Proyecto 2: Machine Learning

Universidad de Alicante
Razonamiento Automático (Machine Learning)

Repositorio del proyecto: github.com/Ginescag/Atari_assault_NN

GINÉS CABALLERO GUIJARRO
RAÚL LUJÁN DOMENECH
RAÚL RUIZ FLORES

19 de enero de 2026

Índice

1. Introducción a los modelos implementados

Este proyecto desarrolla dos componentes principales orientados a aprendizaje automático y optimización: una librería de redes neuronales tipo Perceptrón Multicapa (MLP) y un Algoritmo Genético (GA) genérico y reutilizable, concebidos como base para entrenar, evaluar y comparar modelos capaces de resolver problemas no lineales y de toma de decisiones.

Para validar su funcionamiento se incluyen herramientas de ejecución y pruebas por terminal, mecanismos de control experimental (como reproducibilidad mediante semilla, criterios de parada y gestión de parámetros), y casos de uso que permiten comprobar el comportamiento de los modelos antes de su aplicación final en un entorno de agente basado en *Arcade Learning Environment* (ALE) con el juego *Atari Assault*, dejando el sistema preparado para integrarse en un escenario de evaluación dinámico.

1.1. Introducción a la NN

Las Redes Neuronales Artificiales (ANN, *Artificial Neural Networks*) son modelos computacionales inspirados en la estructura y funcionamiento del sistema nervioso biológico. En este proyecto, nos centramos en la implementación del Perceptrón Multicapa (MLP), una arquitectura feedforward capaz de aprender representaciones no lineales de los datos.

El objetivo de una NN es aproximar una función matemática compleja $f(x) \approx y$ que mapee las entradas del entorno (inputs) a las salidas deseadas (acciones o valores), minimizando el error de predicción. El MLP es una herramienta fundamental en el aprendizaje supervisado debido a su capacidad como aproximador universal de funciones. Esto significa que, con suficiente capacidad (número de neuronas y capas) y una función de activación no lineal adecuada, la red puede modelar teóricamente cualquier función continua. Esta característica es crucial para abordar problemas donde la relación entre las variables de entrada y la decisión óptima es altamente compleja y no puede describirse mediante reglas simples.

1.2. Introducción al Algoritmo Genético (GA)

Los Algoritmos Genéticos (GA, Genetic Algorithms) son técnicas de optimización inspiradas en la evolución biológica. Su objetivo es encontrar soluciones de alta calidad en espacios de búsqueda grandes o complejos mediante un proceso iterativo que simula selección natural: se mantiene una población de soluciones candidatas, se evalúa su calidad mediante una función de fitness y, generación tras generación, se generan nuevas soluciones combinando y modificando las mejores candidatas.

Un GA es especialmente útil cuando el problema presenta alguna de estas características: (1) el espacio de soluciones es muy grande para una búsqueda

exhaustiva, (2) la función objetivo es no lineal o multimodal (muchos óptimos locales), (3) la evaluación puede considerarse una "caja negra" (solo se puede medir el rendimiento de una solución sin disponer de derivadas ni una formulación analítica sencilla), o (4) existen representaciones distintas para la solución (binaria, real, permutaciones, etc.).

Por estas razones, un GA permite abordar desde problemas discretos sencillos hasta problemas continuos de ooptimización numérica y problemas combinatorios, manteniendo el mismo esquema general de búsqueda:

- **Problemas discretos** (genoma binario), donde cada gen representa una decisión de tipo 0/1 (p. ej. XOR, OneMax).
- **Problemas continuos** (genoma real), donde cada gen es un valor real dentro de un rango (p. ej. Sphere, Ackley o Rastrigin).
- **Problemas tipo "caja negra"**, donde el fitness se obtiene ejecutando una simulación o entorno y puede ser costoso (por ejemplo, evaluar una política en ALE a partir del *score* obtenido en el juego).

De forma general, el ciclo de un GA se compone de los siguientes pasos:

1. **Inicialización**: se genera una población inicial de individuos (soluciones) de forma aleatoria o guiada.
2. **Evaluación**: cálculo del fitness de cada individuo según el problema.
3. **Selección**: elección de padres favoreciendo a los individuos con mejor fitness (p. ej. torneo o ruleta).
4. **Cruce**: se combinan genes de dos padres para producir descendencia (por ejemplo, cruce a un punto, uniforme o cruce tipo BLX-α en variables reales).
5. **Mutación**: se introducen pequeñas variaciones aleatorias para mantener diversidad y evitar estancamiento (por ejemplo, bit-flip en binario o ruido gaussiano en reales).
6. **Reemplazo y elitismo**: se construye la nueva población, opcionalmente preservando los mejores individuos de la generación anterior (elitismo).
7. **Criterios de parada**: el algoritmo se detiene cuando se cumple una condición (máximo de generaciones, fitness objetivo o estancamiento).

Este enfoque ofrece un equilibrio entre exploración (probar regiones distintas del espacio de búsqueda) y explotación (refinar las mejores soluciones encontradas). En problemas continuos unimodales como Sphere, suele observarse una convergencia rápida hacia el óptimo global. En funciones multimodales como Ackley o Rastrigin, la convergencia puede verse limitada por óptimos locales, haciendo que la configuración de parámetros (tamaño de

población, tasas de cruce/mutación y elitismo) sea especialmente relevante. En problemas combinatorios como TSP, la representación de las soluciones y la elección de operadores condiciona la calidad del resultado y la velocidad de mejora.

En este proyecto se implementa un GA reutilizable, con operadores intercambiables y parámetros configurables, y se valida su comportamiento en distintos tipos de problemas para analizar su rendimiento y sensibilidad a la configuración.

2. Descripción de tecnologías implementadas y estado operativo

2.1. Red neuronal MLP

El proyecto implementa una librería de Redes Neuronales Artificiales basada en la arquitectura de Perceptrón Multicapa (MLP) con aprendizaje supervisado. El diseño se ha realizado sin dependencias de terceros para el álgebra lineal, implementando una clase propia `Matrix` para gestionar las operaciones de matrices.La implementación permite instanciar redes de topología dinámica y entrenarlas mediante *Backpropagation* con descenso de gradiente estocástico (SGD). A continuación se detallan las características técnicas clave.

2.1.1. Arquitectura y Topología Dinámica

La clase principal `NeuralNetwork` se ha diseñado para ser flexible en cuanto a profundidad y anchura. El constructor acepta un vector de enteros (`vector<unsigned int>`) que define la topología:

- La capa de entrada se ajusta a la dimensión del vector de características.
- Se pueden añadir arbitrariamente capas ocultas.
- La capa de salida se ajusta al número de clases o valores a predecir.

Para la inicialización de los parámetros, no se utilizan valores aleatorios uniformes simples. Se ha implementado el método de Inicialización de Xavier Glorot en la clase `Matrix`. Esta técnica inicializa los pesos dentro del rango $\pm\sqrt{\frac{6}{n_{in}+n_{out}}}$, lo cual es crucial para mantener la varianza de las activaciones controlada en las primeras etapas del entrenamiento, facilitando la convergencia cuando se usan funciones como la tangente hiperbólica.

2.1.2. Funciones de Activación Seleccionables

Tanto feedforward como backpropagate soportan múltiples funciones de activación, seleccionables mediante un campo parametrizado. Esto permite experimentar con diferentes dinámicas de aprendizaje. Se han implementado las siguientes opciones:

- **Tangente Hiperbólica (`"tanh"`)**: Es la función por defecto en esta implementación. Mapea la entrada al rango $[-1, 1]$. Se ha priorizado su uso debido a que, al estar centrada en cero, suele ofrecer un mejor rendimiento en el entrenamiento de redes profundas comparada con la sigmoide clásica.
- **Sigmoide (`"sigmoid"`)**: Implementada para problemas que requieren salidas en el rango $[0, 1]$, típicamente interpretables como probabilidades.
- **Softmax**: Se incluye una implementación estática de *Softmax* con estabilidad numérica (sustracción del máximo), útil para la capa de salida en problemas de clasificación multiclase.
- **Signo (`"Sign"`)**: Tanto su implementación como su uso son anecdóticos ya que al no ser una función continua su uso es muy limitado

2.1.3. Normalización de Datos

Para garantizar el correcto funcionamiento de las funciones de activación (especialmente `tanh` y `sigmoid`, que saturan si los valores de entrada son muy grandes), se ha implementado una etapa de normalización. En la clase auxiliar `DataHelper` y en los generadores de problemas sintéticos, los datos de entrada se normalizan para encontrarse en el rango $[-1, 1]$. Por ejemplo, en la lectura de datos crudos (0-255), se aplica la transformación:

$$x' = \frac{x - 128{,}0}{128{,}0} \tag{1}$$

Esta normalización es crítica para evitar la saturación prematura de las neuronas y asegurar que los gradientes fluyan correctamente durante el proceso de aprendizaje.

2.1.4. Regularización mediante Early Stopping

Uno de los puntos críticos en el entrenamiento de redes neuronales es evitar el *overfitting*. En esta implementación, se ha optado por la técnica de Early Stopping como mecanismo principal de regularización. El método `train` ha sido extendido para aceptar un conjunto de datos de validación (`valInputs`, `valTargets`), distinto al de entrenamiento. El funcionamiento implementado es el siguiente:

1. En cada epoch, tras actualizar los pesos con el conjunto de entrenamiento, la red evalúa su rendimiento (pérdida media o *loss*) sobre el conjunto de validación.

2. Se mantiene un registro del **mejor modelo** observado hasta el momento (copia de seguridad de pesos y sesgos).

3. Se utiliza un contador de **paciencia**. Si la pérdida en validación no mejora tras un número determinado de epochs consecutivas, se asume que la red ha empezado a sobreajustarse.

4. Al dispararse la condición de parada, el entrenamiento se detiene y **se restauran los pesos del mejor modelo guardado**, descartando los últimos cambios que provocaron el sobreajuste.

Este enfoque permite entrenar durante un número elevado de epochs sin riesgo de degradar la capacidad de generalización de la red.

2.1.5. Persistencia y Evaluación

Finalmente, la arquitectura incluye mecanismos de persistencia. Los métodos `saveModel` y `loadModel` permiten serializar la topología y los valores de todas las matrices de pesos y sesgos a ficheros de texto. Esto habilita un flujo de trabajo donde el entrenamiento (costoso computacionalmente) se separa de la fase de test.

Para la evaluación del rendimiento, se incluye el cálculo de la Desigualdad de Hoeffding. Esta métrica teórica permite establecer, con un nivel de confianza, una cota superior para el error de generalización basada en el error observado y el número de muestras de test, aportando rigor estadístico a los resultados obtenidos.

2.2. Algoritmo Genético

En este proyecto se ha implementado un Algoritmo Genético (GA) genérico en C++ (estándar C++17), pensado para poder reutilizarse en varios tipos de problemas: binarios (p. ej. XOR/OneMax), continuos (p. ej. Sphere/Ackley/Rastrigin) y problemas de evaluación costosa o "caja negra" (p. ej. un entorno de simulación como ALE). Esta parte cubre la tecnología implementada y cómo está organizada.

2.2.1. Estructura del código y módulos

La implementación está separada en dos piezas principales:

- `new_ga.(h/cpp)`: núcleo del GA (representación de individuos, operadores genéticos y motor principal).

- `problemas_ga.(h/cpp)`: conjunto de problemas de prueba que implementan una misma interfaz (`IProblem`).

Además, se incluye un ejecutable de pruebas `main_ga.cpp` que permite seleccionar el problema y los hiperparámetros por línea de comandos (población, generaciones, semilla, elitismo, tasas, etc.). Esto sirve como verificación de funcionamiento y como herramienta de experimentación.

2.2.2. Representación de soluciones: `Genome` e `Individual`

El GA trabaja con dos estructuras básicas:

- `Genome`: representa el cromosoma (la solución). Soporta dos tipos:
  - **Binario**: vector de bits (`vector<int>`) para problemas tipo XOR u OneMax.
  - **Real**: vector de reales (`vector<double>`) para optimización continua o para representar parámetros/pesos.
- `Individual`: empaqueta un `Genome` y su **fitness**. Incluye un flag `fitness_valid` para saber si el fitness está actualizado.

El uso de `fitness_valid` es importante cuando la evaluación es cara: si un individuo se copia por elitismo y no cambia su genoma, su fitness no necesita recalcularse.

2.2.3. Interfaz del problema: `IProblem`

Para desacoplar el GA del problema concreto, se usa una interfaz común:

- `genomeType()` y `genomeSize()`: definen el tipo y tamaño del genoma esperado.
- `objective()`: indica si el GA debe **maximizar** o **minimizar**.
- `getBounds()`: devuelve límites por gen (especialmente útil en genomas reales).
- `randomGenome()`: inicializa una solución válida aleatoria.
- `evaluate()`: calcula el fitness del genoma (puede ser costoso).
- `describe()`: genera un texto para depuración/logs.

En los problemas continuos, los límites se almacenan con una estructura `Bounds{lo,hi}`. Estos límites se reutilizan posteriormente para recortar valores (por ejemplo, tras mutación gaussiana o cruce BLX-α).

2.2.4. Operadores intercambiables (patrón estrategia)

Los operadores genéticos se han diseñado para poder cambiarse sin tocar el núcleo del algoritmo. La idea es que el GA tenga "enchufados" tres componentes:

- **Selección**: `ISelection`
- **Cruce**: `ICrossover`
- **Mutación**: `IMutation`

Cada interfaz define el método principal del operador y un `reset()` por si en el futuro se quisiera mantener estado interno. En la implementación actual, los operadores no necesitan memoria interna y `reset()` es trivial.

Selección Se implementan dos estrategias:

- **Torneo** (`TournamentSelection`): elige k candidatos al azar y devuelve el mejor según la función objetivo.
- **Ruleta** (`RouletteSelection`): asigna a cada individuo un peso proporcional a su fitness y selecciona por probabilidad acumulada.

Cruce Se implementan tres cruces:

- **1-punto** (`OnePointCrossover`): válido para binario y real.
- **Uniforme** (`UniformCrossover`): válido para binario y real (intercambio por gen con probabilidad).
- **BLX-α** (`BlendCrossover`): pensado para genomas reales; amplía el intervalo entre los dos padres y samplea dentro.

Mutación Se implementan dos mutaciones:

- **Bit-flip** (`BitFlipMutation`): para genomas binarios; cambia bits con una probabilidad por gen.
- **Gaussiana** (`GaussianMutation`): para genomas reales; suma ruido normal $N(0, \sigma)$ y (opcionalmente) recorta a `Bounds`.

Componente	**Implementaciones**	**Parámetros principales**
Selección	Torneo / Ruleta	`tournament_size`, `epsilon`
Cruce	1-punto / Uniforme / BLX-α	`allow_endpoints`, `swap_prob`, `alpha`
Mutación	Bit-flip / Gaussiana	`flip_prob`, `sigma`, `clamp_to_bounds`

Cuadro 1: Operadores implementados en el GA y sus parámetros configurables.

2.2.5. Configuración del GA y criterios de parada

El GA se controla mediante una configuración global (`GeneticAlgorithm::Config`), donde destacan:

- **Tamaño de población** (`population_size`).
- **Tasa de cruce** (`crossover_rate`) y **tasa de mutación** (`mutation_rate`).
- **Elitismo** (`elitism_count`): número de mejores individuos que pasan a la siguiente generación sin modificarse.
- **Semilla** (`seed`): permite reproducibilidad en experimentos.
- **Histórico** (`keep_history`): guarda métricas por generación (mejor/-media/peor).
- **Parada** (`stop`): máximo de generaciones, fitness objetivo opcional, y estancamiento.

El criterio de **estancamiento** se implementa comparando la mejora del mejor fitness entre generaciones. Si la mejora absoluta es menor que un umbral (`min_delta`) durante `max_stagnant_generations`, se detiene la ejecución.

2.2.6. Flujo de ejecución y estado interno

El motor principal se gestiona con tres métodos:

- `initialize()`: crea población inicial usando `randomGenome()` y evalúa.
- `step()`: ejecuta una generación completa (reproducción, evaluación, actualización de estadísticas y parada).
- `run()`: itera hasta cumplir algún criterio de parada y devuelve un `Result` con el mejor individuo.

En cada generación, la nueva población se construye así:

1. Copia de elites (elitismo).
2. Relleno del resto generando descendencia:
   - selección de dos padres,
   - cruce con probabilidad `crossover_rate`,
   - elección aleatoria de uno de los dos hijos generados,
   - mutación con probabilidad `mutation_rate`,
   - invalidación de fitness del nuevo individuo.

El estado del algoritmo incluye población actual, siguiente población, generación actual, mejor/peor individuo, límites por gen (`bounds`) y contadores de estancamiento.

2.2.7. Métricas y registro

Para facilitar el análisis y las gráficas, se calcula por generación:

- Mejor fitness (`best_fitness`)
- Fitness medio (`mean_fitness`)
- Peor fitness (`worst_fitness`)

Estas métricas pueden guardarse en un vector histórico si `keep_history` está activado. Además, existe un **callback por generación** (`GenerationCallback`) para imprimir logs o capturar datos sin acoplarlo al núcleo del GA.

2.2.8. Soporte para problemas "caja negra"

Para problemas donde evaluar el fitness implica ejecutar una simulación (por ejemplo, jugar episodios en ALE), la arquitectura permite:

- Tener un `evaluate()` costoso sin cambiar el GA.
- Reducir evaluaciones innecesarias gracias a `fitness_valid` (por ejemplo, individuos copiados por elitismo).
- Configurar el problema mediante callbacks (en `AleAtariProblem`) para inicialización, evaluación y descripción.

3. Manual de uso de los modelos implementados

3.1. Red neuronal MLP

Este apartado detalla el uso del ejecutable `mlp_test`, diseñado para verificar el correcto funcionamiento de la librería de redes neuronales (`NeuralNetwork.h`). Este ejecutable está diseñado como una batería de pruebas con configuraciones pre-ajustadas para tres problemas tipo, permitiendo al usuario variar la complejidad del dataset.

3.1.1. Requisitos y compilación

El código fuente se encuentra en la carpeta `MLP/`. La compilación requiere un compilador compatible con C++17. Se puede compilar manualmente o mediante el `Makefile` proporcionado en la raíz:

```
# Opción A: Usando Makefile (Recomendado)
make nn

# Opción B: Manualmente
g++ -O3 -std=c++17 -Wall MLPtests.cpp -o mlp_test
```

Esto genera el binario `mlp_test`.

3.1.2. Formatos de ejecución y problemas

El ejecutable no utiliza *flags*, sino argumentos posicionales para seleccionar el problema y el tamaño de los datos.

```
./mlp_test <problema> [argumentos...]
```

Los tres escenarios implementados para validar la red son:

1. **Problema del Donut** (Clasificación no lineal 2D):

   ```
   ./mlp_test donut <samples>
   ```

   Genera un dataset de puntos (x, y) donde la clase positiva es un anillo concéntrico. Valida la capacidad de la red para generar fronteras de decisión curvas/no lineales.

2. **Problema de Cuadrantes** (Clasificación Multiclase):

   ```
   ./mlp_test quadrant <samples>
   ```

   Divide el plano 2D en 4 clases según el signo de las coordenadas. Valida la capacidad de la red para manejar salidas múltiples (vector *One-Hot* de 4 neuronas).

3. **Problema de Paridad / XOR** (Lógica N-dimensional):

   ```
   ./mlp_test parity <bits> <samples>
   ```

   El reto más complejo. La red debe determinar si la suma de bits es par o impar. Es un problema altamente no lineal que no se puede resolver sin capas ocultas.

   - Ejemplo XOR simple: `./mlp_test parity 2 1000`
   - Ejemplo carga alta: `./mlp_test parity 16 30000`

3.1.3. Monitorización del entrenamiento

Al ejecutar cualquiera de los problemas, el sistema inicia un entrenamiento supervisado. La salida por consola permite monitorizar tres aspectos clave implementados en la librería:

```
Epoch: 50/100 | Avg Error (Loss): 0.023 | patience left: 2
```

- **Avg Error (Loss)**: El Error Cuadrático Medio (MSE) sobre el conjunto de entrenamiento. Debe disminuir conforme avanzan las épocas.
- **Validación y Regularización (Early Stopping)**: El sistema reserva internamente un 20 % de los datos generados para validación (conjunto que la red no usa para entrenar). El parámetro `patience left` indica cuántas épocas quedan antes de detener el entrenamiento si el error de validación no mejora, evitando así el sobreajuste (*Overfitting*).
- **Activación**: Internamente, estos tests configuran la red para usar la función `tanh` (Tangente Hiperbólica) y la inicialización de pesos Xavier, optimizando la convergencia.

3.1.4. Interpretación de resultados finales

Tras finalizar el entrenamiento (ya sea por alcanzar el límite de épocas o por *Early Stopping*), el programa evalúa la red contra un conjunto de **Test** (datos nunca vistos, distintos de Train y Validación) y muestra dos métricas de calidad:

1. **Precisión (Accuracy)**: Porcentaje directo de aciertos en el conjunto de test.
2. **Garantía Teórica (Desigualdad de Hoeffding)**:

```
==================== HOEFFDING INEQUALITY ====================
Numero de muestras (N): 1000
Error en la prueba (E_in): 0.031
Probabilidad de estar dentro del limite...: 98.65%
```

   Calcula la probabilidad estadística de que el error obtenido sea fiable dada la cantidad de muestras de test usadas, aportando rigor matemático a la evaluación empírica.

3.2. Algoritmo Genético (GA)

Este apartado explica cómo ejecutar el Algoritmo Genético desde terminal, cómo ajustar sus parámetros y cómo usarlo para **entrenar** (buscar una buena solución / pesos) y después **reutilizar** el resultado en el uso posterior (por ejemplo, en un bot). El ejecutable de esta parte se llama `main_ga` y se controla por línea de comandos.

3.2.1. Requisitos y compilación

La implementación está pensada para compilar con `g++` en Linux Manjaro usando C++17. Desde la carpeta donde están los ficheros (`main_ga.cpp`, `new_ga.cpp`, `problemas_ga.cpp`):

```
g++ -O3 -std=c++17 -Wall -Wextra -pedantic \
  main_ga.cpp new_ga.cpp problemas_ga.cpp \
  -o main_ga
```

Esto genera el binario `main_ga`.

3.2.2. Ayuda rápida y formato general de ejecución

Para ver la ayuda:

```
./main_ga --help
./main_ga -h
```

Formato general:

```
./main_ga <problem> [options]
./main_ga test [options]
```

Donde `<problem>` puede ser: `sphere`, `rastrigin`, `ackley`, `xor`, `onemax`, `tsp`, `ale`. El modo `test` sirve para comparar combinaciones de operadores y, por diseño, **no ejecuta ALE**.

3.2.3. Problemas disponibles y parámetros específicos

La Tabla 2 resume los problemas implementados y sus opciones principales.

Problema	Genoma	Objetivo	Opciones
`xor`	Binario (4 bits)	Maximizar aciertos	(sin opciones)
`onemax`	Binario (N bits)	Maximizar unos	`-bits N`
`sphere`	Real (D)	Minimizar	`-dim N`
`ackley`	Real (D)	Minimizar	`-dim N`
`rastrigin`	Real (D)	Minimizar	`-dim N`
`tsp`	Real (*random keys*)	Minimizar distancia	`-tsp-n N`
`ale`	Real (S)	Maximizar	`-ale-size N -episodes N -steps N`

Cuadro 2: Problemas incluidos en el ejecutable `main_ga`.

Notas importantes de uso:

- En `tsp` las ciudades se generan aleatoriamente. Por tanto, si se quiere reproducibilidad, es clave fijar la semilla con `-seed`.

- En `ale` (en esta entrega) el fitness está implementado como una función "dummy" con genoma real, pero la estructura está preparada para sustituir la evaluación por una simulación real (ALE) sin tocar el núcleo del GA.

3.2.4. Parámetros del GA (población, generaciones, elitismo, tasas)

Las opciones generales del GA (válidas para cualquier problema) son:

- `-pop N`: tamaño de población. A mayor población, más exploración pero más coste por generación.
- `-gen N`: máximo de generaciones. Es el límite duro del entrenamiento.
- `-seed N`: semilla para reproducir resultados.
- `-elite N`: número de individuos que pasan intactos a la siguiente generación (elitismo).
- `-cross P`: probabilidad de aplicar cruce (p.ej. 0.9). Si no hay cruce, los hijos tienden a ser copias (antes de mutación).
- `-mut P`: probabilidad/intensidad de mutación (según el operador). Controla la diversidad.
- `-target X`: fitness objetivo. Si se alcanza, el algoritmo puede parar antes del máximo de generaciones.

Valores por defecto (si no se especifican): `pop=50`, `gen=200`, `seed=123`, `elite=1`, `cross=0.9`, `mut=0.2`. Además, hay criterios internos para detectar estancamiento (máximo de generaciones sin mejora y mejora mínima esperada), de forma que el entrenamiento no se alargue inútilmente cuando la solución deja de progresar.

3.2.5. Selección de operadores (selección, cruce, mutación)

Se pueden forzar operadores por línea de comandos:

- Selección: `-sel tournament` o `-sel roulette`
- Cruce: `-cx onepoint`, `-cx uniform`, `-cx blend`
- Mutación: `-mx bitflip` o `-mx gauss`

Si el usuario **no especifica** operadores, el programa aplica una elección por defecto coherente con el tipo de genoma:

- Para problemas **binarios**: cruce *one-point* y mutación *bit-flip*.

- Para problemas **reales**: cruce *blend* (BLX-α) y mutación gaussiana.

La idea detrás de esta decisión es práctica: los operadores binarios trabajan con bits (0/1), mientras que en continuos se necesitan operadores que generen valores reales dentro de límites y permitan un ajuste fino.

3.2.6. Interpretación de la salida y seguimiento del entrenamiento

Durante el entrenamiento se imprimen estadísticas periódicas:

```
gen X best=... mean=... worst=...
```

Esto permite monitorizar:

- **Mejor fitness (best)**: progreso real del GA.
- **Fitness medio (mean)**: si la población en conjunto mejora o si solo mejora un individuo aislado.
- **Peor fitness (worst)**: detecta si hay individuos muy malos (diversidad) o si toda la población ha colapsado.

Al finalizar se imprime:

```
DONE
best fitness: ...
best genome:  ...
generations:  ...
```

3.2.7. Modo `test`: comparación rápida de combinaciones

El modo `test` ejecuta una batería de pruebas automáticas para comparar varias combinaciones (*tournament/roulette* + *onepoint/uniform/blend* + *bitflip/gauss*) sobre varios problemas (`sphere`, `rastrigin`, `ackley`, `xor`, `onemax`, `tsp`). Cada caso se repite varias veces con semillas diferentes y se resume cuántas veces:

- mejora respecto al mejor individuo inicial,
- alcanza el fitness objetivo si existe un target.

Ejemplo:

```
./main_ga test --gen 150 --pop 60 --seed 123
```

Este modo es útil para **fine-tuning** porque da evidencia rápida de qué operadores funcionan mejor en cada familia de problemas, sin necesidad de ejecutar manualmente decenas de combinaciones.

3.2.8. Entrenamiento de bots (caso ALE) y uso posterior

En el flujo de trabajo de bots, el GA se utiliza como método de búsqueda de parámetros (por ejemplo, pesos de una red neuronal). El proceso se divide en dos fases: **entrenamiento offline** y **uso posterior**.

1) Entrenamiento offline El comando base para el modo ALE es:

```
./main_ga ale --ale-size 60 --episodes 3 --steps 5000 --pop 80 --gen 200
```

Donde:

- `-ale-size` define el tamaño del genoma real (habitualmente coincide con el número de pesos/parámetros a optimizar).
- `-episodes` y `-steps` controlan el coste de evaluación (más episodios/-pasos = evaluación más fiable pero más lenta).

En esta entrega, el problema `ale` está implementado con una evaluación "dummy" (sirve para validar el pipeline del GA), pero está diseñado para que el fitness se pueda reemplazar por una evaluación real: en un caso real, el fitness sería la media

4. Experimentación: Redes MLP

El desarrollo de un motor de redes neuronales desde cero implica desafíos que van más allá de la correcta implementación de las fórmulas matemáticas. Es necesario verificar que la red es capaz de aprender topologías complejas, generalizar ante datos no vistos y mantener la estabilidad numérica en arquitecturas profundas.

Para validar estos aspectos antes del despliegue final, se ha diseñado una metodología experimental dividida en problemas sintéticos de dificultad incremental y una prueba de integración en un entorno de aprendizaje a través de ALE. A continuación, se detallan los escenarios de prueba y las decisiones de diseño derivadas de la observación del entrenamiento.

4.1. Entornos de Prueba y Problemas Seleccionados

Se han seleccionado cuatro problemas distintos, cada uno dirigido a evaluar una capacidad específica de la arquitectura perceptrón multicapa (MLP) implementada en `MLPtests.cpp`:

1. **No Linealidad Geométrica (Problema del Donut):** Consiste en clasificar puntos en un espacio 2D pertenecientes a dos clases concéntricas.

- *Objetivo:* Verificar que la red es capaz de generar fronteras de decisión curvas y cerradas. Un perceptrón simple o una red mal configurada solo podría trazar cortes lineales, fallando en esta tarea.

2. **Clasificación Multiclase (Cuadrantes):** Clasificación de coordenadas (x, y) en cuatro clases distintas según su cuadrante.

   - *Objetivo:* Validar el funcionamiento de la capa de salida vectorizada (representación *One-Hot* de 4 neuronas).

3. **Lógica Profunda (Paridad N-bits / XOR):** Determinar si la suma de un vector de bits es par o impar. Es un problema clásico donde no existe correlación espacial; cambiar un solo bit invierte la salida.

   - *Objetivo:* Evaluar la capacidad de la red para aprender relaciones lógicas altamente no lineales. Se ha diseñado para probar la red bajo estrés con configuraciones de alta dimensionalidad ($N = 8$ y $N = 16$), donde la memorización es ineficiente y se requiere abstracción real.

4. **Entorno Dinámico (Atari Assault con ALE):** Como prueba de integración final, se ha conectado la red neuronal al *Arcade Learning Environment* (ALE).

   - *Configuración:* La red recibe como entrada el estado del juego (vector de características extraído de la RAM o pantalla) y sus salidas corresponden a las acciones del mando.
   - *Objetivo:* Verificar que la implementación es lo suficientemente eficiente y robusta para operar dentro del bucle de simulación de un agente, procesando entradas continuas y de alta variabilidad que no siguen una distribución estática como los datasets sintéticos anteriores.

4.2. Exploración de Topologías

Uno de los focos de la experimentación ha sido determinar la relación entre la profundidad de la red y su capacidad de aprendizaje. Se han definido experimentos comparativos variando la arquitectura:

- **Redes Superficiales vs. Profundas:** Se ha comparado el rendimiento de redes con una única capa oculta frente a redes multicapa para el problema de Paridad. La hipótesis de trabajo es que, para problemas lógicos complejos, la profundidad otorga una ventaja representacional superior a la anchura.

- **Estructuras Piramidales:** Para los problemas de clasificación (Donut y Cuadrantes), se ha experimentado con topologías que reducen progresivamente el número de neuronas por capa (ej. $24 \rightarrow 12 \rightarrow 6$). El objetivo es forzar a la red a comprimir la información, extrayendo características de alto nivel en las capas finales.

4.3. Estabilidad y Convergencia: Inicialización de Pesos

Durante las fases iniciales de experimentación, especialmente con las topologías más profundas requeridas para el problema de paridad y en el entorno de ALE, se observó que el entrenamiento a menudo se estancaba en las primeras épocas o convergía muy lentamente.

El análisis de los gradientes sugirió un problema de **saturación de neuronas**: al usar una inicialización aleatoria uniforme estándar, las sumas ponderadas en las neuronas alcanzaban valores elevados, empujando la función de activación (Tanh/Sigmoide) a sus regiones asintóticas donde la derivada es cercana a cero.

Para mitigar esto, se incorporó al diseño experimental la comparación entre:

1. **Inicialización Aleatoria Uniforme:** $W \in [-1, 1]$.
2. **Inicialización de Xavier (Glorot):** Ajustando la varianza de los pesos en función del número de conexiones de entrada y salida de cada capa, implementado en la clase `Matrix`.

El objetivo de esta comparativa es demostrar que una inicialización matemáticamente fundamentada es crítica para el entrenamiento de redes neuronales.

4.4. Detección de Sobreajuste y Regularización

En los experimentos con datasets de tamaño limitado (particularmente en Paridad con ruido introducido), se monitorizó la evolución del error. Se observó un fenómeno característico de **sobreajuste (Overfitting)**: a partir de cierto número de épocas, el error de entrenamiento continuaba disminuyendo hacia cero, mientras que el error en datos no vistos comenzaba a aumentar, indicando que la red estaba memorizando el dataset.

Esta observación motivó la implementación y validación de una estrategia de *Early Stopping* en el método `train`:

- Se modificó el protocolo de entrenamiento para separar un subconjunto de validación (distinto de Train y Test).
- Se definió un criterio de "Paciencia"(número de épocas permitidas sin mejora en validación).

- El experimento busca determinar el punto óptimo de parada automatizada, garantizando que el modelo resultante conserve la máxima capacidad de generalización.

4.5. Selección de Funciones de Activación

Finalmente, la experimentación abordó la elección de la función de activación. Dado que los problemas planteados (y el entorno Atari) requieren manejar valores positivos y negativos, se comparó el comportamiento de la función *Sigmoide* frente a la *Tangente Hiperbólica (Tanh)*.

4.6. Análisis de Resultados Experimentales

En esta sección se presentan los datos cuantitativos obtenidos tras la ejecución de la batería de pruebas. El objetivo es evaluar no solo la precisión final, sino la dinámica de convergencia, la eficacia de la regularización y los límites operativos de la arquitectura implementada.

4.6.1. Resultados de experimentación: Problema del donut

Se evaluó el rendimiento de la red en el problema del "Donut"variando el tamaño del dataset de entrenamiento. Este experimento evidencia la dependencia de la red respecto a la densidad de datos para definir fronteras de decisión curvas.

Muestras	Epochs	Loss Final	Accuracy	Observación
250	16 (Stop)	0.748	66.0 %	Subajuste por falta de datos.
900	97 (Stop)	0.118	94.9 %	**Convergencia exitosa.**
2000	44 (Stop)	0.219	94.1 %	Convergencia rápida y estable.

Cuadro 3: Impacto del tamaño del dataset en la precisión (Problema Donut).

Análisis: Para este problema se configuró una topología profunda de tipo piramidal `{2, 24, 12, 6, 2}`, entrenada con una tasa de aprendizaje conservadora ($\eta = 0{,}01$) y activación `tanh`.

Con 250 muestras, la precisión se estanca en un 66 % (apenas superior al azar). Al aumentar la densidad a 900 muestras, la red dispone de información suficiente para aprovechar su profundidad, saltando al 94.9 % de precisión y demostrando que esta arquitectura requiere un muestreo denso para modelar fronteras cerradas no lineales.

4.6.2. Resultado de experimentación: Problema del Cuadrante

El problema de los cuadrantes validó la capacidad de la red para gestionar múltiples salidas simultáneas (capa de salida de 4 neuronas).

Se validó la capacidad de la red para gestionar múltiples salidas con una topología `{2, 10, 10, 4}` y activación `tanh`, entrenada durante 100 épocas.

Muestras	Loss Final	Accuracy	Hoeffding (P)	Conclusión
200	0.039	98.5 %	26.4 %	Fiabilidad estadística baja.
500	0.018	99.0 %	83.6 %	Convergencia estable.
1000	0.012	99.2 %	98.7 %	**Resultado robusto.**

Cuadro 4: Precisión y garantía estadística según tamaño muestral.

Análisis: La red converge fácilmente (>98 % acierto) incluso con pocos datos. Sin embargo, la métrica de Hoeffding revela que con $N = 200$ el resultado es anecdótico ($P = 26{,}4\,\%$). Es necesario escalar a $N = 1000$ para obtener una precisión del 99.2 % con una garantía teórica de generalización superior al 98 %, confirmando la robustez de la arquitectura.

- **Convergencia:** En todas las pruebas ($N = 200, 500, 1000$), la red superó el 98 % de precisión en menos de 100 épocas.
- **Precisión Asintótica:** Con 1000 muestras, se alcanzó un **99.2 % de acierto** y un error de test (E_{in}) de 0.008.
- **Significancia:** Según la Desigualdad de Hoeffding calculada por el sistema, con $N = 1000$ y $\epsilon = 0{,}05$, la probabilidad de que este resultado sea fiable supera el 98.6 %.

Esto confirma que el cálculo de gradientes para vectores *One-Hot* funciona correctamente.

4.6.3. Resultados de experimentación; Paridad N-bits

Este ha sido el test de estrés principal. Se evaluó la capacidad de la red para aprender la función XOR generalizada para N entradas, un problema de complejidad exponencial 2^N.

Bits	Muestras	Accuracy	Estado	Análisis
2 (XOR)	200	**100 %**	Converge	Resuelto en < 40 épocas.
8	1000	**96.9 %**	Early Stop	Generalización excelente.
10	10000	**99.6 %**	Converge	Límite operativo eficiente.
12	10000	50.2 %	No converge	Estancamiento en meseta.
16	30000	50.1 %	No converge	Azar (Gradiente insuficiente).

Cuadro 5: Degradación del rendimiento al aumentar la complejidad dimensional.

La implementación actual es capaz de resolver problemas de paridad de alta complejidad (hasta 10 bits con casi 100 % de acierto), lo cual es un resultado notable. Sin embargo, se identifica una "barrera de complejidad.[en]tre los 12 y 16 bits. En estos casos, la precisión se estanca en $\approx 50\,\%$, indicando que la red no logra encontrar la dirección del gradiente en un paisaje de error tan complejo. Segun nuestra investigación, creemos que se trata de una limitación del SGD sin momento (Momentum/Adam) en problemas de paridad de alta dimensión.

4.6.4. Validación del mecanismo Early Stopping

Las trazas de ejecución demuestran la activación correcta del sistema de regularización automática. Un ejemplo claro se observa en la ejecución de Paridad 8 bits ($N = 1000$):

> *Epoch 50: Avg Error 0.0239 | patience left: 7*
> *Early Stopping activado en epoca 57. Restaurando mejor modelo (Val Loss: 0.031)*

A pesar de que el error de entrenamiento seguía bajando (llegando a 0.023), el sistema detectó que el error de validación dejó de mejorar. La intervención automática en la época 57 evitó el sobreajuste, permitiendo que el modelo final alcanzara un **96.9 % de precisión en test**. Sin este mecanismo, la red habría memorizado el ruido del dataset, degradando su capacidad predictiva final.

5. Experimentación: Algoritmo Genético (GA)

5.1. Metodología experimental

El objetivo de esta fase es justificar empíricamente el comportamiento del Algoritmo Genético (GA) implementado en `main_ga`, analizando cómo los hiperparámetros principales y la elección de operadores afectan al rendimiento en distintos tipos de problemas. Se sigue una metodología *one-factor-at-a-time*: en cada bloque experimental se modifica **una única variable** y se mantiene el resto fijo.

Problemas evaluados. Se emplean problemas sintéticos y combinatorios con propiedades diferentes: (i) minimización continua unimodal (`sphere`), (ii) minimización continua multimodal (`rastrigin`, y en modo `test` también `ackley`), (iii) maximización binaria (`onemax`), (iv) optimización combinatoria (`tsp` con *random keys*), y (v) clasificación booleana (`xor`, en modo `test`).

Métricas y criterio de parada. El ejecutable reporta cada cierto número de generaciones el mejor/medio/peor individuo y, al finalizar, el `best fitness` y el número de generaciones consumidas. En problemas de **minimización** (`sphere/rastrigin/ackley/tsp`) un fitness **menor** implica mejor solución; en problemas de **maximización** (`onemax/xor`) un fitness **mayor** es mejor. El entrenamiento se detiene al alcanzar el máximo de generaciones o cuando el propio problema/implementación alcanza el objetivo (p.ej. `onemax` alcanza el óptimo antes del máximo).

Reproducibilidad. La semilla (`-seed`) fija la aleatoriedad del GA y, en `tsp`, también el conjunto de ciudades (por tanto, es imprescindible para comparaciones justas). En el *baseline* se usan tres semillas (123, 456, 789).

5.2. Configuración base (baseline) y primera hipótesis

Hipótesis inicial (H0). Con una configuración moderada de población y operadores estándar (alta tasa de cruce y mutación moderada), el GA: (i) converge de manera estable en problemas sencillos (unimodales y binarios), (ii) muestra estancamiento en problemas multimodales (mesetas por óptimos locales), (iii) presenta variabilidad entre semillas en paisajes complejos.

Configuración baseline. Se fija:

```
-pop 80   -gen 300   -elite 1   -cross 0.9   -mut 0.2
```

y se ejecuta con semillas 123/456/789 (cuando aplica).

Problema	Seed	Best fitness final	Generaciones
`sphere` (dim=20)	123	$8{,}2186 \times 10^{-14}$	300
`sphere` (dim=20)	456	$1{,}27455 \times 10^{-10}$	300
`sphere` (dim=20)	789	$2{,}92227 \times 10^{-5}$	300
`rastrigin` (dim=20)	123	17.9093	300
`rastrigin` (dim=20)	456	14.9244	300
`rastrigin` (dim=20)	789	21.8891	300
`onemax` (bits=200)	123	200	145
`onemax` (bits=200)	456	200	147
`onemax` (bits=200)	789	200	166
`tsp` (n=25)	123	500.374	143
`tsp` (n=25)	456	440.4	116
`tsp` (n=25)	789	423.398	118

Cuadro 6: Resultados baseline del GA. En `sphere/rastrigin/tsp` (minimización), valores menores son mejores; en `onemax` (maximización), el óptimo es 200.

Discusión del baseline. Se observa: (i) `onemax` llega al óptimo en todas las semillas y, además, antes del máximo de generaciones, lo que valida el funcionamiento general del GA en binario. (ii) En `rastrigin` el mejor individuo entra en meseta (se mantiene constante durante muchas generaciones), consistente con estancamiento en óptimos locales del paisaje multimodal. (iii) En `sphere` aparece una variabilidad marcada: mientras 123 y 456 alcanzan errores muy pequeños, la seed 789 queda atrapada en un valor notablemente peor, evidenciando sensibilidad a la exploración aleatoria y/o pérdida de diversidad.

5.3. E1 — Tamaño de población (`-pop`)

Hipótesis (H1). Aumentar el tamaño de población incrementa la diversidad y la capacidad de exploración: debería mejorar especialmente en problemas multimodales (`rastrigin`), a costa de mayor coste por generación. En problemas unimodales (`sphere`) también podría acelerar el ajuste fino o reducir estancamientos.

Configuración. Se mantiene:

```
-gen 300   -elite 1   -cross 0.9   -mut 0.2
```

y se varía `-pop` en {20, 40, 80, 160}. (En este bloque se dispone de una única seed: 123.)

Pop	Problema	Best final	Observación
20	`sphere`	0.0116293	Se estanca alrededor de 10^{-2} (exploración insuficiente).
40	`sphere`	0.00339039	Mejora respecto a pop=20, pero aún lejos del óptimo.
80	`sphere`	$8{,}2186 \times 10^{-14}$	Ajuste fino excelente; coincide con baseline seed 123.
160	`sphere`	$4{,}29478 \times 10^{-16}$	Mejor y además converge antes (gen 249).
20	`rastrigin`	57.169	Muy estancado (fitness alto).
40	`rastrigin`	43.4879	Mejora clara frente a pop=20.
80	`rastrigin`	17.9093	Mejor que 40; aparece meseta (óptimos locales).
160	`rastrigin`	8.95463	Mejor de los probados; más exploración reduce el estancamiento.

Cuadro 7: E1: efecto del tamaño de población (seed 123).

Conclusión E1. Los resultados apoyan H1: aumentar población mejora drásticamente `rastrigin` (multimodal) y también mejora `sphere` (unimodal), incluso reduciendo generaciones necesarias en pop=160.

5.4. E2 — Tasa de mutación (`-mut`)

Hipótesis (H2). Una mutación más alta ayuda a escapar de óptimos locales (mejor en `rastrigin`), pero puede introducir demasiado ruido y perjudicar el ajuste fino en `sphere`. Se espera un valor intermedio óptimo.

Configuración. Se fija:

```
-pop 80   -gen 300   -elite 1   -cross 0.9
```

y se varía `-mut` en {0.05, 0.10, 0.20, 0.40}. (Seed 123.)

mut	Problema	Best final	Observación
0.05	`sphere`	$1{,}6058 \times 10^{-8}$	Buena, pero no llega al mejor ajuste fino.
0.10	`sphere`	$2{,}10823 \times 10^{-15}$	Mejor del bloque; converge antes (gen 265).
0.20	`sphere`	$8{,}2186 \times 10^{-14}$	Muy buena, pero peor que mut=0.10.
0.40	`sphere`	$1{,}65003 \times 10^{-6}$	Demasiado ruido; empeora el ajuste fino.
0.05	`rastrigin`	16.9143	Mejor del bloque (junto con 0.10); meseta más baja.
0.10	`rastrigin`	16.9143	Similar a 0.05; mejora frente a mut=0.20.
0.20	`rastrigin`	17.9093	Peor; se estanca más arriba.
0.40	`rastrigin`	16.9148	Similar a 0.05/0.10, pero con medias peores (población más dispersa).

Cuadro 8: E2: efecto de la tasa de mutación (seed 123).

Conclusión E2. Se confirma la existencia de un rango intermedio favorable. En `sphere` el mejor valor es mut=0.10 (equilibrio exploración/explotación); mutaciones muy altas degradan el ajuste fino. En `rastrigin` mutaciones bajas-intermedias (0.05–0.10) resultan mejores que 0.20.

5.5. E3 — Elitismo (`-elite`)

Hipótesis (H3). Un elitismo pequeño acelera convergencia preservando progreso, un elitismo alto reduce diversidad y puede aumentar el estancamiento, especialmente en multimodal (`rastrigin`).

Configuración. Se fija:

```
-pop 80   -gen 300   -cross 0.9   -mut 0.2
```

y se varía `-elite`. (Seed 123.)

elite	Problema	Resultado	Observación
0	`onemax` (bits=100)	óptimo 100 (gen 54)	Converge rápido incluso sin elitismo.
1	`onemax` (bits=100)	óptimo 100 (gen 54)	Similar a elite=0.
5	`onemax` (bits=100)	óptimo 100 (gen 57)	Ligera penalización en generaciones.
10	`onemax` (bits=100)	óptimo 100 (gen 61)	Empeora; penalización en generaciones.
0	`rastrigin` (dim=20)	13.9294	Mejor que baseline; diversidad mayor.
1	`rastrigin` (dim=20)	17.9093	Baseline; meseta.
5	`rastrigin` (dim=20)	44.239	Empeora mucho; convergencia prematura.
10	`rastrigin` (dim=20)	36.8134	Empeora; diversidad insuficiente.

Cuadro 9: E3: efecto del elitismo (seed 123).

Conclusión E3. En `onemax` el elitismo apenas cambia el resultado (problema simple y con gradiente de mejora claro). En `rastrigin` se valida H3: elitismos altos causan convergencia prematura y empeoran notablemente el fitness.

5.6. E4 — Tasa de cruce (`-cross`)

Hipótesis (H4). El cruce facilita recombinación de material genético y acelera convergencia en problemas donde existe "composición" de soluciones (p.ej. `onemax`). En continuo, un cruce demasiado agresivo puede destruir ajustes finos, aunque también puede ayudar si recombina buenos bloques.

Configuración. Se fija:

```
-pop 80   -gen 200/300   -elite 1   -mut 0.2
```

y se varía `-cross` en $\{0.0, 0.5, 1.0\}$. (Seed 123.)

cross	Problema	Resultado	Observación
0.0	`onemax` (bits=200)	óptimo 200 (gen 149)	Sin cruce converge, pero más lento.
0.5	`onemax` (bits=200)	óptimo 200 (gen 87)	Más rápido (mejor recombinación).
1.0	`onemax` (bits=200)	óptimo 200 (gen 94)	Similar a 0.5.
0.0	`sphere` (dim=20)	0.0611451	Mutación sola no alcanza ajuste fino.
0.5	`sphere` (dim=20)	0.00413013	Mejora clara, pero aún lejos del óptimo.
1.0	`sphere` (dim=20)	$2{,}24173 \times 10^{-16}$	Mejor del bloque; cruce constante ayuda a recombinar óptimos parciales.

Cuadro 10: E4: efecto de la tasa de cruce (seed 123).

Conclusión E4. En binario, el cruce reduce generaciones (H4 se cumple). En `sphere` el mejor resultado se obtiene con `cross=1.0`, lo que sugiere que, con esta codificación y operador, la recombinación no destruye el ajuste fino sino que lo potencia al combinar componentes cercanos al óptimo.

5.7. E5 — Comparación de combinaciones de operadores (selección/cruce/mutación)

Hipótesis (H5). La combinación de operadores debe adaptarse al tipo de genoma: (i) en binario funcionan mejor operadores discretos (p.ej. *one-point* + *bitflip*), (ii) en continuo funcionan mejor cruces tipo BLX/Blend y mutaciones gaussianas, (iii) ciertas combinaciones pueden inducir *drift* o convergencia a regiones malas (p.ej. *roulette* en paisajes difíciles).

Resultados en continuo (Rastrigin). Se comparan tres combinaciones sobre `rastrigin` (seed 123):

Selección	Cruce	Mutación	Best final	Gen	Observación
tournament	blend	gauss	17.9093	300	Baseline; meseta (óptimos locales).
roulette	blend	gauss	104.373	170	Muy malo; converge a mala región.
tournament	uniform	gauss	30.3516	81	Se detiene pronto, pero en región subóptima.

Cuadro 11: E5 (continuo): combinaciones de operadores en `rastrigin` (seed 123).

Interpretación. El torneo estabiliza la presión selectiva y evita parte del *drift* observado con *roulette*. Además, *uniform* en continuo puede ser demasiado agresivo y "romper" estructuras útiles gen-a-gen, provocando convergencia prematura a un mínimo mediocre.

5.8. E6 — Validación rápida en modo `test`

El modo `test` ejecuta una batería interna para comparar combinaciones de operadores en varios problemas. En cada caso se reporta: (i) cuántas repeticiones mejoran respecto al mejor individuo inicial (`improved`), (ii) cuántas alcanzan un objetivo si existe (`reached_target`). Configuración: 6 problemas $\times$ 5 combinaciones $\times$ 8 trials (240 ejecuciones).

Problema	Combo (sel, cx, mx)	improved	reached_target
sphere	tournament, blend, gauss	8/8	0/8
sphere	roulette, blend, gauss	8/8	0/8
sphere	tournament, uniform, gauss	8/8	0/8
sphere	tournament, onepoint, bitflip	8/8	0/8
sphere	roulette, uniform, bitflip	0/8	0/8
rastrigin	tournament, blend, gauss	8/8	0/8
rastrigin	roulette, blend, gauss	8/8	0/8
rastrigin	tournament, uniform, gauss	8/8	0/8
rastrigin	tournament, onepoint, bitflip	8/8	0/8
rastrigin	roulette, uniform, bitflip	0/8	0/8
ackley	tournament, blend, gauss	8/8	0/8
ackley	roulette, blend, gauss	8/8	0/8
ackley	tournament, uniform, gauss	8/8	0/8
ackley	tournament, onepoint, bitflip	8/8	0/8
ackley	roulette, uniform, bitflip	0/8	0/8
xor	tournament, blend, gauss	0/8	8/8
xor	roulette, blend, gauss	0/8	8/8
xor	tournament, uniform, gauss	0/8	8/8
xor	tournament, onepoint, bitflip	0/8	8/8
xor	roulette, uniform, bitflip	0/8	8/8
onemax	tournament, blend, gauss	0/8	0/8
onemax	roulette, blend, gauss	0/8	0/8
onemax	tournament, uniform, gauss	8/8	0/8
onemax	tournament, onepoint, bitflip	8/8	8/8
onemax	roulette, uniform, bitflip	8/8	7/8
tsp	tournament, blend, gauss	8/8	0/8
tsp	roulette, blend, gauss	8/8	0/8
tsp	tournament, uniform, gauss	8/8	0/8
tsp	tournament, onepoint, bitflip	8/8	0/8
tsp	roulette, uniform, bitflip	0/8	0/8

Cuadro 12: Modo `test`: resumen por problema y combinación (8 trials por caso).

Lectura crítica del modo test. El `test` confirma patrones consistentes: (i) *roulette + uniform + bitflip* es un combo sistemáticamente malo en problemas continuos y TSP (0/8 mejoras). (ii) Para `onemax`, el combo clásico *tournament + onepoint + bitflip* alcanza el target 8/8, mientras que *roulette + uniform + bitflip* queda en 7/8. (iii) En `xor`, todos los combos alcanzan el objetivo (8/8), lo que indica que el problema es demasiado pequeño para discriminar operadores por `improved`.

5.9. Experimento de "mejora" y resultado negativo

Para aportar una narrativa de iteración basada en evidencias, se intentó "mejorar" el baseline de `rastrigin` incrementando exploración:

```
pop=160, mut=0.4, sel=roulette, cx=blend, mx=gauss
```

La hipótesis era que más población y mutación ayudarían a escapar de óptimos locales. Sin embargo, el resultado fue negativo (fitness empeora):

Seed	Baseline	"Mejorada"	Δ (baseline - mejorada)	Notas
123	17.9093	91.3046	-73.3953	Empeora: el fitness sube (peor solución).
456	14.9244	96.7171	-81.7927	Empeora: pérdida de explotación / deriva.
789	109.385	109.385	0	Salida baseline/mejorada parece duplicada; revisar ejecución/copia.

Cuadro 13: Intento de mejora en `rastrigin`: el aumento agresivo de exploración con *roulette* degrada resultados.

Interpretación del fallo. Este experimento es útil precisamente por ser negativo: muestra que "más exploración" no implica "mejor rendimiento" si la presión selectiva no controla la deriva. La combinación de *roulette* (selección proporcional) con mutación alta puede amplificar ruido y fijar soluciones mediocres, especialmente en paisajes multimodales con muchos mínimos locales.

5.10. Conclusiones generales del bloque de experimentación

En conjunto, los experimentos permiten concluir: (i) el tamaño de población es un factor determinante en multimodalidad (mejora fuerte en `rastrigin`), (ii) existe un rango óptimo de mutación (demasiada mutación destruye ajuste fino), (iii) elitismo alto es perjudicial en multimodal (convergencia prematura), (iv) el cruce acelera problemas "componibles" como `onemax` y puede también ayudar en continuo, (v) la elección de operadores importa: ciertas combinaciones fallan de forma sistemática (p.ej. *roulette+uniform+bitflip* en continuo/TSP), y otras son robustas (*tournament+blend+gauss* en continuo, *tournament+onepoint+bitflip* en binario).

6. Conclusiones

La realización de este proyecto ha servido para comprobar que la teoría matemática es solo el primer paso: para que una Inteligencia Artificial funcione en la práctica, es necesario aplicar soluciones de ingeniería que estabilicen el aprendizaje. Hemos verificado que una red neuronal no aprende correctamente sin ayudas como la inicialización de pesos adecuada o la parada temprana (*Early Stopping*), del mismo modo que un algoritmo genético necesita ajustar sus operadores para no quedarse atascado en soluciones mediocres. Al finalizar, hemos conseguido dos motores robustos, desarrollados completamente por nosotros, que superan las pruebas de dificultad y están listos para aplicarse en el entorno de juego.

7. Referencias

[1] Repositorio del proyecto (GitHub): Atari_assault_NN.

[2] Stanley, K. O., & Miikkulainen, R. (2002). *Efficient Evolution of Neural Network Topologies.*

[3] Y. S. Abu-Mostafa, M. Magdon-Ismail, y H.-T. Lin, *Learning From Data.* AMLBook, 2012.

[4] *Chapter 7: Neural Networks.*

[5] *Google Gemini*

[6] *Wikipedia - Red neuronal artificial*

[7] GeeksforGeeks: *Genetic Algorithms* (información general sobre algoritmos genéticos).

[8] Coding with Thomas: *C++ Genetic Algorithm* (algoritmos genéticos en C++).