Entre la Luna y el suelo: estrategias para un aterrizaje perfecto

Javier Milá de la Roca Dos Santos *Universidad de Sevilla*
Sevilla, España
javmildos@alum.us.es - PMN9817

Claudia Oviedo Govantes *Universidad de Sevilla*
Sevilla, España
claovigov@alum.us.es - VJR5418

Resumen

El objetivo principal de este trabajo es implementar y analizar un agente *deep Q-network* (*DQN*) para resolver el problema del entorno *LunarLander* mediante aprendizaje por refuerzo profundo. Se construyó desde cero una arquitectura que utiliza redes neuronales para aproximar la función de valor Q, permitiendo al agente aprender una política que maximiza la recompensa acumulada. Durante el proceso de entrenamiento se monitorizó la evolución de las recompensas y la tasa de exploración, lo que permitió evaluar cómo el agente mejora su comportamiento con el tiempo y ajusta su balance entre exploración y explotación.

Los experimentos realizados muestran que el agente es capaz de aprender a aterrizar el módulo lunar de forma cada vez más eficiente, alcanzando puntuaciones positivas y estabilidad en el rendimiento. Además, se introdujeron mejoras basadas en técnicas como *double DQN* y *prioritized replay*, que consiguieron generar un agente más fiable a pesar de tener un entrenamiento más caótico.

Palabras clave

Aprendizaje por refuerzo, *deep Q-network*, *DQN*, *LunarLander*, *Q-learning*, *double DQN*, *prioritized experience replay*, inteligencia artificial.

I. Introducción

El aprendizaje por refuerzo (*reinforcement learning*) es una rama del aprendizaje automático que permite a un agente aprender a tomar decisiones mediante la interacción con un entorno, con el objetivo de maximizar una recompensa acumulada a lo largo del tiempo [1]. En los últimos años, esta área ha experimentado un gran auge gracias a la combinación de métodos clásicos con redes neuronales profundas [2].

Una de las contribuciones más importantes en este área fue el desarrollo del *deep Q-network* (*DQN*), un algoritmo que integra el *Q-learning* con redes neuronales profundas para aproximar la función de valor Q, superando las limitaciones de los métodos clásicos que no pueden tolerar espacios de estado y acción grandes [1]. Este avance permitió que los agentes de aprendizaje por refuerzo alcanzaran un rendimiento similar al humano en varios juegos *Atari*, sentando un precedente en el campo [2].

El presente trabajo se centra en la aplicación del algoritmo *DQN* en el entorno simulado *LunarLander*, que plantea el reto de aterrizar un módulo lunar sobre una superficie irregular controlando sus propulsores. El objetivo es que el agente aprenda una política que permita resolver el problema con al menos un 30% de éxito. Para ello, se diseña y entrena una red neuronal *feed-forward* que estima los valores Q asociados a las acciones posibles dados los estados observados, aplicando técnicas estándar del aprendizaje por refuerzo profundo.

II. Preliminares

A. LunarLander

LunarLander es un entorno que simula la librería *Gymnasium* de *Python*. El objetivo es aterrizar un módulo lunar en la zona de aterrizaje, un parche plano de una superficie irregular [7]. El entorno se puede observar en la Figura 1.

En *Gymnasium*, el estado actual de un entorno se representa mediante una tupla de estado s. Cada paso, se aplica una acción a entre un número finito posible, dando lugar al siguiente estado s'. Esto se repite hasta llegar a un estado final, cuando se cumplen las condiciones de finalización del entorno. Este proceso se denomina un episodio. En *LunarLander*, el estado contiene la posición x, y, la velocidad v_x, v_y, el ángulo α, la velocidad angular ω, y si las piernas del módulo tocan el suelo (1):

$$S = (x, y, v_x, v_y, \alpha, \omega, leg\_down_l, leg\_down_r) \tag{1}$$

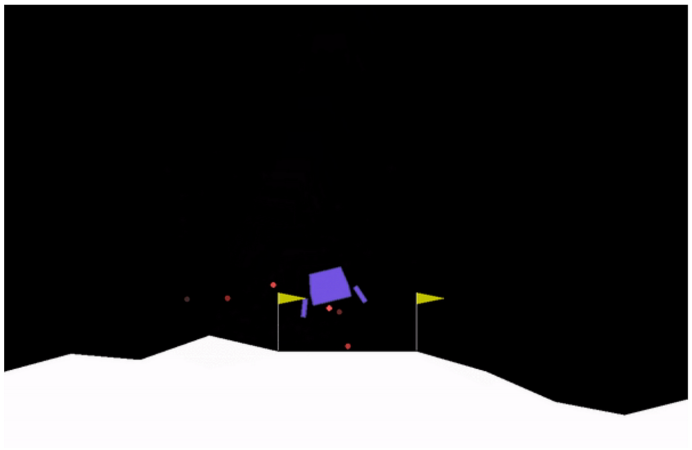

Fig. 1: Ejemplo de *LunarLander*

Las cuatro acciones posibles son: no hacer nada, encender el propulsor de giro izquierdo, encender el propulsor principal y encender el propulsor de giro derecho. Se representan con los números 0, 1, 2 y 3 respectivamente.

El estado inicial de un episodio se determina generando aleatoriamente la posición, la velocidad, el ángulo y la velocidad angular dentro de un rango establecido. El episodio se considera acabado cuando se cumple una de las siguientes condiciones:

- El módulo se estrella.
- El módulo sale de la pantalla.
- El módulo no está en movimiento ni en colisión con ningún cuerpo (es decir, está aterrizado).

Aplicar una acción a un estado resulta en una recompensa r, y el objetivo es maximizar la acumulación de esta recompensa. *LunarLander* define un modelo exitoso como uno que consiga, en promedio, una recompensa superior a 200. Usaremos este número como referencia para evaluar a los agentes entrenados. En el caso de *LunarLander*, la recompensa:

- Incrementa con la cercanía al punto de aterrizaje.
- Decrementa con la velocidad del módulo.
- Decrementa con la inclinación del módulo respecto a la horizontal.
- Incrementa en 10 puntos por cada pierna que toca el suelo.
- Disminuye en 0.03 puntos cada paso que un propulsor de giro está activo.
- Disminuye en 0.3 puntos cada paso que el propulsor principal está activo.
- Al acabar el episodio, se da una recompensa de $+100$ puntos o -100 puntos, dependiendo de si tuvo un aterrizaje seguro.

B. Q-Learning

El algoritmo *Q-learning* tradicional es la base del *deep Q-learning*. Resuelve un problema expresado como un conjunto de estados posibles S y un conjunto de acciones A, que parten de un estado y llevan con distintas probabilidades a otros. Además, cada transición tiene asociada una recompensa inmediata r. La estructura del problema ve representada en la Figura 2. Un episodio se puede representar como una secuencia de estados, acciones y recompensas $s_0, a_0, r_0, s_1, a_1, r_1, ...r_{n-1}, s_n$. El algoritmo resulta en una política (que decide qué acción tomar en cada estado) que maximiza el valor esperado descontado por un factor γ por cada paso R_t (2):

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2}$$

La ecuación de Bellman (3) define una función de estimación de valor ideal $Q^*(s, a)$ tal que el valor de tomar una acción es el valor esperado de, para cada siguiente estado s' posible, la recompensa de ir a ese estado r más el valor obtenible desde ese estado, calculado con la misma función $Q^*(s, a)$, proporcional a la probabilidad de obtener ese estado dada la acción tomada $P(s' \mid a)$ [1]:

$$Q^*(s, a) = E_{s' \sim P(s'|a)}[r + \gamma \max_{a'} Q^*(s', a')] \tag{3}$$

La política que para cada estado s elige tomar la acción a que maximiza $Q^*(s, a)$ es óptima (es decir, maximiza R_t) [2].

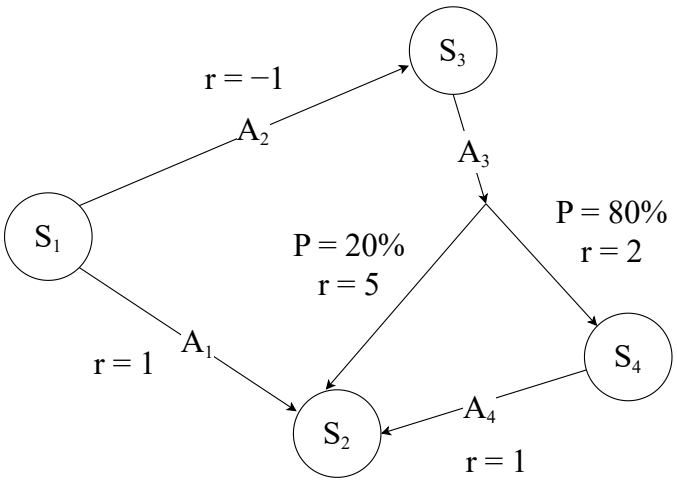

Fig. 2: Grafo que representa un problema en *Q-learning*.

	a_0	a_1
s_0	$q(s_0, a_0)$	$q(s_0, a_1)$
s_1	$q(s_1, a_0)$	$q(s_1, a_1)$

Fig. 3: Tabla de estados para $|S| = 2, |A| = 2$ en *Q-learning*

El algoritmo usa una tabla con los estados s en un eje y las acciones a en otro llamada Q-table y calcula en cada celda el valor de la función $q(s, a)$ usando programación dinámica, como se ve en la figura 3. Esto eventualmente converge a una política óptima en el infinito [2].

Sin embargo, requiere de memoria en el orden de $|S| \times |A|$. Para un dominio como el *LunarLander*, dado que la tupla contiene 6 números flotantes de 32 bits y dos booleanos, hay (muy aproximadamente) $(2^{32})^6 \times 2^2 = 2^{194} \approx 2.51 \times 10^{58}$ estados posibles. Proveer la cantidad de memoria que la tabla de este problema requiere no es viable, por lo que el método *Q-learning* no es aplicable.

C. Deep Q-network

Deep Q-network (*DQN*) es un algoritmo que combina aprendizaje por refuerzo con redes neuronales profundas. Aproxima la función de valor Q con una red neuronal, evitando tener que usar una tabla. Esto permite generalizar en espacios de estado continuos y de altas dimensiónes [4].

La red neuronal tiene un nodo de entrada por cada valor en el estado y un nodo de salida por cada acción posible. Pasando un vector de estado s como entrada, la salida asociada a la acción a es la estimación de la red de $q(s, a)$ [1], [2]. Se ve representado en la Figura 4. La capa de salida no tiene función de activación, ya que $q(s, a)$ puede tomar cualquier valor.

Para que la red interactúe con el entorno, se pasa el estado actual s como entrada y se selecciona la acción a_{max} que maximice la salida $q(s, a)$ (4):

$$a_{max} = \underset{a}{\operatorname{argmax}}(q(s, a)) \tag{4}$$

Sin embargo, para permitir que la red sea entrenada con más caminos que el que haya decidido que es óptimo (aumentando la precisión de la red) [2], con una probabilidad de la tasa de exploración $\mathcal{E}$ se elige una acción aleatoria entre todas las posibles. Esta política se llama $\mathcal{E}$**-voraz**.

Lo estándar es que $\mathcal{E}$ comience con un valor inicial $\mathcal{E}_0$ de 1, y sea multiplicado por un $\mathcal{E}_{decay}$ cada episodio hasta que llegue a un valor mínimo de $\mathcal{E}_{min}$. Estos tres son hiperparámetros del entrenamiento.

$$\mathcal{E} \leftarrow \max(\mathcal{E}_{decay} \times \mathcal{E}, \mathcal{E}_{min}) \tag{5}$$

La red se entrena utilizando la ecuación de Bellman para determinar el error. Como valor esperado y_{exp} se toma el $q(s, a)$ aproximado por la red para la acción a tomada en el estado s. Como valor real y_{real} se toma el valor que se puede conseguir

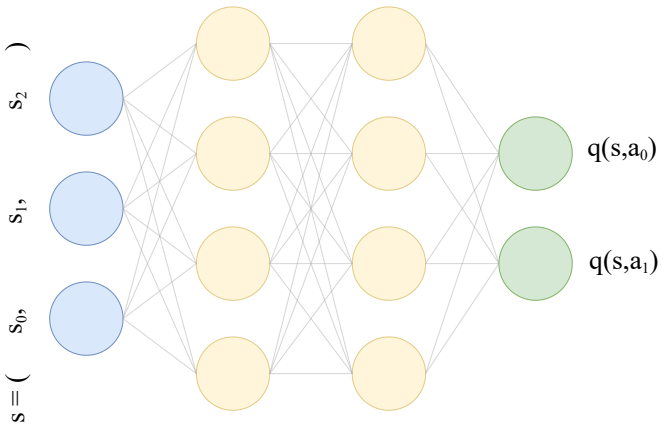

Fig. 4: Representación de una *Q-network*.

desde el estado resultante s'. En caso de ser terminal, es sencillamente la recompensa r. En caso de no serlo, a la recompensa r se le añade el valor máximo que la red estima que puede conseguir desde el estado resultante $\max_{a'}(q(s', a'))$, considerando todas las posibles acciones a' que se podrían tomar en ese estado.

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

$$y_{exp} = q(s, a, \theta)$$

$$y_{real} = \begin{cases} r + \max_{a'}(q(s', a', \bar{\theta})) & done = 0 \\ r & done = 1 \end{cases} \tag{6}$$

$$error = (y_{exp} - y_{real})^2$$

La variable θ representa los pesos de la red que se entrena, llamada ***Q-network***. $\bar{\theta}$ representa los pesos de una copia periódica de la red a entrenar, llamada ***training network***, que se usa por estabilidad [1] [2]. El ritmo de actualización de la *training network* en pasos es un hiperparámetro.

Dado que los casos de entrenamiento de una red neuronal deben ser independientes, y dos pasos consecutivos no son independientes, surge otra necesidad técnica: se usa una memoria de las acciones tomadas por la red llamada ***replay buffer***. Con cada acción que toma la red, se añaden a la memoria los datos mínimos necesarios para calcular su error: $(s, a, r, s', done)$. Esto se llama **almacenar la transición**. Si se supera el tamaño de la memoria, se descartan las transiciones más antiguas. Al momento de entrenar, se entrena con un minilote elegido aleatoriamente de la memoria. El tamaño de la memoria y el tamaño del minilote son hiperparámetros.

D. Mejoras adicionales de DQN

Se realizaron dos mejoras adicionales sobre el *DQN* explicado hasta ahora: el ***double DQN*** y el ***prioritized replay***.

1) Double DQN: El *double DQN* es una técnica que busca integrar en *DQN* las ventajas de *double Q-learning*, que reduce la sobreestimación de los valores Q que tienen los métodos clásicos. Esta ocurre porque se usa la misma función (la *training network*) para determinar la acción que tomar y estimar su valor, generando un sesgo positivo (7): [5]:

$$\begin{aligned} y_{real} &= r + \max_{a'}(q(s', a', \bar{\theta})) \\ max_{a'}(q(s', a', \bar{\theta})) &= q(s', \operatorname{argmax}_{a'}(q(s', a', \bar{\theta}), \bar{\theta}) \end{aligned} \tag{7}$$

[1]En este texto hay necesidades técnicas cuyo por qué no se explica en detalle. La referencia [2] contiene más información.

El *double DQN* busca adaptar a *DQN* la corrección que hace el *double-Q learning* al elegir la acción a tomar con la *Q-network* y calcular su valor con la *training network* [5] (8):

$$y_{real} = r + q(s', \underset{a'}{\operatorname{argmax}}(q(s', a', \theta), \bar{\theta}) \tag{8}$$

2) Prioritized Replay: El *prioritized replay* establece que el muestreo de la memoria se haga de manera proporcional al error de cada transición [6], suponiendo que las transiciones con más error son de los que más se puede aprender. En la memoria se almacena para cada transición i, junto al resto de datos, una prioridad p_i. Durante el muestreo, la probabilidad $P(i)$ de elegir cada transición de la memoria es proporcional a su prioridad.

Inicialmente, las transiciones se añaden a la memoria con una prioridad igual a la máxima ya existente, para priorizar que se entrene con ellas al menos una vez. Si la transición es elegida en un minilote, su prioridad se actualiza con el error obtenido, elevado a un hiperparámetro α (9):

$$p_i \leftarrow (y_{exp,i} - y_{real,i})^{\alpha} \tag{9}$$

Esto introduce un sesgo en los datos de entrenamiento [6] que es corregido escalando la gradiente generada por esa transición por un factor w_i (10). Este factor se atiene al exponente β, que siendo 0 causa que el peso no tenga efecto y siendo 1 anula el sesgo por completo. β incrementa linealmente desde el hiperparámetro β_0 hasta llegar a 1 al final del entrenamiento.

$$w_i = \left(\frac{1}{N} \times \frac{1}{P(i)}\right)^{\beta} \tag{10}$$

III. Metodología

A. Entrenamiento

El agente interactúa con el entorno *LunarLander* durante múltiples episodios.

En cada episodio:

1) Se restablece el entorno a un estado inicial aleatorio.
2) Se ejecuta la simulación, como se explicará a continuación.
3) Se aplica el decaimiento de $\mathcal{E}_{decay}$.
4) Se almacena la puntuación de la simulación y $\mathcal{E}$.
5) Cada 50 episodios, se calcula la puntuación promedio de los últimos 50 episodios y se almacena. También se hace un respaldo del modelo.

Al ejecutar la simulación, en cada paso:

1) Se obtiene el estado actual s del entorno.
2) Se selecciona la acción a con la política $\mathcal{E}$-voraz.
3) Se realiza la acción en el entorno y se obtiene el siguiente estado s', la recompensa r, y si es un estado final $done$.
4) Se almacena la transición $(s, a, r, s', done)$ en la memoria.

Y, un paso de cada tantos, siguiendo el ritmo de actualización de la *training network* [4]:

1) Se muestrea un minilote de transiciones aleatorias desde la memoria.
2) Se calcula el valor esperado y_{exp} y el valor real y_{real} (6) (8).
3) Si se está realizando *priority replay*, se actualiza la prioridad (9).
4) Se realiza la retropropagación del error cuadrático medio entre y_{exp} y y_{real}, usando descenso por gradiente. Si se está realizando *priority replay*, el error de cada caso i se escala con w_i.
5) Se copian los pesos de la *Q-network* a la *training network*.

Este proceso se ve representado en el Algoritmo 1.

Algoritmo 1 Entrenamiento del agente DQN

Entrada: número total de episodios N, hiperparámetros
Salida: modelo entrenado

Inicializar la *Q-network* y la *target network* con los mismos pesos aleatorios.
Inicializar $\mathcal{E}$: $\mathcal{E} \leftarrow \mathcal{E}_0$.
Inicializar la memoria.
Inicializar contador de pasos: $pasos \leftarrow 0$.
for $episodio = 1$ hasta N **do**
    Reiniciar el entorno a un estado inicial aleatorio s.
    while s no es terminal **do**
        Seleccionar la acción a usando política $\mathcal{E}$-greedy en la *Q-network*.
        Ejecutar acción a, obtener recompensa r, siguiente estado s' y $done$.
        Almacenar $(s, a, r, s', done)$ en la memoria.
        if tamaño del *replay buffer* $\geq$ tamaño de minilote **then**
            if $pasos \mod$ frecuencia de actualización de *target network* $= 0$ **then**
                Muestrear un minilote de transiciones $(s, a, r, s', done)$ de la memoria[a].
                Calcular el valor esperado y_{exp} y el valor real y_{real} (6) (8).
                Realizar la retropropagación sobre el error cuadrático medio entre y_{exp} y y_{real}[b].
                Copiar los pesos de la *Q-network* a la *training network*.
            end if
        end if
        Actualizar el estado: $s \leftarrow s'$.
        Incrementar contador de pasos $pasos \leftarrow pasos + 1$.
    end while
    Aplicar $\mathcal{E}_{decay}$: $\mathcal{E} \leftarrow max(\mathcal{E}_{decay} \times \mathcal{E}, \mathcal{E}_{min})$.
    Registrar la puntuación total del entrenamiento y $\mathcal{E}$.
    if $episodio \mod 50 = 0$ **then**
        Registrar puntuación promedio de los 50 episodios y guardar modelo.
    end if
end for
Devolver modelo entrenado

[a] En caso de hacer *priority replay*, se hace con probabilidades proporcionales al peso, y se muestrea $(s, a, r, s', done, w)$. w se calcula con β incrementando linealmente β_0 a 1 a lo largo del entrenamiento
[b] En caso de hacer *priority replay*, se escala el error por un factor de w_i. Además, se actualiza las prioridades (9)

B. Detalles de la red

Elegimos usar para la *Q-network* y *target network* una red *feed-forward* con dos capas ocultas de tamaño 64, ambas con función de activación *leaky ReLU*. La red tiene 8 nodos de entrada y 4 de salida, correspondiendo a la tupla de estados y las acciones del entorno *LunarLander*.

Se implementó en primera instancia con *Keras* usando *TensorFlow* de *back-end*, pero al no poder usar la *GPU* disponible para acelerarlo, el código se volvió a realizar en *PyTorch*.

IV. Resultados

El agente *DQN* se entrenó en el entorno *LunarLander* durante 1500 episodios, utilizando los hiperparámetros especificados en la tabla I:

TABLA I: Configuración de hiperparámetros del agente *DQN*

Parámetro	**Valor**
Número total de episodios	1500
Tasa de descuento (γ)	0.99
Tasa inicial de exploración ($\mathcal{E}_0$)	1.0
Decaimiento de $\mathcal{E}_{decay}$	0.995
Mínimo $\mathcal{E}_{min}$	0.01
Tamaño del *replay buffer*	10,000
Tamaño del minibatch	64
Frecuencia de actualización target network	Cada 10 episodios
Frecuencia de guardado del modelo	Cada 50 episodios
Tamaño de las capas ocultas	64 neuronas

El objetivo del experimento fue comprobar si el agente podía alcanzar una tasa de éxito superior al 30% en la tarea de aterrizaje controlado, y analizar la evolución de la recompensa acumulada durante el entrenamiento.

A. Agente base

En la primera gráfica (Figura 5) se muestra la evolución de la recompensa para cada episodio durante el entrenamiento. La curva azul representa la recompensa episodio a episodio, mientras que la curva naranja refleja la recompensa media de cada intervalo de 50 episodios.

En los primeros episodios las recompensas son muy bajas. Este es el resultado esperado, ya que el agente comienza con una *Q-network* aleatoria y no ha tenido experiencia suficiente para encontrar una política óptima. Además, aunque la tuviese toma acciones aleatorias al tener aún un $\mathcal{E}$ alto. Sin embargo, el agente *DQN* logra aprender rápidamente. La recompensa es positiva a partir del episodio 780, y 40 episodios más tarde se estabiliza por encima de los 200 puntos. La recompensa sigue una trayectoria claramente ascendente.

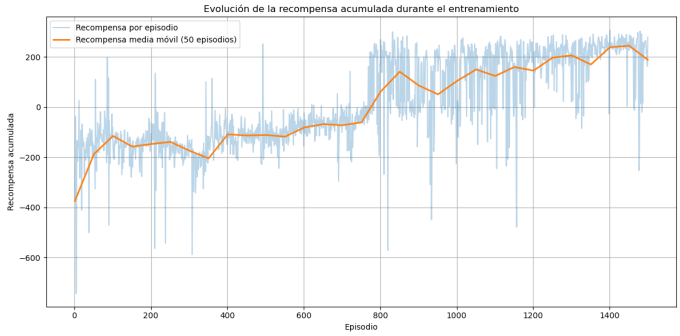

Fig. 5: Evolución de la recompensa para cada episodio durante el entrenamiento para el agente base

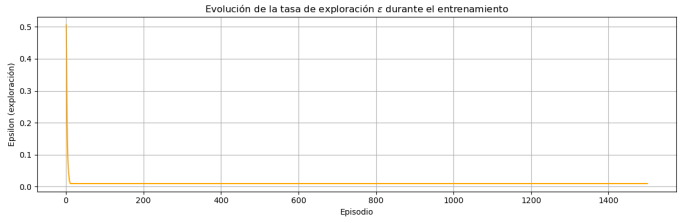

Fig. 6: Evolución de la tasa de exploración $\mathcal{E}$ durante el entrenamiento para el agente base

La tasa de exploración cae de manera repentina, como se puede ver en la Figura 6. Llega a su valor mínimo en el episodio 12. Esto es debido a un error de implementación en el que se aplicaba $\mathcal{E}_{decay}$ cada paso en lugar de cada episodio. Corregir este error resulta en resultados mucho peores, como se puede ver en la Figura 7 y la Figura 8.

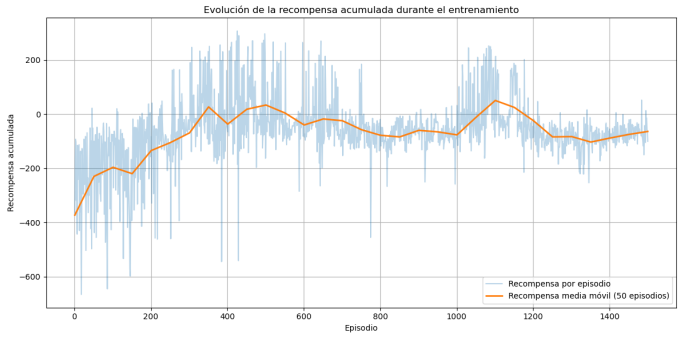

Fig. 7: Evolución de la recompensa para cada episodio durante el entrenamiento para el agente base con el $\mathcal{E}_{decay}$ corregido

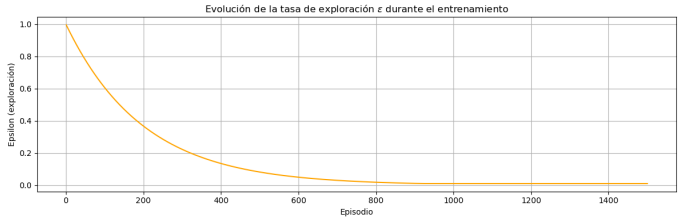

Fig. 8: Evolución de la tasa de exploración $\mathcal{E}$ durante el entrenamiento para el agente base con el $\mathcal{E}_{decay}$ corregido

B. Agente ampliado

El agente ampliado sigue una trayectoria más caótica que el agente base. La Figura 9 muestra los resultados del agente ampliado con *double DQN* y *prioritized replay*, con el error de actualización de $\mathcal{E}$ corregido. Consigue recompensas positivas, y cerca del final del entrenamiento consigue superar los 200 puntos. No tiene una trayectoria únicamente ascendente.

La figura 10 muestra el progreso de la tasa de exploración $\mathcal{E}$ una vez corregido el error. Sigue el comportamiento esperado, decayendo gradualmente hasta llegar a su valor mínimo en el episodio 919.

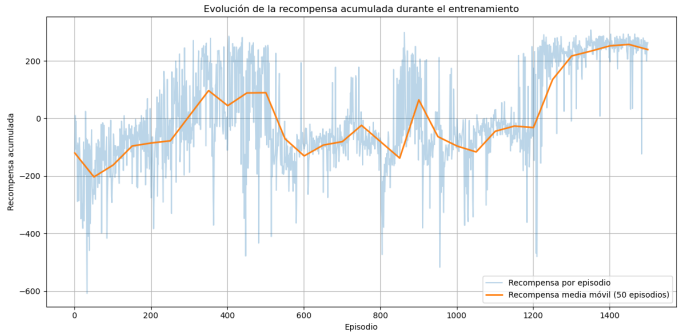

Fig. 9: Evolución de la recompensa para cada episodio durante el entrenamiento con el modelo mejorado con el $\mathcal{E}_{decay}$ corregido

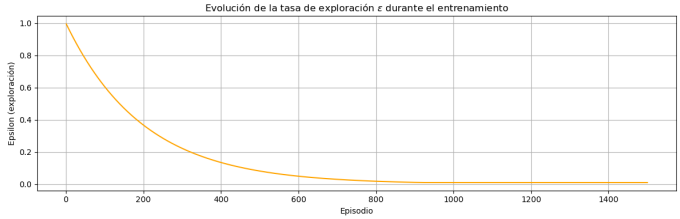

Fig. 10: Evolución de la tasa de exploración $\mathcal{E}$ durante el entrenamiento con el modelo mejorado con el $\mathcal{E}_{decay}$ corregido

A diferencia del modelo base, el modelo ampliado obtiene mejores resultados con el decaimiento de $\mathcal{E}$ corregido que sin corregir. Si no se corrige, $\mathcal{E}$ decae de manera excesivamente rápida, resultando en un entrenamiento más caótico y lento: se

obtienen recompensas negativas hasta el episodio 1300, como muestra la figura 5. En el episodio 1300 ocurre una mejora significativa de la recompensa obtenida, con episodios superando los 200 puntos. Sin embargo, la recompensa promedio nunca supera los 200 puntos, y poco después vuelve a bajar.

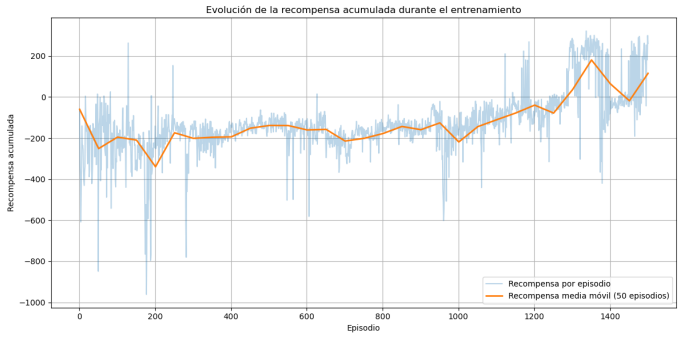

Fig. 11: Evolución de la recompensa para cada episodio durante el entrenamiento con el modelo mejorado sin el $\mathcal{E}_{decay}$ corregido

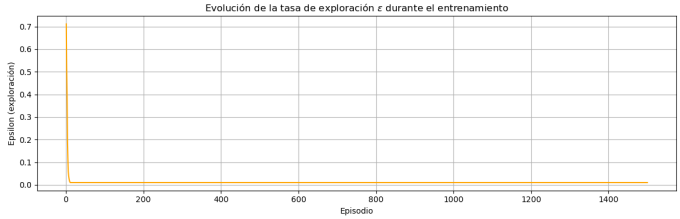

Fig. 12: Evolución de la tasa de exploración $\mathcal{E}$ durante el entrenamiento con el modelo mejorado sin el $\mathcal{E}_{decay}$ corregido

C. Comparación de los agentes

Los dos modelos resultantes fueron ejecutados sobre el entorno *LunarLander* durante 1000 episodios para medir su rendimiento de manera más fiable. El agente ampliado consigue mejorar tanto en la puntuación promedio como en la tasa de episodios con recompensa inferior a 200, como se puede ver en la Tabla II.

TABLA II: Resultados de la ejecución de los modelos.

	Agente base	**Agente sin ampliar**
Recompensa promedio	207.65	222.32
Tasa de episodios con recompensa inferior a 200	22.7%	18.6%

El código usado para el entrenamiento junto con los modelos resultantes puede ser encontrado en https://github.com/miniluz/aprendizaje-modulo-lunar.

V. Conclusiones

En este trabajo se implementó un agente *DQN* para resolver el entorno *LunarLander* mediante aprendizaje por refuerzo profundo. Se diseñó y entrenó una red neuronal *feed-forward* para aproximar la función de valor Q, permitiendo al agente aprender una política efectiva para controlar el módulo lunar.

Los experimentos realizados mostraron que el agente logra mejorar progresivamente su rendimiento, alcanzando recompensas positivas por encima de los 200 puntos en casi la mitad de los episodios, obteniendo los mejores resultados con un decaimiento de la tasa de exploración $\mathcal{E}$ precipitado en comparación con otros estudios relacionados. No conocemos con certeza el motivo por el que esto ocurre. Es posible que el rápido decaimiento del $\mathcal{E}$ resulte en que el agente únicamente refine la heurística de las acciones que prefiere y esto acelere su convergencia a una recompensa alta, a coste de la generalidad del modelo.

Además, se exploraron como ampliaciones las mejoras *double DQN* [5] y *prioritized replay* [6], que permiten corregir sesgos de sobreestimación y centrar el aprendizaje en las experiencias más relevantes, respectivamente. Este modelo funciona mejor con el error en el decaimiento de $\mathcal{E}$, como se espera. Se observan mejoras en el modelo resultante en comparación al producido por el agente base. Sin embargo, es difícil saber si esto se debe a las mejoras implementadas o a la variabilidad inherente al entrenamiento.

Debido a esto, se propone como trabajo futuro (idealmente destinado a un ordenador más potente) el entrenamiento de numerosos agentes con y sin las mejoras para poder evaluar si estas realmente llevan a mejores resultados. También se propone, como proyecto de aprendizaje, profundizar en la integración de otras mejoras sobre *DQN*, como *dueling DQN* [3], acercándose más al nivel conseguido por *rainbow DQN* [3] al combinar las mejoras conocidas en el momento.

VI. Autodeclaración del uso de la inteligencia artificial generativa

Hemos utilizado la inteligencia artificial generativa para:

- Realizar una revisión de la memoria.
- Realizar una revisión general del código y los comentarios que tiene.
- Realizar preguntas puntuales para explicar la documentación durante la investigación.
- Realizar preguntas puntuales para explicar fragmentos código de la documentación de Keras.
- Realizar preguntas puntuales para explicar errores en el código.

REFERENCIAS

[1] V. Mnih *et al.*, "Playing Atari with Deep Reinforcement Learning," *arXiv preprint arXiv:1312.5602*, 2013. [Online]. Available: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

[2] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015. doi:10.1038/nature14236. [Online]. Available: https://www.nature.com/articles/nature14236

[3] M. Hessel *et al.*, "Rainbow: Combining Improvements in Deep Reinforcement Learning," *arXiv preprint arXiv:1710.02298*, 2017. [Online]. Available: https://arxiv.org/pdf/1710.02298

[4] J. Bailey, "Deep Q-Networks explained," *LessWrong*, 2019. [Online]. Available: https://www.lesswrong.com/posts/kyvCNgx9oAwJCuevo/deep-q-networks-explained

[5] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," *arXiv preprint arXiv:1509.06461*, 2015. [Online]. Available: https://arxiv.org/pdf/1509.06461

[6] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," *arXiv preprint arXiv:1511.05952*, 2016. [Online]. Available: https://arxiv.org/pdf/1511.05952

[7] "Lunar Lander Environment," Gymnasium. [Online]. Available: https://gymnasium.farama.org/environments/box2d/lunar_lander/.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

LISTA DE ALGORITMOS