Aprendizaje por refuerzo: LunarLander & DQN

Lucía Campos Díez
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
luccamdie@alum.us.es

Nora Peñaloza Friqui
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
norpennfri@alum.us.es

***Resumen*—El objetivo de este trabajo es diseñar, crear y evaluar un agente inteligente capaz de resolver el entorno de aterrizaje del módulo lunar LunarLander. En concreto el objetivo es conseguir que una nave espacial llegue a la plataforma de aterrizaje sin impactar y sin salirse de la plataforma. Se ha experimentado con técnicas de aprendizaje por refuerzo, en concreto Deep Q-Network (DQN), donde la política de decisiones será una red neuronal Feed Forward simple. Se persigue que el modelo sea capaz de alcanzar al menos un 30 por ciento de aciertos de aterrizajes exitosos. Utilizando hiperparámetros optimizados como la tasa de descuento, de aprendizaje, la tasa de decaimiento de epsilon...**

Los resultados obtenidos muestran que el agente DQN es capaz de aprender una política eficiente. Durante la etapa de entrenamiento el agente ha sido capaz de mejorar su desempeño alcanzando una tasa de éxito superior a la solicitada, demostrando la gran utilidad de las técnicas de aprendizaje por refuerzo en problemas multiagente.

***Palabras clave*—Inteligencia Artificial, DQN, redes neuronales, Q-learning, Deep Q-Network, entorno, agente, estado, acciones, recompensa...**

I. Introducción

En los últimos años, el campo de la inteligencia artificial ha experimentado grandes avances en la tarea de resolución de problemas complejos. El área de aprendizaje por refuerzo (Reinforcement Learning, RL) no se ha quedado atrás, y se ha consolidado como una de las ramas o áreas más activas [1], teniendo una amplia variedad de aplicaciones exitosas, entre ellas el control robótico [2] [3]. RL es un paradigma del aprendizaje automático que permite que un agente autónomo aprenda a tomar decisiones secuenciales mediante la interacción directa con un entorno dinámico [4]. Este agente aprenderá a tomar decisiones óptimas en el entorno mediante prueba y error, guiado por las recompensas o sanciones, de modo que gradualmente desarrollará una estrategia para maximizar las recompensas acumuladas a largo plazo.

Dentro de este marco, el problema que se tratará en este artículo es el clásico problema del aterrizaje lunar, conocido como LunarLander [5]. Este entorno, proporcionado por el paquete Gymnasium, presenta el desafío de controlar una nave espacial para lograr una llegada estable a la plataforma de aterrizaje dentro de la zona limitada. El agente debe aprender a regular con precisión la activación de los motores, la orientación del módulo y la velocidad de descenso para no estrellarse ni salirse de la zona delimitada. El reto está en encontrar la estrategia adecuada para evitar accidentes y fallos en el terreno complejo de la Luna. Mediante las interacciones con el entorno y la guía de las recompensas y sanciones, el agente debe aprender y adaptarse a los diferentes estados y tomar decisiones óptimas para completar con éxito la tarea.

Para lograr esa tarea, será necesario diseñar un agente de aprendizaje por refuerzo capaz de resolver el entorno.En primer lugar, se dará una definición formal del problema y su entorno. [6]

1) *Estados del problema:* consiste en un vector de 8 dimensiones

$$(x, y, v_x, v_y, \theta, \omega, \text{left\_leg}, \text{right\_leg})$$

   - x, y representan las posiciones del módulo lunar en el espacio, coordenadas horizontales y verticales.
   - v_x, v_y son las velocidad lineal en X y en Y.
   - θ es el ángulo del módulo lunar respecto a una referencia fija.
   - ω es la velocidad angular.
   - left_leg, right_leg son indicadores binarios, indican si la pata izquierda o derecha están en contacto con el suelo.

2) *Acciones:* El espacio contiene cuatro acciones posibles:
   - No hacer nada (acción pasiva).
   - Disparar el motor de orientación izquierdo.
   - Disparar el motor principal (motor de empuje vertical).
   - Disparar el motor de orientación derecho.

   Estas acciones permiten controlar el aterrizaje ajustando la posición y la orientación.
3) *Recompensas:* el entorno también devuelve una recompensas para cada estado del problema, la recompensa final es la suma de los siguientes criterios:
   - Se aumenta o disminuye según la cercania de la nave con la plataforma.
   - Se aumenta si la velocidad del módulo es baja y disminuye en caso contrario.
   - Se reduce conforme aumenta la inclinación del módulo.
   - Se otorga 10 puntos por cada pierna que esté en contacto con el suelo.
   - Se resta 0.3 por cada fotograma en que se utilice un motor lateral.

- Se resta 0.3 por cada fotograma en que se utilice el motor principal.

4) *Objetivo:* El problema se considera resulto teóricamente cuando el agente consigue una puntuación de 200 puntos de recompensa.
5) *Observaciones a tener en cuenta:*
   - Basándose en las reglas, la recompensa máxima teórica por episodio deber ser al rededor de 200 puntos. En este caso, se considera resuelto si el agente consigue una puntuación de más de 190 puntos de recompensa en un episodio durante el entrenamiento.
6) *Estado inicial y condiciones de finalización:* El módulo espacial inicia cada episodio en la parte superior central de la ventana de visualización. Posteriormente el agente decidirá que acciones ejecutar y la nave actualizará su estado en consecuencia. Un episodio finaliza si la nave colisiona con la superficie lunar o se desplaza fuera del área visible.

Todo lo anterior esta ya implementado en el paquete Gymnasium, por lo que el enfoque está en el diseño del agente autónomo utilizando el algoritmo Deep Q-Network, una mejora del algoritmo Q-learning que emplea redes neuronales profundas para aproximar el valor de Q, en vez de una tabla como se hace con Q-learning. Esto permite operar eficientemente en entornos donde el espacio es continuo y de alta dimensión, como en LunarLander. [7] Evitando la necesidad de guardar todos los posibles valores estado-acción. La solución propocionada incluye tres elementos claves:

- Q-network, que es la red principal que estima los valores Q dados un estado.
- Una red objetivo (target network), que realiza una copia periódica de los pesos de la red principal.
- Un replay Buffer, una memoria que almacena experiencias pasadas, de las cuales se extraen mini-lotes aleatorios que se usasaran para estimar los valores Q, mejorando la estabilidad del aprendizaje.

Durante el entrenamiento, el agente debe encontrar el equilibrio adecuado entre exploración y explotación. Para ello, la red principal se entrena minimizando la función de pérdida que mide la diferencia entre los valores Q obtenidos por la Q-network y los objetivos calculados por la target network. Esto permite al agente a aprender políticas más óptimas y robustas a largo plazo, mejorando su capacidad de generalizar nuevas situaciones en el entorno.

El rendimiento del modelo se evalúa a lo largo de múltiples episodios de entrenamiento, analizando su capacidad para aprender la política óptima. Se considera que el entorno ha sido resuelto cuando el agente alcanza una puntuación igual o superior a 200 en al menos un 30 por ciento de los episodios, criterio que permite valorar tanto la eficiencia como la consistencia del enfoque propuesto.

El documento se organiza de la siguiente manera: en la sección Preliminares se lleva a cabo una pequeña introducción de las técnicas empleadas y se mencionan trabajos relacionados. Luego en la sección de Metodología, se profundiza en la descripción de los métodos utilizados, se presenta el algoritmo DQN y se detalla su implementación. El siguiente apartado es la sección de Resultados, donde se detallarán los experimentos realizados y los resultados conseguidos. Por último esta la sección de Conclusiones, se expone la conclusión general y se incluye una sección de uso de inteligencia artificial generativa.

II. Preliminares

En esta sección se presenta una introducción general a las técnicas que se han usado a la hora de realizar el trabajo. En primer lugar, se explicará Q-learning, dado que constituye la base teórica sobre la que se construye DQN. A continuación, se describirá el algoritmo DQN, destacando sus componentes fundamentales. Finalmente, se abordará el concepto de red neuronal feed-forward, que es la utilizada como aproximador de la función valor.

A. Métodos empleados

1) *Q-learning:* es un algoritmo fundamental de aprendizaje por refuerzo que permite a un agente aprender a tomar decisiones óptimas, basándose en la idea de aprendizaje mediante ensayo y error. Su objetivo principal es encontrar la estrategia óptima que guíe al agente a maximizar la suma de recompensas acumuladas a largo plazo.
   El aprendizaje se basa en un esquema de prueba y error, donde el agente explora el entorno seleccionando acciones, observa las recompensas inmediatas y los estados resultantes, y actualiza sus estimaciones de la función Q conforme a la siguiente regla:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big] \tag{1}$$

   Aquí, $\alpha \in (0,1]$ representa la tasa de aprendizaje, que controla la influencia de la nueva experiencia sobre el valor Q anterior; $\gamma \in [0,1]$ es el factor de descuento que pondera la relevancia de las recompensas futuras frente a las inmediatas; y r es la recompensa inmediata obtenida por realizar la acción a' en el estado s'. [8] El valor Q representa la suma esperada de recompensas futuras descontadas, por lo que el algoritmo busca aproximar esta función para todos los pares estado-acción. Si el agente conociera esta función a priori, podría seleccionar directamente la acción óptima en cada estado. Sin embargo, inicialmente no dispone de esta información, por lo que debe aprenderla gradualmente a partir de la experiencia.
   Durante el aprendizaje, el agente inicia con valores Q asignados arbitrariamente y va interactuando con el entorno, tomando acciones, observando las recompensas y estados siguientes, y actualizando sus estimaciones de Q mediante una regla que combina el valor anterior con la nueva información obtenida:

$$Q'(s_t,a_t) = (1-\alpha)Q(s_t,a_t) + \alpha\left[r(s_t,a_t) + \gamma \max_{a} Q(s_{t+1},a)\right] \tag{2}$$

Donde α facilita la convergencia estable incluso en ambientes ruidosos. Este proceso se repite iterativamente hasta que los valores convergen a los valores óptimos esperados. En casos donde los espacios de estados y acciones son discretos y finitos, esta actualización puede representarse como una tabla. Sin embargo, cuando el espacio de estados es grande o continuo, como ocurre en problemas con variables continuas (por ejemplo, velocidad o ángulo en el entorno LunarLander), el método tabular no es práctico. En estos escenarios, la función Q puede aproximarse mediante modelos paramétricos como redes neuronales, que permiten manejar espacios continuos y capturar mayor complejidad en la relación entre acciones y estados.

En resumen, Q-learning es un método basado en prueba y error que combina exploración y explotación para que un agente aprenda gradualmente la función Q y, a partir de ella, derive una política óptima que maximiza la recompensa a largo plazo, constituyendo la piedra angular sobre la cual se desarrollan métodos avanzados como los basados en redes neuronales profundas, tal como el algoritmo DQN.

Véase Fig. 1 para el pseudocódigo del algoritmo utilizado.

q-learning()
Entrada: entorno de aprendizaje por refuerzo
Salida: función $Q(s, a)$ aproximada
Inicializar $Q(s, a)$ **arbitrariamente para todos** s, a
para cada episodio hacer
  Inicializar el estado s
  mientras s **no sea terminal hacer**
    Elegir una acción a **usando política** ε**-greedy basada en** Q
    Ejecutar acción a**, observar recompensa** r **y nuevo estado** s'
    Actualizar $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
    $s \leftarrow s'$
  fin mientras
fin para

Fig. 1. Algoritmo de aprendizaje por refuerzo `Q-learning`

2) *DQN:* El algoritmo utiliza un perceptrón multicapa para estimar los valores de la función Q. La entrada de esta red corresponde al vector de estado actual(en LunarLander son 8 valores), y las salidas corresponden a los valores Q asociados a cada posible acción en este estado. Este enfoque permite al agente aprender directamente desde el espacio de estados continuos. La red actúa como una función aproximadora, capturando la relación entre estados, recompensas y acciones. Los detalles específicos sobre la arquitectura de la red y el proceso de entrenamiento se presentan en la sección de Metodología.
3) *Red neuronal feed-forward:* Para el caso específico del problema Lunar Lander, se ha optado por emplear una red neuronal feed-forward, también conocida como red neuronal totalmente conectada (fully connected neural network). Este tipo de red es una de las arquitecturas más simples y directas en aprendizaje profundo. Su estructura consiste en capas de neuronas donde cada unidad de una capa está conectada a todas las unidades de la siguiente. En particular, se implementó una red con dos capas ocultas, cada una con 64 unidades y función de activación ReLU. Esta configuración permite capturar relaciones no lineales entre las variables del entorno sin un elevado costo computacional, ya que la activación ReLU implica un umbral en cero y es computacionalmente eficiente [9]. La función de activación ReLU se define como:

$$f(x) = \max(0, x) \tag{3}$$

La función ReLU umbraliza la entrada: los valores negativos se convierten en cero, mientras que las entradas positivas permanecen sin modificar.

La capa de salida, de dimensión 4, devuelve una estimación de los valores Q correspondientes a las cuatro acciones posibles del agente. Esta configuración ha demostrado ser suficiente para lograr un rendimiento competitivo en el entorno, ofreciendo un buen equilibrio entre capacidad de generalización, rapidez de entrenamiento y complejidad computacional. La red se entrena mediante retropropagación, utilizando como función de pérdida el error cuadrático medio entre el valor actual y el valor objetivo definido por la ecuación de Bellman. [6]

$$L(\theta) = E_{(s,a,r,s')}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right] \tag{4}$$

Esta estrategia permite una aproximación eficiente de la política óptima incluso en espacios de estados continuos como los que presenta LunarLander.

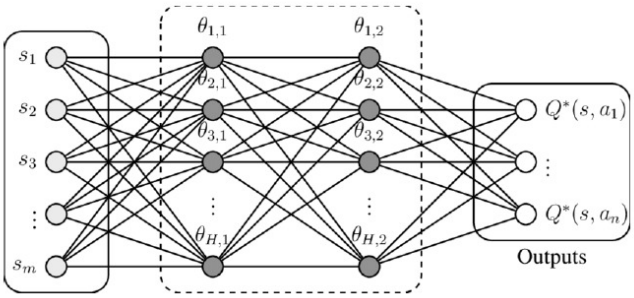

Fig. 2. Ejemplo de red neuronal feed-forward [6]

B. Trabajo Relacionado

A lo largo de los últimos años, el aprendizaje por refuerzo ha evolucionado significativamente, siendo una de las ramas más activas del aprendizaje automático. Una de las contribuciones fundamentales en este campo fue realizada por Sutton y Barto [1], quienes establecieron las bases teóricas del aprendizaje por refuerzo, incluyendo conceptos como los procesos de decisión de Markov (MDP) y el aprendizaje por diferencias temporales (TD). Posteriormente, Mnih et al. [7] introdujeron el algoritmo Deep Q-Network (DQN), que combina redes neuronales profundas con Q-learning, logrando un rendimiento

a nivel humano en videojuegos de Atari. Para entornos con espacios de acción continuos, Silver et al. [10] propusieron los algoritmos de política determinista (DPG) y Lillicrap et al. [11] extendieron esta idea con Deep Deterministic Policy Gradient (DDPG), integrando redes profundas para aprender directamente desde el espacio de estados continuo. Además, Schulman et al. [12] desarrollaron el algoritmo Proximal Policy Optimization (PPO), que mejora la estabilidad del entrenamiento mediante restricciones en las actualizaciones de política. Finalmente, entornos como LunarLander fueron proporcionados como estándar para comparar algoritmos de aprendizaje por refuerzo. Todos estos trabajos han sido fundamentales para el diseño del presente estudio, en el que se utiliza un DQN para resolver el entorno LunarLander, y se evalúa su rendimiento bajo condiciones, convirtiéndose en referentes para la validación de estos algoritmos.

III. Metodología

En este trabajo se resuelve el problema de LunarLander con el algoritmo Deep Q-Network. Este potente algoritmo combina los principios de las redes neuronales profundas con Q-learning, facilitando al agente el aprendizaje de políticas óptimas en escenarios complejos. Resulta ampliamente reconocido que las redes neuronales son una óptima manera de aproximar funciones no lineales, por lo tanto, DQN, usa las redes para aproximar la función Q sin necesidad de la tabla que se utiliza en Q-learning.

El uso de DQN en el problema se fundamenta en la necesidad de manejar un entorno de estado continuo donde las técnicas clásicas de Q-learning resultan inviables debido a todas las combinaciones estado-acción que pueden resultar. Como se ha mencionado anteriormente, el entorno del juego proporciona estados de 8 dimensiones, en las cuales incluye atributos continuos, dificultando así la discretización eficiente sin pérdidas de información significativas.

Este desafío lo resuelve DQN utilizando redes neuronales profundas como aproximadores para estimar la función Q. Esto permite generalizar a estados no vistos previamente, capturando patrones complejos y no lineales en los datos, algo que los métodos como Q-learning no pueden lograr. Además, las redes neuronales feed-forward son adecuadas para representar relaciones entre las características del estado y la calidad esperada de cada acción, garantizando una estimación más precisa y robusta del valor. Otra ventaja fundamental del DQN es la capacidad para entrenar directamente desde experiencias de interacción con el entorno, aplicando técnicas como la experiencia de repetición (experience replay) y la red objetivo (target network), que estabilizan el proceso de aprendizaje y mejoran la convergencia del algoritmo. Esto es crucial en entornos dinámicos y con ruido, como LunarLander, donde las recompensas pueden ser irregulares.

A continuación se presenta una descripción detallada de la estructura empleada en el trabajo. En el caso de DQN se usan en realidad dos redes neuronales con la intención de estabilizar el proceso de aprendizaje [13] La red principal esta representada por los parámetros θ. Es la encargada de estimar los valores Q del estado s y la acción a del presente. La segunda red, conocida como red objetivo o target network, esta parametrizada por θ' y se encargará de aproximar los valores Q del siguiente estado s' y la siguiente acción a'. El aprendizaje se realiza en la principal y no en la objetivo, es más la red objetivo se bloquea de modo que sus parámetros no se cambian en cada iteración sino que la objetivo se actualiza cada cierto número de pasos con los parametros de la principal, transmitiendose el aprendizaje y haciendo que las estimaciones calculadas sean más precisas por parte de la red objetivo.

$$Q(s_t, a_t, \theta) = r + \gamma \max_{a'} Q(s', a', \theta') \tag{5}$$

Se ha de destacar que ambas redes son idénticas en cuanto a arquitectura. En este caso, consta de una capa de entrada, donde lo que entra es el vector de estado actual, capturando toda la información relevante sobre el entorno. También se ha definido capas ocultas, totalmente conectadas al ser una red feed-forward. Cada capa aplica una transformación lineal seguida de una función de activación no lineal, en este caso la función ReLU, para introducir no linealidades que permiten modelar funciones complejas. En cuanto a la profundidad y tamaño de estas capas se suelen elegir para equilibrar la capacidad de modelado y la eficiencia computacional. En nuestro caso, se han usado 2 capas ocultas con 64 neuronas cada una. Normalmente en trabajos como LunarLander se suelen usar entre 2 o 3 capas ocultas con entre 64 y 256 neuronas [7]. Finalmente la capa de salida, para cada acción la red estima el valor Q, en este caso hay 4 acciones por lo que la salida esun vector de 4 valores Q. En la capa de salida, se utiliza una función de activación lineal para permitir la predicción de valores continuos.

Otro elemento a destacar de DQN es el Replay buffer. En aprendizaje por refuerzo, la experiencia del agente (estado, acción, recompensa, siguiente estado y fin de episodio) se almacenan en un buffer llamado Replay buffer. Esto permite al agente entrenar usando datos aleatorios del historial, en lugar de solo usar las transacciones secuenciales por las que acaba de pasar. De este modo, el aprendizaje es más eficiente y estable al mezclar experiencias pasadas y evitar depender solo de las secuencias actuales. En nuestro trabajo se ha implementado utilizando una estructura de datos llamada deque para almacenar de forma eficiente las experiencias.

- Inicialización: Se crea una cola doble con un tamaño fijo (buffersize). Garantiza de este modo que se mantenga un número limitado de experiencias, descartando las más antiguas cuando se llega al limite.
- Almacenamiento de experiencias (push): Cada vez que el agente ejecuta una acción en un estado, recibe una recompensa y observa el siguiente estado. Estos datos se añaden al buffer como una tupla (state, action, reward, nextState, done).
- Muestreo aleatorio (sample): Se extrea un mini-lote de experiencias de tamaño batch size. Es aleatorio y sin orden para romper la correlación temporal, lo que permite que la red sea más eficiente a la hora de estimar la función Q.

Gracias a deque con el tamaño fijo, el buffer actua como una ventana deslizante que siempre contine lo más reciente, permitiendo que el agente aprenda de manera estable y generalice mejor.

ReplayBuffer$(buffer\_size)$
Inicializar: buffer vacío con capacidad máxima $buffer\_size$

push$(state, action, reward, next\_state, done)$
Insertar la experiencia $(state, action, reward, next\_state, done)$ en el buffer

sample$(batch\_size)$
Seleccionar aleatoriamente una muestra de tamaño $batch\_size$ del buffer
Devolver los arrays de estados, acciones, recompensas, siguientes estados y flags de fin

__len__()
Devolver el número actual de experiencias almacenadas en el buffer

Fig. 3. Pseudocódigo del Replay Buffer

Antes de abordar el diseño del agente propuesto, es necesario detallar con mayor precisión la formulación de la función de pérdida utilizada durante el proceso de entrenamiento.

Para poder entrenar la red se necesita de una funicón de perdida o coste, la cual en este caso es la siguiente:

$$L(\theta) = E_{(s,a,r,s')}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right] \tag{6}$$

La función de pérdida en DQN es dependiente de la diferencia entre los Q-valores obtenido por la red principal y los estimados por la red objetivo. [6] Donde:

- θ son los parámetros de la red principal.
- θ^- son los parámetros de la red objetivos, los que se actualizan cada x tiempo.
- r es la recompensa inmediata obtenida tras ejecutar la acción a en el estado s.
- s' es el estado siguiente al aplicar la acción a.
- γ es el factor de descuento que pondera la importancia de recompensas futuras, el valor oscila entre 0 y 1.

Esta fórmula proviene de la ecuación de Bellman (5) y tiene como objetivo minimizar el error cuadrático entre el valor actual estimado $Q(s, a; \theta)$ y el valor objetivo $(r + \gamma \max_{a'} Q(s', a'; \theta^-)$. Esto representa la recompensa inmediata más el valor descontado de la mejor acción posible en el siguiente estado, estimado por la red objetivo.

El uso de la segunda red (la red objetivo) para calcular el valor objetivo tiene un papel crucial en la estabilidad del entrenamiento. Al mantener fijos los parámetros durante varios pasos de entrenamiento, se reduce la correlación entre las predicciones y los objetivos, evitando así oscilaciones o divergencia en el aprendizaje, lo cual es común al entrenar redes neuronales con datos secuenciales y altamente correlacionados como en los entornos de RL. [7]

Una vez establecidas las bases teóricas del algoritmo DQN, incluyendo su arquitectura, componentes fundamentales, asi como técnicas empleadas, se procede a describir detalladamente el diseño e implementación del agente propuesto para el entorno LunarLander.

En primer lugar se crea la instancia donde se configura todos los parámetros y componentes que regirán el comportamiento durante el entrenamiento. Los parámetros que recibe por entrada son los siguientes:

- lunar: instancia del entorno.
- gamma: factor de descuento para recompensas futuras.
- epsilon: tasa inicial de exploración.
- epsilonDecay y epsilonMin: para disminuir la exploración.
- learningRate:tasa de aprendizaje para optimizar la red.
- batchSize: tamaño de los minis-lotes.
- memorySize:capacidad del buffer.
- episodes: número total de episodios de entrenamiento.
- target network update freq: frecuencia para sincronizar.
- replays per episode: cantidad de actualizaciones extra al acabar un episodio para reforzar el entrenamiento.

Además, se inicializa los componentes que se han definido anteriormente, el buffer, la red principal y la objetivo. Y otro componente del que no se ha hablado hasta ahora,el optimizador. Este se encarga de actualizar los pesos de la red principal usando el gradiente calculado a partir de la función pérdida. Nuestro agente está compuesto por seis funciones principales, de las cuales tres son las más relevantes para el correcto funcionamiento y entrenamiento del modelo. Por tanto, se realizará un análisis detallado de estas tres funciones clave, mientras que las restantes, por su simplicidad, se describirán de manera más concisa en primer lugar.

1) *Actualización de la red objetivo:* Como se he indicado ya varias veces a lo largo del documento, la red objetivo no cambia sus parámetros en cada iteración sino que realiza una copia de la red principal cada cierto número de pasos. De este modo la función update target network permite que la red objetivo se sincronice con la principal solo cada ciertos números de episodios definidos por el parámtro target-network-update-freq. Realizándose una copia de los pesos.
2) *Guardar y cargar los modelos:* Las funciones save-model y load-model permiten persistir el estado de la red principal, para guardarlo o cargar un modelo previamente entrenado.
3) *Selección de acción:* La función act() implementa la política de elección de acciones del agente, basándose en el método epsilon-greedy. En primer lugar se obtiene el estado actual del entorno. A continuación, genera un número aleatorio para decidir si el agente debe explorar o explotar: si este número es menor que el valor de epsilon, el agente selecciona una acción aleatoria, lo que fomenta la exploración y evita que el modelo se estanque en políticas subóptimas. Si, en cambio, decide explotar, convierte el estado en un tensor compatible con PyTorch y lo pasa por la red neuronal principal para obtener los valores Q estimados para todas las acciones posibles. La acción escogida será aquella con el valor Q más alto, reflejando la mejor predicción del modelo en ese estado. Luego, la función ejecuta la

acción en el entorno, obteniendo el nuevo estado, la recompensa y si el episodio ha finalizado. Finalmente, devuelve estos valores para que puedan ser usados en el proceso de aprendizaje y toma de decisiones. Esta función es esencial para que el agente interactúe con el entorno de manera inteligente, combinando aleatoriedad e información aprendida para mejorar su desempeño progresivamente.

4) *Actualizar:* La función update-model() es responsable de actualizar los parámetros de la red neuronal principal mediante el proceso de experience replay, técnica clave en el algoritmo DQN para mejorar la estabilidad y eficiencia del aprendizaje, que se ha explicado anteriormente. Primero, se extrae un lote aleatorio de experiencias almacenadas en la memoria, donde cada experiencia consiste en un estado, una acción tomada, la recompensa recibida, el siguiente estado y un indicador de si el episodio terminó. Estos datos se convierten en tensores adecuados para ser procesados por PyTorch y se trasladan al dispositivo de cómputo (CPU o GPU). Luego, la red principal calcula los valores Q correspondientes a las acciones tomadas en los estados del lote, mientras que la red objetivo calcula el valor Q máximo para los siguientes estados, utilizando .detach() para evitar que esta parte del cálculo afecte la propagación de gradientes. A partir de estos valores, se construyen los valores objetivo Q esperados, que incorporan la recompensa inmediata y el valor descontado del futuro según el factor gamma, ignorando los estados terminales. El error o pérdida se calcula como el error cuadrático medio entre los valores Q predichos y los esperados. Finalmente, se realiza la retropropagación para ajustar los pesos de la red principal, minimizando esta pérdida, y se devuelve el valor de la pérdida para monitorear el entrenamiento. Esta función permite que el agente aprenda a predecir mejor los valores de las acciones y, por tanto, tome decisiones más óptimas en el entorno.
5) *Entrenar el agente:* La función train() es el núcleo del proceso de entrenamiento del agente DQN en el entorno LunarLander. En cada episodio, el agente reinicia el entorno para obtener un estado inicial y comienza a interactuar con él hasta que el episodio finaliza. Durante la interacción, el agente utiliza la función act() para seleccionar y ejecutar acciones, almacenando cada experiencia —formada por el estado, la acción tomada, la recompensa obtenida, el siguiente estado y si el episodio terminó— en el buffer. A medida que se acumulan suficientes experiencias, el agente actualiza la red neuronal principal mediante el método update-model(), ajustando sus parámetros para reducir el error entre las predicciones y los valores objetivos calculados. Tras concluir cada episodio, el agente realiza un repaso intensivo llamado "experience replay", ejecutando múltiples actualizaciones adicionales con experiencias almacenadas, lo que fortalece la generalización del aprendizaje. Paralelamente, el factor de exploración epsilon se reduce gradualmente para favorecer la explotación de políticas aprendidas sobre la exploración aleatoria. Además, periódicamente se sincronizan los pesos de la red objetivo con la red principal para mejorar la estabilidad del entrenamiento. Durante todo el proceso, se registra en consola información relevante como el número de episodio, la recompensa total obtenida, el valor actual de epsilon y la pérdida en la actualización de la red. Finalmente, se lleva un conteo de episodios exitosos —definidos al inicio de este documento— para calcular la tasa de éxito global al finalizar el entrenamiento.

A continuación se muestran los pseudocódigos de las tres funciones principales del agente DQN

act()
Entrada: estado actual s
Salida: siguiente estado s', recompensa r, `done`, acción a
si rand() $< \varepsilon$ **entonces**
    Seleccionar acción aleatoria $a \sim$ **Uniform**(A)
sino
    Calcular valores $Q(s,a)$ **para todas las acciones** a
    Seleccionar $a \leftarrow \arg\max_a Q(s,a)$
Ejecutar a **en el entorno y observar** s'**,** r**,** `done`
devolver $s', r,$ `done`$, a$

Fig. 4. Selección de acción con política ε-greedy

update_model()
Entrada: batch de experiencias $(s, a, r, s', done)$
Salida: pérdida de entrenamiento
Muestrear un minibatch aleatorio de la memoria de repetición
Calcular $Q(s,a)$ **usando la red principal**
Calcular $Q'(s',a') \leftarrow \max_{a'} Q_{\textbf{target}}(s',a')$
Calcular $Q_{\textbf{target}} \leftarrow r + \gamma Q'(s',a') \cdot (1 - done)$
Calcular la pérdida $L = \textbf{MSE}(Q(s,a), Q_{\textbf{target}})$
Actualizar los pesos de la red principal mediante retropropagación
devolver pérdida L

Fig. 5. Actualización de la red principal mediante aprendizaje por diferencia temporal

train()
para episodio = 1 **hasta** N
    Inicializar estado $s \leftarrow$ **reset**()
    mientras `done` **= False**
        Ejecutar *act*() $\rightarrow (s', r,$ `done`$, a)$
        Guardar transición $(s, a, r, s',$ `done`$)$ **en la memoria**
        si memoria suficiente entonces
            Ejecutar *update_model*()
        $s \leftarrow s'$
    fin mientras
    para repeticiones adicionales de entrenamiento
        si memoria suficiente entonces
            Ejecutar *update_model*()
    fin para
    Decaer $\varepsilon \leftarrow \max(\varepsilon_{\textbf{min}}, \varepsilon \cdot \textbf{decay})$
    si episodio $\bmod f = 0$ **entonces**
        Actualizar red objetivo $Q' \leftarrow Q$
fin para

Fig. 6. Entrenamiento del agente DQN

IV. Resultados

Durante el desarrollo del proyecto, se entrenó el agente bajo diferentes configuraciones de hiperparámetros con el objetivo de optimizar su rendimiento en el entorno LunarLander. Este entorno, de tipo continuo y parcialmente estocástico, requiere que el agente aprenda una política capaz de aterrizar de forma segura una nave espacial en una plataforma, maximizando la recompensa acumulada.

El proceso de entrenamiento incluyó la exploración sistemática de distintos valores para los hiperparámetros más relevantes del algoritmo DQN, como la tasa de aprendizaje (learning rate), el factor de descuento , el tamaño del batch, la capacidad del replay buffer, el parámetro de exploración (epsilon) y su tasa de decremento, así como la frecuencia de actualización de la red objetivo (target network update). Estos valores influyen directamente en la estabilidad del aprendizaje y en la velocidad de convergencia del modelo.

El objetivo establecido fue alcanzar al menos una tasa de éxito del 30 por ciento, entendida como el porcentaje de episodios en los que el agente lograba aterrizar con éxito. Este umbral inicial sirvió como referencia para validar las primeras configuraciones. A lo largo de sucesivos experimentos, y mediante el uso de técnicas como el experience replay y la separación entre la red principal y la red objetivo, se consiguió mejorar sustancialmente el rendimiento del agente. Al finalizar el proceso de ajuste fino de hiperparámetros, se obtuvo una tasa de éxito superior al 70 por ciento, lo que demuestra la capacidad del agente para aprender una política eficiente y robusta en un entorno complejo y con ruido.

A continuación se describen en detalle los experimentos realizados. Se presenta la configuración utilizada en cada uno de ellos, así como los objetivos específicos perseguidos y los resultados cuantitativos obtenidos. Además, se analiza el impacto de cada hiperparámetro en el rendimiento general, lo que permite identificar cuáles son los más determinantes en el proceso de aprendizaje profundo por refuerzo.

La primera evaluación del modelo se realizó utilizando una configuración base de hiperparámetros, sin realizar ajustes previos ni procesos de optimización. El objetivo de esta prueba inicial era establecer una línea base de rendimiento para el agente DQN en el entorno LunarLander. A continuación, se detallan los valores utilizados:

TABLA I
HIPERPARÁMETROS UTILIZADOS EN LA EVALUACIÓN INICIAL

Hiperparámetro	**Valor**
Entorno	`LunarLander`
Factor de descuento (γ)	0.99
ϵ inicial	1.0
ϵ mínimo	0.01
Decaimiento de ϵ	0.995
Tasa de aprendizaje	0.001
Tamaño del batch	64
Capacidad del buffer de memoria	10,000
Frecuencia actualización red objetivo	10
Episodios de entrenamiento	1,500
Actualizaciones por episodio	1

Es importante destacar que el parámetro actualizaciones por episodio, encargado de definir cuántas veces se actualiza la red neuronal por episodio, se fijó en 1. Esto implica que únicamente se realiza una actualización del modelo por cada episodio jugado, lo que ralentiza significativamente el proceso de aprendizaje.

Durante esta primera evaluación, el agente comenzó con un comportamiento aleatorio, como resultado de una política ϵ-greedy con ϵ =1.0. A medida que se incrementaron los episodios y decayó, el agente fue acumulando experiencia y mejorando paulatinamente su rendimiento.Aunque en algunas ocasiones el agente logró alcanzar e incluso superar el umbral mínimo de rendimiento fijado, en otras oportunidades sólo se acercó a dicho valor sin lograr superarlo de manera consistente. Esta variabilidad se atribuye principalmente a la baja frecuencia de actualización de la red objetivo, lo que resultó en una mejora general lenta y poco estable.

Para evaluar los resultados de cada configuración, se utilizó como métrica la tasa de acierto, definida como el cociente entre el número de episodios con una recompensa total mayor o igual a 190 y el número total de episodios de entrenamiento. Esta métrica refleja la proporción de episodios exitosos y permite comparar de forma objetiva el desempeño del agente bajo diferentes configuraciones.

Este experimento evidenció la necesidad de modificar algunos de los hiperparámetros críticos del modelo —especialmente el número de actualizaciones por episodio— para acelerar el aprendizaje y permitir que el agente mejore su comportamiento de forma más eficiente.

En el siguiente experimento que se realizó, se modificaron alguunos hiperparámetros clave para ver su influencia con el rendimiento del agente. Las modificaciones con respecto al primer intento son las siguintes:

- $\epsilon_{\min}$: se exploraron valores en el rango $[0.01, 0.05]$, con el objetivo de analizar cómo afecta la reducción de la exploración mínima al rendimiento a largo plazo del agente.
- Tasa de aprendizaje (α): se evaluaron tasas entre 0.0005 y 0.001, buscando un equilibrio entre velocidad de aprendizaje y estabilidad.
- Frecuencia de actualización de la red objetivo: se redujo de 10 a 5 episodios para favorecer una adaptación más ágil de la política aprendida.

Tras múltiples pruebas combinando estas variaciones, la configuración que logró el mejor rendimiento fue aquella con $\epsilon_{\min}$= 0.05, una tasa de aprendizaje de 0.0005, y actualización de la red objetivo cada 5 episodios. Esta combinación permitió alcanzar una tasa de éxito superior al 40 por ciento, mejorando significativamente respecto a los experimentos anteriores.

Tras estas últimas pruebas, el agente ya había superado con éxito la tasa mínima requerida. Por lo tanto, a partir de este punto, los experimentos se enfocaron en maximizar la tasa de acierto, buscando optimizar el rendimiento dentro de las limitaciones computacionales de un equipo sin GPU. A continuación se presentan los principales experimentos realizados y sus resultados:

TABLA II
RESULTADOS DE LOS EXPERIMENTOS CON DIFERENTES CONFIGURACIONES DE HIPERPARÁMETROS

Experimento	Episodios	$\epsilon_{\min}$	Replays/Episodio	Tasa de acierto (%)
1	2500	0.05	150	60
2	3000	0.05	150	65
3	3000	0.01	150	66
4	5000	0.01	200	70

En estos experimentos se realizaron modificaciones adicionales, no reflajados en la tabla por ser comunes, respecto a los anteriores, destacando principalmente el aumento de la capacidad del buffer de experiencia a 50,000 transiciones y la ligera reducción de la tasa de decaimiento de epsilon de 0.995 a 0.996. La ampliación del buffer permitió almacenar una mayor diversidad y cantidad de experiencias acumuladas durante el entrenamiento, favoreciendo un muestreo más representativo y evitando el sobreajuste a transiciones recientes, lo que mejora la estabilidad y robustez del aprendizaje. Por otro lado, al incrementar ligeramente la tasa de decaimiento de epsilon, se prolongó la fase de exploración del agente, permitiendo que este continúe explorando el entorno durante más episodios antes de reducir la exploración para favorecer la explotación. Esta exploración extendida facilitó la identificación de estrategias más óptimas y evitó caer prematuramente en políticas subóptimas, contribuyendo a una mejora en el desempeño final del agente.

En el primer experimento se mantuvo igual el factor de descuento (este no se verá modificado en ningun experimento), el tamaño de los mini-lotes y la frecuencia de actualización. Por otro lado, se fijo $\epsilon_{\min}$= 0.05 y se elevo el número de replays por episodio a 150. El valor mínimo de $\epsilon_{\min}$=0.05 mantuvo cierto nivel de exploración residual en fases avanzadas, ayudando al agente a seguir descubriendo trayectorias beneficiosas. El aumento del número de episodios, de 1500 a 2500, proporcionó al agente más oportunidades de interacción con el entorno, lo que permitió una exploración más exhaustiva del espacio de estados y una mejora progresiva de la política aprendida. Finalmente, el incremento a 150 replays por episodio permitió actualizar con mayor frecuencia los pesos de la red neuronal, acelerando la convergencia. El objetivo principal era proporcionar al agente una mayor base de experiencias para el aprendizaje y permitir más pasos de entrenamiento por episodio, esperando una mejora en la estabilidad y precisión del aprendizaje. Como resultado de estas modificaciones, el agente logró alcanzar una tasa de acierto del 60 por ciento, evidenciando una mejora significativa respecto a los experimentos iniciales y situándose por encima del umbral de rendimiento mínimo establecido.

En el segundo experimento, se mantuvieron todos los hiperparámetros constantes respecto a la configuración anterior, salvo por un incremento en el número de episodios de entrenamiento, que pasó de 2500 a 3000. Este ajuste tuvo como objetivo otorgar al agente más tiempo de entrenamiento para consolidar y refinar la política aprendida, permitiéndole una mejor explotación de la experiencia acumulada.

La única modificación —el aumento de 500 episodios adicionales— resultó en una mejora notable de la tasa de acierto, la cual ascendió del 60 al 65 por ciento. Este resultado sugiere que, dada una configuración adecuada de los hiperparámetros restantes, proporcionar al agente más tiempo para interactuar con el entorno tiene un efecto positivo directo sobre su rendimiento. El mayor número de episodios permitió reforzar decisiones previamente aprendidas, estabilizar la política y reducir la varianza en los resultados obtenidos en los episodios finales del entrenamiento

En el tercer experimento, se mantuvieron inalterados todos los hiperparámetros utilizados en el experimento anterior, salvo el valor de $\epsilon_{\min}$, que fue reducido de 0.05 a 0.01. Esta modificación tenía como propósito disminuir el grado de exploración mínima en las fases finales del entrenamiento, favoreciendo una mayor explotación del conocimiento previamente adquirido por el agente.

El resultado de este ajuste se tradujo en una mejora ligera pero apreciable en la tasa de acierto, la cual aumentó del 65 al 66 por ciento. Esta ganancia puede atribuirse a que, al limitar la aleatoriedad en la selección de acciones en las últimas etapas del entrenamiento, el agente logró consolidar una política más estable y centrada en las acciones óptimas ya descubiertas. En otras palabras, al permitir que el agente se concentre más en la explotación que en la exploración una vez completado el proceso de aprendizaje, se maximiza el uso eficiente de la experiencia acumulada.

Si bien el incremento en la tasa de acierto no fue drástico, este experimento demuestra que ajustes aparentemente sutiles en parámetros relacionados con la exploración pueden tener un efecto positivo en el rendimiento final del agente.

En el cuarto y último experimento, se introdujeron dos modificaciones principales respecto al experimento anterior. Por un lado, se incrementó el número total de episodios de entrenamiento de 3000 a 5000, y por otro, se aumentó el número de repeticiones por episodio (replays) de 150 a 200. Ambos cambios estaban orientados a reforzar el proceso de aprendizaje del agente y a consolidar su política de toma de decisiones.

El aumento del número de episodios proporcionó al agente una mayor cantidad de interacciones con el entorno, permitiéndole acumular más experiencias y realizar más actualizaciones de su red neuronal. Esta prolongación del entrenamiento es particularmente beneficiosa en entornos complejos como LunarLander, donde una política eficiente requiere la integración de múltiples patrones de comportamiento.

Asimismo, el incremento del número de replays por episodio facilitó un aprendizaje más intensivo dentro de cada ciclo de entrenamiento. Esto permitió realizar más pasos de retropropagación de error por cada episodio jugado, aprovechando de manera más efectiva las experiencias almacenadas en el buffer de memoria. En conjunto, estos ajustes promovieron una mejora sustancial en la calidad de la política aprendida.

Como resultado, la tasa de acierto alcanzó un valor del 70 por ciento, superando ampliamente el umbral mínimo establecido del 30 por ciento. Este experimento puso de manifiesto la

importancia de disponer de un entrenamiento más extenso y de una estrategia de actualización más intensiva para maximizar el rendimiento del agente bajo restricciones computacionales moderadas (sin uso de GPU).

V. Conclusiones

A lo largo de este trabajo se ha implementado y entrenado un agente basado en Deep Q-Network (DQN) con el objetivo de resolver el entorno LunarLander. El proceso incluyó tanto el diseño del agente como la experimentación con diferentes configuraciones de hiperparámetros. Inicialmente, se estableció una tasa mínima de acierto del 30 por ciento como criterio de éxito. A través de una serie de experimentos progresivos, y gracias al ajuste cuidadoso de parámetros como el número de episodios, el valor mínimo de $\epsilon_{\min}$, la cantidad de replays por episodio y el tamaño del buffer de memoria, se logró superar ampliamente este umbral, alcanzando finalmente una tasa de acierto del 70 por ciento.

En términos de resultados, se observó una mejora continua en el rendimiento del agente a medida que se aplicaban modificaciones estructurales en la configuración. El aumento del tamaño del buffer permitió conservar más experiencias relevantes, el incremento de los replays favoreció un aprendizaje más intensivo por episodio, y la disminución del valor $\epsilon_{\min}$ contribuyó a una mayor explotación del conocimiento adquirido. Igualmente, extender el número total de episodios de entrenamiento ofreció al agente más oportunidades para consolidar una política efectiva. Cada cambio individual tuvo un impacto medible, pero fue la combinación estratégica de estos ajustes lo que permitió alcanzar el máximo rendimiento.

Como conclusión principal, se constata que la optimización de un agente DQN en entornos complejos como LunarLander depende críticamente del ajuste fino de los hiperparámetros. Aunque el algoritmo DQN cuenta con una estructura robusta, su desempeño puede variar significativamente según cómo se configuren elementos como la política de exploración, el ritmo de aprendizaje y la frecuencia de actualización. Asimismo, incluso sin el uso de GPU, mediante una planificación cuidadosa es posible lograr un rendimiento competente dentro de los recursos computacionales disponibles.

Para futuras líneas de trabajo, una primera mejora posible sería la incorporación de técnicas más avanzadas como Double DQN, Dueling DQN o Prioritized Experience Replay, que podrían mejorar tanto la estabilidad como la velocidad de convergencia. Además, contar con recursos computacionales más potentes (especialmente con aceleración por GPU) permitiría realizar más experimentos, extender los episodios o aumentar la capacidad de memoria sin afectar la viabilidad temporal del entrenamiento.

En resumen, este proyecto demuestra no solo la eficacia de los métodos de RL basados en redes neuronales profundas, sino también la importancia de un enfoque sistemático y experimental para lograr soluciones robustas y eficientes en entornos dinámicos y estocásticos.

VI. Uso de Inteligencia Artificial Generativa

Durante el desarrollo de este trabajo se ha hecho uso de herramientas de inteligencia artificial generativa con el objetivo de mejorar la calidad formal y técnica del documento. En concreto se ha usado para las siguientes tareas: En primer lugar, para la corrección ortográfica y gramatical; para detectar errores y garantizar una redacción clara y adecuada. Por otro lado, se ha hecho uso también para evitar repeticiones innecesarias y enriquecer el vocabulario utilizado. Finalmente, se consultó para generar expresiones en lenguaje Latex, especialmente en pseudo-código y en las fórmulas matemáticas.

Los prompts proporcionadas a la IA fueron siempre específicos, incluyendo el fragmento de texto a revisar o bien la expresión matemática que debía representarse. Cabe destacar que la IA solo ha actuado como apoyo técnico y linguístico, no como generador de contenido.

Referencias

[1] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.

[2] Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement learning in robotics: A survey. *International Journal of Robotics Research*, 32(11), 1238–1274.

[3] Silver, D., Huang, A., Maddison, C. J., et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529, 484–489.

[4] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

[5] Farama Foundation. *Gymnasium LunarLander Environment Documentation*. Disponible en: https://gymnasium.farama.org/environments/box2d/lunar_lander/

[6] Propuesta de trabajo del profesor Miguel Bermudo Bayo. *Aprendizaje por Refuerzo en Lunar Lander*. PDF proporcionado al inicio del trabajo.

[7] Mnih, V., et al. (2015). Human-level control through deep reinforcement learning. *Nature*.

[8] Sánchez, F. J. (2025). Aprendizaje por refuerzo: algoritmo Q Learning. Disponible en: https://www.cs.us.es/~fsancho/Blog/posts/Aprendizaje_por_Refuerzo_Q_Learning.md. Consultado: 4 de junio de 2025.

[9] DataCamp. (2023). *Introducción a las funciones de activación en redes neuronales*. Recuperado de https://www.datacamp.com/es/tutorial/introduction-to-activation-functions-in-neural-networks

[10] Silver, D., et al. (2014). Deterministic Policy Gradient Algorithms. *International Conference on Machine Learning (ICML)*.

[11] Lillicrap, T. P., et al. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

[12] Schulman, J., et al. (2017). Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*.

[13] Markel Sanz. https://markelsanz14.medium.com/introducci%C3%B3n-al-aprendizaje-por-refuerzo-parte-3-q-learning-con-redes-neuronales-algoritmo