

UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA INFORMÁTICA EN
TECNOLOGÍAS DE LA INFORMACIÓN



Entrenamiento de agentes inteligentes.
Aplicación a videojuegos.

TRABAJO FIN DE GRADO

Junio - 2025

AUTOR: Héctor Sanz González
DIRECTOR: Jesús Javier Rodríguez Sala

RESUMEN

Este Trabajo de Fin de Grado se centra en la aplicación de la Inteligencia Artificial para el entrenamiento de agentes autónomos, abordando una comparación entre dos paradigmas clave: el aprendizaje por refuerzo, a través del algoritmo Deep Q-Learning (DQN), y los algoritmos genéticos. El objetivo principal del proyecto fue la creación de un entorno simulado, específicamente el videojuego Lunar Lander, para el entrenamiento de un agente, analizando sus acciones y recompensas para optimizar su desempeño.

El desarrollo de este trabajo se fundamenta en el uso de herramientas y bibliotecas como Python, Gymnasium para la simulación del entorno, y Keras para la implementación de redes neuronales. Los modelos desarrollados incluyen tanto la implementación de algoritmos genéticos con diversas configuraciones de operadores (selección, cruce y mutación) como la configuración y ajuste de arquitecturas de redes neuronales para DQN.

Los resultados obtenidos revelan que, si bien los algoritmos genéticos pueden generar políticas estables bajo configuraciones específicas, su rendimiento es altamente dependiente de una correcta selección de parámetros y operadores. En contraste, DQN demostró una mayor robustez y capacidad de generalización en el aprendizaje de políticas óptimas, ofreciendo un rendimiento superior y más consistente, aunque a costa de una mayor complejidad en su implementación y ajuste de hiperparámetros.

AGRADECIMIENTOS

Este Trabajo de Fin de Grado no solo representa mi esfuerzo, sino también el apoyo y el cariño de muchas personas que me han acompañado en este camino. En primer lugar, quiero agradecer a mi tutor, Jesuja, por su ayuda y por guiarme en el desarrollo de este proyecto desde el principio. A mi familia, gracias por estar siempre ahí, y en especial a mis padres, por creer en mí desde el primer momento, por su apoyo incondicional y por darme el ánimo que muchas veces necesitaba para no rendirme en el proceso. A mis amigos y compañeros, gracias por los buenos momentos, por las risas y por hacer que todo este proceso fuera mucho más fácil. Por último, agradezco a Dios por haberme acompañado y guiado en este proceso, por ser mi fortaleza en los momentos de debilidad y por brindarme una vida llena de aprendizajes, experiencias y, sobre todo, felicidad.



ÍNDICE GENERAL

1. Introducción	9
1.1. Inteligencia Artificial en la sociedad	9
1.2 Justificación del proyecto	10
1.3 Objetivos	12
2. Antecedentes y estado de la cuestión	13
2.1 Tipos de Aprendizaje	14
2.2 Aprendizaje por Refuerzo	15
2.2.1 Representación Formal	15
2.2.2 Funciones de valor	18
2.2.3 Q-learning	19
2.2.4 Deep Q-learning	21
2.3 Computación Evolutiva	22
2.3.1 Selección	24
2.3.2 Cruce	25
2.3.2.1 Operadores de cruce codificación discreta	25
2.3.2.2 Operadores de cruce codificación real	27
2.3.3 Mutación	28
2.3.4 Reemplazo de individuos	29
2.4 Resumen	29
3 Hipótesis de trabajo	31
3.1 Miniforge	32
3.2 Visual Studio Code	33
3.3 Lunar Lander	34
3.4 Google Colab	35
3.5 Equipo de Desarrollo	35
4. Metodología y resultados	37
4.1 Planificación del proyecto	37
4.1.1 Ciclo de vida del proyecto	37
4.1.2 Planificación temporal del proyecto	38
4.2 Conceptos previos Lunar Lander	39
4.3 Algoritmo Genético	40
4.3.1 Codificación	40
4.3.2 Flujo del algoritmo	41
4.3.3 Implementación	42
4.3.3.1 Operadores de selección	42

4.3.3.2 Operadores de cruce	44
4.3.3.3 Mutación	46
4.3.3.4 Función de fitness	46
4.3.4 Resultados Algoritmos Genéticos	47
4.4 Aprendizaje por refuerzo	62
4.4.1 Codificación	62
4.4.2 Flujo del algoritmo	62
4.4.3 Implementación	63
4.4.4 Experimentación Aprendizaje por Refuerzo	68
4.5 Comparación de enfoques	76
5 Conclusiones y trabajo futuro	78
5.1 Conclusiones	78
5.2 Posibles desarrollos futuros	79
6 Bibliografía	81



ÍNDICE DE TABLAS

Tabla 2.1 Tabla Q (Q-Table)	20
Tabla 4.1 Hiperparametros para Entrenamiento de Agentes de Aprendizaje por Refuerzo	63
Tabla 4.2 Valores para experimentación DQN.	69
Tabla 4.3 Resumen resultados obtenidos DQN	75



ÍNDICE DE FIGURAS

Figura 1.1 Inversión en inteligencia artificial	10
Figura 2.1 Aprendizaje Supervisado	14
Figura 2.2 Aprendizaje no supervisado	15
Figura 2.3 Interacción del agente con el entorno	16
Figura 2.4 Ejemplo Markov Decision Process	17
Figura 2.5 Algoritmo Q-Learning	20
Figura 2.6 Algoritmo Deep Q-Learning	21
Figura 2.7 Proceso de algoritmo evolutivo	23
Figura 2.8 Representación algoritmo genético	23
Figura 2.9 Exploración vs Explotación	24
Figura 2.10 Operador de cruce en un punto	26
Figura 2.11 Operador de cruce en dos puntos	26
Figura 2.12 Operador de cruce uniforme	27
Figura 2.13 Operador de cruce plano	28
Figura 2.14 Mutación de un gen codificación real	29
Figura 3.1 Entorno miniforge	32
Figura 3.2 Plugin de Python en VS code	33
Figura 3.3 Plugin de Jupyter en VS code	34
Figura 3.4 Plugin de GitHub Copilot VS code	34
Figura 3.5 Juego Lunar Lander Atari	34
Figura 3.6 Entorno Colab	35
Figura 4.1 Diagrama Gantt	38
Figura 4.2 Lunar Lander en el entorno de simulación OpenAI Gym	39
Figura 4.3 Red de neuronas codificadas por un individuo y vector real	41
Figura 4.4 Gráfica entrenamiento cruce plano sin mutación	49
Figura 4.5 Resultado de simulaciones cruce plano sin mutación.	49
Figura 4.6 Gráfica de entrenamiento cruce plano con mutación 0.1.	49
Figura 4.7 Resultado de simulaciones cruce plano con mutación 0.1.	50
Figura 4.8 Gráfica de entrenamiento cruce plano con mutación 0.3	50
Figura 4.9 Resultado de simulaciones cruce plano con mutación 0.3.	50
Figura 4.10 Gráfica de entrenamiento cruce morfológico sin mutación.	51
Figura 4.11 Resultado de simulaciones cruce morfológico sin mutación.	52
Figura 4.12 Gráfica de entrenamiento cruce morfológico con mutación 0.1.	52
Figura 4.13 Resultado de simulaciones cruce morfológico con mutación 0.1.	52
Figura 4.14 Gráfica de entrenamiento cruce morfológico con mutación 0.3.	53
Figura 4.15 Resultado de simulación cruce morfológico con mutación 0.3.	53
Figura 4.16 Gráfica de entrenamiento cruce BLX con $\alpha = 0.3$.	54
Figura 4.17 Resultado de simulación cruce BLX con $\alpha = 0.3$.	54

Figura 4.18 Gráfica de entrenamiento cruce BLX con $\alpha = 0.5$.	55
Figura 4.19 Resultado de simulación cruce BLX con $\alpha = 0.5$	55
Figura 4.20 Gráfica de entrenamiento cruce BLX con $\alpha = 0.7$.	55
Figura 4.21 Resultado de simulación cruce BLX con $\alpha = 0.7$.	56
Figura 4.22 Gráfica de entrenamiento cruce BLX con $\alpha = 0.5$ y mutación 0.1.	57
Figura 4.23 Resultado de simulación cruce BLX con $\alpha = 0.5$ y mutación 0.1.	57
Figura 4.24 Gráfica de entrenamiento selección torneo con $k=3$.	58
Figura 4.25 Resultado de simulación selección torneo con $k=3$.	58
Figura 4.26 Gráfica de entrenamiento selección torneo con $k=7$.	59
Figura 4.27 Resultado de simulación selección torneo con $k=7$.	59
Figura 4.28 Gráfica de entrenamiento selección ruleta con $e=15$.	60
Figura 4.29 Resultado de simulación selección ruleta con $e=15$.	60
Figura 4.30 Gráfica de entrenamiento selección ruleta con $e=30$.	60
Figura 4.31 Resultado de simulación selección ruleta con $e=30$.	61
Figura 4.32 Entrenamiento red 128 sin Target Network.	70
Figura 4.33 Entrenamiento red 64 sin Target Network.	70
Figura 4.34 Entrenamiento red 32 sin Target Network.	70
Figura 4.35 Entrenamiento red 128 con Target Network actualización cada episodio.	71
Figura 4.36 Entrenamiento red 64 con Target Network actualización cada episodio.	71
Figura 4.37 Entrenamiento red 32 con Target Network actualización cada episodio.	72
Figura 4.38 Entrenamiento red 128 con Target Network actualización cada 1000 pasos.	72
Figura 4.39 Entrenamiento red 64 con Target Network actualización cada 1000 pasos.	73
Figura 4.40 Entrenamiento red 32 con Target Network actualización cada 1000 pasos.	73
Figura 4.41 Entrenamiento red 128 con Target Network actualización con soft update.	74
Figura 4.42 Entrenamiento red 64 con Target Network actualización con soft update.	74
Figura 4.43 Entrenamiento red 32 con Target Network actualización con soft update.	74

ÍNDICE DE ALGORITMOS

Algoritmo 2.1: Q-Learning	21
Algoritmo 4.1: Selección por torneo	42
Algoritmo 4.2: Selección por ruleta con elitismo	43
Algoritmo 4.3: Operador de cruce plano	44
Algoritmo 4.4: Operador de cruce BLX- α	44
Algoritmo 4.5: Operador de cruce morfológico	45
Algoritmo 4.6: Mutación	46
Algoritmo 4.7: Fitness	47
Algoritmo 4.8: DQN (Deep Q-Network)	63
Algoritmo 4.9: ReplayMemory	66
Algoritmo 4.10: Entrenamiento del modelo	66



Capítulo 1

Introducción

1.1.- INTELIGENCIA ARTIFICIAL EN LA SOCIEDAD

Vivimos en una era donde el uso de la tecnología se ha convertido en un pilar fundamental de nuestra sociedad. Esto se debe a la gran evolución que ha experimentado el mundo de la informática desde sus inicios a mitad del siglo pasado hasta la actualidad. Lo observamos en el uso cotidiano de computadoras, dispositivos móviles, y todo tipo de servicios en la nube, ofreciendo una conectividad global [1].

Dentro del marco de esta evolución que ha transformado todos los sectores y forma de entender nuestra sociedad la inteligencia artificial (IA) ha emergido como una de las innovaciones más significativas y transformadoras. La IA consiste en la capacidad de los sistemas para realizar tareas que suelen requerir inteligencia humana, como aprender de la experiencia, tomar decisiones o la resolución de problemas, todas estas habilidades están proporcionando resultados sorprendentes en un amplio número de campos [2].

La inteligencia artificial está muy presente en nuestra vida diaria, por ejemplo, en los asistentes virtuales como Alexa o Siri, los algoritmos de búsqueda de Google, en los navegadores para calcular rutas más eficientes o en los sistemas de recomendación de plataformas como Netflix o Amazon. Estos ejemplos muestran cómo la IA no solo está cambiando la tecnología, sino también la forma en que interactuamos con el mundo.

1.2.- JUSTIFICACIÓN DEL PROYECTO

El campo de la inteligencia artificial es una de las áreas más influyentes y revolucionarias de la tecnología actual, con el desarrollo de tecnologías como el procesamiento del lenguaje natural (Chat GPT), generación de imágenes (Dal-E) o transcripción de audio (Siri) entre otras [3]. Esta tecnología genera un incremento en la eficiencia y la productividad automatizando tareas cotidianas y repetitivas. En campos como el de la salud, permite analizar datos médicos, ayudando en el diagnóstico e identificando patrones en el historial de los pacientes; en educación permite crear experiencias de aprendizaje adaptadas al usuario; también puede ayudar a personas con discapacidades mejorando sus vidas, comunicación e independencia [4]. La importancia de esta tecnología la vemos en la evolución de la inversión en IA en la última década:

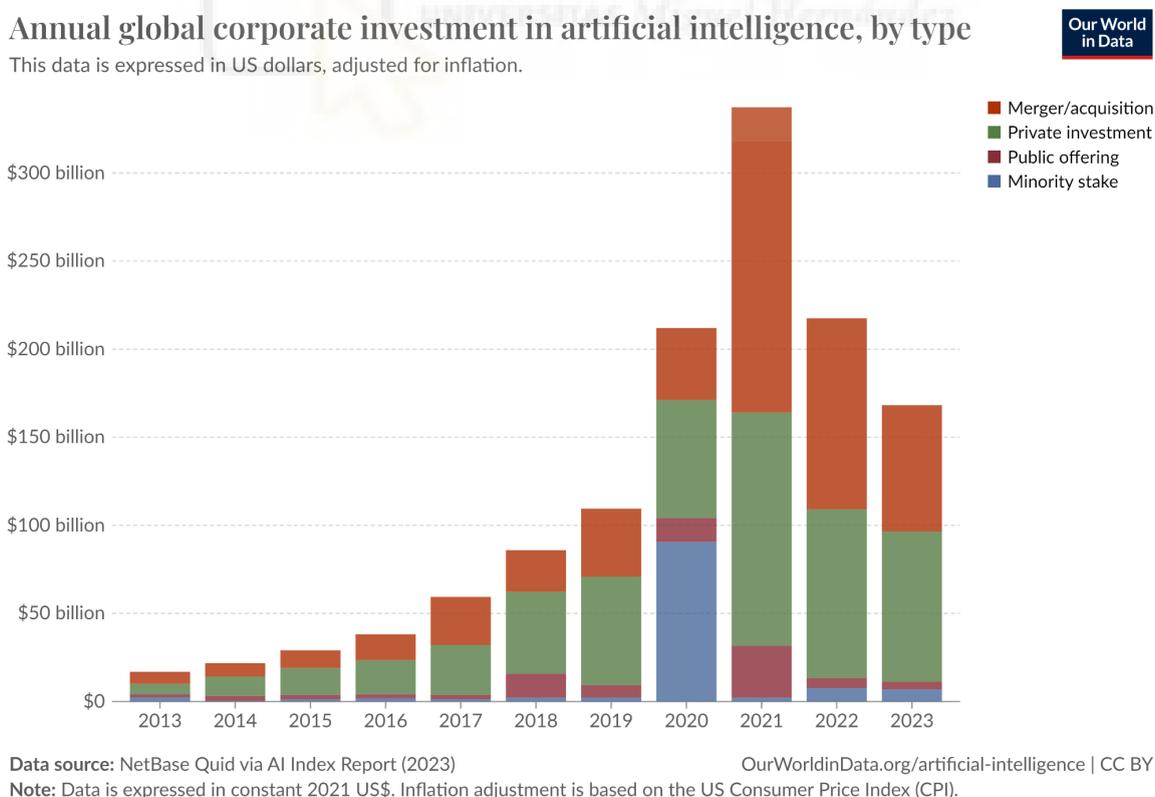


Figura 1.1: Inversión en inteligencia artificial [5]

1.2.1.- Innovación y Futuro

La inteligencia artificial está reinventando la forma de trabajar en prácticamente todos los sectores, mejorando la automatización de tareas complejas, optimizando procesos y permitiendo el desarrollo de soluciones más personalizadas. Tiene la capacidad de generar un impacto significativo en el futuro cercano, superando a las tecnologías actuales [6].

La IA está redefiniendo constantemente los límites de los conocimientos científicos y generando nuevas tecnologías que producen avance en multitud de áreas. El futuro de esta área es más que relevante debido al acceso a cantidades de datos cada vez más grandes y un poder de cómputo cada vez mayor, el ejemplo más evidente de todo esto son los sistemas de aprendizaje profundo o deep learning [7].

1.2.2.- Relevancia y Actualidad

La IA y los videojuegos son sectores en constante evolución, permiten el desarrollo de diferentes algoritmos y funcionalidades, cuyos usos se pueden extrapolar a otros ámbitos, como en el entorno Unity, que posibilita el desarrollo de entornos 2D y 3D para la utilización de agentes autónomos que permite tanto avanzar en el campo de la investigación, como para el desarrollo comercial, y dando la posibilidad de extrapolar estos desarrollos a otros campos como el de la robótica [8].

1.2.3.- Oportunidad de Aprender y Aplicar Tecnologías de Interés

Mi interés en desarrollar un proyecto utilizando Unity ML-Agents surgió tras ver un video del canal de Youtube DotCSV (administrado por el divulgador científico Carlos Santana) sobre la creación de agentes inteligentes en entornos simulados. En este video, se expone de manera clara cómo el aprendizaje por refuerzo, junto con la simulación en 3D, permite entrenar agentes autónomos en tareas complejas, como la toma de decisiones y el aprendizaje por ensayo y error [9]. Este enfoque me motivó a explorar las posibilidades de aplicar estos conceptos con Gymnasium, una plataforma que facilita la experimentación con IA en contextos simulados desarrollada por OpenAI. Y respecto al uso de la computación evolutiva para resolver también problemas en entornos de simulación viene dado por el interés en aprender el funcionamiento de esta tecnología basada en la evolución biológica. Este proyecto representa una oportunidad única para profundizar en tecnologías avanzadas y su potencial en áreas como la robótica, la automatización y los videojuegos, campos en los que la IA está teniendo un impacto transformador.

1.3.- OBJETIVOS

El objetivo principal de este trabajo es crear un entorno donde entrenar a un agente con diferentes algoritmos tanto de aprendizaje por refuerzo como algoritmos genéticos, realizando la configuración de estos para que aprendan a jugar a un videojuego. Para alcanzar esta meta, este objetivo principal se concreta en objetivos más específicos, agrupados en dos categorías, en primer lugar, en lo referente al entender el entorno de simulación:

- Entender y probar el entorno de Lunar Lander.
- Analizar las diferentes acciones que puede realizar el agente y las recompensas que va a recibir.

Por otro lado, los objetivos referentes a la creación del agente y la implementación de la metodología de Machine Learning:

- Estudiar la metodología a seguir para entrenar a los agentes inteligentes.
- Determinar la información necesaria del entorno para el correcto aprendizaje del agente.
- Configurar el entrenamiento de los agentes.
- Entrenar a los agentes y estimar el tiempo de entrenamiento necesario para obtener un resultado exitoso.

Para la realización del algoritmo genético:

- Diseñar un esquema de codificación de soluciones para representar individuos en la población como redes neuronales densas.
- Implementar un algoritmo genético para el agente de Lunar Lander.
- Evaluar la eficiencia del enfoque evolutivo, e identificar la configuración óptima de operadores que maximiza el desempeño.

Por último, realizar una comparación entre los diferentes algoritmos de entrenamiento usados.

Capítulo 2

Antecedentes y estado de la cuestión



El aprendizaje automático (o machine learning, ML) es un campo de estudio dentro de la inteligencia artificial que dota a los ordenadores de la capacidad de aprender sobre algún tema sin estar explícitamente programados para ello [10]. Los sistemas de aprendizaje automático aprenden a partir de unos datos para llevar a cabo una tarea específica, no sólo ajustando un modelo a los datos disponibles sino aplicando también lo aprendido para obtener buenos resultados con nuevos datos no conocidos a este término, a esto se le llama generalización del modelo.

Por otra parte, la computación evolutiva, enfocada en los algoritmos genéticos, permite resolver problemas de optimización y búsqueda inspirándose en los principios de la evolución biológica [18]. Este enfoque se analizará en paralelo con el aprendizaje por refuerzo, comparando sus características, ventajas y aplicaciones.

En este capítulo, mostraremos en primer lugar los diferentes tipos de aprendizaje y a continuación se profundizará especialmente en el aprendizaje por refuerzo. También se

realizará un estudio sobre la computación evolutiva y los algoritmos genéticos. Por tanto, en este proyecto se busca combinar el aprendizaje por refuerzo y los algoritmos genéticos.

2.1.- TIPOS DE APRENDIZAJE

Los algoritmos de aprendizaje pueden ser organizados y clasificados en cuatro categorías principales, las cuales se determinan según la atención que se presta y el tipo de supervisión que se lleva a cabo durante la fase de entrenamiento. Estas categorías son: aprendizaje supervisado, aprendizaje no supervisado, aprendizaje semi-supervisado y aprendizaje por refuerzo. Cada una de estas modalidades de aprendizaje tiene sus propias características específicas que las diferencian entre sí, y que son fundamentales para entender cómo se desarrollan y aplican en diferentes contextos. A continuación, se procede a describir con más detalle cada uno de estos tipos de aprendizaje, resaltando sus particularidades:

- Aprendizaje supervisado: utiliza un set de datos que alimenta el algoritmo incluyendo el valor de la variable que se intenta predecir. De esta forma el algoritmo puede comparar la salida que ha obtenido con la proporcionada por el set de datos, calcular una medida de error y ajustar los componentes del modelo para conseguir mejores resultados. El algoritmo de aprendizaje, con un entrenamiento suficiente, produce un modelo para realizar predicciones de las variables objetivos, para cualquier entrada de datos nuevos. Los principales ejemplos de uso del aprendizaje supervisado son tareas de clasificación y regresión.

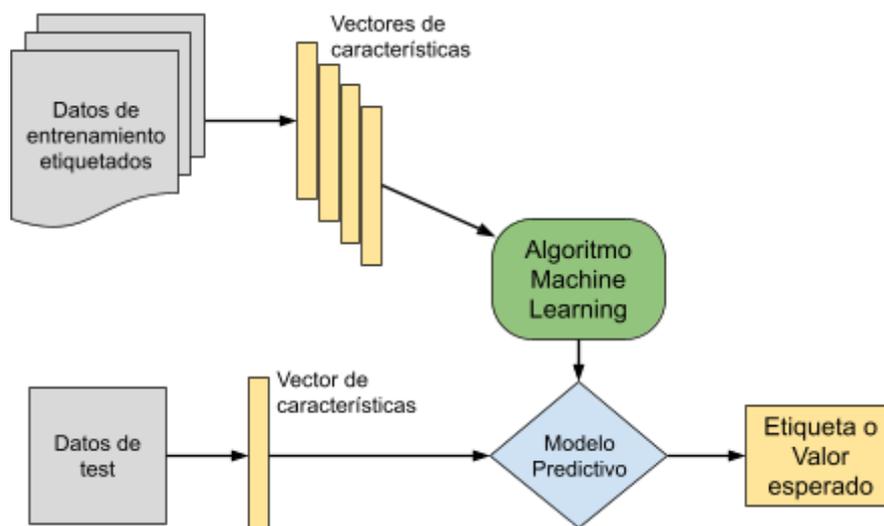


Figura 2.1: Aprendizaje Supervisado [12]

- Aprendizaje no supervisado: utiliza un set de datos el cual no incluye una variable objetivo para predecir. El algoritmo encuentra patrones o relaciones entre los datos

de entrenamiento. Algunos ejemplos de uso serían el agrupamiento o clustering y la asociación de características.

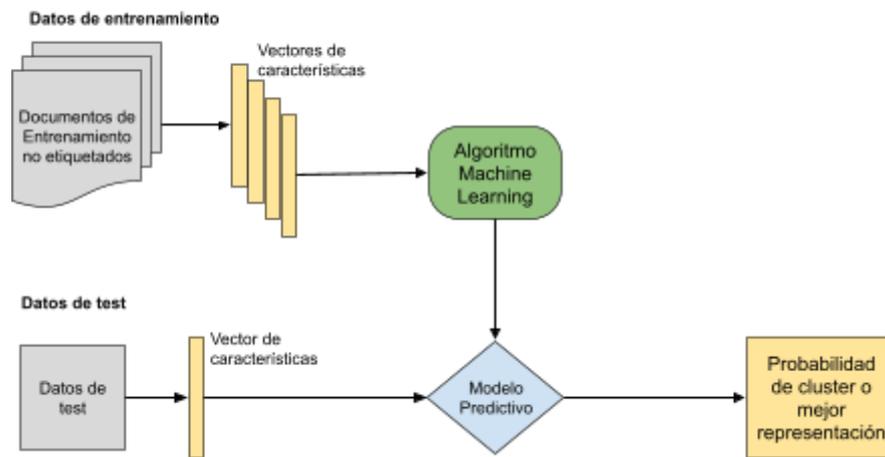


Figura 2.2: Aprendizaje no supervisado

- Aprendizaje semi-supervisado: es un enfoque que combina elementos del aprendizaje supervisado y no supervisado. Utiliza un conjunto de datos que contiene una pequeña cantidad de ejemplos etiquetados y una gran cantidad de ejemplos no etiquetados. Este método permite a los modelos aprender patrones y estructuras en los datos no etiquetados, mejorando su rendimiento al aprovechar la información de los datos etiquetados.
- Aprendizaje por refuerzo: método de aprendizaje a través de prueba y error, otorga recompensas positivas por acciones que coinciden o se acercan a la solución esperada, y recompensas negativas en caso contrario [11] (ver a continuación).

2.2.- APRENDIZAJE POR REFUERZO

2.2.1.- Representación Formal

El aprendizaje por refuerzo (o Reinforcement Learning RL) [15] se centra en el entrenamiento interactuando con el entorno, con el principal objetivo de maximizar una recompensa a lo largo del tiempo. Utiliza un agente autónomo que toma diferentes decisiones y aprende de las acciones realizadas, a veces sin un conocimiento previo. El RL se basa en varios conceptos claves: Estado, Acciones, Política, Recompensas y el modelo del entorno. Cada uno de estos componentes juega un papel importante que define las interacciones del agente con el entorno y en el proceso general del aprendizaje.

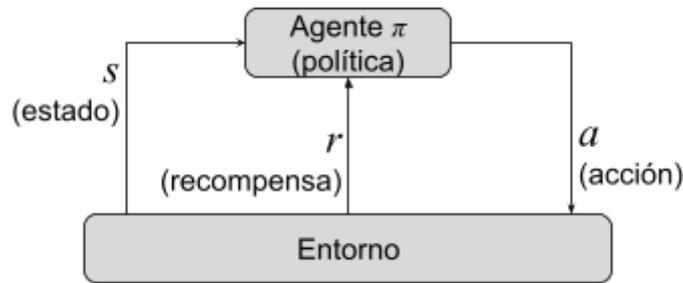


Figura 2.3: Interacción del agente con el entorno

- Estado ($s \in S$): Representa una condición o configuración específica del entorno en un momento dado como lo percibe el agente. Un estado prepara el escenario para que el agente tome decisiones y seleccione determinadas acciones, lo que describe todos los estados entre los cuales un agente puede elegir para cada acción que realice. En resumen, los estados son las situaciones en las que un agente se puede encontrar y observar para tomar decisiones. El estado puede tomar valores tanto discretos como continuos.
- Acciones ($a \in A$): Son los diferentes movimientos o decisiones que puede realizar el agente en su interacción con el entorno. La realización de una acción concreta es parte de la estrategia seguida por el agente para conseguir una mayor recompensa de acuerdo con su estado y política actual.
- Política (π): Determina la próxima acción que va a realizar el agente al transformar los estados percibidos del entorno en acciones. Una política pueden ser:
 - Determinista: Asigna directamente un estado a una acción específica

$$a = \pi(s)$$

- Estocástico: Define una distribución de probabilidad sobre posibles acciones para un estado dado

$$\pi(a|s) = P(a|s)$$

En el enfoque estocástico el agente elige el movimiento mediante un muestreo de la distribución de probabilidad, lo que introduce una diversidad en la toma de decisiones, esto resulta muy útil para explorar nuevos movimientos en entornos inciertos. Por otro lado, el enfoque determinista es eficiente en entornos bien definidos y predecibles.

- Recompensas ($r \in R$): Las recompensas son medidas cuantitativas que se usan para tomar decisiones, las cuales el agente busca en cada paso definiendo metas

tanto a nivel local como global. Las recompensas diferencian eventos positivos y negativos, lo que ayuda a actualizar la política según los resultados de las acciones. Las recompensas dependen del estado actual y de la acción tomada.

- Determinista: $r = R(s, a)$ asigna un único valor fijo a cada par acción/estado.
 - Estocástica: $r \sim P(r|s, a)$ la cual nos indica que el agente cada vez que realiza la misma acción en el mismo estado puede recibir una recompensa diferente debido a la distribución de probabilidad.
- **Modelo del entorno:** es una predicción que determina el estado y las recompensas futuras en función del estado y la acción actuales. Estos modelos son útiles para planificar acciones según eventos futuros. Los métodos de aprendizaje por refuerzo (RL) que utilizan modelos, se denominan model-based, mientras que aquellos que aprenden solo por prueba y error, son model-free. Esta distinción es crucial para seleccionar el algoritmo adecuado para un problema determinado.

Un proceso de Decisión de Markov (MDP) es una representación matemática para la toma de decisiones secuenciales en el que las acciones del agente afectan a las recompensas inmediatas y a los estados futuros del sistema. Se utilizan en el marco del RL para formalizar la interacción entre un agente y su entorno. Los MDPs cumplen con la propiedad de Markov, lo que significa que la probabilidad de transición a un estado futuro s' depende únicamente del estado actual s y la acción tomada a , sin considerar otros estados anteriores, simplificando la resolución del problema.

$$P(s_{t+1}|s_t) = P(s_{t+1}|s_t, s_{t-1}, \dots, s_1) \quad (\text{propiedad de Markov})$$

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_1) \quad (\text{MDP})$$

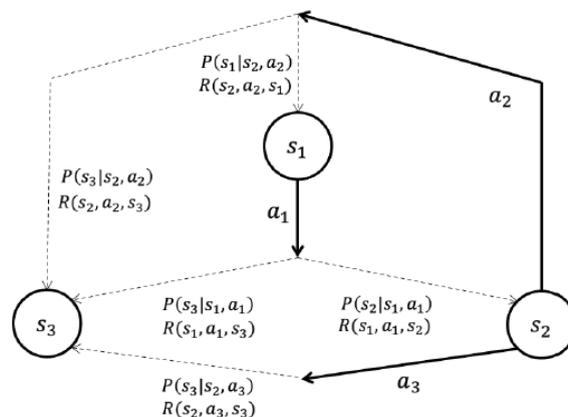


Figura 2.4: Ejemplo Markov Decision Process (MDP)

2.2.2.- Funciones de valor

Una función de valor estima el rendimiento esperado dado que el agente se encuentra en un determinado estado o realice una acción en un determinado estado. Dependiendo de las acciones seleccionadas, los factores pueden variar. Las funciones de valor se puede dividir en dos categorías con una versión general y óptima:

- Función de valor $V(s)$: Se usa para cuantificar cómo de bueno es un estado. Determina el rendimiento esperado si se empieza en el estado 's' y luego se actúa de acuerdo con una determinada política 'π'.

$$V^\pi(s) = \mathbb{E}_{\tau \sim \pi}[R(\tau)|s_1 = s]$$

donde:

- $V^\pi(s)$: El valor esperado del estado s bajo la política π.
 - $\mathbb{E}_{\tau \sim \pi}$: La esperanza o valor promedio de la trayectoria τ según la política π.
 - $R(\tau)$: La recompensa total obtenida a lo largo de la trayectoria τ.
 - $s_1 = s$: El proceso inicial en el estado s.
- Función de valor óptima $V^*(s)$: Determina el rendimiento esperado si se comienza en el estado 's', actuando siempre con la política óptima del entorno, es decir, elegir la mejor acción en cada estado para obtener la mayor recompensa a largo plazo.

$$V^*(s) = \max_{\pi} V^\pi(s) = \max_{\pi} \mathbb{E}_{\tau \sim \pi}[R(\tau)|s_1 = s]$$

donde:

- $V^*(s)$: El mejor valor esperado del estado s entre todas las políticas π.
 - $\max_{\pi} V^\pi(s)$: De todas las políticas se selecciona la que devuelve el mayor valor respecto del estado s.
- Función de valor-acción $Q(s,a)$: Cuantifica cuánto de buena es una acción dado un estado. Determina el rendimiento esperado si, en el estado 's', se toma la acción 'a' y, a partir de ahí, el agente actúa siguiendo una política π previamente establecida para tomar decisiones.

$$Q(s, a) = \mathbb{E}[R(\tau)|s_1 = s, a_1 = a]$$

donde:

- $\mathbb{E}[\]$: Esperanza matemática o valor esperado.
- $R(\tau)$: La recompensa acumulada en la trayectoria.
- Función de valor-acción óptima $Q^*(s,a)$: Determina cuál es el rendimiento de una acción 'a' en un estado 's', toma una acción arbitraria y luego actúa en el entorno de acuerdo con la política óptima del agente.

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) = \max_{\pi} \mathbb{E}_{\tau \sim \pi}[R(\tau)|s_1 = s, a_1 = a]$$

donde:

- $Q^*(s, a)$: Es el mejor valor que puedes obtener si, estando en el estado s, tomas la acción a y luego sigues siempre la política óptima.
- $\max_{\pi} Q^{\pi}(s, a)$: De todas las políticas se selecciona la de mayor valor.
- $\tau \sim \pi$: La trayectoria τ generada siguiendo la política π .

2.2.3.-Q-learning

El algoritmo Q-learning [16], ampliamente reconocido como uno de los pilares fundamentales en el ámbito del aprendizaje por refuerzo, permite que un agente inteligente aprenda a tomar decisiones óptimas en entornos complejos y dinámicos. Este enfoque, clasificado como model-free, opera bajo el marco teórico de los Procesos de Decisión de Markov (MDP). En este contexto, el entorno se describe mediante estados, acciones posibles y recompensas asociadas. Su versatilidad lo ha convertido en una herramienta clave en aplicaciones como la robótica, los juegos estratégicos y la optimización de sistemas complejos.

El objetivo principal del Q-learning es encontrar la función de valor-acción $Q^*(s, a)$ óptima, que se define como el valor máximo esperado al tomar una acción 'a' en un estado 's', y seguir la política óptima a partir de este momento. En la práctica, el Q-learning utiliza una estructura llamada Q-Table para almacenar y actualizar los valores de $Q(s, a)$ para cada par {estado/acción}, donde cada entrada representa la utilidad esperada de

ejecutar una acción específica en un estado dado. A través de un proceso de exploración y explotación, el agente actualiza estos valores incorporando tanto la recompensa inmediata como una estimación del rendimiento futuro. Esta tabla se caracteriza por:

- Las filas representan los estado posibles (s).
- Las columnas representan las acciones posibles (a).
- Las celdas almacenan el valor $Q(s, a)$ que mide la utilidad esperada de realizar una acción a en un estado s .

Tabla 2.1: Tabla Q (Q-Table)

	a_1	a_2	...	a_n
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$		$Q(s_1, a_n)$
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$		$Q(s_2, a_n)$
...				
s_n	$Q(s_n, a_1)$	$Q(s_n, a_1)$		$Q(s_n, a_n)$

Se actualiza iterativamente mediante la regla de actualización de Q-learning donde se actualizan los valores de la tabla:

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

Donde:

- α : Tasa de aprendizaje, controla el peso que se asigna al nuevo valor.
- r : Recompensa recibida al tomar la acción a en el estado s .
- s' : Estado al que transita el sistema tras ejecutar la acción a en el estado s .
- γ : Factor de descuento, pondera la importancia de las recompensas futuras.

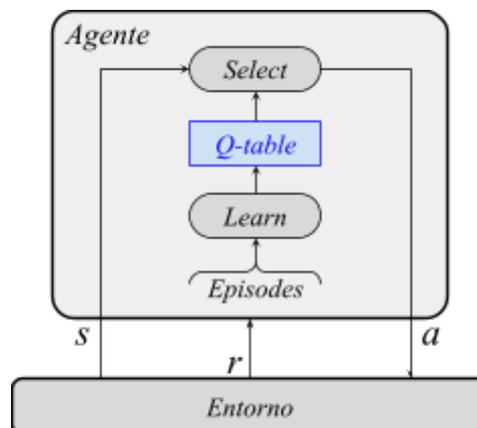


Figura 2.5: Algoritmo Q-Learning

Algoritmo 2.1: Q-Learning

```
1  $Q(s,a) \leftarrow$  inicialización a cero  $\forall s, \forall a$ 
2
3 repetir
4    $s \leftarrow$  estado inicial (e.g estado aleatorio)
5   repetir
6      $a \leftarrow$  seleccionar( $s$ )
7      $s', r \leftarrow$  resultado de una acción del entorno
8      $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$ 
9      $s \leftarrow s'$ 
10  hasta que  $s$  sea un estado terminal
11 hasta el número máximo de iteraciones
```

2.2.4.-Deep Q-learning

El Deep Q-Learning (DQL) [17] es una extensión del algoritmo Q-Learning que combina el aprendizaje por refuerzo con redes neuronales profundas. Este enfoque permite al agente manejar espacios de estado complejos y de alta dimensionalidad, como imágenes en bruto, superando las limitaciones de los métodos tradicionales basados en tablas Q. El DQL fue introducido por DeepMind y demostró ser efectivo en tareas como juegos de Atari, alcanzando rendimientos comparables o superiores a los de jugadores humanos.

El DQL combina el algoritmo Q-Learning con el uso de una red neuronal profunda, llamada Q-Network, para aproximar la función $Q(s, a)$. Esta red recibe como entrada una representación del estado s y genera como salida los valores $Q(s, a)$ para todas las posibles acciones en ese estado.

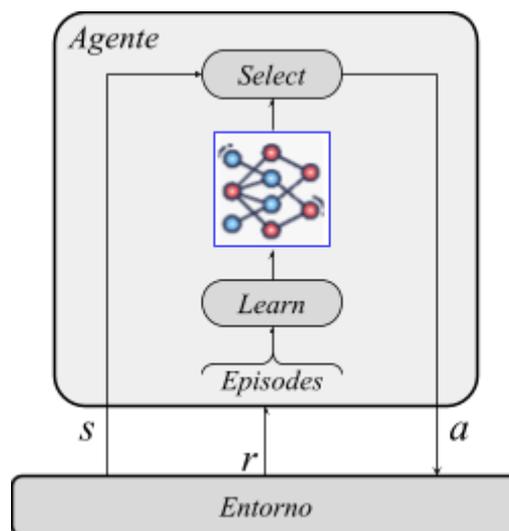


Figura 2.6 : Algoritmo Deep Q-Learning

En la práctica, se usan dos redes neuronales, una principal que realiza la estimación de los valores Q del estado s y las acciones a , y la segunda llamada red neuronal objetivo, la cual tiene la misma arquitectura que la primera pero su finalidad es aproximar los valores Q del siguiente estado y siguiente acción. El proceso de aprendizaje se realiza en la red principal. Para el entrenamiento se utilizan dos fórmulas, la ecuación de Bellman y una función de pérdida:

- Ecuación de Bellman:

$$Q(s, a; \theta) = r + \gamma \max_{a'} (s', a'; \theta')$$

Donde:

- r : Recompensa inmediata
 - γ : Factor de descuento
 - θ : Parámetros de la red principal.
 - θ' : Parámetros de la red objetivo.
- Función de pérdida:

$$L(\theta) = \mathbb{E}[(r + \gamma \max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2]$$

Esta es la función que se minimiza utilizando el algoritmo de descenso de gradientes.

2.3.- COMPUTACIÓN EVOLUTIVA

La computación evolutiva es un campo de la informática y la inteligencia artificial que se basa en el proceso de evolución biológica para resolver problemas complejos de búsqueda y optimización. Al igual que en la evolución natural, estos algoritmos parten de una población original inicializada de forma aleatoria con soluciones candidatas.

De esta población se seleccionan los individuos mejor adaptados para la generación de nuevos individuos, para los cuales se calcula su aptitud (función fitness o evaluación) sobre el problema que se pretende resolver. Se realiza un reemplazo de la población a partir de la descendencia generada eliminando los individuos menos adaptados, se realiza esto de forma iterativa hasta conseguir una solución óptima, es decir la convergencia de la población, o al alcanzar un número determinado de generaciones [18].

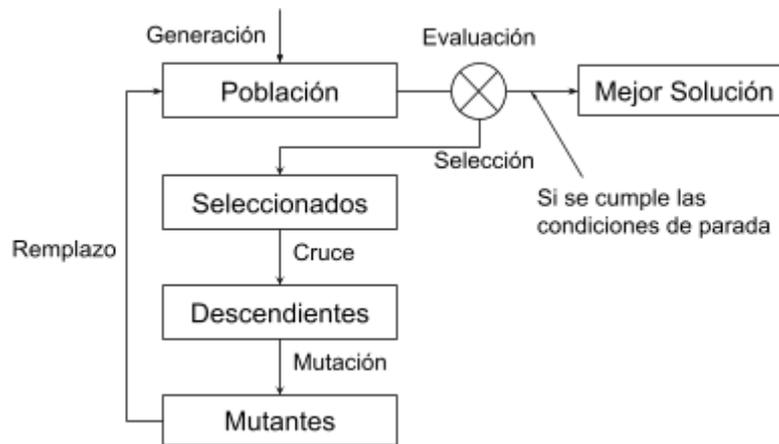


Figura 2.7: Proceso de algoritmo evolutivo

Los algoritmos genéticos (AAGG) constituyen la rama con más repercusión de la computación evolutiva [18]. Se caracterizan por el uso de codificación de los individuos basadas en vectores de dimensión fija, tradicionalmente con el uso de codificación binaria pero también se puede usar codificación real la cual presenta ciertas ventajas en problemas donde la función de evaluación es continua, obteniendo una interpretación directa, mayor precisión, no estando limitado a las potencias de 2 y tiene una mayor exploración a funciones de variables reales de forma gradual. Estos algoritmos están compuestos por los siguientes conceptos básicos:

- Individuo o cromosoma: Cada uno de los puntos del espacio de búsqueda. Se busca aquel ‘individuo’ que resuelve de forma óptima el problema.
- Gen: cada uno de los parámetros de un individuo. Se crearán nuevos individuos alterando y/o combinando los genes de otros individuos.
- Alelo: cada uno de los posibles valores de un gen. Típicamente serán valores cero (0) y uno (1), aunque dependerá del problema concreto que se esté abordando.
- Población: El conjunto de puntos del espacio de búsqueda (individuos) en un momento dado del proceso de evolución como puede ser la población inicial.

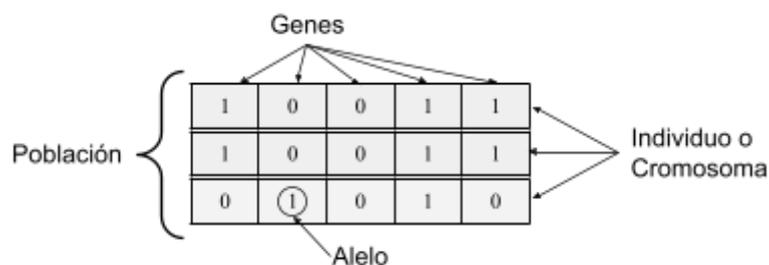


Figura 2.8: Representación algoritmo genético

Para la codificación del problema se define cómo se representan las posibles soluciones al problema dentro del algoritmo. La codificación establece la relación entre el espacio de búsqueda y el espacio de soluciones. El espacio de búsqueda define el conjunto de todas las posibles representaciones codificadas de las soluciones. El espacio de soluciones es el conjunto de soluciones válidas al problema.

La convergencia de estos algoritmos hacia el óptimo global que mejora la función de evaluación (o función fitness), se considera cuando dentro de la población todos los genes han convergido, se considera que un gen ha convergido cuando el 95% de la población comparte ese mismo valor para ese gen en particular. Para lograr esta convergencia de forma óptima se busca encontrar un equilibrio entre exploración que trata de buscar soluciones alejadas de las actuales y la explotación o búsqueda local que trata de mantener las soluciones buenas y refinarlas.

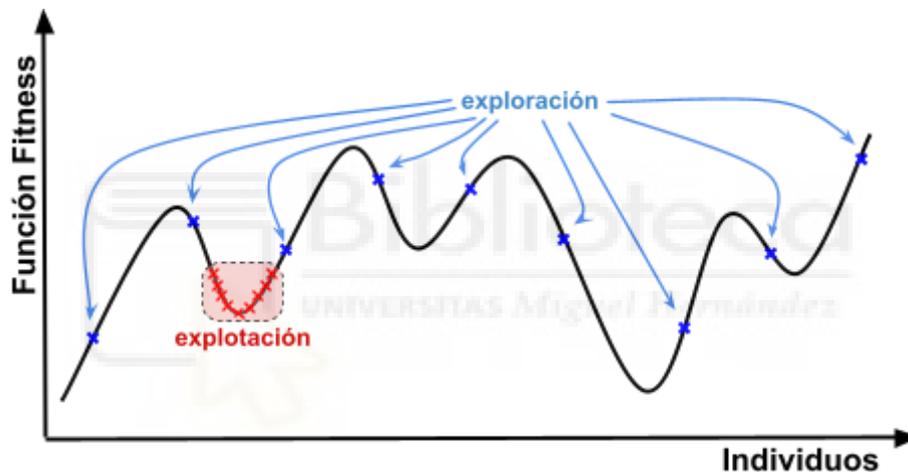


Figura 2.9: Exploración vs Explotación

A continuación, se van a presentar los elementos de selección, cruce, mutación y reemplazo de individuos para obtener los mejores resultados posibles:

2.3.1.- Selección

La selección es un proceso fundamental en los algoritmos evolutivos, pues determina qué individuos serán utilizados para generar la siguiente generación. Los métodos de selección son los siguientes:

- **Selección proporcional:** La probabilidad de que un individuo sea seleccionado es proporcional a su grado de adaptación. Este método tiene el inconveniente de favorecer la selección de individuos con un grado de adaptación superior a la media de la población, lo que puede provocar caer en óptimos locales.

- Selección por orden: La probabilidad de que un individuo sea seleccionado es proporcional a su grado de adaptación dentro de la población. Este método soluciona el problema de la selección proporcional, pero presenta un inconveniente común a ambos operadores: la importancia de la aleatoriedad en la selección.
- Muestreo universal estocástico o método de la ruleta: La probabilidad de que un individuo sea seleccionado es proporcional a la diferencia entre su aptitud y la de sus competidores. Cada individuo se elige tantas veces como sea necesario para cumplir con la cantidad de población de la siguiente generación. Además, en este operador se preservan los mejores 'n' individuos (elitismo), asegurando que los mejores de cada generación siempre se mantengan.
- Selección por torneo: La selección por torneo es un método en el que se elige a los mejores individuos de un subconjunto aleatorio de la población. En este caso, se selecciona un grupo de individuos aleatorios (denominado torneo) y se elige al individuo con la mayor aptitud como ganador del torneo.

2.3.2.- Cruce

El operador de cruce es el operador más importante de los algoritmos genéticos debido a su capacidad de generar nuevos individuos para la convergencia de la población. Este operador se basa en la selección de dos individuos de una población para producir nuevos individuos (codificaciones de soluciones) con características de los padres. Existen diferentes operadores para realizar esta tarea ya sea para valores discretos o reales.

2.3.2.1.- Operadores de cruce codificación discreta

Se realiza sobre los individuos llamados padres o progenitores (normalmente 2) y se combinan para obtener el mismo número de hijos o descendencia. Estos hijos heredan características de cada padre y para realizarse el cruce se utiliza una probabilidad de cruce p_c , si esta probabilidad no se cumple la descendencia serán los mismos padres. Para este tipo de codificación se tienen las siguientes estrategias:

- Operador de cruce basado en un punto: Este operador selecciona dos padres, se elige un punto de cruce de forma aleatoria y por último se combinan las subcadenas para generar la descendencia. Las ventajas de este operador es la alta explotación debido a la preservación de bloques de genes y las desventaja es la baja exploración debido a la poca combinación de los genes.

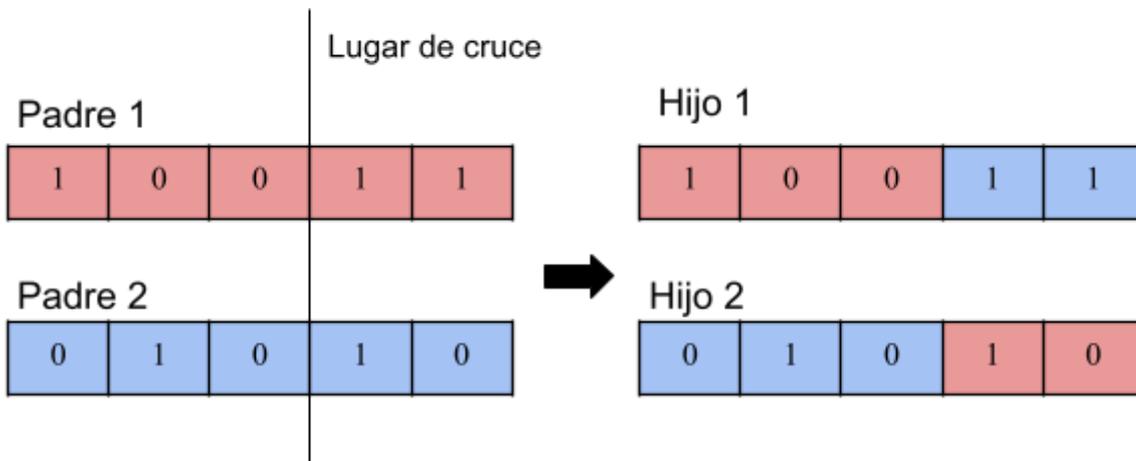


Figura 2.10: Operador de cruce en un punto

- Operador de cruce basado en dos puntos: El funcionamiento de este operador es similar al anterior donde la diferencia radica en que se seleccionan dos puntos de cruce de forma aleatoria. Las ventajas de este operador es la alta exploración debido a la preservación de bloques de genes y la desventaja es la baja exploración debida a la poca combinación de los genes.

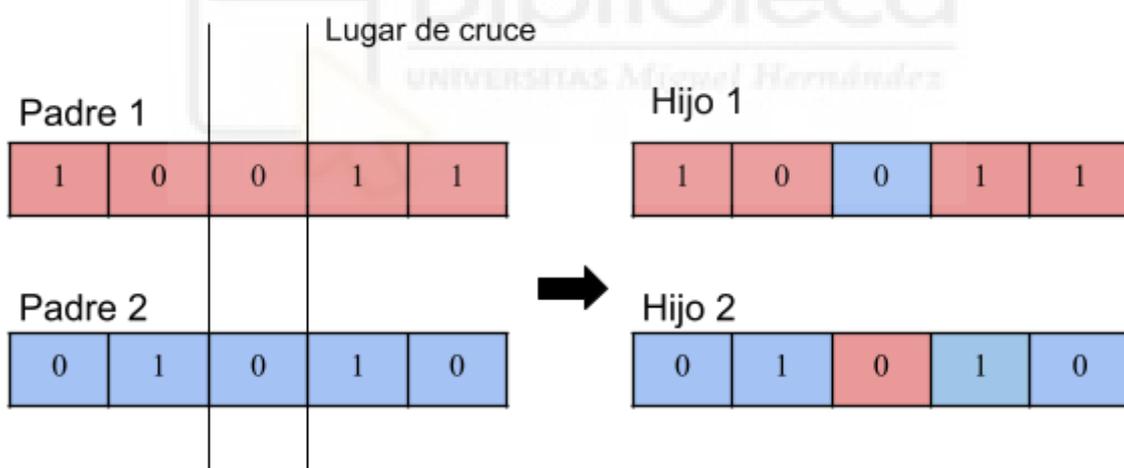


Figura 2.11: Operador de cruce en dos punto

- Operador de cruce uniforme: Este operador selecciona dos padres y se genera una máscara de cruce de forma aleatoria para generar la descendencia. Para la asignación de los genes se asignan a_i si $m_i = 0$ y $b_i =$ si $m_i = 1$. Para el primer hijo se aplica la máscara con lo anterior y para el segundo con la máscara inversa. Este operador presenta una mayor exploración debido a la alta combinación lo que puede producir la fragmentación de cadenas de genes beneficiosas.

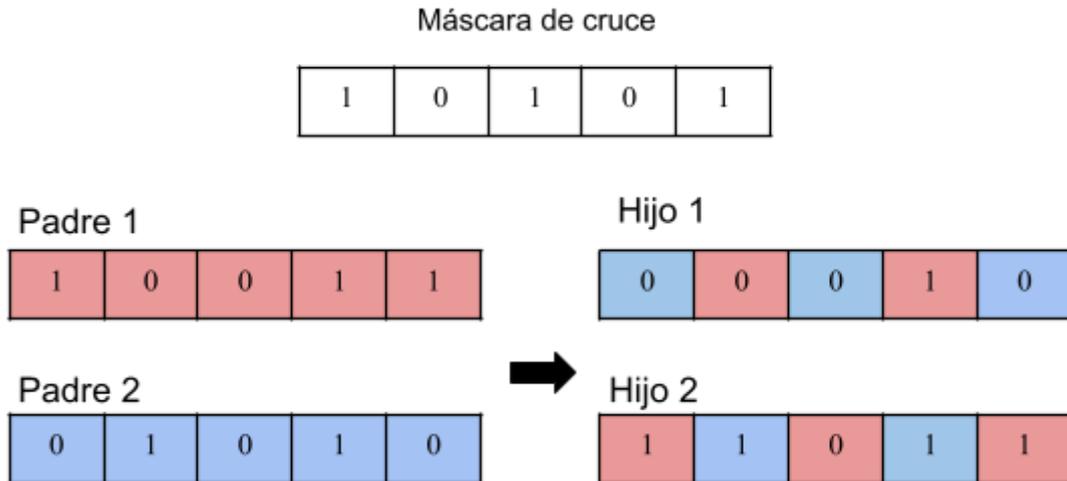


Figura 2.12: Operador de cruce uniforme

- Operador de cruce generalizado: Dados dos padres, esta operador genera la descendencia a partir de dos puntos dentro de un intervalo la descendencia donde el hijo a' se calcula de forma aleatoria en el intervalo y el segundo será su simétrico:

$$a' \in g^{-1}([g(a \text{ and } b), g(a \text{ or } b)])$$

$$b' = g^{-1}(g(a) + g(b) - g(a'))$$

La exploración y la explotación de este operador depende de los valores de los padres.

2.3.2.2.- Operadores de cruce codificación real

La utilización de operadores de codificación real se aplica cuando la codificación real contiene infinitos valores y los operadores de cruce discretos, que se basan en la recombinación de los genes originales, no convergen hacia una solución óptima. A parte de esto presenta las ventajas de que se obtiene una interpretación directa entre el espacio de búsqueda y el espacio de soluciones y una exploración más precisa lo que ayuda a encontrar mejores soluciones. Para codificación real encontramos los siguientes operadores:

- Operador de cruce plano: Este operador utiliza dos progenitores y genera una descendencia de dos individuos. Para la generación de los nuevos individuos se genera un intervalo de cruce:

$$C_i = [\min(a_i, b_i), \max(a_i, b_i)]$$

Los valores de los hijos se generan a partir de valores aleatorios dentro del intervalo de los padres. Este operador ofrece poca exploración y mucha explotación lo que supone una dificultad para salir de óptimos locales.

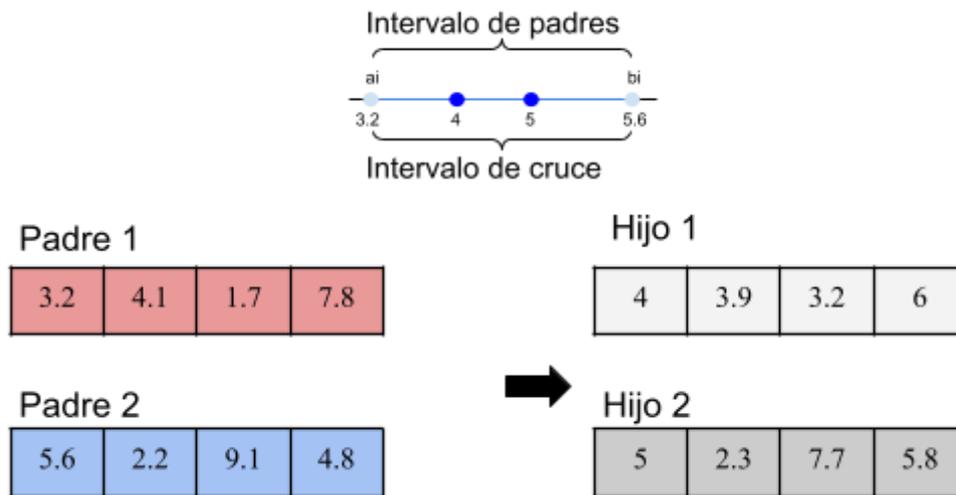


Figura 2.13: Operador de cruce plano

- Operador de cruce combinado (BLX-p): Este operador funciona de forma parecida al anterior introduciendo las siguientes variaciones para el cálculo del intervalo de cruce. Se añade una constante α a cada gen progenitor, se trata de una constante preseleccionada tal que $\alpha \in [0, 1]$ y una variable I que es la diferencia entre los valores de los genes progenitores $I = b_i - a_i$.

$$C_i = [a_i - \alpha I, b_i + \alpha I]$$

El cálculo de los genes del primer hijo es un valor aleatorio dentro del intervalo mientras que el segundo es su complementario. Este operador es versátil ya que puede obtener una buena exploración y explotación (ver figura 2.9) pero con el inconveniente de que dependen del valor de α y de I .

- Operador de cruce morfológico: El cruce morfológico se basa en la idea de usar la diversidad de la población para controlar el rango de los valores generados para los hijos. Este enfoque combina principios de la morfología matemática, donde se utilizan operaciones de expansión y contracción para generar nuevas soluciones.

2.3.3.- Mutación

El operador de mutación es un componente de los algoritmos evolutivos que introduce variabilidad en la población ayudando a la convergencia del algoritmo. Actúa, con una

muy baja probabilidad, directamente sobre las cadenas, cambiando de forma aleatoria alguno de sus valores ayudando a escapar de mínimos locales.

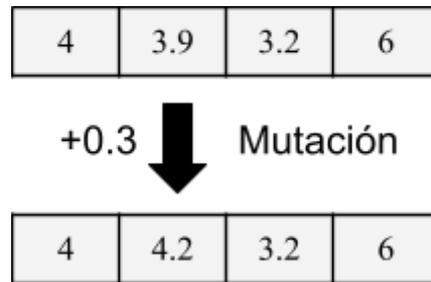


Figura 2.14: Mutación de un gen codificación real.

2.3.4.- Reemplazo de individuos

El reemplazo de individuos determina la forma de reemplazo de los individuos dentro de una población para obtener una nueva generación la cual esté más adaptada. Está determinado por dos factores: la tasa de reemplazo que indica la cantidad de individuos que sustituimos para la próxima generación y la tasa de elitismo que determina la cantidad de los mejores individuos que pasan a la siguiente generación de forma directa.

2.4.- RESUMEN

En este capítulo se ha realizado una revisión del estudio de los enfoques de aprendizaje por refuerzo, como Q-Learning y Deep Q-Learning y los algoritmos genéticos que pertenecen a la computación evolutiva. Estos dos enfoques se utilizan para resolver problemas de inteligencia artificial con sus diferencias en su funcionamiento y aplicaciones.

A grandes rasgos, el aprendizaje por refuerzo se enfoca en maximizar recompensas mediante la interacción permanente con el entorno. Los agentes aprenden a tomar decisiones óptimas a lo largo del tiempo basándose en prueba y error. Dentro de este enfoque, Q-Learning utiliza una tabla para almacenar valores estado-acción, siendo ideal para problemas discretos y de menor complejidad. Por otro lado, Deep Q-Learning (DQL) extiende este enfoque al emplear redes neuronales profundas, lo que permite abordar problemas con espacios de estado amplios y de alta dimensionalidad, aunque a costa de mayores recursos computacionales y tiempos de entrenamiento.

Respecto a los algoritmos genéticos, estos adoptan un enfoque inspirado en la evolución biológica. Estos no dependen de interacciones directas con un entorno, sino que trabajan sobre una población de posibles soluciones. Mediante operadores de selección, cruce y mutación, buscan de manera iterativa optimizar las soluciones. Si bien destacan por su

capacidad para explorar grandes espacios de búsqueda, su convergencia puede ser más lenta y existe el riesgo de quedar atrapados en óptimos locales.



Capítulo 3

Hipótesis de trabajo



En este capítulo se presentan de manera detallada las herramientas y bibliotecas que han sido fundamentales para el desarrollo del proyecto. Se inicia con Miniforge, una distribución de Conda que no solo facilita la administración de paquetes, sino que también permite la creación y gestión de entornos virtuales en Python, lo que resulta esencial para mantener un entorno de trabajo limpio y organizado. A continuación, se describen las bibliotecas más relevantes que se han empleado a lo largo del proyecto, tales como Matplotlib, Numpy y Keras. Estas bibliotecas son fundamentales para diversas tareas, incluyendo la manipulación eficiente de datos, la visualización gráfica de resultados y el desarrollo de modelos de aprendizaje profundo, que son cruciales para el análisis y la interpretación de datos complejos.

También se abordan otras bibliotecas como Gymnasium, para la implementación y evaluación de algoritmos de aprendizaje por refuerzo, una técnica que constituye la base del proyecto. Esta biblioteca permite simular entornos en los que los algoritmos pueden aprender y mejorar su rendimiento a través de la interacción. Además también se comentan

los módulos Swing y Box2D, que se utilizan en Gymnasium para interfaces y creación de físicas, respectivamente.

Finalmente, se menciona Visual Studio Code (VSCoDe), el editor de código elegido para el desarrollo del proyecto. Se destacan sus características avanzadas, como la integración de herramientas de depuración y la personalización de entornos de trabajo, que optimizan significativamente el flujo de desarrollo y facilitan la colaboración entre los miembros del equipo.

3.1.- MINIFORGE

Miniforge es una distribución de Conda, que es un gestor de paquetes y un sistema de gestión de entornos para Python. Conda es ampliamente utilizado para la gestión de paquetes, dependencias y entornos virtuales en Python, facilitando la instalación y mantenimiento de librerías y aplicaciones. Miniforge ofrece un control más preciso sobre estos paquetes y sus dependencias dentro de un proyecto [19]. Permite tener diferentes configuraciones de entornos de trabajo con diferentes configuraciones, con distintas versiones de Python instaladas y las diferentes librerías necesarias para cada proyecto, lo cual nos permite no tener problemas de compatibilidad de versiones y poder definir un entorno de trabajo único para cada proyecto que deseemos desarrollar.

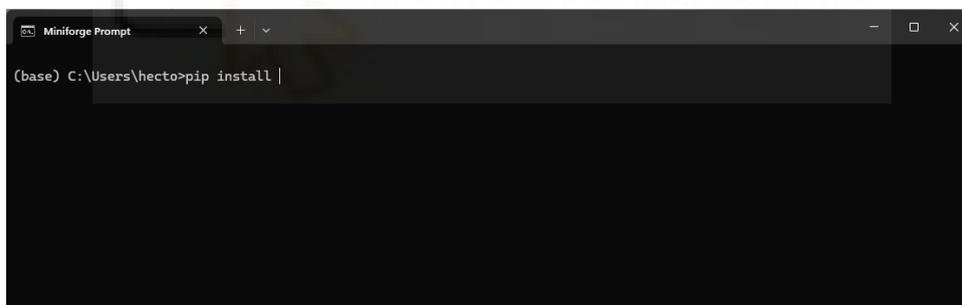


Figura 3.1:Entorno miniforge.

- Python: lenguaje de programación ampliamente utilizado en las aplicaciones web, el desarrollo de software, la ciencia de datos y el machine learning (ML). Es ampliamente utilizado debido a su eficiencia y su facilidad de uso, además de que se puede ejecutar en múltiples plataformas [20]. Aparte para este lenguaje existe una gran cantidad de librerías que ayudan a la hora de realizar proyectos de ciencia de datos y machine learning (ML).
- Matplotlib: biblioteca para la creación de gráficos y visualizaciones de datos. Es especialmente popular debido a su flexibilidad y capacidad de generar gráficos en 2D y 3D de alta calidad [21].

- Numpy: biblioteca para el cálculo numérico y el manejo eficiente de matrices. Proporciona herramientas para realizar operaciones matemáticas avanzadas, incluyendo álgebra lineal, transformadas de Fourier y generación de números aleatorios [22].
- Keras: módulo para la creación y entrenamiento de redes neuronales en Python. Proporciona una interfaz de alto nivel, que permite desarrollar modelos de deep learning de manera rápida y eficiente [23].
- Gymnasium: biblioteca desarrollada por OpenAI utilizada para desarrollar y probar algoritmos de aprendizaje por refuerzo. Proporciona un conjunto de entornos simulados donde los agentes pueden ser entrenados y evaluados [24].
- Swing: módulo de renderizado que se utiliza para la creación de interfaces gráficas de usuario. Se utiliza en Gymnasium para la visualización de entornos [25].
- Box2D: motor de simulación de física en 2D, diseñado para modelar colisiones y movimientos realistas de cuerpos rígidos. Junto con Gymnasium, se utiliza para simular entornos [26].

3.2.- VISUAL STUDIO CODE

Visual Studio Code es un editor de código desarrollado por Microsoft que combina la simplicidad de un editor con potentes herramientas de desarrollo. Su ciclo de edición, compilación y depuración es ágil. Ofrece soporte para cientos de lenguajes de programación, resaltado de sintaxis, coincidencia de corchetes, auto-indentación, selección de bloques y fragmentos de código [27]. A continuación, se muestran los diferentes plugins que se han utilizado para la realización del proyecto.

- Python: Plugin VSCode [29]: Soporte para trabajar con Python en VSCode.

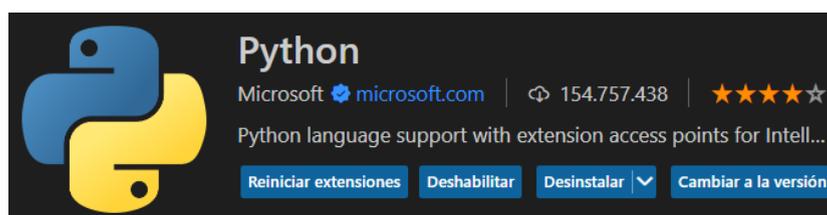


Figura 3.2: Plugin de Python en VS Code

- Jupyter Plugin VSCode [30]: Utilización de cuadernos de Jupyter con cualquier entorno Python y kernels.

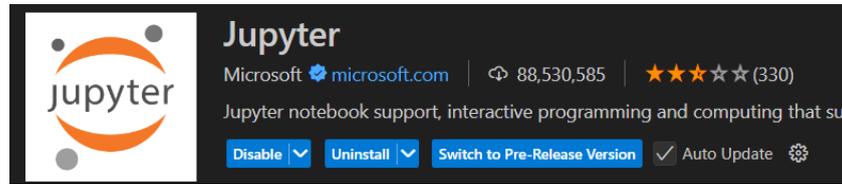


Figura 3.3: Plugin de Jupyter en VS Code

- GitHub Copilot plugin VSCode [31]: IA para la ayuda de la generación de código.

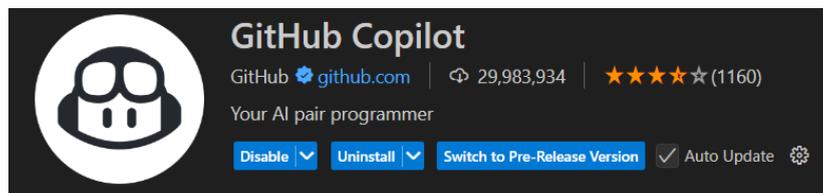


Figura 3.4: Plugin de GitHub Copilot en VS Code

3.3.- LUNAR LANDER

Lunar Lander se trata de un videojuego arcade desarrollado por Atari y lanzado en el año 1979, donde se controla un módulo lunar con el objetivo de aterrizar de forma controlada en áreas designadas en la superficie lunar, manejando el combustible el cual es limitado y evitando colisiones [32].

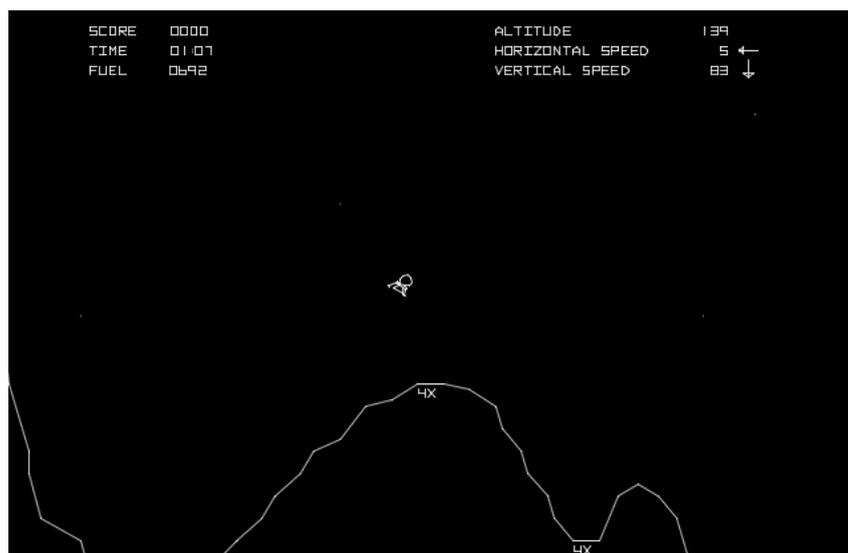


Figura 3.5: Juego Lunar Lander Atari

A partir de este clásico juego, utilizamos el entorno Lunar Lander de Gymnasium que se basa en el juego anterior para desarrollar y probar algoritmos para el control autónomo de descenso y aterrizaje.

3.4.- GOOGLE COLAB

Google Colab es una herramienta gratuita de Google donde se ejecuta código en Python desde la nube. Está basado en los notebooks de Jupyter y proporciona a los usuarios recursos computacionales como GPU y TPU, lo que permite realizar tareas de alta demanda computacional como IA o análisis de datos. Además integra la opción de poder utilizar Google Drive para la gestión de los datos. Una de las grandes ventajas de Google Colab es que no requiere configuración previa, lo que permite comenzar a trabajar de forma inmediata desde cualquier dispositivo con acceso a internet. Su entorno permite combinar código, texto y visualizaciones, lo que resulta muy útil para documentar y probar modelos de forma ordenada y clara.

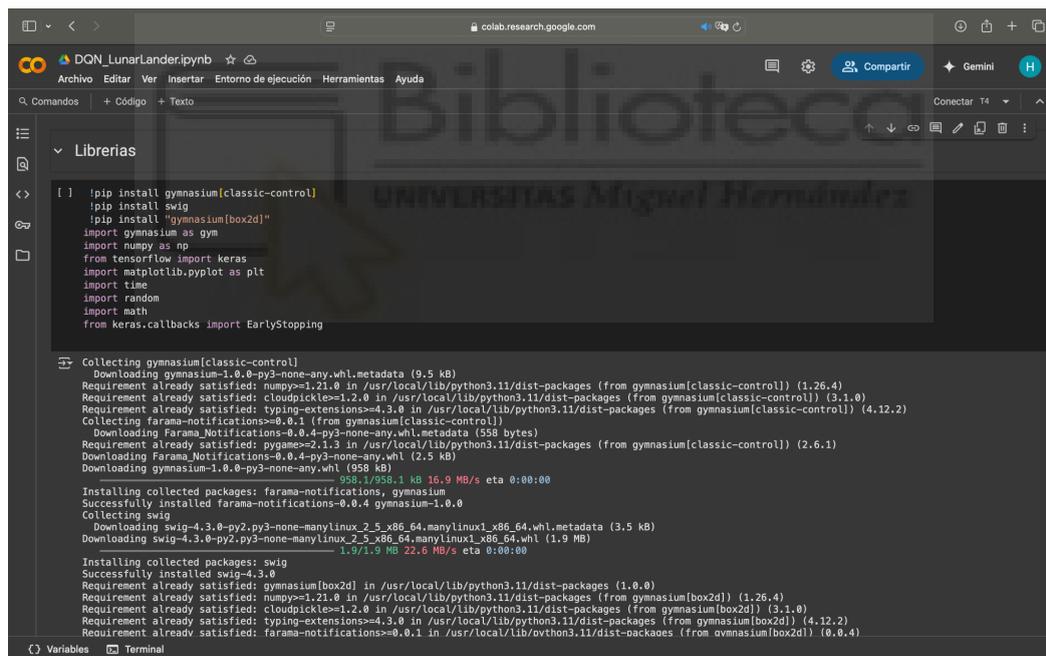


Figura 3.6: Entorno Colab

3.5.- EQUIPO DE DESARROLLO

El desarrollo del proyecto se ha llevado a cabo utilizando como equipo principal un MacBook Pro con chip M4 Pro. Gracias a su arquitectura de alto rendimiento y su gran cantidad de núcleos de CPU, este equipo permite un muy buen rendimiento para la realización de un proyecto de aprendizaje por refuerzo. Además, la integración nativa con macOS facilita

el uso de herramientas de línea de comandos (Homebrew, iTerm2) y entornos de desarrollo (Docker, Python virtual environments), lo que agiliza enormemente el flujo de trabajo diario.

Para complementar las pruebas locales y comparar rendimiento en la nube, hemos utilizado instancias virtuales de Google Colab. Esta combinación híbrida ha permitido iterar de forma rápida, medir tiempos de simulación en distintos entornos y validar que el rendimiento es coherente independientemente de la plataforma utilizada.



Capítulo 4

Metodología y resultados

4.1.- PLANIFICACIÓN DEL PROYECTO

4.1.1.- Ciclo de vida del proyecto

El ciclo de vida seleccionado para el desarrollo de este proyecto ha sido un modelo de ciclo de vida ágil basado en Scrum, adaptado a un contexto de trabajo individual. Esta metodología nos permite el desarrollo del proyecto en iteraciones cortas denominadas sprints, en las cuales se definen objetivos concretos y versiones funcionales. Dado que el proyecto fue individual, los diferentes roles propios de la metodología Scrum recaen sobre la misma persona. Estas son las funciones de los diferentes roles:

- Product Owner: Define los objetivos del proyecto.
- Scrum Master: Supervisa el proceso, organiza los sprints y evalúa los avances.

- **Development Team:** Implementa los algoritmos, ejecuta los experimentos y analiza el resultado de los mismos.

El desarrollo se ha dividido en los siguientes sprints:

- **Sprint 1:** Preparación del entorno de entrenamiento donde se hace un análisis del problema Lunar Lander y se realiza la configuración del entorno de entrenamiento.
- **Sprint 2:** Implementación del algoritmo genético y de todos los métodos utilizados en el proyecto.
- **Sprint 3:** Optimización y pruebas de los algoritmos genéticos con los diferentes operadores de mutación, selección y tasas de mutación.
- **Sprint 4:** Implementación del algoritmo Deep Q-Learning y pruebas para determinar los hiperparámetros óptimos.
- **Sprint 5:** Optimización y pruebas con diferentes redes neuronales, utilizando target network y evaluación del rendimiento.
- **Sprint 6:** Comparación de rendimiento entre algoritmos genéticos y algoritmo de aprendizaje por refuerzo y preparación de notebooks.

Esta planificación ha permitido una gestión eficaz del proyecto, fomentando la iteración, la mejora continua y una adecuada organización del trabajo, alineándose con los principios de desarrollo ágil.

4.1.2.- Planificación temporal del proyecto

Se ha realizado la elaboración de un diagrama de Gantt para proporcionar una visión clara y estructurada de la distribución del tiempo de los sprints y las tareas que se han realizado a lo largo del proyecto. Aunque la metodología Scrum no se basa en tiempos fijos, nos resulta útil para la gestión del tiempo, el control del progreso.

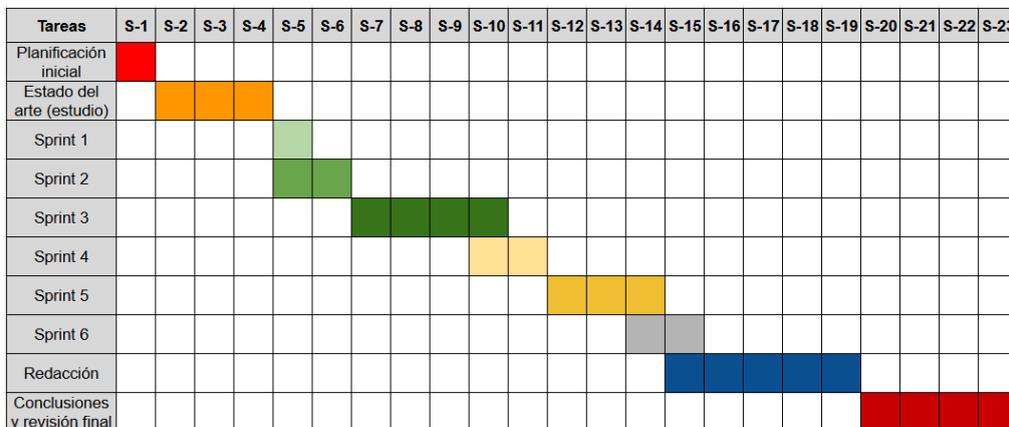


Figura 4.1: Diagrama Gantt

4.2.- CONCEPTOS PREVIOS LUNAR LANDER

El problema del Lunar Lander es un desafío clásico dentro de los entornos de simulación de OpenAI Gym, basado en Box2D [28], y observado en la Figura 1. Representa un problema de optimización de trayectorias para un cohete, donde el objetivo es aterrizar el módulo lunar en una plataforma con coordenadas (0,0). Este es lanzado desde una posición inicial y con una velocidad aleatoria.

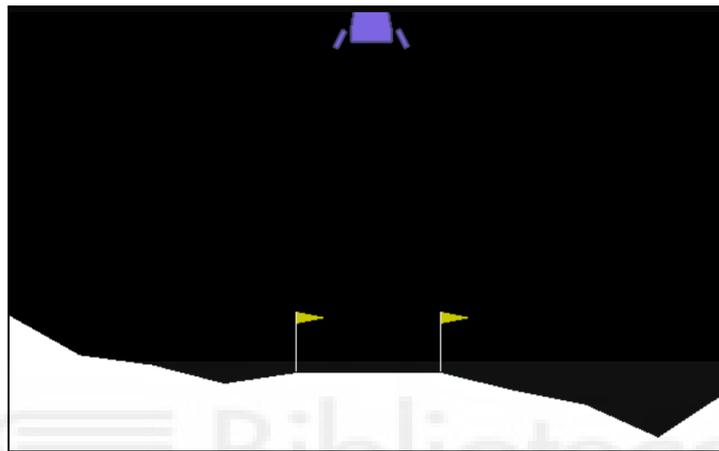


Figura 4.2: Lunar Lander en el entorno de simulación OpenAI Gym

En este trabajo, se presentan soluciones al problema del Lunar Lander utilizando algoritmo de aprendizaje por refuerzo y un algoritmo genético. A continuación, se especificarán conceptos relevantes en el entorno del Lunar Lander, incluyendo las observaciones recopiladas, posibles acciones a tomar y recompensas recibidas al ejecutar una determinada acción en el entorno:

- Espacio de observaciones: está formado por un vector de 8 valores que describe la posición, velocidad, ángulo y estado de los motores del cohete. En concreto, estas incluyen:
 - Coordenadas X e Y: Posiciones horizontales y verticales del módulo lunar en el entorno, con valores en el rango $[-1.5, 1.5]$ para cada una.
 - Velocidad en X e Y: Velocidades horizontal del módulo en metros por segundo (m/s), en el rango $[-5, 5]$ para cada una.
 - Ángulo respecto al suelo: Orientación del módulo en radianes, en el rango $[-3.14, 3.14]$.
 - Velocidad angular: Velocidad de rotación del módulo en radianes por segundo (rad/s), en el rango $[-5, 5]$.
 - Contacto de la pata derecha y pata izquierda: Indicadores binarios que señalan si la pata derecha o izquierda está tocando el suelo.

- Espacio de acciones: un conjunto de 4 acciones discretas, que incluyen:
 - No hacer nada.
 - Encender el motor izquierdo.
 - Encender el motor derecho.
 - Encender el motor central.

- Recompensas: valores que reflejan lo deseable que son cada uno de los posibles resultados de tomar una acción. Penalizan el consumo de combustible y los aterrizajes fuera de la plataforma, e incentivar aterrizajes suaves y precisos en el objetivo:
 - Uso del motor central: genera una recompensa de -0.3.
 - Uso de los motores laterales: genera una recompensa de -0.03. Es más deseable que usar el motor central.
 - Choque: genera una recompensa de -100. Es el peor resultado posible, y una acción terminal que provoca la finalización del juego.
 - Contacto de una pata con el suelo: genera una recompensa de +10.
 - Aterrizaje suave: genera una recompensa de +100. Es una acción terminal.
 - Aterrizaje entre las banderas: genera una recompensa de +200. Es el mejor resultado posible, y una acción terminal.

4.3.- ALGORITMO GENÉTICO

4.3.1.- Codificación

Cada individuo de la población codifica una red de neuronas densa, sin capas ocultas donde sus componentes son:

- Entradas: el vector de observaciones del entorno (dimensión 8).
- Salidas: el conjunto de acciones posibles (dimensión 4).
- Pesos: el conjunto de conexiones entre las entradas (observaciones) y salidas (acciones).
- Bias: el conjunto de términos que ajustan las actividades de cada salida (acción).

Los individuos están compuestos por un vector real, que representa los pesos y bias de la red neuronal. La dimensión total del genotipo es igual al número de pesos necesarios para conectar las 8 entradas con las 4 salidas debido a que la red está totalmente conectada, más los 4 términos de bias. En total cada individuo tiene 36 dimensiones.

Para determinar la acción a tomar, se realiza el producto matricial entre el vector de observaciones (dimensión 8) y la matriz de pesos del individuo (dimensión 8x4), obteniendo un vector de 4 valores. Luego, se aplica la función softmax() y se identifica la acción con mayor valor en este vector, que será seleccionada.

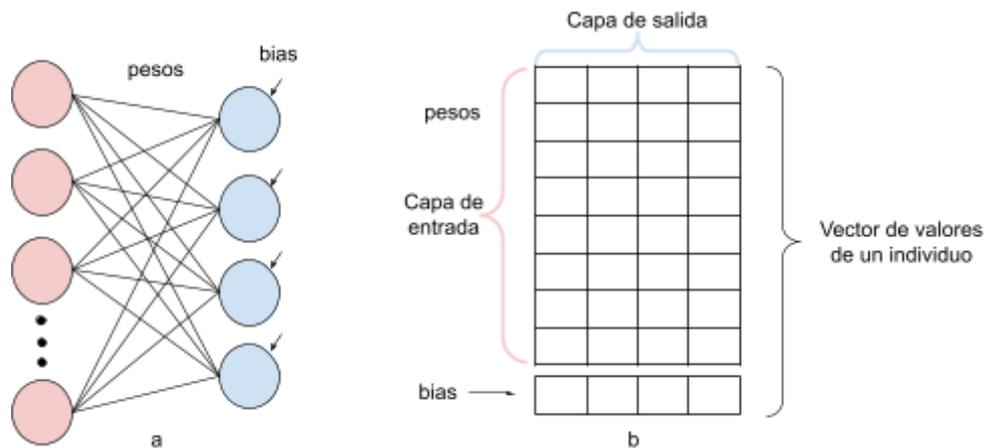


Figura 4.3: a) Red de neuronas codificada por un individuo, compuesta por 8 neuronas de entrada, 4 de salida, y un conjunto de pesos y bias. b) Vector real de individuo que codifica una red de neuronas, mediante un conjunto de pesos y bias.

4.3.2.- Flujo del algoritmo

El enfoque implementado en este apartado del proyecto se basa en un algoritmo genético, un tipo de algoritmo evolutivo inspirado en los procesos de selección natural. El objetivo es encontrar una solución óptima al problema del Lunar Lander, representado como un individuo que maximiza la recompensa acumulada. El algoritmo se compone de los siguientes pasos principales:

1. **Inicialización:** Se genera una población inicial de individuos de manera aleatoria. Cada Individuo representa una posible solución, codificada como los pesos y sesgos de una red neuronal densa (36 parámetros en total).
2. **Evaluación:** Cada individuo se evalúa en el entorno del Lunar Lander para determinar su desempeño. Esto se hace ejecutando múltiples episodios con la red neuronal representada por el individuo y acumulando la recompensa obtenida. La recompensa promedio se utiliza como su valor de fitness.
3. **Selección:** Se seleccionan los mejores individuos de la población en función de su fitness, utilizando un esquema como torneo o ruleta. Este paso prioriza individuos más aptos para transmitir sus características a la siguiente generación.

4. Cruce: Se aplica un operador de cruce entre pares de individuos seleccionados, generando nuevos descendientes que combinan las características de sus progenitores. Este paso busca explorar nuevas regiones del espacio de soluciones.
5. Mutación: Con una probabilidad predefinida, se altera aleatoriamente alguno de los genes de los descendientes, introduciendo diversidad en la población y reduciendo el riesgo de convergencia prematura.
6. Reemplazo: Se crea una nueva generación reemplazando parcialmente o completamente la población anterior con los descendientes generados.
7. Criterio de parada: El proceso iterativo se detiene cuando se alcanza un número máximo de generaciones o se cumple una condición de desempeño satisfactorio.

4.3.3.- Implementación de Operadores

A continuación, se van a presentar los diferentes operadores de selección, cruce y mutación implementados, así como la función objetivo utilizada para resolver el problema del Lunar Lander.

4.3.3.1- Operadores de selección

Operador de selección por torneo

Algoritmo 4.1: Selección por torneo

```

1 def tournament_selection(population, fitness_values, k=3):
2     selected = []
3     for _ in range(len(population)):
4         indices = np.random.choice(len(population), k, replace=False)
5         best = indices[np.argmax([fitness_values[i] for i in indices])]
6         selected.append(population[best])
7     return np.array(selected)

```

Explicación:

1. Se toma un subconjunto aleatorio de 'k' individuos de la población.
2. Se calcula la aptitud de cada individuo en el torneo y se selecciona el que tenga la mayor aptitud.
3. Este proceso se repite para cada individuo de la población.
4. El operador devuelve el mejor individuo de cada torneo, que será utilizado en la siguiente fase del algoritmo evolutivo.

Operador de selección por ruleta con elitismo

La selección por ruleta con elitismo se basa en asignar probabilidades a cada individuo, proporcionalmente a su aptitud. Después, se realiza una selección de manera aleatoria siguiendo estas probabilidades. Además, en este operador se preservan los mejores ‘n’ individuos (elitismo), asegurando que los mejores de cada generación siempre se mantengan.

Algoritmo 4.2: Selección por ruleta con elitismo

```
1 def roulette(population, fitness_values, num_selected = 60, num_elites=15):
2     # Elitismo: selecciona los mejores individuos directamente
3     elite_indices = np.argsort(fitness_values)[-num_elites:]
4     elites = [population[i] for i in elite_indices]
5
6     # Normaliza los valores de fitness para obtener probabilidades
7     total_fitness = sum(fitness_values)
8     probabilities = fitness_values / total_fitness
9
10    # Distribuye los marcadores en la ruleta
11    cumulative_probabilities = np.cumsum(probabilities)
12    markers = np.linspace(0, 1, num_selected - num_elites, endpoint=False)\
13              + np.random.uniform(0, 1 / (num_selected - num_elites))
14
15    # Selecciona individuos segun los marcadores
16    selected = []
17    for marker in markers:
18        for i, cp in enumerate(cumulative_probabilities):
19            if marker <= cp:
20                selected.append(population[i])
21                break
22
23    # Combina los elites y los seleccionados
24    return elites + selected
```

Explicación:

1. **Elitismo:** Se seleccionan los mejores ‘n’ mejores individuos de la población para asegurarse de que los mejores resultados no se pierdan en generaciones posteriores.
2. **Probabilidades:** La aptitud de cada individuo se normaliza, dividiendo por la suma total de las aptitudes de la población. Esto genera un vector de probabilidades.
3. **Ruleta:** Los individuos se seleccionan aleatoriamente según sus probabilidades acumuladas, con una ruleta que determina qué individuos se eligen en función de su aptitud. Cada marcador en la ruleta corresponde a una probabilidad acumulada, y se selecciona el individuo cuyo marcador caiga dentro de esa probabilidad.
4. **Retorno:** El operador devuelve los individuos seleccionados, que son utilizados para la siguiente generación.

Ambos operadores tienen ventajas y desventajas. La selección por torneo tiende a favorecer la diversidad genética, mientras que la selección por ruleta con elitismo garantiza que los mejores individuos se preserven en cada generación.

4.3.3.2- Operadores de cruce

Operador de cruce plano

Algoritmo 4.3: Operador de cruce plano

```
1 def flat_crossover_operator(parent1,parent2):
2     # Calcular el intervalo de cruce
3     min_vals = np.minimum(parent1, parent2)
4     max_vals = np.maximum(parent1, parent2)
5
6     # Generar descendientes
7     child1 = np.random.uniform(min_vals, max_vals)
8     child2 = np.random.uniform(min_vals, max_vals)
9     return child1, child2
```

Explicación:

1. Se calculan los valores mínimos y máximos de cada gen de los padres.
2. Se generan aleatoriamente valores para los genes del primer hijo dentro del intervalo de cruce.
3. Se generan aleatoriamente valores para los genes del segundo hijo dentro del intervalo de cruce.

Operador de cruce combinado BLX- α

Algoritmo 4.4: Operador de cruce BLX- α

```
1 def crossover_BLX(parent1,parent2,alpha=0.5):
2     # Inicializar los hijos
3     child1 = np.zeros_like(parent1)
4     child2 = np.zeros_like(parent2)
5
6     # Generar los hijos utilizando el cruce basado en el intervalo
7     for i in range(len(parent1)):
8         # Calcular el intervalo I
9         I = parent2[i] - parent1[i]
10        # Calcular el intervalo de cruce
11        lower_bound = parent1[i] - alpha * I
12        upper_bound = parent2[i] + alpha * I
13        # Seleccionar un valor aleatorio dentro del intervalo de cruce
14        child1[i] = np.random.uniform(lower_bound, upper_bound)
15        # Calcular el valor complementario para el hijo2
16        child2[i] = parent1[i] + parent2[i] - child1[i]
17    return child1, child2
```

Explicación:

1. Inicializar los hijos como vectores de ceros con el mismo tamaño que los padres.
2. Se realiza un bucle por cada gen para calcular I y el intervalo de cruce.
3. Generar un valor aleatorio para el primer hijo dentro del intervalo de cruce.
4. Calcular el valor complementario para el segundo hijo.

Operador de cruce morfológico

Algoritmo 4.5: Operador de cruce morfológico

```
1 def morphological_crossover(population):
2     child1 = np.zeros_like(population[0])
3     child2 = np.zeros_like(population[0])
4     norm_population = (population - np.min(population))\
5                       / (np.max(population) - np.min(population))
6
7     for i in range(len(population[0])):
8         gene_values = [ind[i] for ind in norm_population]
9
10        diversity = max(gene_values) - min(gene_values)
11        #print(diversity)
12        phi_value = phi(diversity) # valor  $\phi$ 
13        min_val = min(gene_values) + phi_value
14        max_val = max(gene_values) - phi_value
15
16        child1[i] = np.random.uniform(min_val, max_val)
17        child2[i] = min_val + max_val - child1[i]
18
19        child1 *= (np.max(population) - np.min(population)) + np.min(population)
20        child2 *= (np.max(population) - np.min(population)) + np.min(population)
21
22    return child1, child2
```

Explicación:

1. Inicialización y normalización: Se prepara la población y se normalizan los valores de los genes para que estén dentro de un rango uniforme.
2. Cálculo de la diversidad de la población: Para cada gen en la población, se calcula la diversidad, que probablemente es una medida de la variabilidad de los valores de ese gen entre los individuos de la población.
3. Cálculo de un valor ϕ : El valor ϕ es un ajuste basado en la diversidad, y es utilizado para determinar los límites de los genes en los hijos.
4. Generación de los hijos: Para cada gen de los padres, se calcula un valor mínimo y máximo basado en la diversidad y se usa para generar dos hijos. La idea es ajustar la exploración y explotación (ver apartado 2.3) de los valores de los genes en función de la diversidad en la población.
5. Reescalado de los hijos: Finalmente, se reescalan los valores generados para asegurar que los valores de los genes de los hijos estén dentro de los rangos adecuados.

4.3.3.3- Mutación

El operador de mutación se aplica a los individuos de la población con una probabilidad predefinida, conocida como tasa de mutación. Si un gen es seleccionado para ser mutado (según esta tasa), se ajusta su valor de forma aleatoria dentro de un intervalo determinado.

Algoritmo 4.6: Mutación

```
1 def mutate(chromosome, mutation_rate):
2     for i in range(len(chromosome)):
3         if np.random.rand() < mutation_rate:
4             chromosome[i] += np.random.uniform(-0.5, 0.5)
5
6     return chromosome
```

Explicación:

1. Se recorre cada gen del cromosoma.
2. Para cada gen, se genera un número aleatorio entre 0 y 1.
3. Si el número generado es menor que la tasa de mutación, se realiza una alteración en el valor del gen, incrementando o decrementando en un valor aleatorio dentro del intervalo de [0.5, 0.5].
4. Si el número aleatorio es mayor que la tasa de mutación, entonces el gen no se modifica.

4.3.3.4- Función de fitness

La función de fitness es un componente crítico en el algoritmo evolutivo, ya que mide qué tan bien se desempeña un individuo en el entorno del Lunar Lander. Este trabajo, se define como la recompensa promedio obtenida por el individuo al ejecutar varios episodios de prueba (ver algoritmo 4.7).

Explicación:

1. Configuración inicial:
 - Se inicializa el entorno del Lunar Lander.
 - Se configura el modelo de red neuronal del individuo a evaluar con los pesos y sesgos codificados en su cromosoma.
2. Ejecución de episodios:
 - Para cada episodio, el entorno se reinicia y el agente comienza desde una posición inicial aleatoria.
 - En cada paso, se utiliza el modelo para seleccionar una acción en función de las observaciones actuales, aplicando el producto matricial entre las observaciones y los pesos del individuo.

- La acción con mayor valor tras aplicar la función softmax es seleccionada y ejecutada en el entorno.
 - El entorno devuelve una nueva observación, una recompensa por la acción tomada y un indicador de si el episodio ha terminado.
3. Acumulación de recompensas:
- Durante el episodio, se acumulan todas las recompensas obtenidas, penalizando comportamientos indeseados (como choques) y recompensando logros (como aterrizajes suaves).
 - Cálculo del fitness promedio, después de completar los episodios, se calcula el promedio de las recompensas totales, que se utiliza como el valor de fitness del individuo.

Algoritmo 4.7: Fitness

```

1 def fitness(chromosome):
2     env = gym.make("LunarLander-v3", render_mode=None)
3     model = Perceptron(8, 4)
4     model.from_chromosome(chromosome)
5     total_reward = 0
6
7     for _ in range(5): # Generalization with multiple episodes
8         observation, _ = env.reset()
9         episode_reward = 0
10        while True:
11            action = np.argmax(model.forward(observation))
12            observation, reward, terminated, truncated, _ = env.step(action)
13            episode_reward += reward
14            if terminated or truncated:
15                break
16
17        #total_reward += (episode_reward + 500) / 800 # Normalization
18        total_reward += episode_reward
19    env.close()
20
21    return total_reward / 5

```

Esta función garantiza que los individuos que aterrizan el módulo lunar de manera más eficiente, utilizando menos combustible y evitando choques, obtengan un fitness más alto. Esto guía la evolución hacia soluciones que cumplen con los objetivos del problema.

4.3.4.- Resultados Algoritmos Genéticos

En esta sección se presentan los resultados obtenidos al aplicar el algoritmo evolutivo con diferentes combinaciones de operadores e hiperparámetros. El objetivo es identificar la configuración más beneficiosa para resolver el problema. Para cada combinación, se realizaron entrenamientos del algoritmo durante 100 generaciones, evaluando el rendimiento en términos de la recompensa media obtenida (fitness) por el mejor individuo

a lo largo de 100 simulaciones. Adicionalmente, se consideró la velocidad de convergencia hacia una solución óptima como criterio complementario.

Para garantizar un análisis sistemático, se varió un único elemento (ya sea un operador o un hiperparámetro asociado) en cada experimento. Las configuraciones óptimas identificadas en experimentos previos se utilizaron como base para iteraciones posteriores, permitiendo así aproximarnos progresivamente a la mejor combinación de operadores. Inicialmente, se utilizó el operador de selección por torneo con $k=5$. Para dicha configuración se hicieron los siguientes experimentos variando los operadores de cruce y sus parámetros, ya que es el operador más importante y resulta determinante para conseguir la convergencia del problema:

- Cruce plano con mutaciones $m=0$, $m=0.1$, $m=0.3$
- Cruce morfológico con mutaciones $m=0$, $m=0.1$, $m=0.3$
- Cruce BLX- α con valores α :
 - $\alpha=0.3$ y mutación $m=0$
 - $\alpha=0.5$ y mutación $m=0$, $m=0.1$
 - $\alpha=0.7$ y mutación $m=0$

Para la realización de las pruebas los hiperparámetros utilizados han sido los siguientes:

- **population_size**: Tamaño de la cantidad de individuos por generación.
- **chromosome_size**: Tamaño de cada individuo que será siempre igual a un vector de 36 valores que contiene los Pesos ($8*4$) y los Bias 4.
- **generations**: Número de generaciones sobre las que itera el algoritmo.
- **mutation_rate(m)**: Porcentaje de probabilidad que un gen mute.

Experimentos con cruce plano y mutación 0 / 0.1 / 0.3

La gráfica 4.4 muestra el valor de la función fitness a lo largo de 100 generaciones con mutación nula ($m=0$). La línea azul indica el desempeño del mejor individuo de cada generación. Las líneas naranja y verde muestran la media y la mediana, respectivamente, de todos los individuos de cada generación. Como puede observarse, todos los valores son negativos, por lo que esta combinación de operadores no proporciona un resultado satisfactorio (en el juego, la nave se estrella).

Por su parte, la figura 4.5 muestra el rendimiento del mejor individuo de este experimento durante 100 simulaciones. Se observa como el mejor individuo producido por las 100 generaciones obtiene un valor medio de reconconsa $fitness = -136.67$, y en ningún caso de entre dichas simulaciones este valor de fitness resulta satisfactorio.

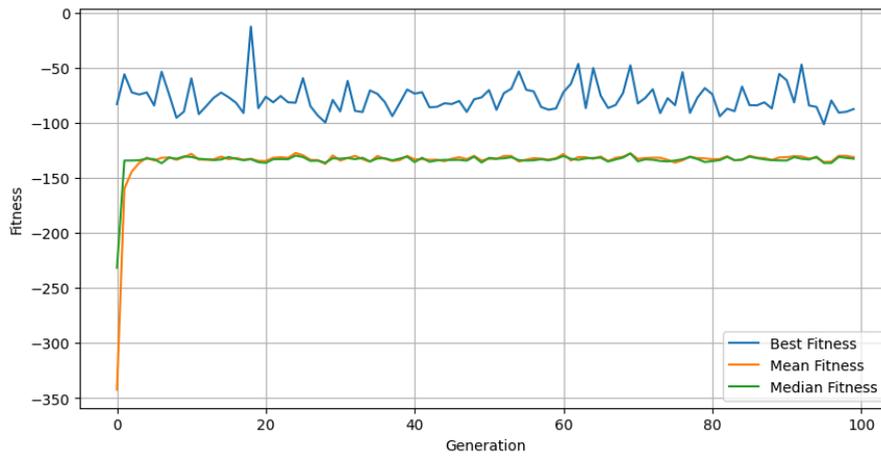


Figura 4.4: Gráfica entrenamiento cruce plano sin mutación.

Resultados despues de 100 simulaciones:
 Recompensa Media : -136.67
 Desviación Típica : 49.70
 Inter. de conf. 95% : ±9.74

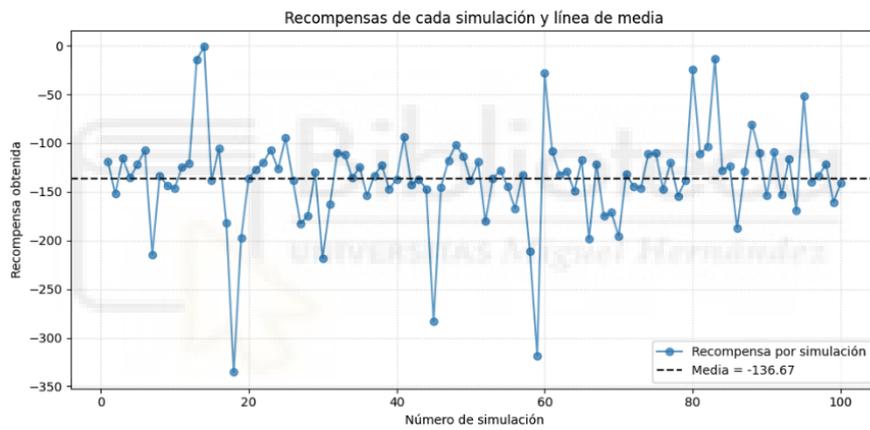


Figura 4.5: Resultado de simulaciones cruce plano sin mutación.

Las gráficas 4.6 y 4.7 muestran el resultado obtenido aplicando una mutación $m=0.1$

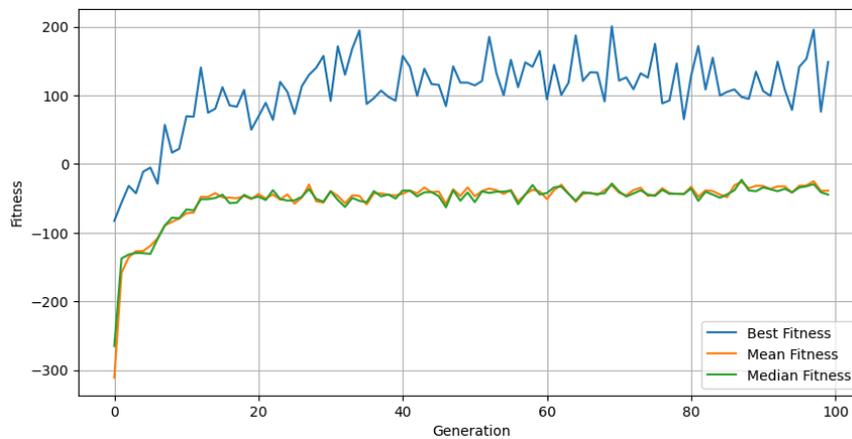


Figura 4.6: Gráfica de entrenamiento cruce plano con mutación 0.1.

Resultados despues de 100 simulaciones:
 Recompensa Media : -64.54
 Desviacion Típica : 143.05
 Inter. de conf. 95% : ±28.04

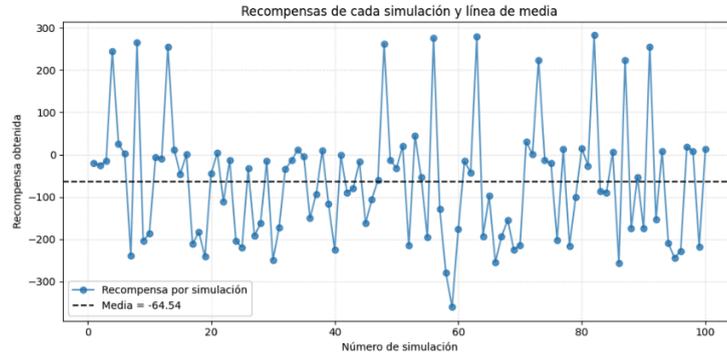


Figura 4.7: Resultado de simulaciones cruce plano con mutación 0.1.

Si bien, para mutación $m=0.1$, el mejor individuo es capaz de aterrizar la nave con suavidad ($fitness \geq 100$), e incluso puede aterrizar entre las banderas ($fitness \geq 200$), esto no ocurre en todos los casos, de hecho, ocurre en muy pocos, por lo que conseguir un resultado satisfactorio (aunque solo sea en parte) depende demasiado de factores aleatorios.

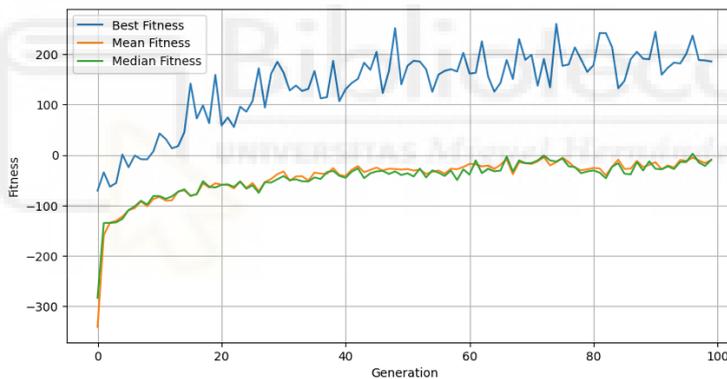


Figura 4.8: Gráfica de entrenamiento cruce plano con mutación 0.3

Resultados despues de 100 simulaciones:
 Recompensa Media : 2.98
 Desviacion Típica : 194.32
 Inter. de conf. 95% : ±38.09

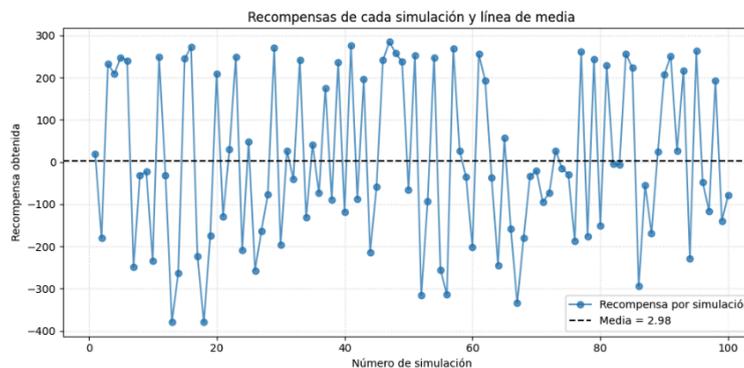


Figura 4.9: Resultado de simulaciones cruce plano con mutación 0.3.

Por último, las figuras 4.8 y 4.9, para mutación $m=0.3$, muestran unos resultados que, en promedio, no mejoran el experimento anterior ($m=0.1$), aunque sí se ha obtenido un mejor individuo que consigue aterrizar de forma óptima en un mayor número de ocasiones.

Como podemos observar en los resultados de las figuras 4.4 a 4.9, el operador de cruce plano no permite la convergencia del algoritmo hacia soluciones óptimas. Esto se puede atribuir a que este operador de cruce tiene mucha explotación y prácticamente ninguna exploración. Al introducir tasas de mutación superiores a 0, se ha intentado que el algoritmo converja, pero apenas sin éxito. Observando la evolución del valor de fitness a lo largo de las iteraciones, se ve que con un aumento en la tasa de mutación, el valor de la función de fitness aumenta más rápidamente a lo largo de las iteraciones. Por otro lado, si nos fijamos en la desviación típica, la variación de resultados no es estable y al aumentar la mutación este fenómeno se acentúa. De esto, se puede concluir que añadir un poco de mutación es beneficioso para encontrar mejores soluciones, pero no suficiente para encontrar la solución óptima, y genera dependencia en la aleatoriedad lo que supone que en cada entrenamiento podemos obtener un rendimiento diferente. De esta prueba se puede observar que la selección y ajuste del operador de cruce es crucial para la resolución del problema.

Experimentos con cruce morfológico y mutación 0 / 0.1 / 0.3

Para el siguiente conjunto de pruebas se utiliza el operador de cruce morfológico, que introduce mayor exploración a partir de la función de ϕ . Como en los experimentos anteriores, se empleó selección por torneo con $k=5$ y se evaluaron diferentes tasas de mutación.

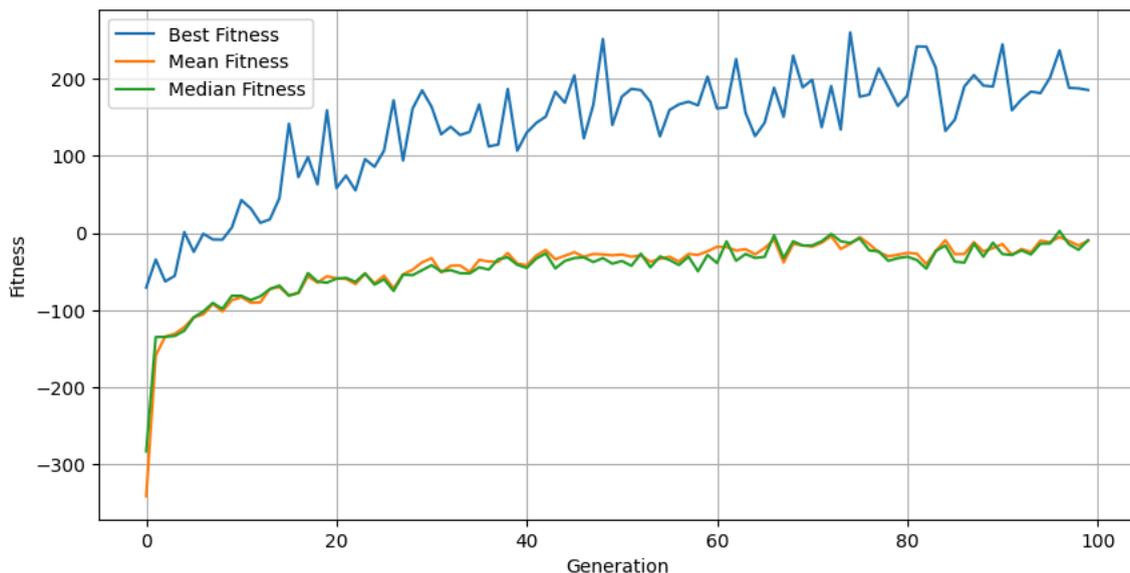


Figura 4.10: Gráfica de entrenamiento cruce morfológico sin mutación.

Resultados despues de 100 simulaciones:
 Recompensa Media : -131.65
 Desviacion Tipica : 128.28
 Inter. de conf. 95% : ±25.14

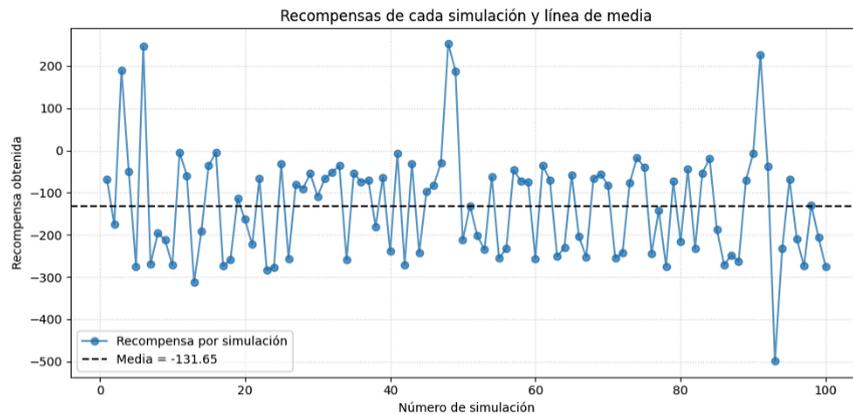


Figura 4.11: Resultado de simulaciones cruce morfológico sin mutación.

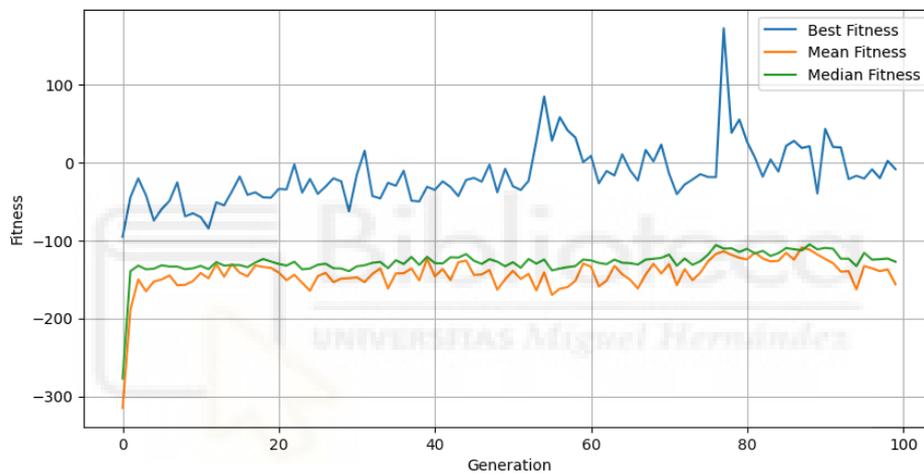


Figura 4.12: Gráfica de entrenamiento cruce morfológico con mutación 0.1.

Resultados despues de 100 simulaciones:
 Recompensa Media : -106.94
 Desviacion Tipica : 106.38
 Inter. de conf. 95% : ±20.85

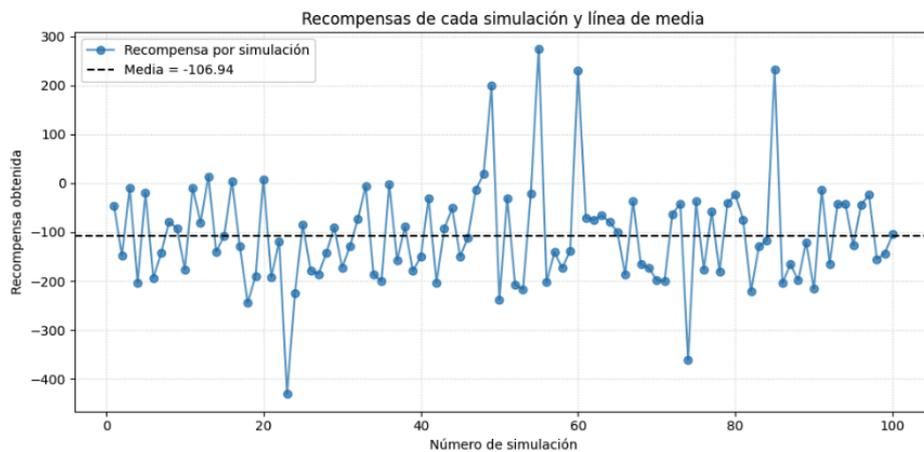


Figura 4.13: Resultado de simulaciones cruce morfológico con mutación 0.1.

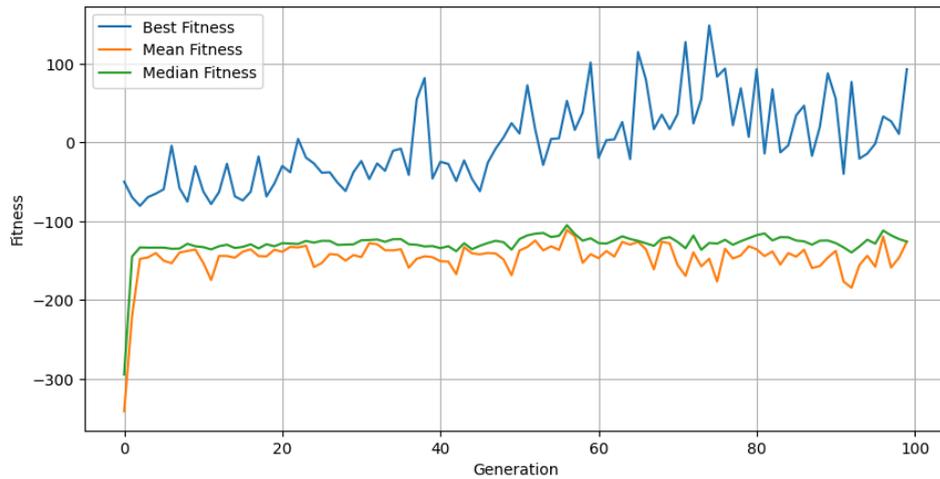


Figura 4.14: Gráfica de entrenamiento cruce morfológico con mutación 0.3.

Resultados despues de 100 simulaciones:
 Recompensa Media : -41.92
 Desviacion Típica : 107.39
 Inter. de conf. 95% : ±21.05

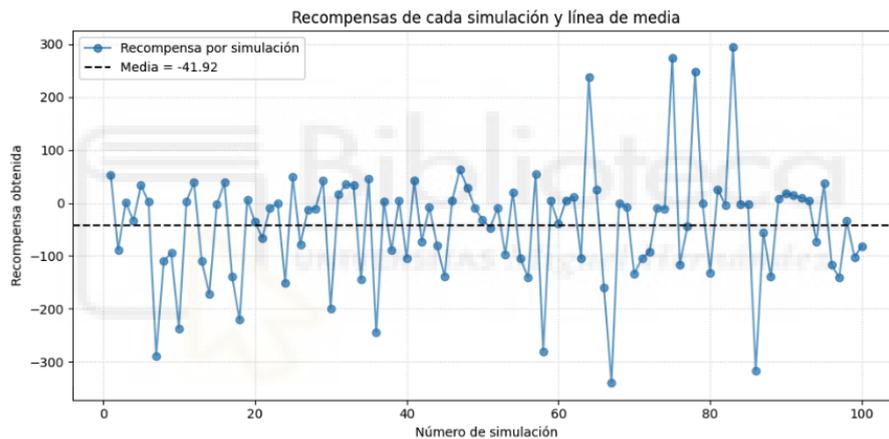


Figura 4.15: Resultado de simulación cruce morfológico con mutación 0.3.

Los resultados reflejan que este operador tampoco logra la convergencia del algoritmo, incluso con la ventaja exploratoria proporcionada por ϕ , como se puede observar en las figuras 4.10 a 4.15. La adición de mutación, debido a ser un elemento que añade aleatoriedad en estas pruebas, no fue suficiente para lograr la convergencia. Esto sugiere que, aunque la exploración que proporciona este operador es esencial, no es suficiente por sí sola para guiar el algoritmo hacia soluciones óptimas en este caso.

Experimentos con cruce BLX- α para α 0.3 / 0.5 / 0.7

Tras observar que los operadores de cruce plano y morfológico no lograron resultados satisfactorios debido a su dificultad para converger hacia soluciones óptimas, se evaluó el operador BLX- α .

Este operador tiene la ventaja de permitir un equilibrio entre exploración y explotación mediante el ajuste del parámetro α lo cual lo convierte en el operador con mayor posibilidad de conseguir que el algoritmo converja a una solución óptima para el problema. A continuación, se va a probar con distintos valores para obtener el mejor parámetro α para nuestro objetivo. Se va a probar con tres valores de α distintos, $\alpha = 0.3$ (más explotación), $\alpha = 0.5$ (equilibrio entre exploración y explotación), y $\alpha = 0.7$ (más exploración).

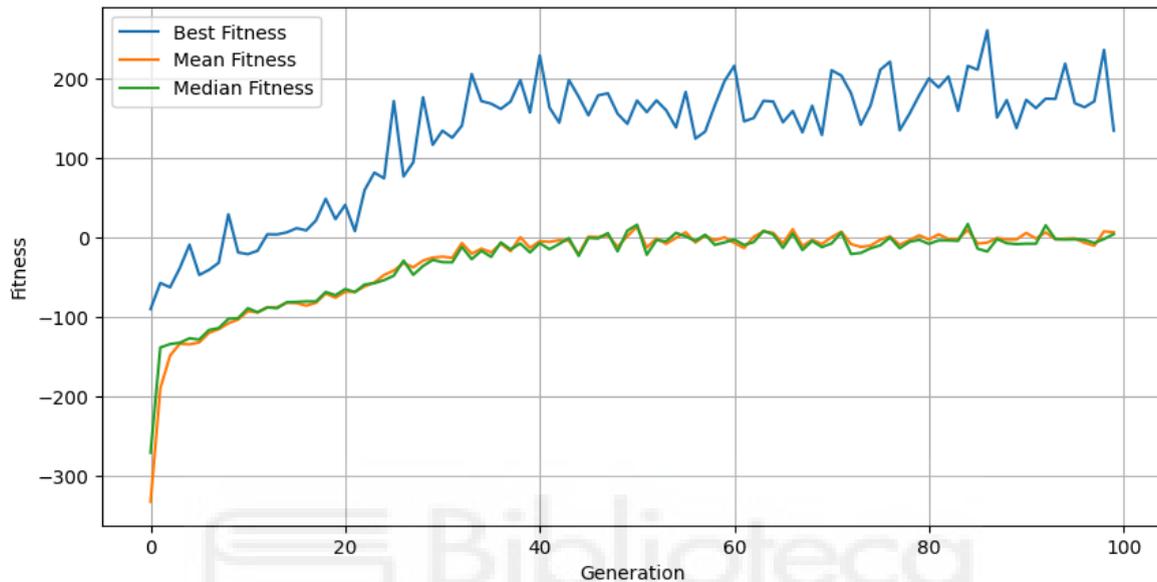


Figura 4.16: Gráfica de entrenamiento cruce BLX con $\alpha = 0.3$.

Resultados despues de 100 simulaciones:
 Recompensa Media : -16.24
 Desviacion Típica : 156.26
 Inter. de conf. 95% : ± 30.63

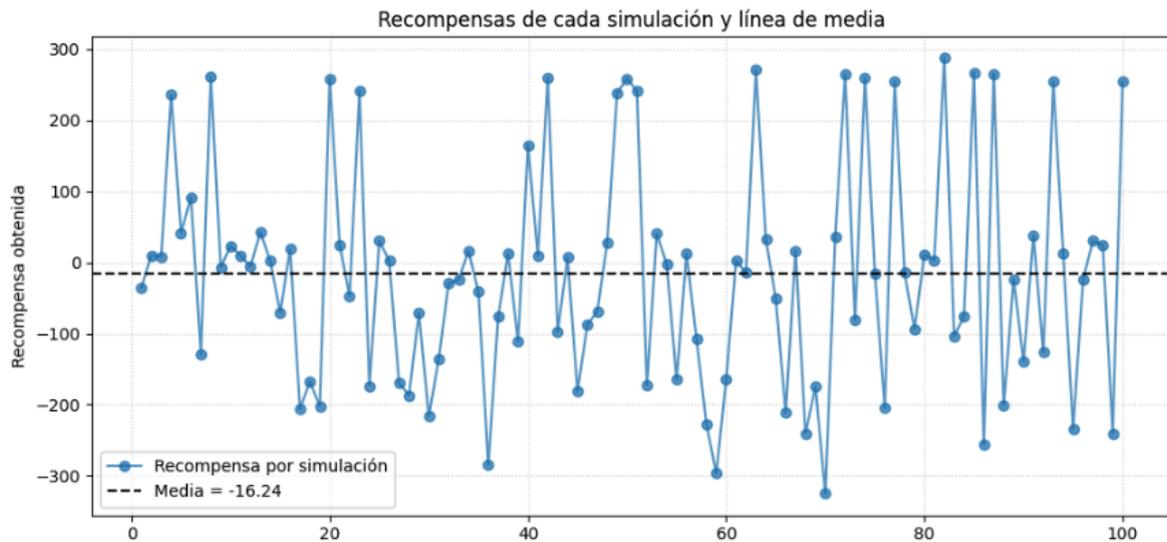


Figura 4.17: Resultado de simulación cruce BLX con $\alpha = 0.3$.

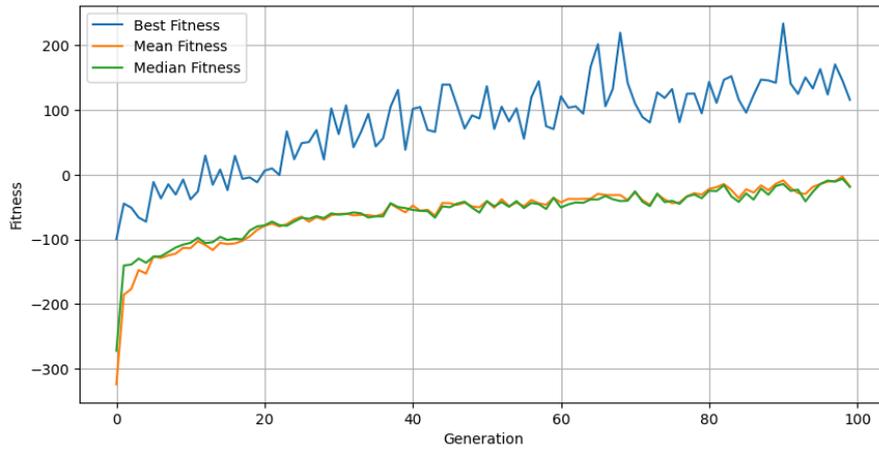


Figura 4.18: Gráfica de entrenamiento cruce BLX con $\alpha = 0.5$.

Resultados despues de 100 simulaciones:
 Recompensa Media : -7.30
 Desviación Típica : 130.26
 Inter. de conf. 95% : ± 25.53

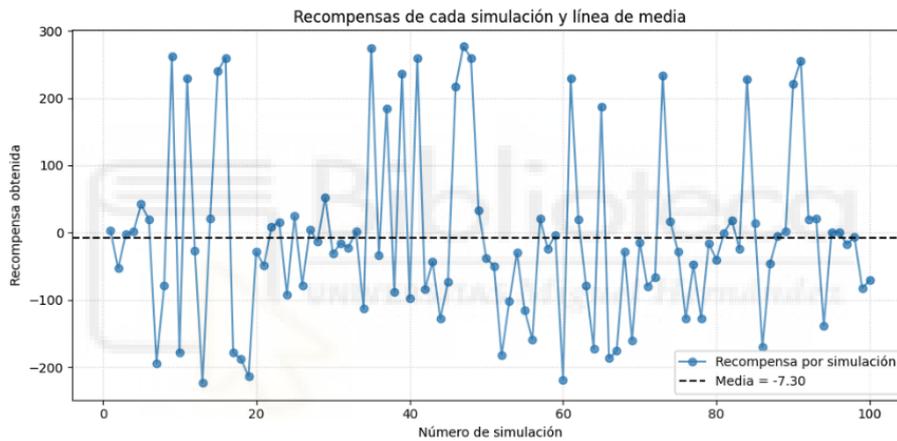


Figura 4.19: Resultado de simulación cruce BLX con $\alpha = 0.5$

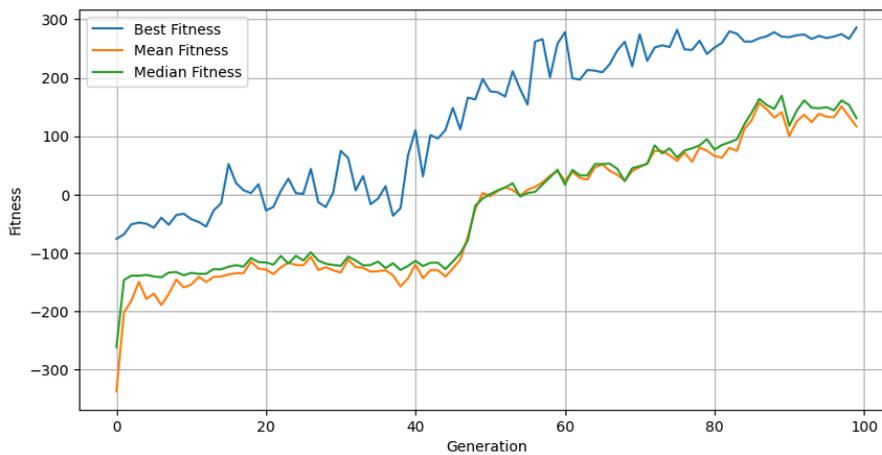


Figura 4.20: Gráfica de entrenamiento cruce BLX con $\alpha = 0.7$.

Resultados despues de 100 simulaciones:
 Recompensa Media : 221.73
 Desviacion Tipica : 87.94
 Inter. de conf. 95% : ± 17.24

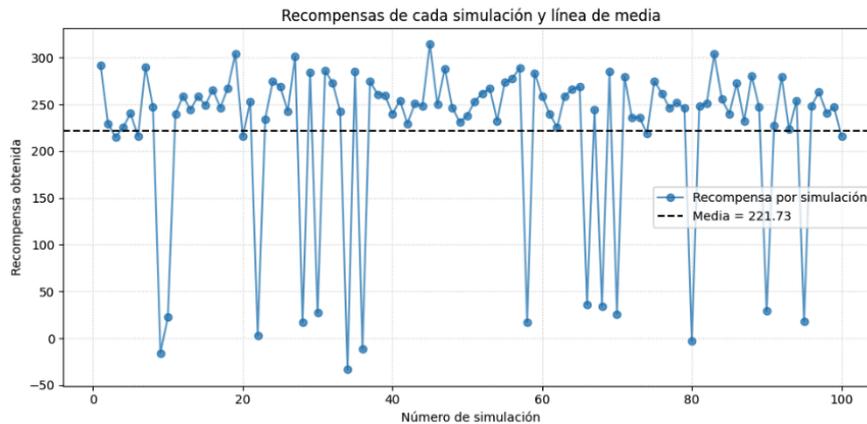


Figura 4.21: Resultado de simulación cruce BLX con $\alpha = 0.7$.

Para la Figura 4.16 se ha hecho el aprendizaje con un valor de $\alpha = 0.3$. Este valor genera una búsqueda muy limitada, priorizando la explotación sobre la exploración. Como resultado, el aprendizaje se estancó y no se observaron mejoras significativas a lo largo de las generaciones. Los cromosomas generados obtuvieron una recompensa media negativa, lo cual es consistente con la baja capacidad exploratoria esperada.

En la Figura 4.18 el valor utilizado para el aprendizaje es de $\alpha = 0.5$, un valor un poco más alto que el anterior que tendría que favorecer el balance entre exploración y explotación. Como se puede observar en la gráfica, este valor de α proporciona un aprendizaje constante a lo largo de las 100 generaciones y tiene una recompensa media cercana a 0 lo que no consigue la convergencia del algoritmo, lo que podría ayudar es introducir mutación para que se pueda salir de máximos locales.

La última prueba para el parámetro $\alpha = 0.7$ (ver figura 4.20) muestran una convergencia del algoritmo con muy buenos resultados en la simulación (ver figura 4.21), tiene más exploración que ayuda a escapar de los máximos locales y encontrar una solución óptima sin tener que recurrir a añadir la aleatoriedad introducida por la mutación. Gracias a la comparación entre los diferentes valores de α se llega a la conclusión de que el mejor parámetro para el cruce BLX- α es de 0.7, ya que todas las simulaciones realizadas con este valor han convergido hacia un resultado significativamente mejor que el inicial, simplemente con el algoritmo de cruce y selección con torneo.

Una vez identificado el mejor valor de α , se evaluará el impacto de introducir mutación a este mismo operador con el valor $\alpha=0.5$, debido a que ha obtenido su máximo valor con una cantidad menor de generaciones y es el más utilizado en la literatura. Se probará una mutación de $m = 0.1$.

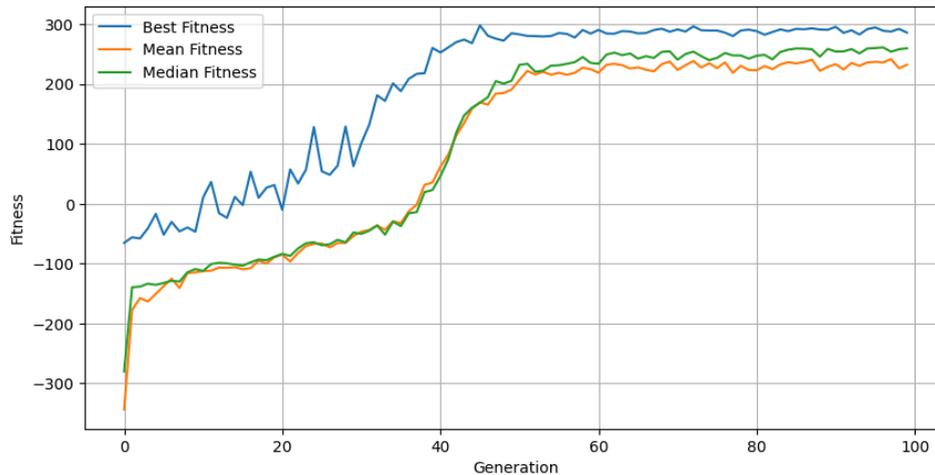


Figura 4.22: Gráfica de entrenamiento cruce BLX con $\alpha = 0.5$ y mutación 0.1.

Resultados despues de 100 simulaciones:
 Recompensa Media : 267.28
 Desviacion Tipica : 37.40
 Inter. de conf. 95% : ± 7.33

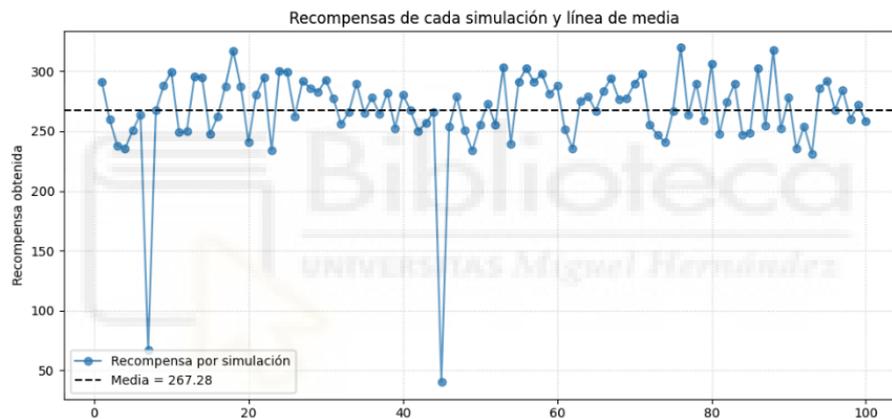


Figura 4.23: Resultado de simulación cruce BLX con $\alpha = 0.5$ y mutación 0.1.

Los resultados obtenidos muestran un muy buen desempeño con un rendimiento medio de los individuos que obtienen recompensas entre 200 y 250 con la ayuda de la mutación al introducir aleatoriedad para escapar de los óptimos locales. Por lo tanto el operador BLX- α es el único que proporciona una convergencia del algoritmo y las mejores versiones son $\alpha=0.5$ con mutación $m=0.1$ (para que no sea tan agresivo), y $\alpha=0.7$ sin mutación.

Métodos de selección

Para realizar las distintas pruebas de los distintos métodos de selección se va a utilizar como operador de cruce BLX- α , ya que es el operador que mejores resultados ha proporcionado, se elige el valor $\alpha = 0.7$ sin mutación para evitar los efectos desfavorables que introduce la aleatoriedad. Se van a realizar pruebas con el método de torneo variando el número de participantes k , se va a probar con 3 y 7 individuos.

Por otro lado, también se van a realizar pruebas de selección por ruleta con elitismo variando el número de individuos ‘e’ que se mantendrán en la nueva generación, concretamente $e=15$ y $e=30$. En definitiva, se va a combinar el método de cruce BLX- α (con $\alpha=0.7$) con los siguientes operadores de selección:

- Selección por torneo con $k=3$, $k=7$
- Selección por ruleta con elitismo con $e=15$, $e=30$

Experimentos con selección por torneo para $k=3/7$

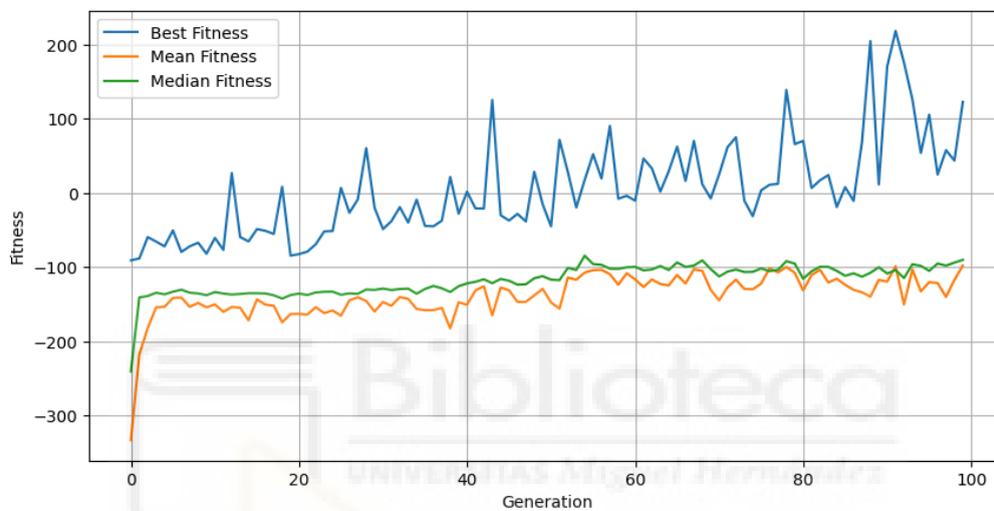


Figura 4.24: Gráfica de entrenamiento selección torneo con $k=3$.

Resultados después de 100 simulaciones:
 Recompensa Media : 9.58
 Desviación Típica : 151.59
 Inter. de conf. 95% : ± 29.71

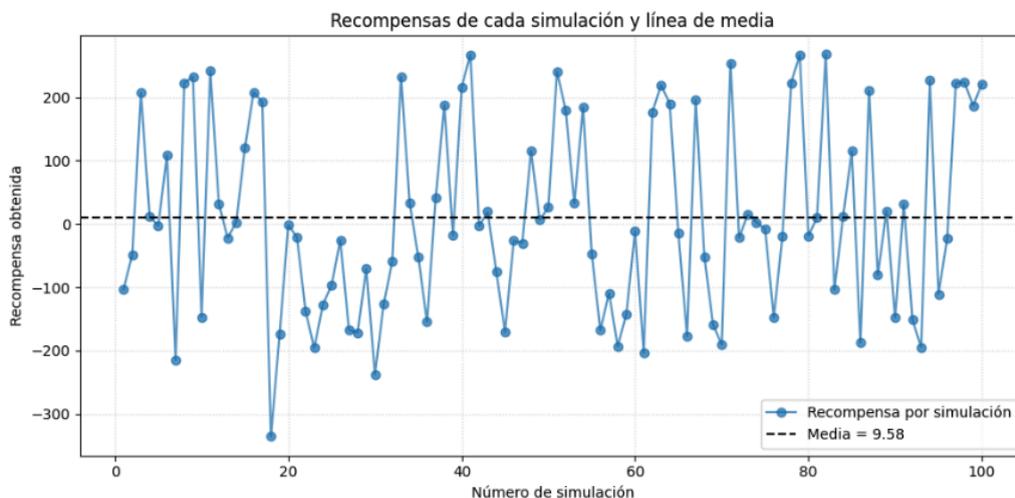


Figura 4.25: Resultado de simulación selección torneo con $k=3$.

Las gráficas 4.24 y 4.25 muestran como con un torneo de tan pocos individuos ($k=3$) no se consigue un buen resultado.

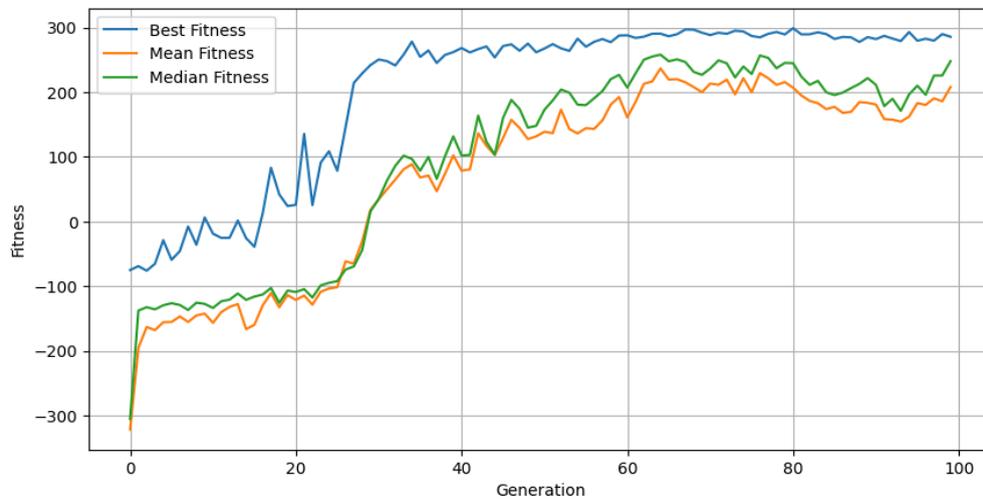


Figura 4.26: Gráfica de entrenamiento selección torneo con $k=7$.

Resultados despues de 100 simulaciones:
 Recompensa Media : 265.21
 Desviacion Tipica : 52.18
 Inter. de conf. 95% : ± 10.23

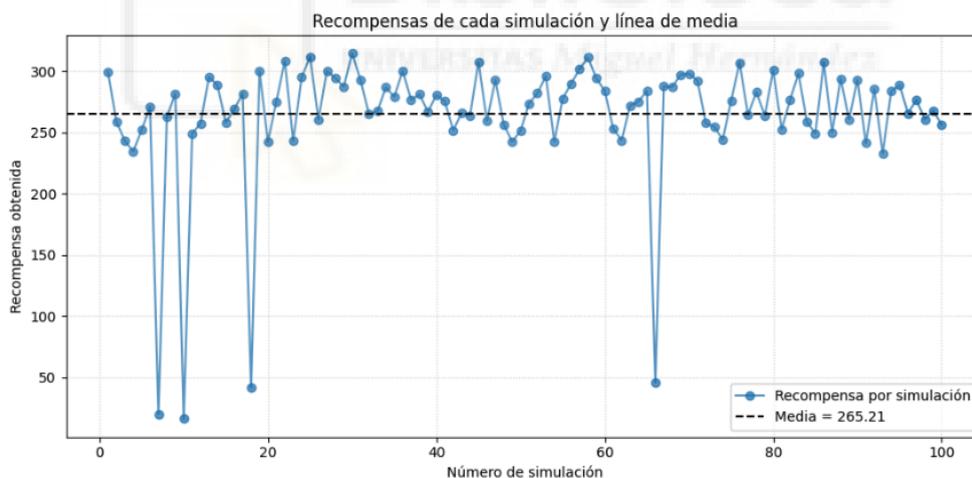


Figura 4.27: Resultado de simulación selección torneo con $k=7$.

En los resultados tras probar distintos valores de individuos que participan en la selección por torneo, se observa que al utilizar un valor bajo con 3 individuos no se consigue que la población converja hacia resultados óptimos con el paso de las generaciones, mientras que cuando se utilizan valores de 5 (ver experimentos anteriores) y 7 individuos, el algoritmo converge. En resumen, este método de selección si nos proporciona buenos resultados a partir de un valor $k \geq 5$ individuos para realizar el torneo, pero a medida que aumentamos este valor se vuelve computacionalmente más demandante.

Experimentos con selección por ruleta con elitismo para $e = 15 / 30$

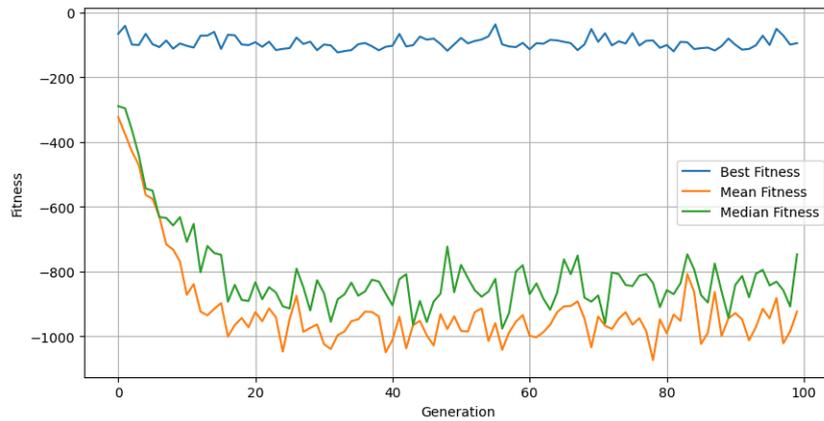


Figura 4.28: Gráfica de entrenamiento selección ruleta con $e=15$.

Resultados después de 100 simulaciones:
Recompensa Media : -121.61
Desviación Típica : 52.88
Inter. de conf. 95% : ± 10.36

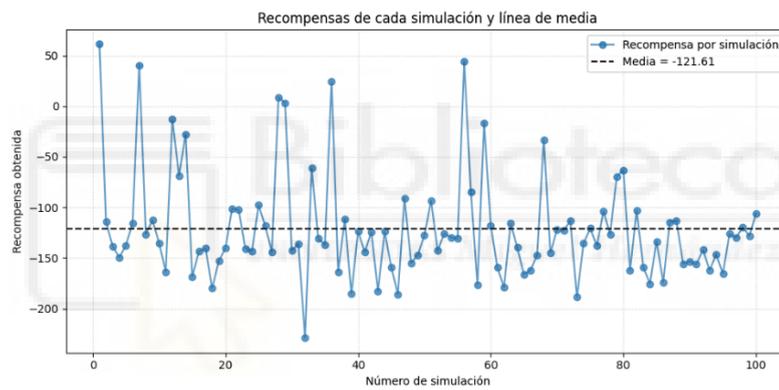


Figura 4.29: Resultado de simulación selección ruleta con $e=15$.

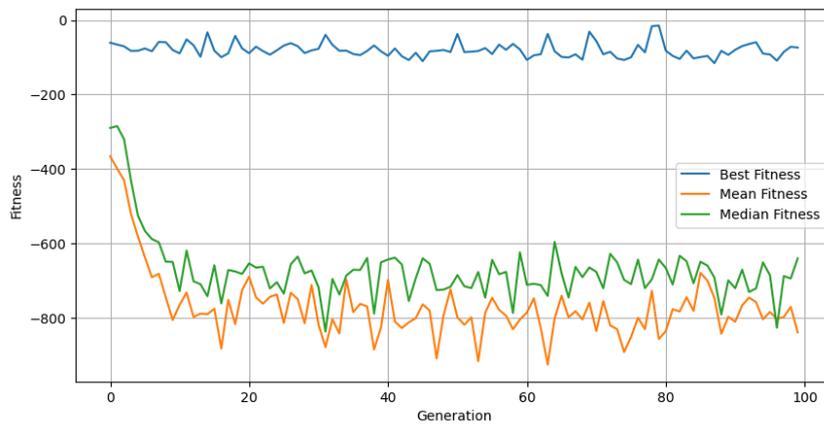


Figura 4.30: Gráfica de entrenamiento selección ruleta con $e=30$.

Resultados despues de 100 simulaciones:
Recompensa Media : -81.02
Desviacion Típica : 81.96
Inter. de conf. 95% : ±16.06

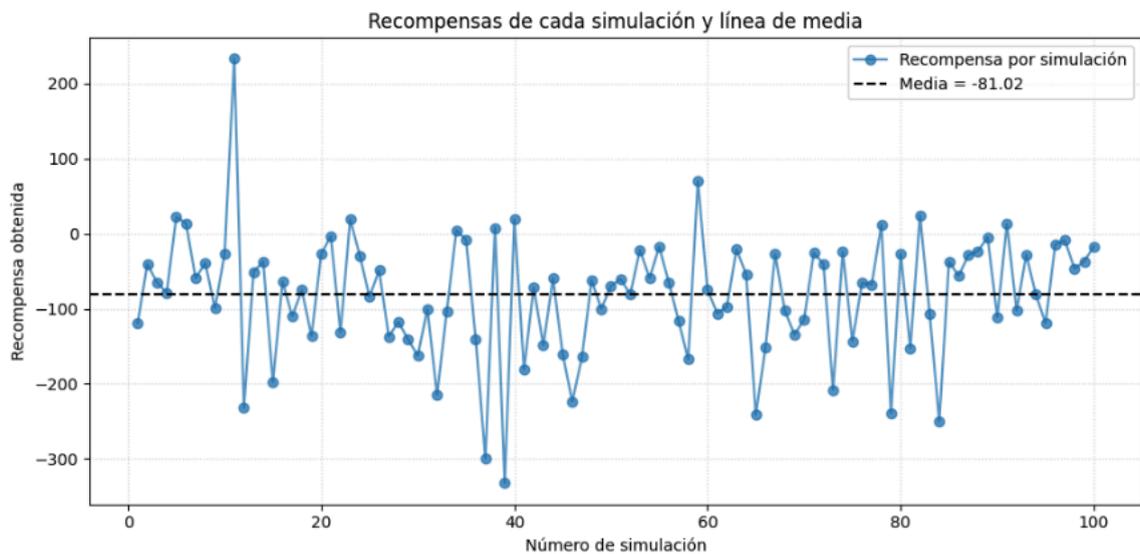


Figura 4.31: Resultado de simulación selección ruleta con $e=30$.

Los resultados obtenidos con la utilización del operador de selección por ruleta con elitismo muestran que no se consigue que con el paso de las generaciones el algoritmo converja, como se observa en las figuras 4.28 y 4.30. Esto se debe a que los individuos obtenidos inicialmente no son lo suficientemente buenos, y estos son los que se propagan entre generaciones, propagando por tanto su mal rendimiento.

Valoración de resultados

Tras la realización de los experimentos realizados con diferentes configuraciones de operadores, tanto de cruce como de selección, y considerando diferentes tasas de mutación, se puede concluir que una buena combinación de estos es crucial para que el algoritmo converja a resultados óptimos, es decir, un buen aterrizaje en el entorno Lunar Lander. También se observa que introducir un poco de mutación puede ayudar a obtener un mejor rendimiento.

La mejor combinación de operadores obtenida está formada por la elección del método de selección de torneo con valores $k=5$ y $k=7$, siendo mejor elección $k=5$, debido a que es menos demandante a nivel computacional y proporciona un resultado similar. Para el cruce, proporciona los mejores resultados el método BLX- α con valor de $\alpha = 0.7$ sin mutación, que presenta una mayor exploración que ayuda a la convergencia, también podemos destacar el rendimiento de $\alpha = 0.5$ cuando se le aplica mutación, lo que permite a los individuos escapar de óptimos locales obteniendo un buen balance entre exploración y explotación a lo largo del entrenamiento.

4.4.- Aprendizaje por Refuerzo

4.4.1.- Codificación

La arquitectura de la red principal del algoritmo Deep Q-learning se constituye de los siguientes elementos:

- Entradas: el vector de observaciones del entorno (dimensión 8).
- Salidas: el conjunto de acciones posibles (dimensión 4).
- Capas ocultas: Número de capas y neuronas ocultas para aprender representaciones no lineales.

4.4.2.- Flujo del algoritmo

El enfoque implementado en este apartado consiste en aplicar el algoritmo Deep Q-Learning, el cual es una extensión del algoritmo Q-Learning que combina el aprendizaje por refuerzo con redes neuronales profundas para realizar la aproximación de los valores de 'Q'. Este algoritmo está compuesto por diferentes componentes claves que se describen a continuación:

- Red Neuronal 'Q': Se trata de la red neuronal que recibe un estado s y produce todos los valores 'Q' para todas las acciones disponibles.
- Memoria de experiencias: Almacena las transacciones (estado, acción, recompensa, siguiente estado) en un búfer. Durante el entrenamiento, se muestrean aleatoriamente pequeños lotes (mini-batches) de estas transiciones para actualizar la red.
- Red objetivo: Esta red es una copia de la red 'Q', pero sus pesos se actualizan de forma menos frecuente. Durante el cálculo de los valores 'Q', se emplea esta red en lugar de la red principal, para reducir la variación en las actualizaciones.
- Función de pérdida: Mide la diferencia entre el valor 'Q' estimado y el valor 'Q' objetivo, calculado mediante la ecuación de Bellman.
- Epsilon-Greedy: Política para regular la exploración y explotación durante el entrenamiento.

4.4.3.- Implementación

Hiperparámetros

Los parámetros utilizados para el entrenamiento del algoritmo para obtener un buen equilibrio entre explotación, exploración y eficiencia computacional.

Tabla 4.1: Hiperparámetros para Entrenamiento de Agentes de Aprendizaje por Refuerzo

Hiperparámetro	Intervalo Típico	Descripción
LEARNING_RATE	$[1^{-4}, 1^{-2}]$	Tasa de aprendizaje de la red neuronal controla la actualización de los pesos.
BATCH_SIZE	[32, 256]	Tamaño de muestras para cada iteración.
GAMMA	[0.95, 0.999]	Determina la importancia de las recompensas futuras.
MEMORY_SIZE	$[10^4, 10^6]$	Memoria máxima para almacenar transiciones pasadas.
EXPLORATION_RATE_MAX	[0.9, 1]	Valor inicial de la política epsilon greedy.
EXPLORATION_RATE_MIN	[0.01, 0.1]	Valor mínimo al que puede decrecer la política epsilon greedy.
MAX_EPISODES_FOR_TRAINING	[500, 2000]	Número de episodios para entrenar al agente.
TRAINING_GOAL	[200, ∞]	Recompensa media para considerar que el algoritmo ha convergido.
EPISODES_TO_CHECK_TRAINING_GOAL	[5, 100]	Número de episodios cada vez que se comprueba el rendimiento del agente.
EPISODES_TO_EVALUATE_MODEL_PERFORMANCE	[10, 100]	Número de episodios donde se prueba el agente al acabar el entrenamiento.

Clase Deep Q-Network

Esta clase implementa el agente que aprende los valores de la función Q a partir de una red neuronal.

Algoritmo 4.8: DQN (Deep Q-Network)

```
1 # Clase Deep Q-Network
2 class DQN:
3
4 def __init__(self, number_of_observations, number_of_actions, modelo):
5     # Initialize variables and create neural model
6     self.number_of_actions = number_of_actions
7     self.number_of_observations = number_of_observations
8     self.scores = []
9     self.memory = ReplayMemory(number_of_observations)
10    self.model = modelo
11    self.model.compile
12    (optimizer=keras.optimizers.Adam(learning_rate=LEARNING_RATE), loss='mse')
13
```

```

14 def remember(self, state, action, reward, next_state, terminal_state):
15     # Store a tuple (s, a, r, s') for experience replay
16     state = np.reshape(state, [1, self.number_of_observations])
17     next_state = np.reshape(next_state, [1, self.number_of_observations])
18     self.memory.store_transition(state, action, reward, next_state, terminal_state)
19
20 def select(self, state, exploration_rate):
21     # Generar una acción para un estado dado, utilizando la política
22     # epsilon-greedy
23     if np.random.rand() < exploration_rate:
24         return random.randrange(self.number_of_actions)
25     else:
26         state = np.reshape(state, [1, self.number_of_observations])
27         q_values = self.model(state).numpy()
28     return np.argmax(q_values[0])
29
30 def select_greedy_policy(self, state):
31     state = np.reshape(state, [1, self.number_of_observations])
32     q_values = self.model(state).numpy()
33     return np.argmax(q_values[0])
34
35 def learn(self):
36     # Aprende el valor Q utilizando una muestra de ejemplos
37     # de la memoria de repetición
38     if self.memory.current_size < BATCH_SIZE: return
39
40     states, actions, rewards, next_states, terminal_states = \
41     self.memory.sample_memory(BATCH_SIZE)
42
43     q_targets = self.model(states).numpy()
44
45     if self.use_target_network:
46         q_next_states = self.target_model(next_states).numpy()
47     else:
48         q_next_states = self.model(next_states).numpy()
49
50     for i in range(BATCH_SIZE):
51         if (terminal_states[i]):
52             q_targets[i][actions[i]] = rewards[i]
53         else:
54             q_targets[i][actions[i]] = rewards[i] + GAMMA *
55             np.max(q_next_states[i])
56     self.model.train_on_batch(states, q_targets)
57
58 def update_target_network(self):
59     self.target_model.set_weights(self.model.get_weights())
60
61 def soft_update_target_network(self, tau=0.005):
62     model_weights = self.model.get_weights()
63     target_weights = self.target_model.get_weights()
64     new_weights = []
65
66     for model_w, target_w in zip(model_weights, target_weights):
67         updated_w = tau * model_w + (1.0 - tau) * target_w
68         new_weights.append(updated_w)
69
70     self.target_model.set_weights(new_weights)

```

```

71
72 def add_score(self, score):
73     # Añadir la puntuación obtenida a una lista
74     self.scores.append(score)
75
76 def delete_scores(self):
77     # Elimina las puntuaciones
78     self.scores = []
79
80 def average_score(self, number_of_episodes):
81     # Calcular la puntuación media de los últimos episodios
82
83     index = len(agent.scores) - number_of_episodes
84     return np.mean(self.scores[max(0, index):(len(agent.scores))])

```

En definitiva, el agente DQN observa el entorno, decide las acciones a realizar, recuerda lo que sucede y luego usa esos recuerdos para aprender y mejorar sus decisiones durante el proceso de entrenamiento.

Explicación de los métodos:

- **__init__**: Es el constructor de la clase donde se inicializa el agente, se encarga de establecer sus componentes básicos, define el número de acciones y observaciones que puede manejar, crea la memoria de las repeticiones y la red neuronal utilizada.
- **remember**: Almacena la información de la transición que se ha ejecutado.
- **select**: Genera la próxima acción para el agente.
- **select_greedy_policy**: Selecciona la acción más óptima para el estado actual.
- **learn**: Es el proceso de entrenamiento del agente. Donde el agente toma las experiencias pasadas para actualizar su red neuronal y mejore su valor a lo largo del entrenamiento.
- **update_target_network**: Actualiza los pesos de la red target copiando los pesos de la red principal.
- **soft_update_target_network**: Actualiza los pesos de la red target mezclando los pesos de la red principal y de la red target.
- **add_score**: Guarda la puntuación obtenida en un episodio.
- **delete_score**: Elimina el historial de las puntuaciones guardadas.
- **average_score**: Calcula la puntuación promedio de un número determinado de episodios.

Clase Replay Memory

Esta clase se encarga de almacenar y gestionar las experiencias del agente para que posteriormente las pueda utilizar para aprender de ellas de forma eficiente, es decir, almacena las transiciones del agente.

Algoritmo 4.9: ReplayMemory

```
1 class ReplayMemory:
2
3     def __init__(self, number_of_observations):
4         # Crear memoria de repetición
5         self.states = np.zeros((MEMORY_SIZE, number_of_observations))
6         self.states_next = np.zeros((MEMORY_SIZE, number_of_observations))
7         self.actions = np.zeros(MEMORY_SIZE, dtype=np.int32)
8         self.rewards = np.zeros(MEMORY_SIZE)
9         self.terminal_states = np.zeros(MEMORY_SIZE, dtype=bool)
10        self.current_size=0
11
12    def store_transition(self, state, action, reward, state_next, term_st):
13        # Guarda las transiciones (s,a,r,s') en la memoria de repetición
14        i = self.current_size
15        self.states[i] = state
16        self.states_next[i] = state_next
17        self.actions[i] = action
18        self.rewards[i] = reward
19        self.terminal_states[i] = term_st
20        self.current_size = i + 1
21
22    def sample_memory(self, batch_size):
23        # Generar una muestra de transiciones a partir de la memoria
24        # de repetición
25        batch = np.random.choice(self.current_size, batch_size)
26        states = self.states[batch]
27        states_next = self.states_next[batch]
28        rewards = self.rewards[batch]
29        actions = self.actions[batch]
30        terminal_states = self.terminal_states[batch]
31        return states, actions, rewards, states_next, terminal_states
```

Explicación de los métodos:

- **__init__**: Es el constructor de la clase el cual es el encargado de inicializar la memoria.
- **store_transition**: Almacena una transición (s,a,r,s').
- **sample_memory**: Devuelve una transacción de forma aleatoria de las que hay almacenadas en la memoria.

Entrenamiento del modelo

Este fragmento de código realiza el proceso completo para entrenar uno o varios agentes en un entorno.

Algoritmo 4.10: Entrenamiento del modelo

```
1 resultados = []
2 tiempos_entrenamiento = []
3 agentes_entrenados = []
4
```

```

5  for modelo in modelos:
6      # Definición de los hiperparametros necesarios para el entrenamiento
7      LEARNING_RATE = 0.001
8      BATCH_SIZE = 64
9      # ...
10     # ESTABLECER RESTO DE PARÁMETROS DE LA TABLA 4.2
11
12     agent = DQN(number_of_observations, number_of_actions, modelo)
13     episode = 0
14     start_time = time.perf_counter()
15     total_steps = 1
16     exploration_rate = EXPLORATION_RATE_MAX
17     goal_reached = False
18
19     episodios = []
20     scores = []
21     while (episode < MAX_EPISODES_FOR_TRAINING) and not(goal_reached):
22         # Bucle de entrenamiento
23         episode += 1
24         score = 0
25         state, info = environment.reset()
26         end_episode = False
27
28         while not(end_episode):
29             # Seleccionar una acción para el estado actual
30             action = agent.select(state, exploration_rate)
31
32             # Ejecutar acción en el entorno y avanzar en el problema con la
33             # acción elegida, step devuelve el próximo estado, recompensa, si
34             # ha acabado el episodio
35             state_next, reward, terminal_state, truncated, info =
36             environment.step(action)
37
38             # Almacenar en memoria la transición (s,a,r,s') y actualizar
39             # q-table con el nuevo estado y recompensa obtenidos
40             agent.remember(state, action, reward, state_next, terminal_state)
41
42             score += reward # Se actualiza el score del episodio
43
44             # Aprender usando el batch de experiencias almacenadas en memoria.
45             agent.learn()
46
47             # Detectar si se ha finalizado el episodio
48             if terminal_state or truncated:
49                 end_episode = True
50                 agent.add_score(score)
51                 average_score = \
52                 agent.average_score(EPISODES_TO_CHECK_TRAINING_GOAL)
53                 if average_score >= TRAINING_GOAL: goal_reached = True
54                 print("Episode {0:>3}: ".format(episode), end = '')
55                 print("score {0:>3} ".format(math.trunc(score)), end = '')
56                 print("(exploration rate: %.3f," % exploration_rate,end = '')
57                 print("score: {0:>3},".format(round(average_score)),end = '')
58                 print("transitions: " + str(agent.memory.current_size) +")")
59                 episodios.append(episode)
60                 scores.append(score)
61             else:

```

```

62         state = state_next
63         total_steps += 1
64
65     # Disminuir la tasa de exploración para comenzar con alta exploración,
66     # y se irá convergiendo hacia una solución enfocándose progresivamente
67     # en la explotación.
68     exploration_rate *= EXPLORATION_RATE_DECAY
69     exploration_rate = max(EXPLORATION_RATE_MIN, exploration_rate)

```

Explicación por pasos:

- **Bucle de Entrenamiento:** Para preparar y entrenar distintas configuraciones de redes neuronales.
- **Configuración y preparación por Modelo:** Se definen todos los hiperparámetros necesarios, se crea el agente DQN y las estructuras necesarias.
- **Bucle de Episodios de entrenamiento:** Ejecutar múltiples episodios para que se realice el aprendizaje del agente.
- **Bucle de pasos dentro de un Episodio:** Simulación de la interacción paso a paso del agente con el entorno dentro de un episodio, donde selecciona acciones, observa resultados, almacena experiencias y aprende.
- **Ajuste de la tasa de exploración:** Se reduce gradualmente la aleatoriedad en la toma de decisiones del agente.

4.4.4.- Experimentación Aprendizaje por Refuerzo

Para la experimentación con la implementación del algoritmo DQL se van a realizar diversas pruebas con distintas configuraciones de redes neuronales. Todas las redes comparten una estructura fundamental:

- Una capa de entrada que recibe las ocho observaciones del entorno.
- Dos capas ocultas densas con activación ReLU con 128, 64 y 32 neuronas por capa.
- Una capa de salida con cuatro neuronas, correspondiente a las acciones.

La estructura de las tres redes neuronales propuestas se debe a la naturaleza del problema de Lunar Lander, para que tengan una capacidad adecuada para aprender la función de Q evitando sobreajuste en el proceso de entrenamiento, también el uso de dos capas ocultas es un estándar habitual en problemas de DQN. Definidas estas redes neuronales, se van a combinar con las siguientes implementaciones para los experimentos:

- Sin el uso de “*target network*” (ver definición a continuación).
- Usando target network actualizando cada episodio (“*hard update*”)
- Usando target network actualizando cada 1000 pasos (“*hard update*”)
- Usando target network con actualización “*soft update*”

Un red “*target network*” es una copia de la red principal cuyos pesos se actualizan cada cierto número de iteraciones, de forma que el valor objetivo que se utiliza para actualizar la red principal se calcula con esta red target, haciendo que el entrenamiento sea más suave, con menos oscilaciones. La estrategia de actualización “*soft update*” consiste en actualizar los pesos de la target network gradualmente, haciendo una mezcla entre sus propios pesos y los de la red principal, por el contrario, la estrategia “*hard update*” hace un volcado completo de todos los pesos de la red principal en la target cada cierto número de episodios o pasos.

Los hiperparámetros utilizados en la experimentación se han seleccionado siguiendo los estándares del estado del arte, y basados en los resultados obtenidos en la experimentación previa. Los valores definidos para el entrenamiento de los agentes son los siguientes:

Tabla 4.2: Valores para experimentación DQN.

Parámetro	Valor
LEARNING_RATE	0.001
BATCH_SIZE	64
GAMMA	0.99
MEMORY_SIZE	1000000
EXPLORATION_RATE_MAX	1
EXPLORATION_RATE_MIN	0.01
MAX_EPISODES_FOR_TRAINING	1000
TRAINING_GOAL	230
EPISODES_TO_CHECK_TRAINING_GOAL	100
EPISODES_TO_EVALUATE_MODEL_PERFORMANCE	100

Para garantizar la reproducibilidad de la experimentación se han generado semillas para que el entrenamiento de los modelos y su testeo se enfrenten a las mismas condiciones. Para que no se alargue en exceso el proceso de entrenamiento se ha establecido como condición de parada obtener una media de 230 de puntuación (fitness) en los últimos 100 episodios. Finalizado el entrenamiento se ha realizado un test de los agentes con diferentes semillas para probar la generalización de los mismos, realizando simulaciones durante 100 episodios de evaluación por semilla.

Experimento sin Target Network

Para el primer experimento se ha realizado el entrenamiento de tres agentes con los diferentes tamaños de red neuronal citados (128, 64 y 32 neuronas por capa), sin el uso de

la target network, se ha utilizado la configuración más básica del algoritmo DQN. Los resultados obtenidos se muestran en las figuras 4.32, 4.33 y 4.34.

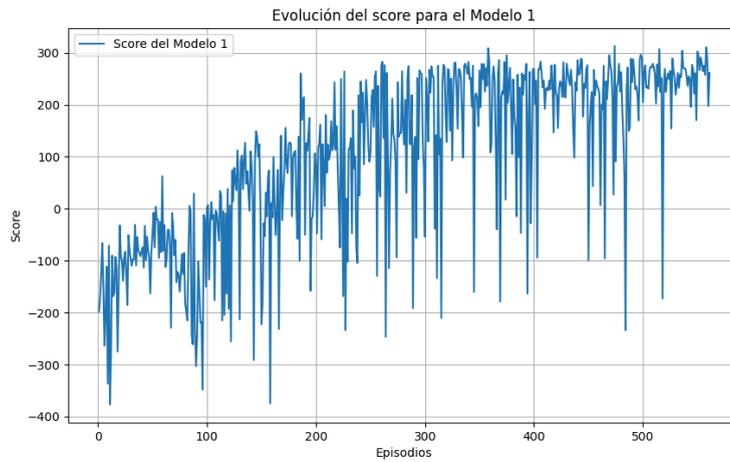


Figura 4.32: Entrenamiento red 128 sin Target Network.

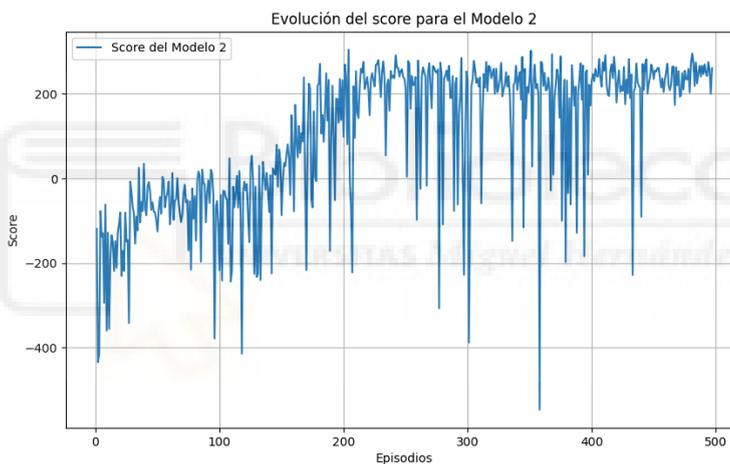


Figura 4.33: Entrenamiento red 64 sin Target Network.

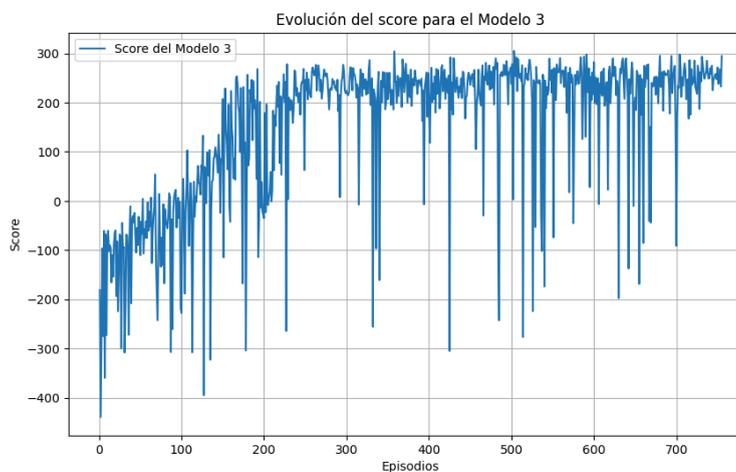


Figura 4.34: Entrenamiento red 32 sin Target Network.

Como podemos observar en el entrenamiento sin el uso de target network las tres redes convergen hacia resultados superiores a puntuaciones de más de 200, lo que indica que el agente ha superado el objetivo, sin embargo se aprecia que el aprendizaje a lo largo de los episodios muestra gran inestabilidad. También se observa que para conseguir la condición de parada, los modelos de mayor tamaño necesitan menos episodios para conseguirlo.

Experimento con Target Network actualizando cada episodio (“hard update”)

Con el uso de la “target network” se busca que el entrenamiento del agente sea más estable y no produzca grandes oscilaciones. Para este experimento se ha añadido la target network, actualizado sus pesos cada episodio (“hard update”), para probar si la actualización frecuente de la target network proporciona la estabilidad buscada o inestabilidad.

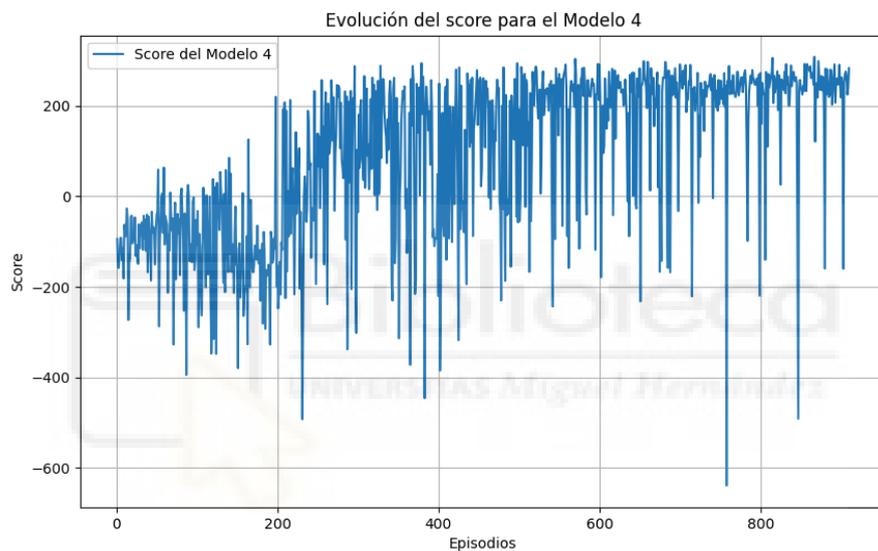


Figura 4.35: Entrenamiento red 128 con Target Network actualización cada episodio.

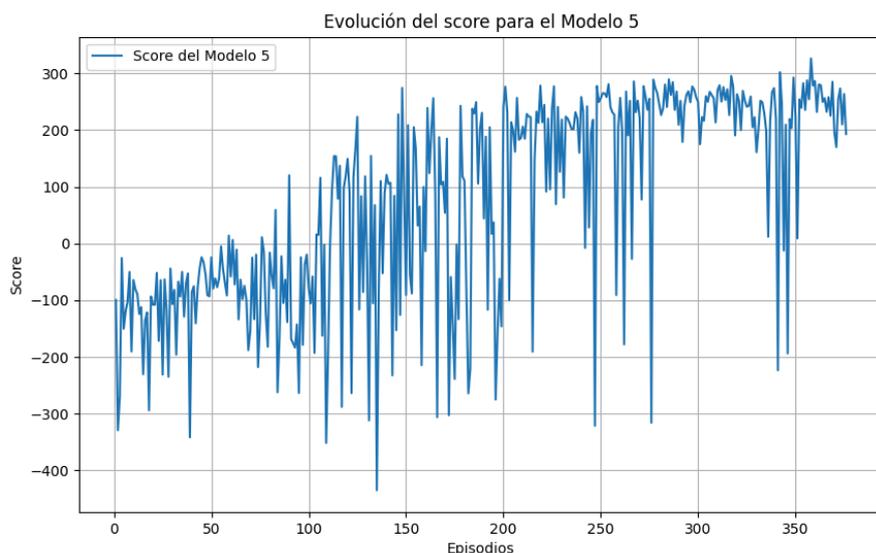


Figura 4.36: Entrenamiento red 64 con Target Network actualización cada episodio.

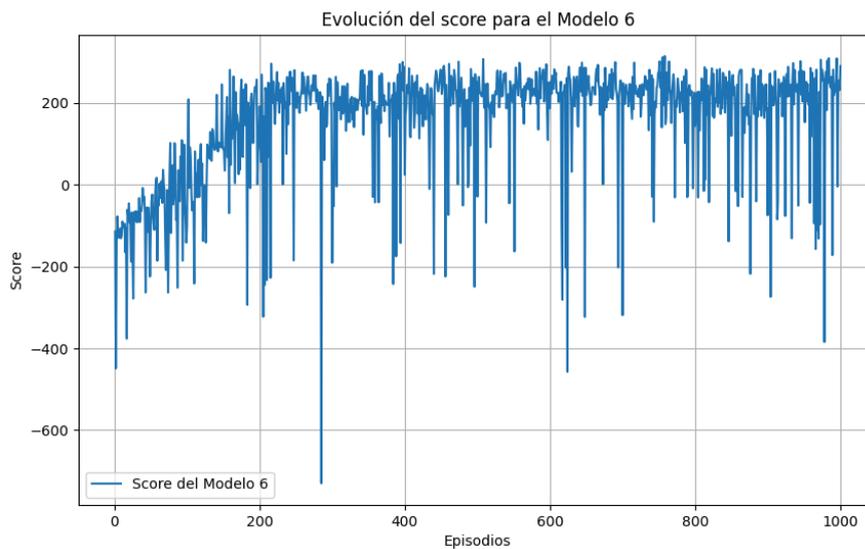


Figura 4.37: Entrenamiento red 32 con Target Network actualización cada episodio.

Para esta configuración también se obtienen resultados con más de 200 de puntuación durante el entrenamiento pero se sigue observando una gran inestabilidad en los modelos (ver figuras 4.35, 4.36 y 4.37).

Experimento con Target Network actualizando cada 100 pasos (“hard update”)

Para esta configuración se ha optado por realizar una actualización de la target network cada 1000 pasos, esta es una configuración clásica en este tipo de problemas.

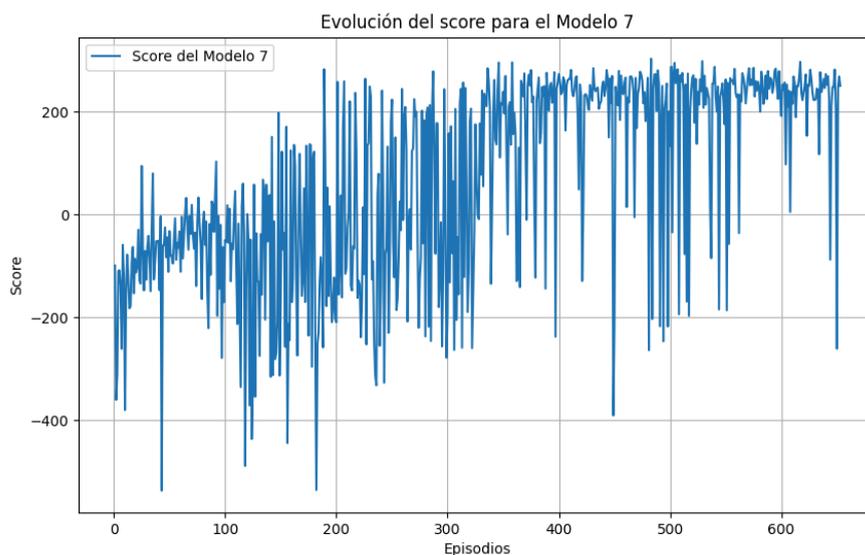


Figura 4.38: Entrenamiento red 128 con Target Network actualización cada 1000 pasos.

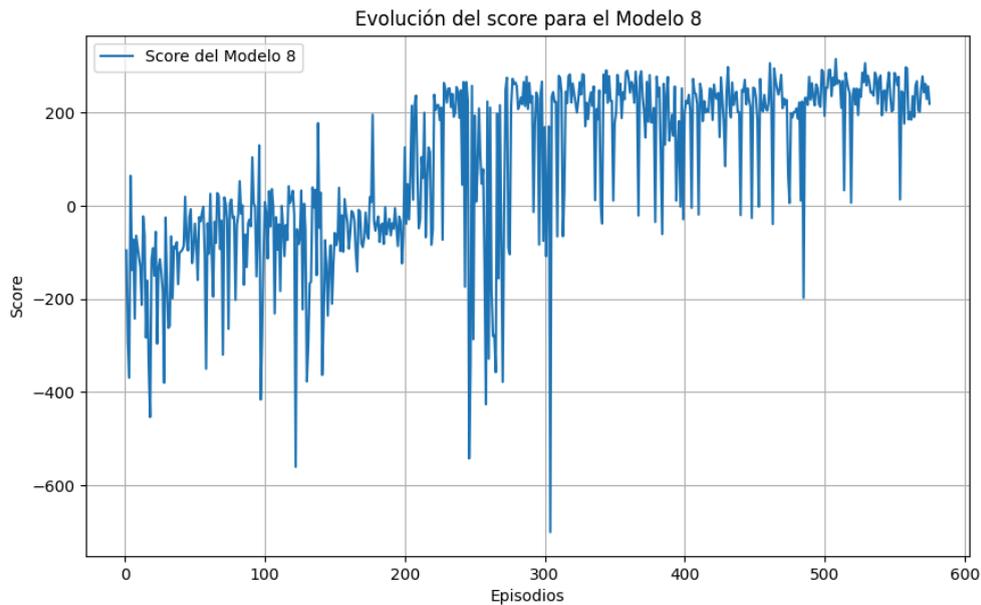


Figura 4.39: Entrenamiento red 64 con Target Network actualización cada 1000 pasos.

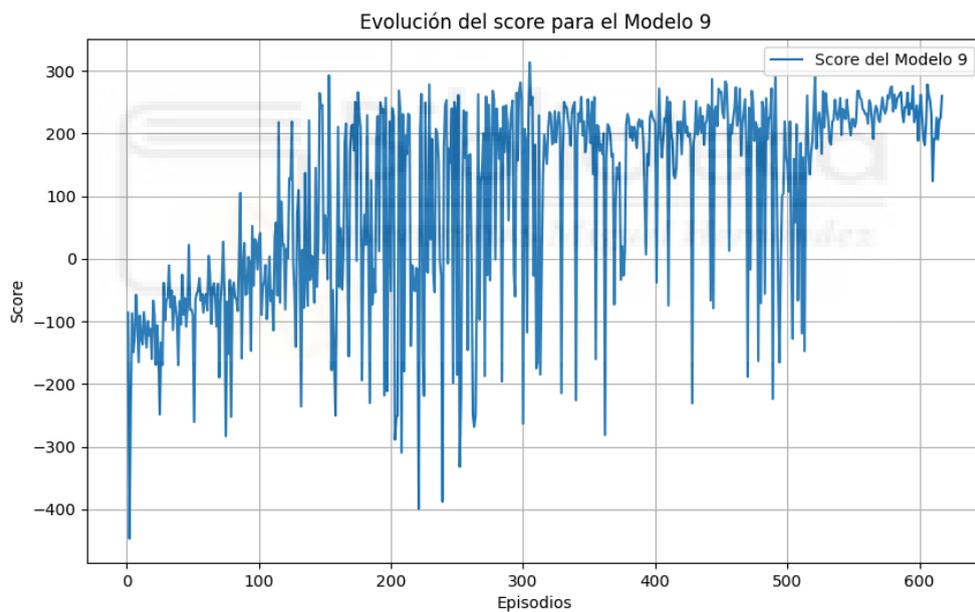


Figura 4.40: Entrenamiento red 32 con Target Network actualización cada 1000 pasos.

Se siguen observando resultados parecidos a los obtenidos en los experimentos realizados anteriormente con una diferencia que observamos una mayor estabilidad de los modelos en los últimos episodios de entrenamiento.

Experimento con Target Network con actualización “soft update”

Por último se ha implementado una configuración que realiza una actualización de la target network con una estrategia de actualización soft update.

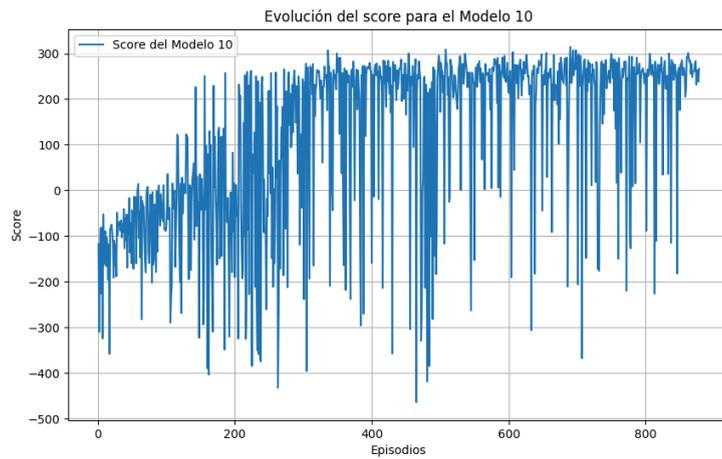


Figura 4.41: Entrenamiento red 128 con Target Network actualización con soft update.

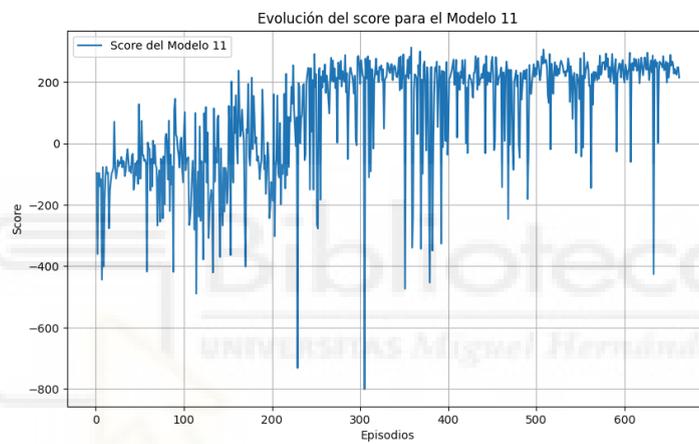


Figura 4.42: Entrenamiento red 64 con Target Network actualización con soft update.

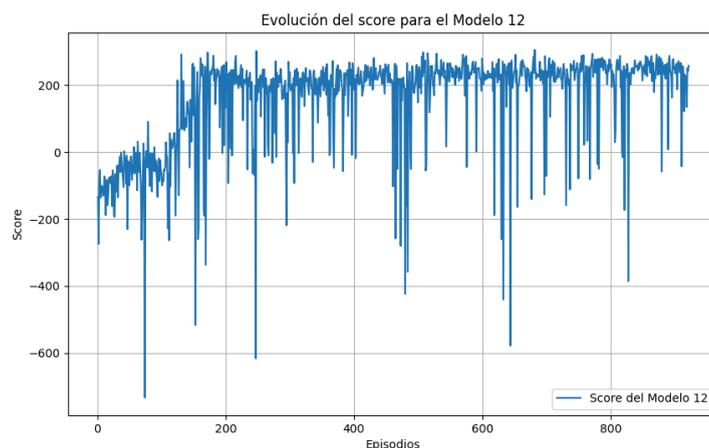


Figura 4.43: Entrenamiento red 32 con Target Network actualización con soft update.

En este último experimento se observa que, con el uso de soft update, con la red más grande aún existe inestabilidad (figura 4.41), pero en las redes más pequeñas (figuras 4.42 y 4.43) sí que se aprecia mayor estabilidad en los modelos.

En conclusión podemos confirmar que con aprendizaje por refuerzo, la mayoría de pruebas durante el proceso de entrenamiento presentan un resultado objetivo con la media de 230 de puntuación. La tabla 4.3 muestra los resultados de los test realizados a los agentes durante 100 episodios con 5 semillas diferentes.

Tabla 4.3: Resumen resultados obtenidos DQN

Nombre Agente	Episodios y Score de Entrenamiento	Tiempo de Entrenamiento	Promedio Test y Desviación Estándar
128 sin Target Network	561 / 231	12 mins	164.48 / 92.856
64 sin Target Network	497 / 231	9 mins	228.38 / 73.7
32 sin Target Network	755 / 233	12 mins	234.68 / 26.55
128 con Target Network actualizado cada episodio	910 / 230	17 mins	136.21 / 116.43
64 con Target Network actualizado cada episodio	376 / 231	6 mins	215.95 / 87.18
32 con Target Network actualizado cada episodio	1000 / 186	17 mins	221.04 / 98.14
128 con Target Network actualizado cada 1000 pasos	653 / 231	11 mins	192.89 / 144.72
64 con Target Network actualizado cada 1000 pasos	575 / 231	10 mins	-180.11 / 199.79
32 con Target Network actualizado cada 1000 pasos	617 / 231	11 mins	194.56 / 32.7
128 con Target Network actualizado con soft update	878 / 230	16 mins	197.34 / 125.64
64 con Target Network actualizado con soft update	662 / 230	14 mins	228.09 / 96.39
32 con Target Network actualizado con soft update	922 / 231	19 mins	17.50 / 140.18

Según los resultados de las pruebas realizadas, se puede observar que los agentes entrenados sin target network logran buenos resultados de generalización. En particular, el agente con la red de 32 alcanza una media de 234.68 con una desviación estándar baja de 26.55, se concluye que la política es estable y robusta.

Por otro lado, utilizando las target network los resultados varían dependiendo del método de actualización de la red target utilizado. Con actualización cada episodio, las redes de 32 y 64 neuronas obtienen buenos resultados con una media superior a 200, y una desviación

estándar en torno los 80 puntos. Para la red de 128 el rendimiento es inestable, con una media de 136.21 y una desviación estándar alta de 116. En el caso de la actualización cada 1000 pasos, el modelo de red de 128 neuronas produce un valor inestable, obteniendo un rendimiento medio de 192.89 y una desviación estándar muy elevada de 144.72, lo que evidencia inestabilidad. La red de 64 presenta una media negativa, lo que indica una muy mala generalización. Para la red de 32 neuronas se obtiene una media de 194.56 y una baja desviación de 32.7, mostrando un desempeño aceptable.

Finalmente, para los modelos entrenados con actualización soft update, la red de 64 neuronas obtiene una media elevada de 228, pero un alto valor de desviación de 96.39. El modelo de 128 produce una media de score de 192 y un muy alto valor de desviación de 140. Por otro lado, la red de 32 no logra generalizar, con una media de 17 y una alta desviación de 140.18.

En conclusión, los resultados obtenidos son coherentes con el marco teórico propuesto. Si bien el uso de una red target puede estabilizar el entrenamiento, su efectividad depende del método de actualización empleado para la red target, así como del tamaño de la red principal. En arquitecturas pequeñas, el uso de target network con actualización por episodio o incluso sin target puede ofrecer políticas estables y generalizables. Sin embargo, en redes medianas y grandes, una configuración inadecuada del target network, como una actualización poco frecuente o un mal ajuste del soft update, puede provocar una alta inestabilidad o una pobre generalización. Por tanto, se resalta la importancia de ajustar adecuadamente la estrategia de actualización al tamaño de la arquitectura utilizada para lograr un aprendizaje efectivo y generalizable.

4.5.- COMPARACIÓN DE ENFOQUES

Una vez se ha finalizado la experimentación con ambos enfoques (algoritmos genéticos y aprendizaje por refuerzo), se observan conclusiones relevantes sobre su comportamiento y rendimiento frente al problema de Lunar Lander. En primer lugar se destacaría la capacidad de obtener un resultado óptimo a partir de ambos enfoques. A pesar de que ambos consiguen una solución válida, lo hacen siguiendo estrategias de aprendizaje muy diferentes.

El enfoque basado en algoritmos genéticos (AG) destaca por su simplicidad y facilidad de implementación. Esto lo convierte en una opción especialmente interesante para entornos no diferenciables o con estructuras poco conocidas. Además, en las mejores configuraciones, como el cruce BLX- α con $\alpha=0.7$ sin mutación, los resultados obtenidos mostraron no solo una buena recompensa media, sino también una desviación estándar relativamente baja, lo que sugiere que, bajo ciertas condiciones, los AG pueden generar

políticas estables. No obstante, su rendimiento general es altamente dependiente de una correcta selección de operadores y del ajuste fino de parámetros como la tasa de mutación, lo que dificulta su uso óptimo sin una exploración previa intensiva.

Por otro lado, el enfoque de Deep Q-Learning (DQN), aunque más complejo en cuanto a implementación y ajuste de hiperparámetros, ha demostrado una mayor capacidad para aprender políticas eficaces y generalizables. En particular, el agente con una arquitectura de 32 neuronas sin target network alcanzó una recompensa media de 234.68 con una desviación estándar de tan solo 26.55, lo que indica un rendimiento tanto elevado como consistente.

En términos comparativos, DQN ofrece mayor estabilidad en el aprendizaje, mejor rendimiento medio y menor dependencia de la aleatoriedad, siempre que se configure correctamente. Sin embargo, su principal desventaja es la complejidad asociada al ajuste de hiperparámetros como la tasa de exploración, la frecuencia de actualización de la red objetivo o el tamaño de los lotes de entrenamiento.



Capítulo 5

Conclusiones y trabajo futuro

5.1.- CONCLUSIONES

Este proyecto ha logrado los objetivos establecidos inicialmente, explorando y comparando el aprendizaje por refuerzo (Deep Q-Learning, DQN) y los algoritmos genéticos para entrenar un agente en el juego Lunar Lander.

Se estudió y aprobó el entorno Lunar Lander, analizando acciones y recompensas, esto fue fundamental como base para el desarrollo de los distintos enfoques propuestos para el entrenamiento del agente.

Para modelado basado en algoritmo genético, se implementaron diferentes versiones del algoritmo combinando los distintos operadores y sus parámetros de ajuste para optimizar el resultado, finalmente se concluyó que el mejor modelo basado en estos algoritmos fue el generado con el método de selección de torneo con $k=5$, el operador de cruce BLX- α con

valor de $\alpha=0.7$ y sin uso de mutación. Se obtienen valores similares con el algoritmo formado por el método de selección de torneo con $k=5$, el operador de cruce BLX- α con valor de $\alpha=0.5$ y con una mutación de 0.1.

Para el DQN, se configuró el algoritmo de aprendizaje por refuerzo con redes neuronales, ajustando sus hiperparámetros y utilizando target networks, con diferentes estrategias de ajuste, para estabilizar el entrenamiento y mejorar la calidad del aprendizaje del agente. El mejor modelo basado en este tipo de aprendizaje fue el formado por la red neuronal con dos capas ocultas de 32 neuronas y sin utilizar target network. Con el uso de target network el mejor modelo ha sido el de una red neuronal con dos capas ocultas de 64 neuronas y soft update como estrategia de actualización.

Ambos enfoques lograron una solución óptima para Lunar Lander. El algoritmo genético destacó por su simplicidad. Por su parte, DQN ofreció mayor estabilidad y rendimiento, gracias a su proceso de aprendizaje continuo y el uso de target networks.

Por último, mencionar que todo el código empleado para la realización des este proyecto está disponible en el siguiente repositorio de GitHub:

<https://github.com/HectorSanz/TFG-Experimentos>

5.2.- POSIBLES DESARROLLOS FUTUROS

Como desarrollos y pruebas futuras respecto a lo realizado en este proyecto, se propone utilizar, en el caso del aprendizaje por refuerzo, otras técnicas más avanzadas como SAC (Soft Actor-Critic) o PPO (Proximal Policy Optimization), para probar si se consigue una mayor estabilidad y rapidez en el entrenamiento.

El algoritmo SAC [33] es un algoritmo de aprendizaje por refuerzo off-policy con arquitectura actor-crítico. Su principal característica es que maximiza tanto la recompensa esperada como la entropía de la política, ayudando a obtener una exploración más amplia y estable en entornos de acciones continuas.

El algoritmo PPO [34] pertenece a la familia de métodos de gradiente de política, ampliamente utilizado en entornos de acciones continuas y discretas. Busca un equilibrio entre la facilidad de implementación y un rendimiento estable, limitando el tamaño de las actualizaciones de la política en cada paso de entrenamiento. Esto se logra a través de una función objetivo "recortada" (clipped surrogate objective) que evita cambios excesivamente grandes que podrían desestabilizar el entrenamiento, asegurando que la nueva política se mantenga "próxima" a la antigua.

Por otro lado, en cuanto a los algoritmos genéticos, se podría implementar un algoritmo guiado por gramática, ya que este enfoque permite explorar de forma más estructurada el espacio de soluciones. Los algoritmos genéticos guiados por gramática [35] son una variante de los algoritmos genéticos que utilizan una gramática formal para generar soluciones válidas y estructuradas durante el proceso evolutivo. Esto permite explorar el espacio de soluciones de manera más eficiente y evitar soluciones inválidas, asegurando que las soluciones cumplan con reglas específicas definidas por la gramática. Son especialmente útiles cuando las soluciones deben seguir estructuras complejas, como expresiones matemáticas o programas.

Otra propuesta interesante es entrenar a los agentes variando las condiciones del entorno, por ejemplo, modificando la gravedad o la resistencia del aire, con el objetivo de evaluar qué tan robustos y adaptables son los algoritmos frente a diferentes escenarios.

Por último, sería interesante aplicar esta metodología a otros videojuegos o entornos similares para comprobar si estos métodos de entrenamiento de modelos funcionan bien en contextos distintos, y validar su capacidad de generalización.





Bibliografía

- [1] Población que usa Internet (en los últimos tres meses). Tipo de actividades realizadas por Internet
https://www.ine.es/ss/Satellite?L=es_ES&c=INESeccion_C&cid=1259925528782&p=1254735110672&pagename=ProductosYServicios%2FPYSLayout
16/06/2024
- [2] ¿Qué es la inteligencia artificial o IA?
<https://cloud.google.com/learn/what-is-artificial-intelligence?hl=es-419#section-2>
16/06/2024
- [3] La nueva era de la inteligencia artificial desvelando sus impactos y desafíos
Autores: Juan Gabriel García Huertas (coord.), Pablo Garrido Pintado (coord.), Diego Botas Leal (coord.). Editores: Sindéresis
24/09/2024

- [4] SALVI, Reena; SINGH, Rashmi. Artificial Intelligence and Human Society. International Journal of Social Science and Human. Research, 2023, vol. 6, no 09.
<https://jndmeerut.org/wp-content/uploads/2024/01/10-6.pdf>
24/09/2024
- [5] Annual global corporate investment in artificial intelligence, by type
<https://ourworldindata.org/grapher/corporate-investment-in-artificial-intelligence-by-type>
25/09/2024
- [6] JARRAHI, Mohammad Hossein. Artificial intelligence and the future of work: Human-AI symbiosis in organizational decision making. Business horizons, 2018, vol. 61, no 4, p. 577-586.
<https://www.sciencedirect.com/science/article/pii/S0007681318300387>
26/09/2024
- [7] LÓPEZ DE MÁNTARAS, Ramón. Algunas reflexiones sobre el presente y futuro de la inteligencia artificial. 2015.
<https://digital.csic.es/bitstream/10261/136978/1/NOV234%282015%2997-101.pdf>
27/09/2024
- [8] JULIANI, Arthur. Unity: A general platform for intelligent agents. arXiv preprint arXiv:1809.02627, 2018.
<https://arxiv.org/pdf/1809.02627>
27/09/2024
- [9] ¡Esta IA juega al ESCONDITE demasiado bien!
https://www.youtube.com/watch?v=5SkQuT3kZOc&ab_channel=DotCSV
27/09/2024
- [10] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," in IBM Journal of Research and Development, vol. 3, no. 3, pp. 210-229, July 1959, doi: 10.1147/rd.33.0210.
<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5392560>
29/11/2024
- [11] From artificial cells to deep learning. An evolutionary story. Archivo Digital UPM, Madrid.
<https://oa.upm.es/66864/>
29/11/2024

- [12] Clasificación multilingüe de documentos utilizando machine learning y la Wikipedia - Multilingual document classification using machine learning and Wikipedia - Scientific Figure on ResearchGate.
https://www.researchgate.net/figure/Figura-21-Aprendizaje-supervisado_fig1_322910101
29/11/2024
- [13] Artificial intelligence and Machine Learning for Real-world problems (A survey) - Scientific Figure on ResearchGate.
www.researchgate.net/figure/Unsupervised-Learning_fig2_355128875
29/11/2024
- [14] Machine Learning for Intrusion Detection in Industrial Control Systems: Applications, Challenges, and Recommendations - Scientific Figure on ResearchGate.
https://www.researchgate.net/figure/Semi-Supervised-Learning_fig4_358846616
29/11/2024
- [15] GHASEMI, Majid, et al. An Introduction to Reinforcement Learning: Fundamental Concepts and Practical Applications.
<https://arxiv.org/pdf/2408.07712>
01/12/2024
- [16] Watkins, C. J. C. H. y Dayan, P. Q-learning. Machine Learning, vol. 8(3), páginas 279–292, 1992. ISSN 1573-0565.
https://www.researchgate.net/publication/220344150_Technical_Note_Q-Learning
16/12/2024
- [17] Playing Atari with Deep Reinforcement Learning, Mnih et al, 2013.
<https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>
16/12/2024
- [18] John H. Holland. 1962. Outline for a Logical Theory of Adaptive Systems. J. ACM 9, 3 (July 1962), 297–314.
<https://doi.org/10.1145/321127.321128>
25/01/2025
- [19] Miniforge
<https://github.com/conda-forge/miniforge>
29/01/2025

- [20] Python
<https://www.python.org/>
29/01/2025
- [21] Matplotlib
<https://matplotlib.org/>
29/01/2025
- [22] Numpy
<https://numpy.org/about/>
29/01/2025
- [23] Keras
<https://keras.io/>
29/01/2025
- [24] Gymnasium
<https://github.com/Farama-Foundation/Gymnasium>
29/01/2025
- [25] Swing
<https://swig.org/>
29/01/2025
- [26] Box2D
<https://github.com/Farama-Foundation/Gymnasium/tree/main/gymnasium/envs/box2d>
29/01/2025
- [27] VSCode
<https://code.visualstudio.com/docs/editor/whyvscode>
30/01/2025
- [28] Gymnasium documentation. Lunar Lander - Gymnasium Documentation. (n.d).
Available
https://github.com/Farama-Foundation/Gymnasium/blob/main/gymnasium/envs/box2d/lunar_lander.py
30/01/2025
- [29] Python plugin VSCode
<https://marketplace.visualstudio.com/items?itemName=ms-python.python>
02/03/2025

- [30] Jupyter plugin VSCode
<https://marketplace.visualstudio.com/items?itemName=ms-toolsai.jupyter>
02/03/2025
- [31] GitHub Copilot plugin VSCode
<https://marketplace.visualstudio.com/items?itemName=GitHub.copilot>
02/03/2025
- [32] Lunar Lander
[https://en.wikipedia.org/wiki/Lunar_Lander_\(1979_video_game\)](https://en.wikipedia.org/wiki/Lunar_Lander_(1979_video_game))
02/03/2025
- [33] SAC
<https://medium.com/intro-to-artificial-intelligence/soft-actor-critic-reinforcement-learning-algorithm-1934a2c3087f>
11/06/2025
- [34] PPO
<https://medium.com/@danushidk507/ppo-algorithm-3b33195de14a>
11/06/2025
- [35] Programación Genética
https://es.wikipedia.org/wiki/Programaci%C3%B3n_gen%C3%A9tica
11/06/2025