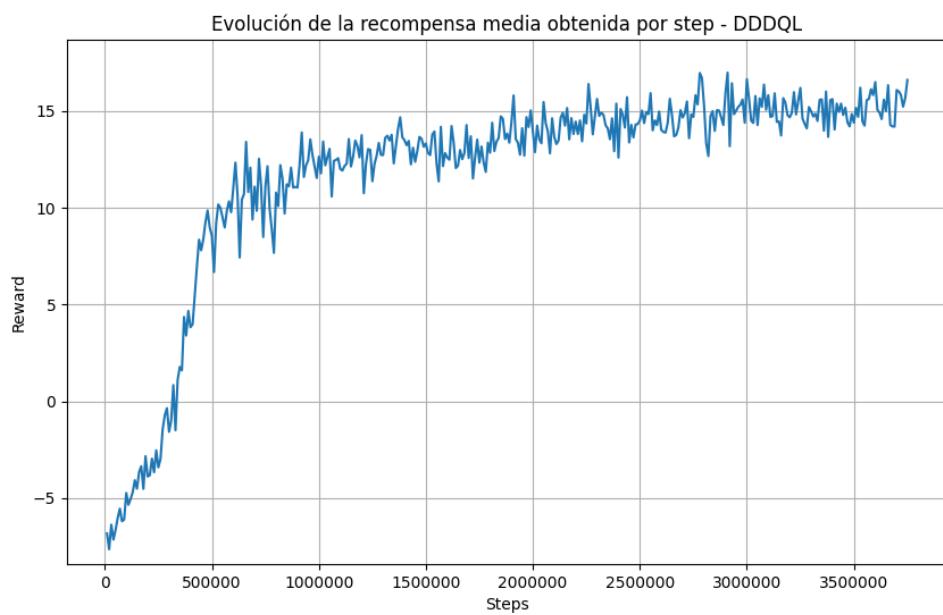


Figura 5.3: Gráfica con la evolución de la recompensa media obtenida por cada acción realizada durante el entrenamiento del algoritmo DDDQL.

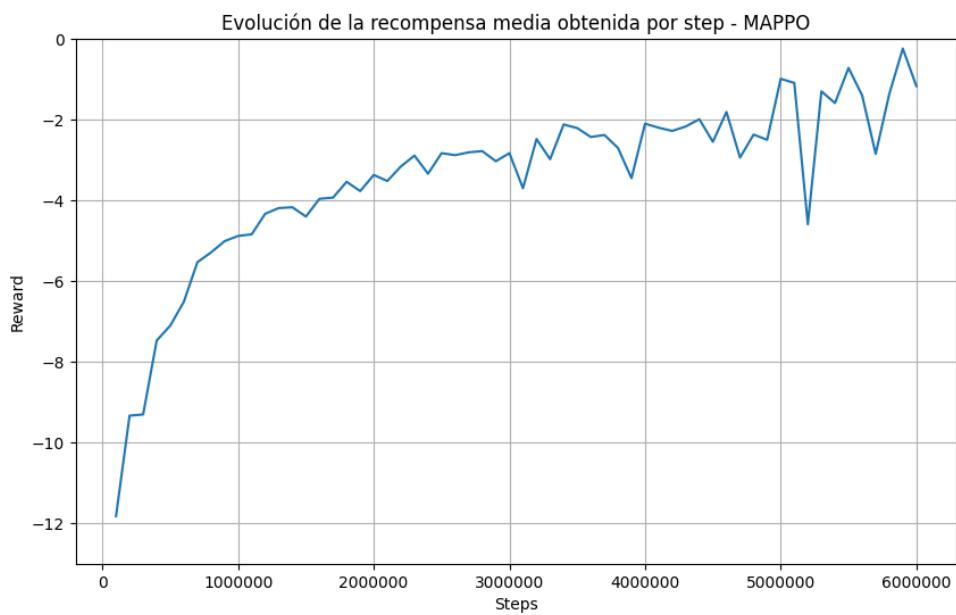


Figura 5.4: Gráfica con la evolución de la recompensa media obtenida por cada acción realizada durante el entrenamiento del algoritmo MAPPO.

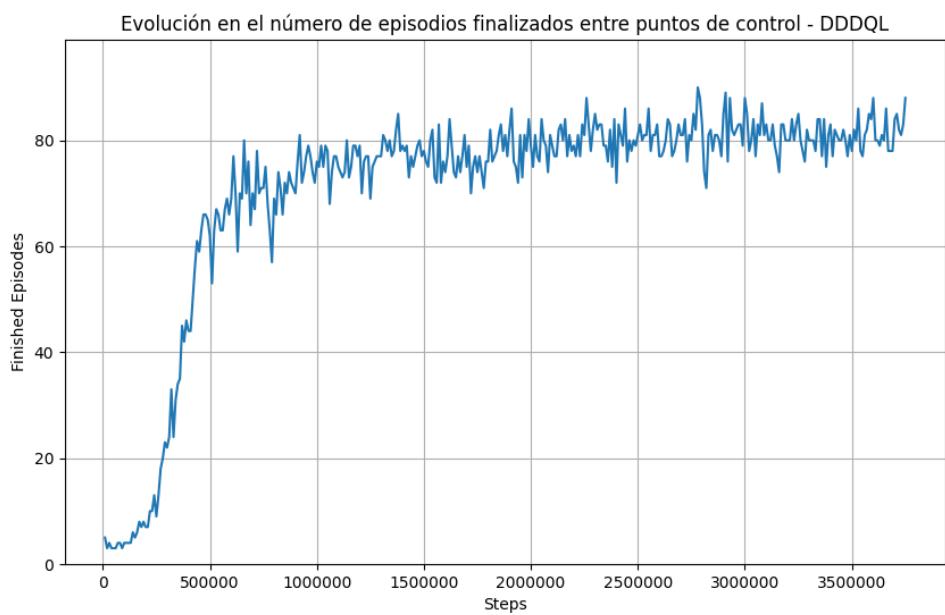


Figura 5.5: Gráfica con la evolución del número de episodios finalizados entre puntos de control durante el entrenamiento del algoritmo DDDQL.

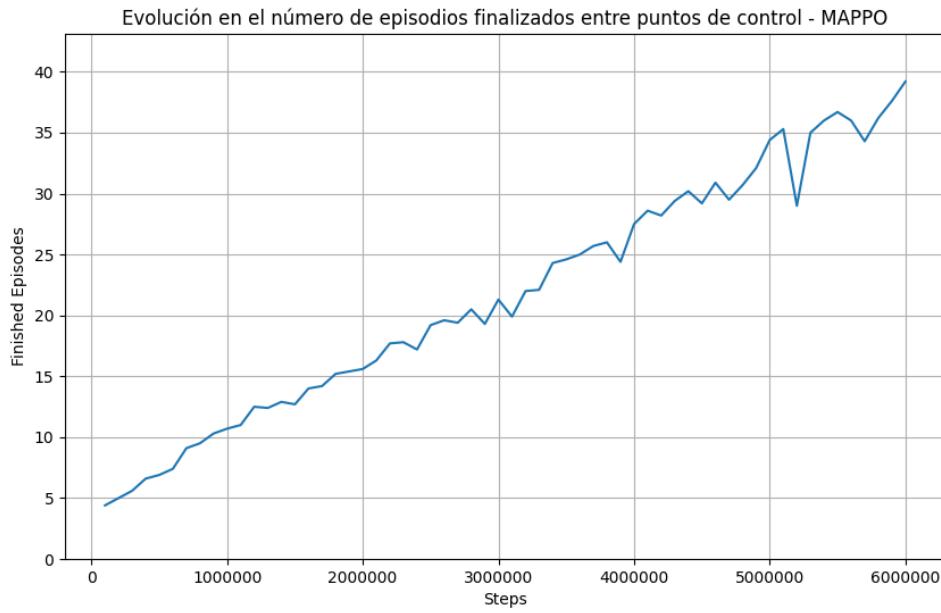


Figura 5.6: Gráfica con la evolución del número de episodios finalizados entre puntos de control durante el entrenamiento del algoritmo MAPPO.

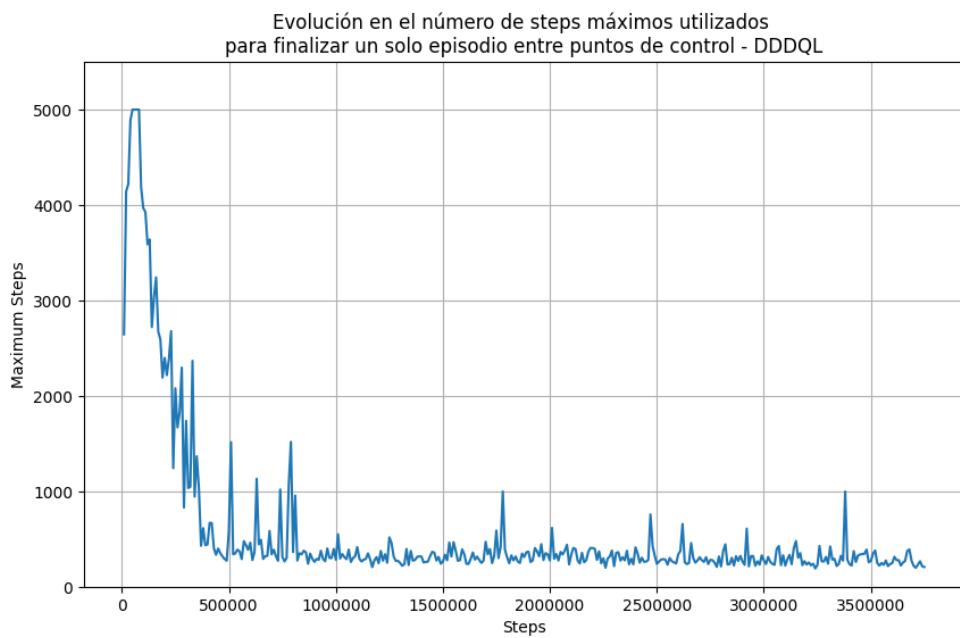


Figura 5.7: Gráfica con la evolución del número de acciones máximas utilizadas para finalizar un solo episodio entre puntos de control durante el entrenamiento del algoritmo DDDQL. Aplicado límite de 5000 steps por episodio para no sobreentrenar circunstancias específicas.

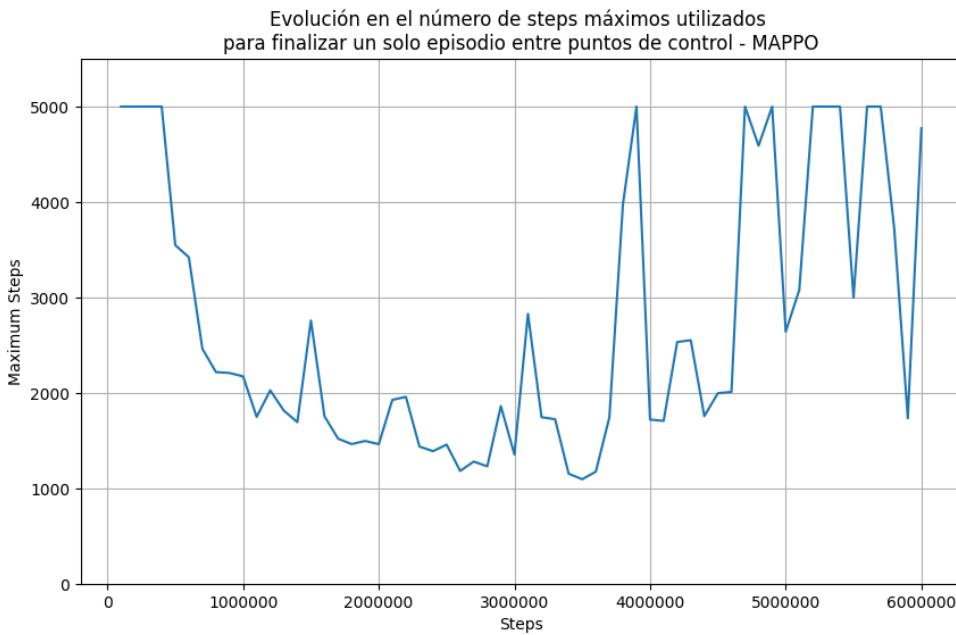


Figura 5.8: Gráfica con la evolución del número de acciones máximas utilizadas para finalizar un solo episodio entre puntos de control durante el entrenamiento del algoritmo MAPPO. Aplicado límite de 5000 steps por episodio para no sobreentrenar circunstancias específicas.

Es importante destacar que todos los datos recopilados durante el entrenamiento incluyen factores de exploración para cada algoritmo. Aunque estos factores van disminuyendo a medida que avanza el entrenamiento, permanecen presentes a lo largo de todo el proceso, influyendo en las métricas de entrenamiento reflejadas en las gráficas.

Como se puede apreciar, el algoritmo DDDQL muestra una evolución positiva en su entrenamiento. Tras un incremento inicial en la pérdida durante las primeras actualizaciones, debido a la alta tasa de exploración y a la continua exposición a nuevas situaciones, se observa un descenso casi constante en la pérdida 5.1. Este comportamiento se correlaciona con el incremento continuo de la recompensa media obtenida por cada acción realizada 5.3, lo que indica una mejora en la calidad de las decisiones tomadas por el algoritmo. Como resultado, se consigue finalizar un mayor número de episodios entre puntos de control 5.5 siendo necesarios un menor número de pasos 5.7.

En contraste, la evolución del entrenamiento del algoritmo MAPPO no es satisfactoria. Aunque se aprecia un aumento casi lineal en el número de episodios completados entre puntos de control 5.6, lo que podría sugerir que el algoritmo está aprendiendo a resolver el problema, la gráfica de la recompensa media por acción 5.4 revela que esta mejora es insuficiente, ya que la recompensa media permanece constantemente negativa. Además, durante gran parte del entrenamiento, el algoritmo alcanza el límite máximo de acciones permitidas por episodio 5.7, a diferencia del algoritmo DDDQL que apenas llega a este límite en las primeras iteraciones de su entrenamiento. En definitiva, el algoritmo *mappo* no ha logrado un aprendizaje adecuado

como para resultar en un modelo aplicable sobre nuestro caso de estudio.

Debido a estos resultados, se ha decidido profundizar en el modelo entrenado por el algoritmo *dddql*, con el que se han realizado pruebas para demostrar su eficiencia en la resolución del caso de estudio. En estas pruebas se ha de utilizar un agente de inferencia sin factor de exploración, asegurando que siempre se selecciona la mejor acción según el modelo entrenado, es decir, la acción con el mayor valor Q, y manteniendo la aplicación de máscaras para evitar acciones inválidas.

Estas pruebas consisten en aplicar el modelo entrenado por el algoritmo *dddql* en cuatro entornos distintos, generados bajo las mismas condiciones configurables que se utilizaron durante el entrenamiento, es decir, un tamaño de 12x12, con tres agentes, cada uno con un rango de visión de tres casillas a su alrededor. En cada uno de estos entornos, se realizan dos pruebas. En la primera, los agentes no tienen conocimiento previo de la ubicación de sus minas y la mezcladora, al igual que durante el entrenamiento, por lo que deben descubrirlas a través de la exploración. En la segunda, los agentes poseen información previa sobre estas posiciones clave.

A continuación, se muestran las rutas seguidas por los agentes de inferencia en los cuatro entornos generados para las dos pruebas realizadas en cada uno de ellos:

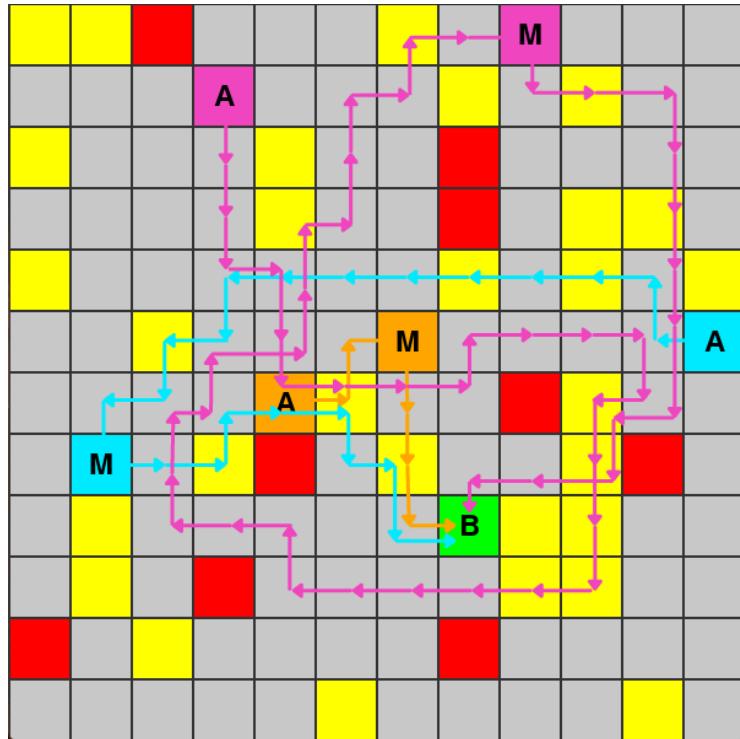


Figura 5.9: Rutas seguidas por los agentes sin el conocimiento previo de las posiciones relevantes en el primer entorno de prueba.

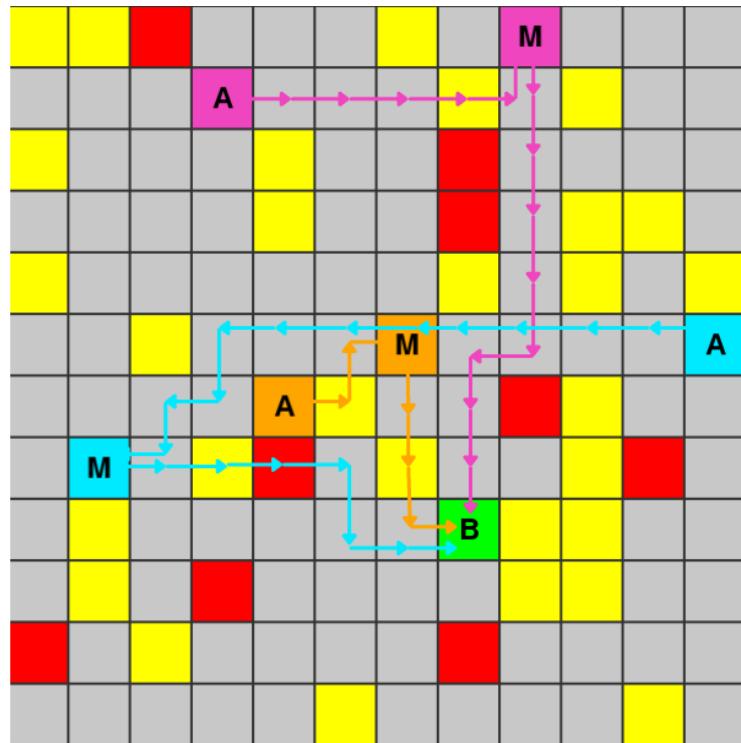


Figura 5.10: Rutas seguidas por los agentes con el conocimiento previo de las posiciones relevantes en el primer entorno de prueba.

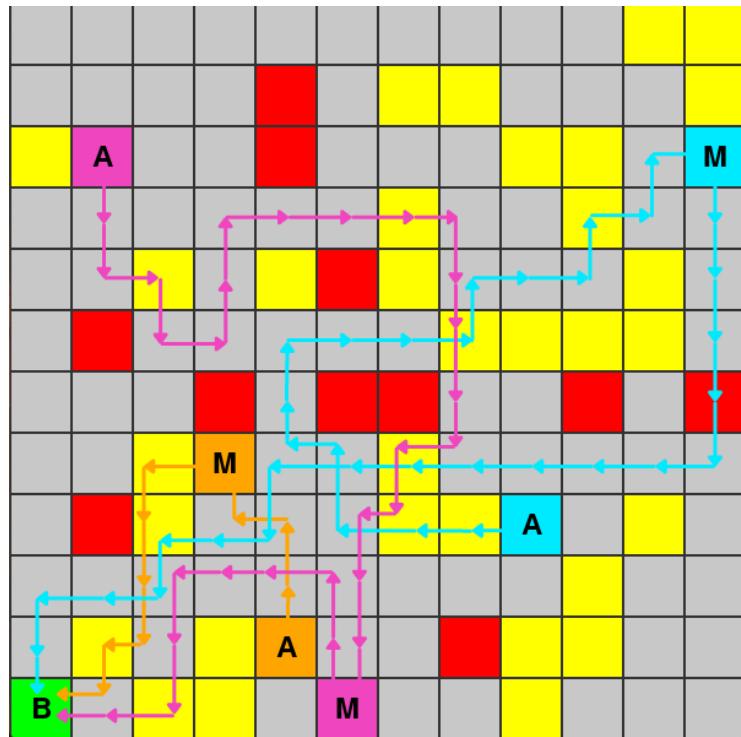


Figura 5.11: Rutas seguidas por los agentes sin el conocimiento previo de las posiciones relevantes en el segundo entorno de prueba.

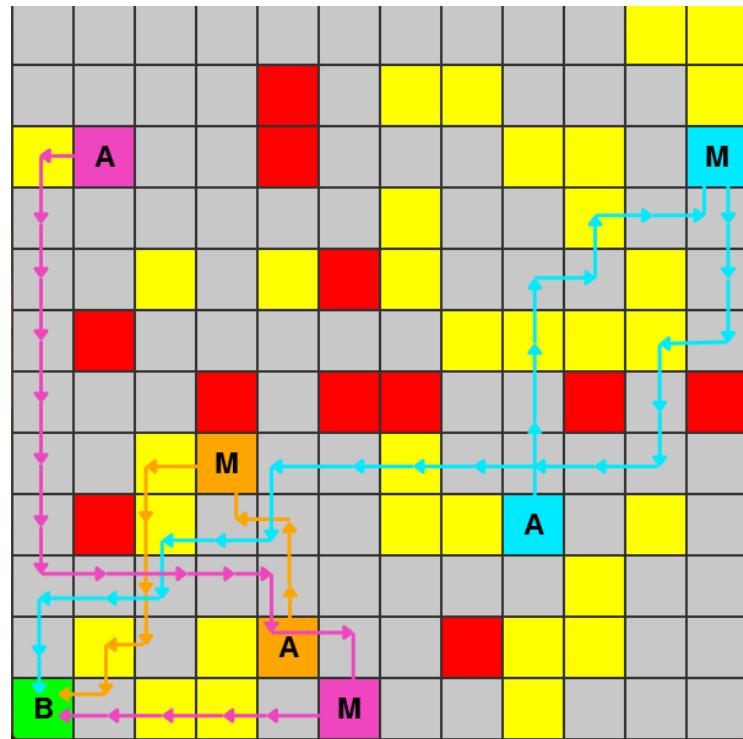


Figura 5.12: Rutas seguidas por los agentes con el conocimiento previo de las posiciones relevantes en el segundo entorno de prueba.

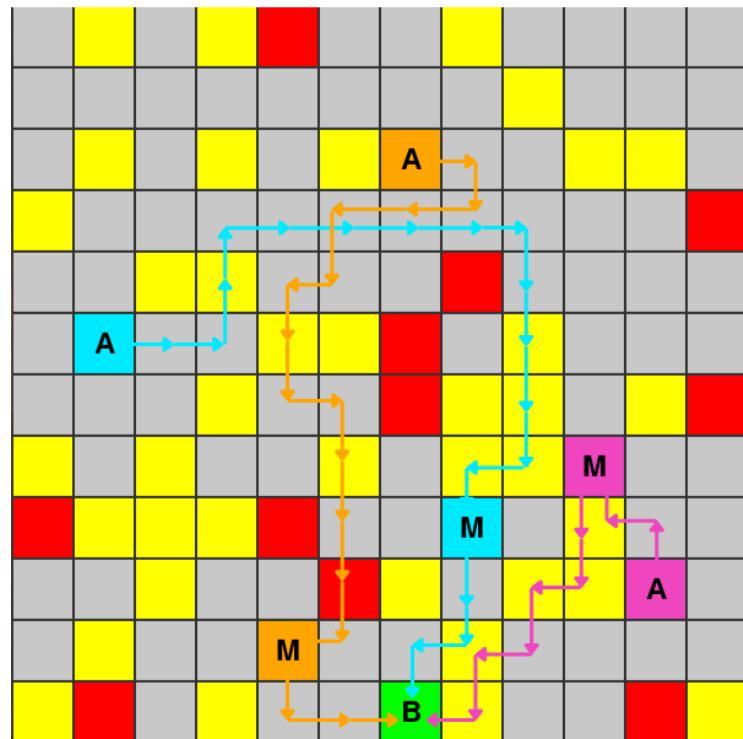


Figura 5.13: Rutas seguidas por los agentes sin el conocimiento previo de las posiciones relevantes en el tercer entorno de prueba.

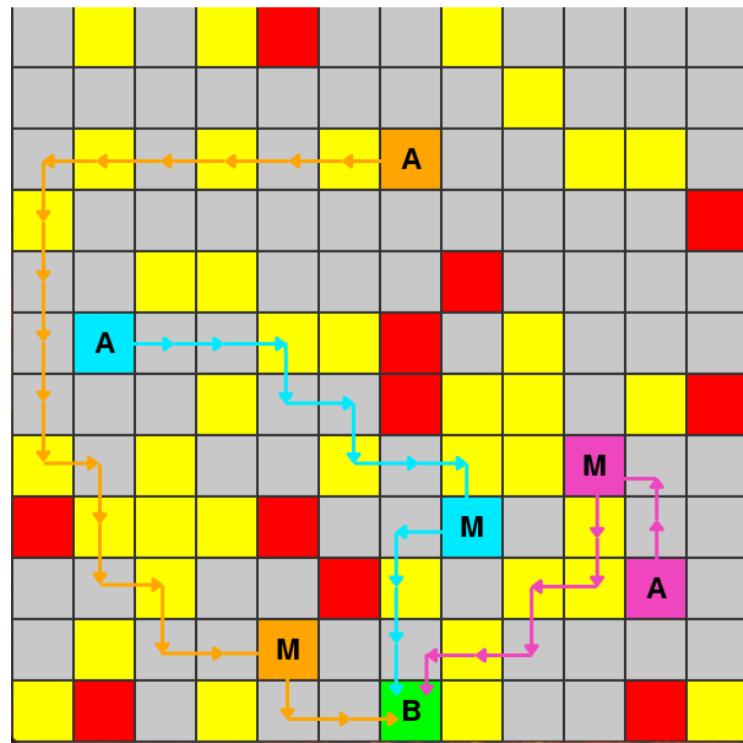


Figura 5.14: Rutas seguidas por los agentes con el conocimiento previo de las posiciones relevantes en el tercer entorno de prueba.

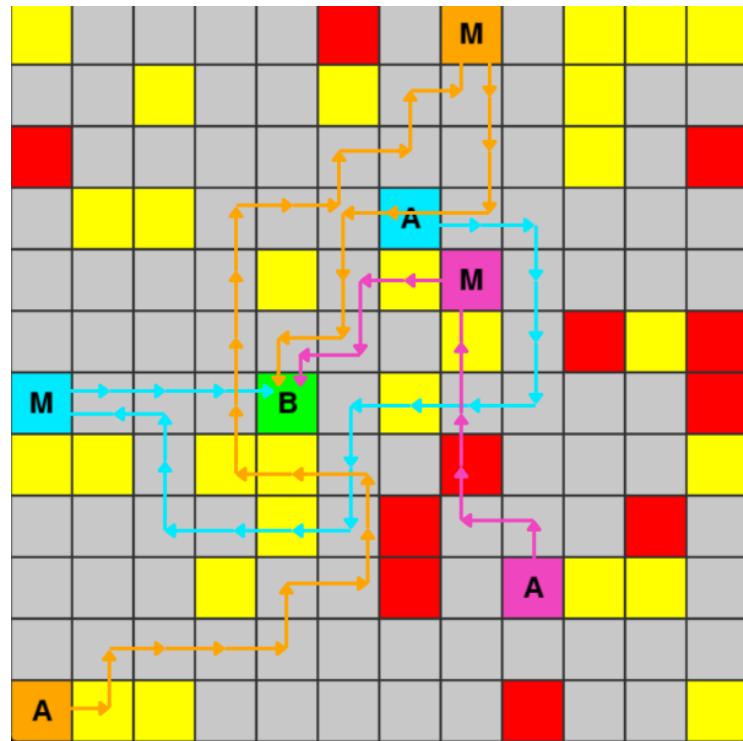


Figura 5.15: Rutas seguidas por los agentes sin el conocimiento previo de las posiciones relevantes en el cuarto entorno de prueba.

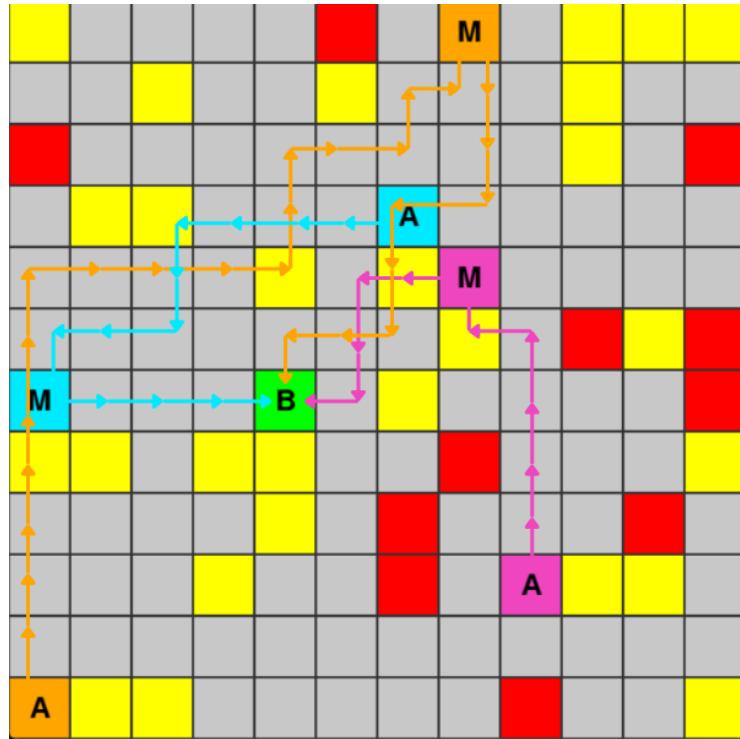


Figura 5.16: Rutas seguidas por los agentes con el conocimiento previo de las posiciones relevantes en el cuarto entorno de prueba.

La siguiente tabla presenta un resumen de las pruebas ilustradas en las imágenes anteriores, detallando el número total de pasos empleados para completar cada prueba y la recompensa global obtenida de manera conjunta por todos los agentes en cada una de ellas:

Figura	Entorno	Posiciones Conocidas	Pasos totales	Recompensa Total
5.9	1	No	83	2186
5.10	1	Si	41	2382
5.11	2	No	78	2194
5.12	2	Si	60	2422
5.13	3	No	46	2209
5.14	3	Si	44	2426
5.15	4	No	63	2232
5.16	4	Si	51	2447

Las pruebas realizadas demuestran el excelente desempeño del modelo entrenado con el algoritmo DDDQL, que logra completar cada entorno de prueba en un número reducido de pasos para todos los agentes. Además, se observa que los *lunar rovers* ejecutan una exploración eficiente en busca de la localización de sus objetivos, primero la mina y luego la mezcladora, evitando áreas ya visitadas. Una vez identificada la posición a la que debe desplazarse, los movimientos de los agentes son precisos y directos. Este comportamiento se manifiesta

también en las pruebas, donde los agentes poseen el conocimiento previo de la ubicación de su propia mina y la mezcladora, lo que les permite evitar los movimientos para la exploración del entorno y dirigirse directamente a los objetivos. La eficiencia del modelo en la resolución del caso de estudio abordado respalda el análisis previo de las gráficas sobre la evolución de las métricas durante el entrenamiento, confirmando que el algoritmo DDDQL ha logrado un aprendizaje efectivo.

Conclusiones y Trabajo Futuro

6

Este Trabajo de Fin de Máster ha consistido en una amplia investigación del estado del arte sobre algoritmos de aprendizaje por refuerzo profundo aplicados a problemas multiagente, con el objetivo de identificar alternativas más eficaces a los métodos tradicionales de planificación de rutas en este tipo de entornos. Tras un análisis minucioso, se seleccionaron dos algoritmos particularmente prometedores, *Double Dueling Deep Q-Learning* (DDDQL) y *Multi-Agent Proximal Policy Optimization* (MAPPO), los cuales fueron desarrollados desde cero, adaptándolos específicamente a las características del problema abordado.

Para validar el correcto funcionamiento de los algoritmos desarrollados, se diseñó y desarrolló un simulador que reproduce las condiciones del entorno lunar descrito en el caso de estudio. Este simulador permitió la implementación práctica de los algoritmos sobre entornos simulados, facilitando la creación de escenarios aleatorios con parámetros configurables, como el tamaño del mapa y el número de agentes, esenciales para la fase de entrenamiento de ambos algoritmos. Además, el simulador también se utilizó para generar varios entornos fijos sobre los que se evaluó el rendimiento del modelo final obtenido tras el entrenamiento del algoritmo DDDQL.

El proceso de entrenamiento fue supervisado y controlado, siendo continuamente ajustado para optimizar la evolución del aprendizaje. Todas las métricas relevantes fueron registradas y almacenadas detalladamente durante la fase de entrenamiento, lo que permitió un análisis exhaustivo de los resultados obtenidos al concluir esta fase, sirviendo también como base para identificar áreas clave para futuras mejoras.

Durante la fase de experimentación y pruebas, el algoritmo DDDQL demostró ser altamente eficiente en la resolución de las tareas tratadas, alcanzando excelentes resultados en los entornos de prueba. En contraste, el algoritmo MAPPO no logró desarrollar un aprendizaje efectivo, resultando en un modelo ineficiente e incapaz de resolver el problema planteado. Esta disparidad en los resultados puede atribuirse a unas de las principales diferencias fundamentales entre ambos algoritmos. DDDQL, al ser un algoritmo *off-policy*, se beneficia de un aprendizaje continuo basado en una amplia variedad de experiencias, recolectadas por todos los agentes a lo largo de los últimos episodios de entrenamiento. En cambio, MAPPO, debido a su naturaleza *on-policy*, depende exclusivamente de las trayectorias generadas por la política actual, generalmente durante un único episodio, lo que limita la diversidad de experiencias utilizadas en cada iteración de entrenamiento y lo hace menos robusto frente a la aleatoriedad en la generación

de episodios, una característica que parece representar una ventaja significativa para DDDQL y un obstáculo para MAPPO. Es probable que MAPPO pueda desarrollar un aprendizaje efectivo en un entorno con condiciones fijas o mayormente controladas, donde la aleatoriedad no sea un factor determinante en la creación de cada nuevo entorno de entrenamiento, pero en nuestro caso, estas condiciones no se ajustan al problema estudiado.

Al analizar las pruebas realizadas con el modelo entrenado por DDDQL, se observó que las trayectorias seguidas por los agentes hacia sus objetivos son, en general, eficientes, evitando en su mayoría los obstáculos grandes. Sin embargo, se detectó una tendencia de los agentes a sobreponerse a pequeños obstáculos que podrían haber sido evitados, lo que sugiere que las rutas aún pueden optimizarse. Por esta razón, considero que sería beneficioso continuar el entrenamiento del algoritmo DDDQL durante un mayor número de *steps*, ya que las gráficas muestran una tendencia de mejora continua en la evolución de sus métricas. Además, sería valioso experimentar con una mayor variedad de distribuciones de recompensas e hiperparámetros, en busca de la combinación que mejor optimice el comportamiento de los agentes en aquellas rutas donde no evitan ciertos obstáculos, a pesar de que esquivarlos podría, en última instancia, mejorar la recompensa final, aunque implique realizar algunos movimientos adicionales. Con las combinaciones de distribuciones de recompensas e hiperparámetros probadas, los agentes parecen priorizar una ruta corta sobre evitar obstáculos menores, que no representan una gran penalización.

Un factor que alarga las rutas es la fase inicial de exploración del entorno, durante la cual los agentes buscan la ubicación de su mina asignada y la mezcladora. Esta exploración, en gran medida aleatoria debido a la falta de conocimiento previo sobre la ubicación de los objetivos, a veces resulta en trayectorias significativamente más largas, especialmente cuando un agente, por mala fortuna, toma un desvío considerable antes de localizar su objetivo dentro de su rango de visión.

Este último factor desapareció en las pruebas realizadas en los mismos entornos, pero con la diferencia de que los agentes contaban con conocimiento previo de la ubicación exacta de sus minas y la mezcladora, donde se observó que las rutas seguidas fueron directas hacia sus objetivos, eliminando la necesidad de exploración. Esto resultó en trayectorias con un menor número de movimientos y una mayor recompensa global, como se refleja en la tabla resumen de las pruebas realizadas. Por lo tanto, un posible trabajo futuro podría centrarse en implementar una alternativa intermedia entre no proporcionar ninguna información a los agentes y darles la información exacta sobre la posición de los objetivos desde el inicio. Por ejemplo, un enfoque intermedio podría consistir en equipar a cada *lunar rover* con una especie de brújula que indique en todo momento la dirección de su mina o mezcladora, incluso cuando estos objetivos no estén dentro de su rango de visión. Esto podría ayudar a los agentes a llevar a cabo una fase de exploración inicial más breve y eficiente, guiados por la orientación otorgada por la brújula.

Cabe destacar que en ninguna de las pruebas realizadas en los distintos entornos se produjo colisión alguna entre los agentes, a pesar de que en múltiples situaciones las rutas seguidas

por cada uno los llevaron a ocupar posiciones adyacentes simultáneamente. Esto demuestra que el modelo resultante no solo es efectivo como planificador de rutas, sino que también cumple con creces como mecanismo de evitación de colisiones.

Por otro lado, sería interesante estudiar la escalabilidad del modelo DDDQL al incrementar el número de agentes o la complejidad del entorno, incorporando más obstáculos o aumentando el tamaño del mapa. Esto permitiría evaluar la robustez del algoritmo frente a situaciones más complejas y dinámicas.

Finalmente, como objetivo más amplio, sería enriquecedor aplicar estos algoritmos a otros tipos de problemas multiagente para determinar si los resultados y conocimientos obtenidos en este proyecto son extrapolables a otros casos de estudio, más allá del contexto de planificación de rutas, abriendo nuevas oportunidades de investigación y aplicación en diversas áreas de la inteligencia artificial.

Lista de Acrónimos

CTCE *Centralized Training and Centralized Execution.*

CTDE *Centralized Training and Decentralized Execution.*

DDDQL *Double Dueling Deep Q-Learning.*

DL *Deep Learning o Aprendizaje Profundo.*

DQL *Deep Q-Learning.*

DQN *Deep Q-Network.*

DRL *Deep Reinforcement Learning o Aprendizaje por Refuerzo Profundo.*

DTDE *Decentralized Training and Decentralized Execution.*

GAE *Generalized Advantage Estimation o Estimación Generalizada de Ventajas.*

IPPO *Independent Proximal Policy Optimization.*

MADRL *Multi-Agent Deep Reinforcement Learning o Aprendizaje por Refuerzo Profundo Multiagente.*

MAPP *Multi-Agent Path Planning o Planificación de Rutas Multiagente.*

MAPPO *Multi-Agent Proximal Policy Optimization.*

MARL *Multi-Agent Reinforcement Learning o Aprendizaje por Refuerzo Multiagente.*

MDP *Mákov Decision Process o Proceso de Decisión de Mákov.*

ORCA *Optimal Reciprocal Collision Avoidance.*

PPO *Proximal Policy Optimization.*

RL *Reinforcement Learning o Aprendizaje por Refuerzo.*

Bibliografía

- Chen, Y., Dong, Q., Shang, X., Wu, Z., y Wang, J. (2023). Multi-uav autonomous path planning in reconnaissance missions considering incomplete information: A reinforcement learning method.
- Chollet, F. (2015). Keras repository.
- Chung, J., Fayyad, J., Younes, Y., y Najjaran, H. (2023). Learning to team-based navigation: A review of deep reinforcement learning techniques for multi-agent pathfinding.
- Dudfield, R., Lindstrom, L., y Shinners, P. (2000). Pygame repository.
- Foundation, F. y OpenAI (2023). Gymnasium documentation.
- Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Os-trovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., y Silver, D. (2017). Rainbow: Combining improvements in deep reinforcement learning.
- Huang, Y., Wei, G., y Wang, Y. (2018). V-d d3qn: the variant of double deep q-learning network with dueling architecture.
- Kulathunga, G. (2022). A reinforcement learning based path planning approach in 3d environment.
- Long, P., Fan, T., Liao, X., Liu, W., Zhang, H., y Pan, J. (2018). Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning.
- Lu, S., Xu, R., Li, Z., Wang, B., y Zhao, Z. (2024). Lunar rover collaborated path planning with artificial potential field-based heuristic on deep reinforcement learning.
- Lumholt, S., van Rossum, G., y Lundh, F. (2000). Pygame repository.
- Ma, Z., Luo, Y., y Ma, H. (2021). Distributed heuristic multi-agent path finding with communication.
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., y Riedmiller, M. (2013). Playing atari with deep reinforcement learning.
- Oliphant, T. (2006). Numpy repository.
- Park, J., Kim, J., y Park, T. K. (2022). Implementation of ppo for multi-agent path finding with dynamic obstacles.
- Saifullah, M., Papakonstantinou, K. G., Andriotis, C. P., y Stoffels, S. M. (2024). Multi-agent deep reinforcement learning with centralized training and decentralized execution for transportation infrastructure management.
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A., y Klimov, O. (2017). Proximal policy optimization algorithms.
- Singh, R., Ren, J., y Lin, X. (2023). A review of deep reinforcement learning algorithms for mobile robot path planning.
- Stanford, M. E. (2016). Multi-agent deep reinforcement learning.
- Team, G. B. (2015). Tensorflow repository.
- Thanh Ha, V. y Vinh, V. (2024). Experimental research on avoidance obstacle control for mobile robots using q-learning (ql) and deep q-learning (dql) algorithms in dynamic environments.
- van den Berg, J., Guy, S., Lin, M., y Manocha, D. (2011). Reciprocal n-body collision avoidance.
- Yang, Y., Juntao, L., y Lingling, P. (2020). Multi-robot path planning based on a deep reinforcement learning dqn algorithm.
- Yao, Q., Zheng, Z., Qi, L., Yuan, H., Guo, X., Zhao, M., Liu, Z., y Yang, T. (2020). Path planning method with improved artificial potential field—a reinforcement learning perspective.
- Yu, C., Velu, A., Vinitsky, E., Gao, J., Wang, Y., Bayen, A., y Wu, Y. (2022). The surprising effectiveness of ppo in cooperative, multi-agent games.
- Yu, X., Wang, P., y Zhang, Z. (2021). Learning-based end-to-end path planning for lunar rovers with safety constraints.
- Zhu, G., Shen, Z., Liu, L., Zhao, S., Ji, F., Ju, Z., y Sun, J. (2022). Auv dynamic obstacle avoidance method based on improved ppo algorithm.