De cero a aterrizar naves: construyendo nuestro agente sin estrellarnos (demasiado)

Carlos Cerdá Morales
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
carcermor@alum.us.es KCG5658

José Luis Moraza Vergara
dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
josmorver@alum.us.es NBL3749

Este documento constituye la memoria del trabajo realizado por parte de los alumnos Carlos Cerdá Morales y José Luis Moraza Vergara, correspondiente a la sección de Aprendizaje por Refuerzo, centrado específicamente en el entorno *LunarLander* del paquete *Gymnasium* y en la implementación del algoritmo *Deep Q-Network* (*DQN*).

I. Introducción

En esta memoria se presenta una explicación introductoria sobre el funcionamiento de las *Deep-Q Networks*, utilizadas para el desarrollo de este proyecto en lenguaje Python. Se parte de unos conocimientos previos que incluyen fundamentos básicos de redes neuronales y retropropagación del error, así como nociones sobre los algoritmos de *Q-learning* basados en tablas. A diferencia de estos últimos, que almacenan valores Q en estructuras tabulares, las *DQN* emplean redes neuronales profundas como función de aproximación para estimar los valores de acción, lo que permite resolver problemas con espacios de estados grandes o continuos, como es el caso de *LunarLander*.

A lo largo del documento se describe el trabajo realizado y la metodología seguida, detallando los distintos pasos llevados a cabo para construir el agente, entrenarlo, ajustar sus hiperparámetros y evaluar su rendimiento. Se incluyen además pruebas con diferentes configuraciones de red y episodios, junto a un análisis de los resultados obtenidos y las dificultades encontradas durante la implementación y entrenamiento del modelo.

Todas las fuentes utilizadas para la elaboración de este trabajo han sido recopiladas y se encuentran referenciadas en el apartado de bibliografía del final del documento.

II. Preliminares

A. *Lunar Lander de Gymnasium*

El entorno *LunarLander* forma parte de la colección de simulaciones que ofrece la librería *Gymnasium* en *Python*. Este entorno recrea el desafío de controlar un módulo lunar que debe descender y aterrizar con precisión en una zona concreta de una superficie irregular. Dicha zona de aterrizaje es plana, y el objetivo principal es lograr un aterrizaje suave y controlado dentro de sus límites.

Cada episodio comienza en un estado inicial generado aleatoriamente, definiendo aspectos como la posición del módulo, sus velocidades lineales y angulares, así como su inclinación. El estado del entorno se representa mediante una tupla de ocho elementos que reflejan la dinámica del módulo: su posición en los ejes x e y, su velocidad en ambas direcciones, el ángulo de inclinación α, su velocidad angular ω, y dos valores *booleanos* que indican si las patas izquierda y derecha del módulo están tocando el suelo.

Durante cada paso del episodio, el agente puede elegir entre cuatro acciones: no hacer nada, activar el propulsor principal, o activar los propulsores laterales (izquierdo o derecho). Estas acciones están codificadas como los enteros 0, 1, 2 y 3. Al aplicar una acción, el entorno evoluciona al siguiente estado, generando también una recompensa asociada a esa transición. Este ciclo se repite hasta que el episodio llega a su fin, lo cual puede ocurrir por tres razones principales: el módulo se estrella, sale del área visible o se detiene por completo tras aterrizar con éxito.

El sistema de recompensas de *LunarLander* está diseñado para guiar al agente hacia comportamientos seguros y eficientes. Se premia acercarse al punto de aterrizaje, mantener una orientación horizontal adecuada y reducir la velocidad de caída. Además, el contacto de cada pata con el suelo otorga una bonificación de 10 puntos. Por el contrario, se penaliza el uso prolongado de los propulsores: cada paso con un propulsor lateral activo resta 0.03 puntos, y el uso del propulsor central cuesta 0.3 puntos por paso. Al final del episodio, se otorga una recompensa significativa: +100 si el aterrizaje fue exitoso o −100 si terminó en accidente.

Para evaluar el comportamiento del agente, se observa la recompensa acumulada que este obtiene a lo largo de los episodios. Cuanto más suave, preciso y eficiente sea el aterrizaje, mayor será la recompensa final. En general, un agente que ha aprendido correctamente tiende a obtener recompensas positivas de forma consistente, lo que indica que

ha desarrollado una política efectiva para completar la tarea de aterrizaje de manera segura.

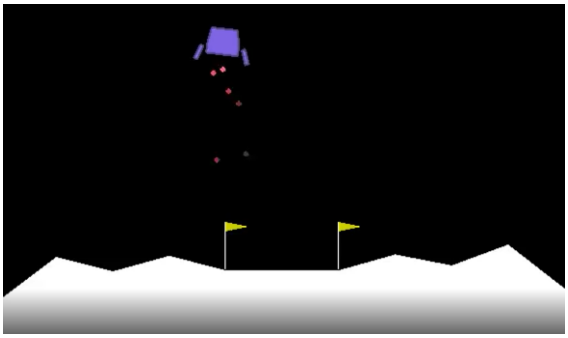

Fig 1. Captura de *LunarLander* durante la ejecución de un episodio

B. *Deep Q-Learning*

En este proyecto se ha implementado el algoritmo *Deep Q-Learning*, una técnica de aprendizaje por refuerzo basada en el uso de redes neuronales para aproximar la función de valor Q. Este método permite a un agente aprender a tomar decisiones en un entorno mediante la interacción directa con él.

Para ello, se emplean mecanismos como el uso de redes neuronales duales, una política de exploración , y una memoria de experiencias (replay buffer), que en conjunto estabilizan y optimizan el proceso de aprendizaje.

Componentes del algoritmo *Deep Q-Learning*:

- *Q-Network*:
  La *Q-network* es una red neuronal encargada de estimar los valores Q dado un estado de entrada. Formalmente, devuelve Q(s, a; θ), donde θ representa los pesos de la red y s es el estado del entorno. Su objetivo es aproximar la función de acción-valor óptima, ayudando al agente a decidir qué acción tomar en cada situación. Esta red es la que se entrena activamente durante el aprendizaje mediante retropropagación.

- *Target Network*:
  La *target network* tiene la misma arquitectura que la *Q-network*, pero sus parámetros θ^- se mantienen congelados temporalmente para proporcionar objetivos de entrenamiento más estables. Se actualiza copiando los pesos de la *Q-network* cada cierto número de episodios. Esta red se usa para calcular los valores Q de los siguientes estados al estimar la función objetivo.

- *Replay Buffer*
  El *replay buffer* es una memoria de experiencias que almacena transiciones pasadas en forma de tuplas (s, a, r, s′, done). Estas representan el estado actual, la acción tomada, la recompensa recibida, el estado siguiente y si el episodio ha finalizado. Durante el entrenamiento, se extraen aleatoriamente lotes (batches) de esta memoria para actualizar la *Q-network*. Esta técnica rompe la correlación temporal entre las experiencias y mejora la eficiencia del aprendizaje.

Funcionamiento del algoritmo y entrenamiento:

El entrenamiento del agente se realiza mediante una función *train()*, que ejecuta una serie de episodios en los que el agente interactúa con el entorno, acumula experiencia y actualiza su red neuronal. En cada episodio, se inicializa el entorno y se obtiene el estado inicial. Luego, en cada paso, el agente decide qué acción tomar usando una política *ε-greedy*.
La política *ε-greedy* se basa en un equilibrio entre exploración y explotación:

- Con probabilidad ε, se selecciona una acción aleatoria.
- Con probabilidad 1 − ε, se elige la acción con el mayor valor Q estimado por la *Q-network*.

El valor de ε disminuye progresivamente siguiendo una política de decaimiento inverso:

$$\varepsilon = \text{máx}(1/N), \tag{1}$$

donde N es el número total de episodios. Épsilon además no podrá ser menor a un mínimo establecido.
Cada vez que se almacena una experiencia en el buffer, si hay suficientes muestras, se entrena la *Q-network*. Para ello, se extrae un lote aleatorio del *replay buffer*, se calculan las predicciones actuales Q (s, a; θ) y los valores objetivo usando el target network Q (s′, a′; θ^-). La función de pérdida se basa en el error cuadrático medio entre estas dos estimaciones:

$$L(\theta) = \mathbb{E}\left[\left(r + \gamma \,\text{máx}_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right] \tag{2}$$

Siendo r la recompensa por estar en el estado y γ un factor de descuento que permite dar más importancia a las recompensas obtenidas futuras que a las inmediatas.

La *target network* se actualiza cada ciertos episodios copiando los pesos de la *Q-network*.

Parámetros a tener en cuenta durante el entrenamiento:

- ***episodes*:** número total de episodios que se entrenará el agente.
- ***batch_size***: número de muestras que se extraen del buffer para cada paso de entrenamiento.
- ***gamma***: factor de descuento para el cálculo del retorno esperado a futuro.
- ***epsilon***: valor inicial de ε, lo que implica exploración total al principio.
- ***epsilon_min***: valor mínimo que puede tomer ε.

- ***target_network_update_freq***: cada cuántos episodios se actualiza la target network.
- ***memory_size***: número máximo de experiencias que puede almacenar el replay buffer.
- ***replays_per_episode*:** número de veces que se ejecuta un solo episodio.
- ***learning_rate***: tasa de aprendizaje del optimizador utilizado (Adam).
- ***epsilon_decay*,** cuanto decrece el parámetro épsilon por cada episodio.

III. METODOLOGÍA

Para abordar este proyecto, el primer paso fue realizar una lectura detenida del documento en formato PDF que recogía los requisitos y objetivos del trabajo. A continuación, analizamos el archivo *lunar.py*, con el objetivo de comprender el funcionamiento del entorno *LunarLander*, y el archivo base *DQN.py*, que contenía la estructura principal del agente a implementar.

En este último fichero observamos que el código estaba dividido en tres clases principales: *DQN*, *ReplayBuffer* y *DQNAgent*. La clase *DQNAgent*, en particular, contenía ya definidas todas las funciones necesarias, aunque sin implementar, lo cual nos proporcionaba una guía clara sobre los componentes que debíamos construir.

En este punto, ambos integrantes del grupo decidimos investigar en profundidad el algoritmo Deep Q-Network (*DQN*), utilizando recursos como vídeos explicativos, artículos técnicos, foros especializados, y la documentación oficial de *Keras*, *Gymnasium*, así como algunos repositorios en GitHub. Todas estas fuentes han sido recopiladas y se encuentran detalladas en la bibliografía.

Durante esta fase de investigación, comprobamos que la estructura proporcionada en *DQN.py* era común a muchas implementaciones del algoritmo *DQN*, lo cual nos permitió comprender con claridad la funcionalidad de cada una de las tres clases:

a) *DQN*: define la estructura de la red neuronal que actuará como aproximador de la función Q.

b) *ReplayBuffer*: implementa el buffer de memoria donde se almacenan experiencias pasadas para su posterior entrenamiento mediante "experience replay".

c) *DQNAgent*: representa el agente que interactúa con el entorno, recoge experiencias, entrena el modelo y ajusta su política.

Una vez comprendida la funcionalidad de cada componente, comenzamos el desarrollo progresivo del proyecto.

1) Definición de la red neuronal (clase *DQN*):

Una de las primeras partes desarrolladas en el proyecto fue la clase encargada de representar la red neuronal profunda utilizada por el agente para estimar los valores Q.

Esta clase extiende la funcionalidad de la librería *keras.Model* [4] para definir un modelo personalizado. Su responsabilidad es construir una red neuronal con una arquitectura adecuada para recibir como entrada el estado del entorno, que queda representado como un vector de ocho valores, y devuelve como salida una estimación de los Q valores para cada una de las acciones posibles, en caso de *LunarLander*, cuatro acciones discretas.

Se define la inicialización de la clase con los parámetros:

- **state_size:** referencia el tamaño de los estados o de la entrada de la red, en este caso, 8.
- **action_size**: referencia el tamaño de las acciones o de la salida de la red, en este caso, 4.
- **hidden_size:** referencia el tamaño que tendrán las capas intermedias de la red. En el caso del modelo entregado, tras diversas pruebas que luego serán expuestas, se eligió un tamaño de 64.

La red está compuesta por una capa de entrada conectada a dos capas ocultas densamente conectadas con un tamaño configurable mediante *hidden_size*. Estas capas son de tipo *keras.Dense* [6].

Como se aprecia en la primera capa oculta, esta recibe un parámetro *input_shape*, que viene referenciando el tipo que recibe de entrada, que no es necesario declarar [6].

Ambas capas ocultas utilizan la función de activación *ReLu* para introducir no linealidad en el modelo. Finalmente, la salida está compuesta por una capa lineal, que genera un valor Q para cada acción posible.

La función *call*, requerida por *keras.Model* [4], establece la lógica de propagación hacia delante, recibiendo como entrada un estado del entorno y procesado a través de las capas internas para obtener la salida correspondiente.

Esta clase se utiliza dos veces en la incialización del agente *DQN* para generar tanto la *Q-Network* principal como la objetivo. Ambas comparten la misma arquitectura, sin embargo, la red objetivo se actualiza con menos frecuencia para proporcionar estabilidad durante el entrenamiento.

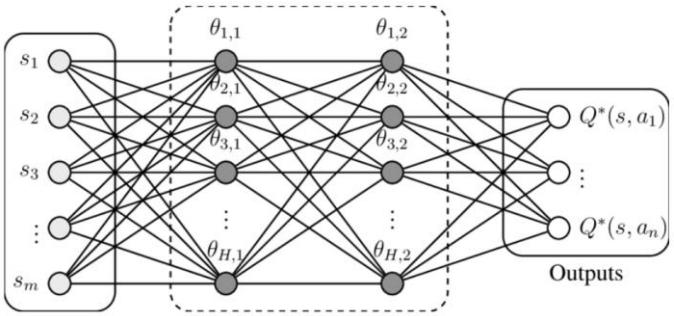

Fig 2. Red Neuronal que se construye. Imagen del PDF de requisitos.

2) Gestión de experiencias (clase *ReplayBuffer*):

El entrenamiento del agente se apoya en una técnica fundamental del algoritmo *DQN* denominada *experience replay*. La finalidad de esta es romper la correlación temporal

entre las experiencias consecutivas del entorno, y para implementar esta, se ha desarrollado la clase *ReplayBuffer*.

Esta actúa como almacén, es una cola, con un tamaño limitado, que guarda experiencias anteriores. Cada una viene representada con una tupla (*s, a, r, s', done*):

- *s*: representa el estado actual del entorno.
- *a*: representa la acción tomada en ese momento.
- *r*: representa la recompensa obtenida.
- *s'*: representa el siguiente estado.
- *done*: indica si el episodio ha terminado tras ese paso.

Este *ReplayBuffer* permite añadir nuevas experiencias. Durante el entrenamiento, de forma aleatoria se extraen pequeños conjuntos o *mini-batches* del buffer, que se utilizarán para actualizar la red neuronal. De esta manera, este pequeño muestreo aleatorio permite al agente aprender de distintas situaciones diversas y evita que se sobreajuste a secuencias concretas de estados.

La clase *ReplayBuffer* está formada por cuatro métodos (contando el de inicialización *init* y el de longitud del buffer *len*).

Cabe destacar que en el método *sample* es necesario convertir las listas de estados (*states*) y próximos estados (*next_states*), representados como vectores de dimensión (8,). Al almacenarse en el buffer quedan representados como *arrays* de 1 dimensión, sin embargo, para poder utilizarse en el entrenamiento deben mostrarse como arrays en 2 dimensiones de forma (*batch_size, dimensión_estado*), como si fuera un *batch* de vectores.

Entonces, esta función *vstack* (*Vertical Stack)* [12], permite apilar estos vectores unos sobre otros para alimentar la red neuronal y garantizar una matriz correctamente estructurada.

3) Lógica de entrenamiento e interacción con el entorno (clase *DQNAgent*):

La clase *DQNAgent* constituye el núcleo del sistema, pues es el responsable de coordinar la interacción entre el agente y el entorno, gestiona la toma de decisiones, entrena la red y controla los elementos implicados en el aprendizaje por refuerzo.

Esta clase integra los componentes comentados previamente (*DQN* y *ReplayBuffer*), y los hiperparámetros correspondientes necesarios. Se explicarán los métodos uno a uno, con una profundización suficiente para el comprendimiento de cada uno. Todos los métodos que se especifican ahora son los mismos que presentaban el esqueleto de la clase DQN:

- Método *__init__*: inicializa todos los hiperparámetros explicados previamente, junto al *ReplayBuffer*, el entorno, que es el *LunarLander*. Además, se inicializan ambas redes neuronales, la *q_network* y la *target_network*, las dos con la misma estructura para poder ir actualizando una respecto de la otra. Por último, se inicializa el optimizador, que en este caso se ha utilizado *keras.Optimizers.Adam* [10] tras haber revisado las distintas opciones que hay por parte de *keras.Optimizers* [11], y haber comprobado que *Adam* es la más utilizada en *DQN*.

- Método *act*: este método toma una acción según el estado actual del sistema. Puede ser elegida de forma aleatoria, en caso de que el número elegido aleatoriamente sea menor que la épsilon actual, o elegirse la mejor acción según los *Q-valores* almacenados, siendo la mejor la que tenga un mayor *q-value*.

  Una vez elegida la acción, se toma la acción (método *take_action* del entorno de *LunarLander*), y se almacenan en la memoria la tupla necesaria (*s, a, r, s', done)*. Finalmente, se retorna: (*s', r, d, a)* para su posterior uso en el entrenamiento.

  NOTA: importante utilizar el método *expand_dims* de *Numpy* para que la red reciba la entrada del estado en forma de *batch*, a pesar de ser una única muestra. Entonces, con este método, el estado pasará de forma (8,) a forma (1,8), que es la esperada por los modelos de *Keras*. [12]

Procedimiento *act*
Salida:
- Próximo estado *s'*
- Recompensa *r*
- Terminado *done*
- Acción *a*

Algoritmo:
1. Establecemos estado actual *s* al estado del entorno.
2. Si n_rand <= ε
   a ← acción aleatoria.
3. Sino:
   a ← argmax(q_values de la red)
4. *s', r, done* ← Método *take_action(a)*
5. Mandar a memoria la tupla (*s, a, r, s', done)*
6. Devolver *(s', r, done, a)*

Pseudocódigo 1. Método *act*.

- Método *update_model*: este método es el encargado de entrenar la red neuronal *q_network* usando un conjunto de experiencias almacenadas en el *ReplayBuffer*. Este proceso es fundamental para que el agente aprenda a aproximar de manera correcta los *Q-valores* asociados a cada acción en diversos estados.

  El funcionamiento se basa en la técnica de *experience replay*, verificando primero que el buffer tenga suficientes experiencias para formar un lote (*batch*) de entrenamiento. De ser así, extrae la ya conocida tupla de forma aleatoria (*s, a, r, s', done*).

  Una vez extraída y convertidos en tensores para poder ser correctamente tratados por la red neuronal, el objetivo es que las predicciones se acerquen al máximo a los *Q-targets*, calculados con la siguiente fórmula:

$$Q_{target} = r + \gamma \cdot máx_{(a)} Q_{target}(s',a') \cdot (1 - done) \qquad (3)$$

Siendo:

- *r:* recompensa inmediata.
- *γ:* factor de descuento.
- *s'*: siguiente estado.
- *done*: indica si el episodio terminó.

Pues, una vez calculadas las *Q-values* y *Q-targets*, se calcula la pérdida con la fórmula mostrada anteriormente, que se resume en el *MSE* entre ambas *Q*s, para posteriormente calcular el gradiente y así actualizar los pesos de la *q_network*.

Procedimiento *update_model*
Salida:
- **Pérdida entre los *Q-values* y *Q-targets***

Algoritmo:
1. Si no hay memoria suficiente:
   Devolver *None*
2. Si hay memoria:
   (*s, a, r, s', done)* ← *Memoria.sample()*
3. Transformar a tensores
4. Calcular siguientes (s*Q-values)* y el máximo
5. Aplicar $Q_{target} = r + \gamma \cdot máx_{(a)} Q_{target}(s',a') \cdot (1 - done)$
6. Calcular actuales (a*Q-values)*
7. pérdida ← MSE(s*Q-values*, a*Q-values*)
8. Actualizar pesos con gradiente.
9. Devolver pérdida.

Pseudocódigo 2. Método *act*.

- Método *update_target_network*: este método se centra en copiar los pesos de la *q_network* a la *target_network*.
- Métodos *save_model* y *load_model*: ambos reciben el parámetro de entrada *path*. El método *save* consiste en almacenar los pesos que tiene la *q_network* en el *path* que se ha indicado.

  El método *load* trabaja al revés, cargando primero los pesos en el *path* indicado, en un *dummy_input*, que viene inicializando a 0 los pesos de ambas redes para luego poder cargarle las del *path* solicitado.

  Ambos con los métodos proporcionados por *Keras* para el guardado y carga de pesos [8].
- Método *train*: este método representa el ciclo completo de entrenamiento del agente a lo largo de un número determinado de episodios. Durante este proceso, el agente interactúa con el entorno, toma decisiones, recopila experiencias y ajusta su red neuronal para mejorar su comportamiento.

  Además cuenta con una serie de variables que se omitirán en la explicación, cuyo fin son la de graficar el recorrido de las recompensas totales y del epsilon a lo largo del entrenamiento.

  Iterando por episodios, en cada uno se reinicia el entorno para obtener un estado inicial, y a partir de este, ejecutar una serie de acciones seleccionadas con la política *ε-greedy* explicada en el método *act()*. Por cada paso:

  1) Se selecciona y ejecuta una acción.
  2) Se obtiene la recompensa y el próximo estado.
  3) Se almacena la experiencia en memoria.
  4) Se entrena la red con un batch aleatorio en caso de haber memoria.

  A lo largo del entrenamiento, y con estos pasos repitiéndose en cada repetición de episodio (parámetro *replays_per_episodes* que se nos recomendó eliminar, pero ya estaba funcionando con este parámetro), se produce un decaimiento progresivo de ε, reduciendo el grado de aleatoriedad y permita al agente centrarse en explotar lo aprendido, bajando hasta llegar a su valor mínimo.

  Además, según la frecuencia de actualización de la *target_network*, (parámetro *target_updt_freq*), se sincroniza la red objetivo aportando estabilidad, evitando fluctuaciones bruscas en el aprendizaje.

  Durante el proceso de entrenamiento, se implementó un sistema de guardado periódico de los pesos del modelo. Este mecanismo tiene como objetivo preservar versiones intermedias del agente, lo cual resulta útil tanto para la recuperación ante errores como para analizar la evolución del aprendizaje en distintos puntos del entrenamiento.

  Concretamente, cada 100 episodios se guardan los pesos actuales de la red neuronal principal, generando un archivo cuyo nombre incluye el número de episodio correspondiente. Además, al finalizar todo el entrenamiento, se guarda una versión final del modelo, que representa el conocimiento adquirido tras completar todos los episodios.

  Este enfoque permite comparar modelos entrenados con diferentes cantidades de experiencia, facilita la reanudación del entrenamiento en caso de interrupciones, y proporciona la posibilidad de realizar evaluaciones específicas sobre versiones concretas del agente, que quizás puedan ser mejores que la versión final.

  Pseudocódigo 3. Método *train*.

Procedimiento *train*
Salida:
- **Modelo entrenado**
- **Evolución de recompensas**
- **Evolución de ε**
- **Modelos periódicos**

Algoritmo:
1. Inicializar arrays de recompensas y de ε.
2. Para cada episodio, desde 1 hasta *episodios*:
   Reiniciar entorno, obtener estado *s*.
   Inicializar recompensa *R* ← 0.
3. Para cada *replays_per_episode:*
   (s', r, done, a) ← *act()*
   (s', r, done) ← *take_action(a)*
   Guardar (*s, a, r, s', done*) en memoria
4. Si hay suficientes experiencias:
   update_model()
   s ← *s'; R* ← *R + r*
   Si episodio % target_updt_freq == 0:
   update_target_network()
   Si *done*, salir del bucle.
   Decaer ε y guardar su evolución.
5. Si *episodio* % 100 == 0:
   Save_model()
6. Finalizar entrenamiento y guardar modelo final.

IV. RESULTADOS

A lo largo de este proyecto se han probado distintas configuraciones con el objetivo de encontrar la combinación de parámetros que proporcione los mejores resultados, permitiendo que el modelo sea el óptimo. Uno de los aspectos clave en este proceso ha sido el ajuste de la estrategia de exploración, concretamente del parámetro ε (épsilon).

Inicialmente, se empleó un factor de reducción constante, concretamente $\varepsilon = \varepsilon \times 0.995$ tras cada episodio. Esta técnica permite que, en las primeras etapas del entrenamiento, el agente explore el entorno de forma intensiva (con valores altos de ε), para luego ir reduciendo gradualmente la exploración en favor de una mayor explotación del conocimiento adquirido.

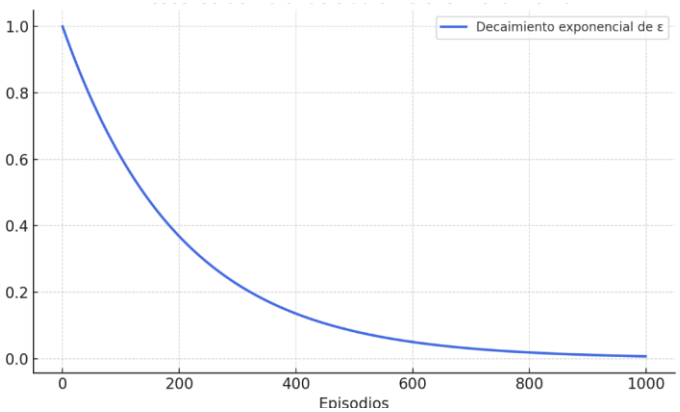

Fig 3 : Gráfica de decaimineto de epsilon con valor de decaimiento de 0.995

Tras entrenar al agente utilizando esta política de exploración, con un factor de decaimiento exponencial y estableciendo un valor mínimo de ε de 0.01, se obtuvo un modelo cuyo rendimiento no alcanzó niveles óptimos. En la siguiente gráfica se muestra la evolución de la recompensa acumulada, calculada de los últimos 100 episodios a lo largo de 3000 episodios de entrenamiento.

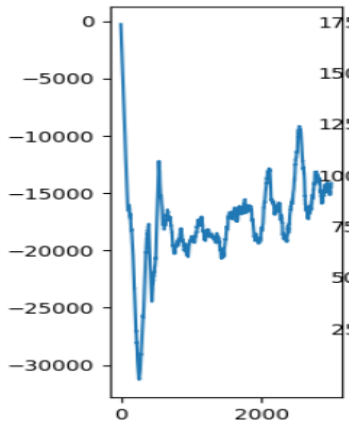

Fig 4: Recompensa acumulada de los 100 ultimos episodios

Tal como se aprecia, la tendencia general es creciente. Sin embargo, a pesar de esta evolución positiva, el modelo final alcanzó una recompensa acumulada en los últimos 100 episodios cercana a los -15000, lo cual refleja que aún no logra un desempeño adecuado.

Las conclusiones que pudimos sacar de esto es que el agente no ha podido explorar lo suficiente y comienza a explotar la política en etapas demasiado tempranas.

Por esta razón, posteriormente, se probó una segunda estrategia alternativa basada en el decaimiento inversamente proporcional al número total de episodios, es decir, utilizando la expresión $\varepsilon = 1/N$, donde N representa el número de episodios de entrenamiento. Este método garantiza una mayor exploración al principio y una transición más suave hacia políticas explotativas. Este tipo de decaimiento se puede ver en acción en la lista de videos [14].

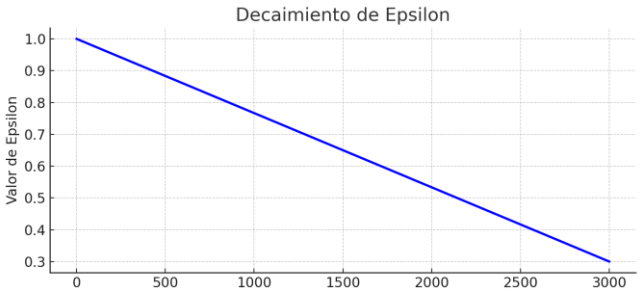

Fig 5 : Gráfica de decaimineto de epsilon dependiente del número de episodios, durante 3000 episodios

El entrenamiento del agente con este decamiento de epsilon, en este caso de $\varepsilon = \varepsilon \times 1/3000$, y siendo ε mínimo igual a 0.01, dio como resultados la siguiente gráfica de la recompensa acumulada:

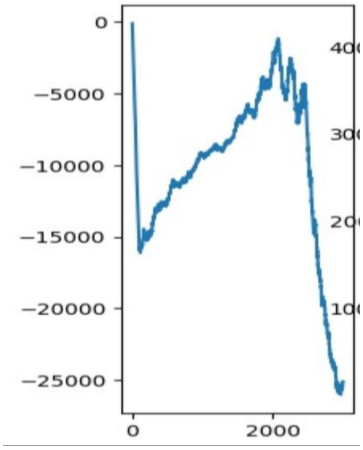

Fig 6: Recompensa acumulada de los 100 ultimos episodios con epsilon dependiente de los episodios y epsilon_min = 0.01

Como podemos ver, la recompensa acumulada ha conseguido tener una tendencia creciente con una pendiente mucho mayor que con la anterior política de decaimiento de

epsilon, pero cuando este esta cerca de su valor mínimo, la política del agente comienza a dar resultados cada vez más negativos hasta llegar a una recompensa de casi -25000 en los ultimos 100 episodios.

Esto puede haberse originado por el valor mínimo que puede tomar epsilon, al ser un valor tan cercano a 0, las probabilidades de explorar en las etapas más avanzas del aprendizaje son casi nulas, provocando que se explote una política que aún no ha llegado a ser la más óptima posible.

La prueba de esto queda expuesta en el siguiente gráfico, el cuál representa solo la recompensa acumulada positiva de los ultimos 100 episodios, donde se ve claramente como la cantidad de recompensas positivas incrementan en la etapa intermedia pero dejan de darse al alcanzar el final del entrenamiento:

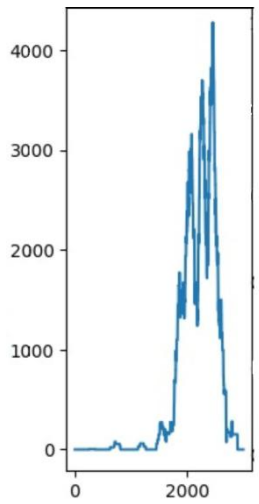

Fig 7: Recompensa positiva acumulada en base a los 100 últimos episodios

A continuación, para solventar este problema, se modificó el valor de epsilon_min, con el fin de que epsilon nunca pueda llegar a tener un valor tan cercano a cero, en el siguiente entrenamiento este parámetro será de 0.3. La recompensa acumulada se muestra en este gráfico:

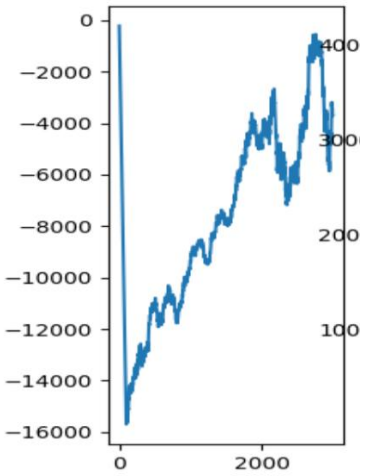

Fig 8: Recompensa acumulada de los 100 ultimos episodios con epsilon dependiente de los episodios y epsilon_min = 0.3

La curva ahora dejó de tener una bajada al final del entrenamiento y la tendencia es completamente creciente a lo largo de todas las fases del exceptuando la inicial.

Previo a estos entrenamientos, se modificó el código del con el fin de guardar un modelo cada 100 episodios, para así poder hacer un estudio más preciso de como el agente ha ido mejorando a lo largo de las iteraciones. Por ello, se probaron no solo el modelo final, sino los modelos cercanos al pico máximo de la grafica, con el fin de obtener el más óptimo posible. A pesar de esto, el modelo con mejores resultados fue el almacenado en el episodio número 3000 tras finalizar todo el entrenamiento, el cuál es capaz de aterrizar el módulo de forma efectiva hasta en un 60-70% por ciento de las veces aproximadamente.

```
Episode finished, score: 111.53916783263942
Episode finished, score: 271.4312850269466
Episode finished, score: -1.908999097667163
Episode finished, score: 275.5017858635982
Episode finished, score: 251.79295016142086
Episode finished, score: 135.04368379403823
Episode finished, score: 48.02943979069188
Episode finished, score: 62.10952574401192
Episode finished, score: 251.9991569942944
Episode finished, score: 149.96349267699946
```

Fig 9: Recompensa obtenida en 10 aterrizajes

V. Conclusiones

El desarrollo de este proyecto nos ha permitido comprender en profundidad la utilidad del algoritmo *Deep Q-Network* (*DQN*) como herramienta eficaz para entrenar agentes inteligentes en entornos complejos.

Durante la implementación, hemos interiorizado tanto los conceptos teóricos del aprendizaje por refuerzo como su aplicación práctica en código. En particular, hemos comprendido la importancia de técnicas como el experience replay, el uso de una red objetivo para estabilizar el aprendizaje y la política de exploración *ε-greedy*, fundamentales para lograr un entrenamiento eficaz.

Una mejora clave introducida fue la optimización del proceso de entrenamiento mediante guardados periódicos del modelo, lo cual no solo nos permitió preservar avances ante posibles fallos, sino también analizar el rendimiento del agente en diferentes etapas del aprendizaje.

Asimismo, la generación de gráficas durante el entrenamiento nos proporcionó una herramienta valiosa para ajustar y analizar el comportamiento del agente. Gracias a ellas, pudimos identificar tendencias, estabilizar parámetros como epsilon, y detectar cuándo el aprendizaje se volvía inestable o dejaba de mejorar. Esta retroalimentación visual ha sido esencial para afinar el modelo y tomar decisiones más informadas durante el desarrollo.

Referencias

[1] Página Web con un ejemplo de programación en Python. https://pythonprogramming.net/deep-q-learning-dqn-reinforcement-learning-python-tutorial/.

[2] Clase Env de Gymnasium https://gymnasium.farama.org/api/env/.

[3] Lectura recomendada 1 https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf.

[4] Clase Model de Keras https://keras.io/api/models/model/.

[5] Clase Layer de Keras https://keras.io/api/layers/base_layer/

[6] Clase Dense de Keras https://keras.io/2/api/layers/core_layers/dense/.

[7] Activaciones de Keras https://keras.io/api/layers/activations/.

[8] Carga y guardado de pesos https://keras.io/api/models/model_saving_apis/weights_saving_and_loading/.

[9] Vídeo DQN de CodeEmporium en Youtube. https://www.youtube.com/watch?v=x83WmvbRa2I.

[10] Optimizador Adam de Keras https://keras.io/api/optimizers/adam/.

[11] Optimizadores de Keras https://keras.io/api/optimizers/.

[12] Operaciones de Numpy en Keras https://keras.io/api/ops/numpy/

[13] Foro StackExchange https://ai.stackexchange.com/questions/20384/what-is-the-target-q-value-in-dqns.

[14] Lista de vídeos de DQN ejemplo Flappy Bird, canal de YouTube de Johnny Code https://www.youtube.com/playlist?list=PL58zEckBH8fCMIVzQCRSZVPUp3ZAVagWi.

[15] GitHub de Johnny Code, con ejemplo del Lago Congelado en DQN. https://github.com/johnnycode8/gym_solutions/blob/main/frozen_lake_dql.py.