# Entrenamiento de un agente de Arkanoid con Algoritmos Genéticos

Victoria Molina M. y José Gabriel T. Perez

***Resumen*—Este trabajo presenta un agente para un juego tipo *Arkanoid*, entrenado directamente en el navegador usando un Algoritmo Genético (AG) sobre una política lineal sencilla. El agente observa un vector de ocho características normalizadas del entorno y, a partir de eso, decide mover la barra a la izquierda, derecha o dejarla quieta. Se explica cómo se codifican los individuos, cuáles son los operadores de selección, cruce y mutación, y cómo se define la función de *fitness*, que combina ladrillos destruidos, puntuación, vidas restantes y tiempo de supervivencia. Luego se muestran resultados experimentales con distintas configuraciones de hiperparámetros, comparando una configuración base con variantes en tamaño de población, número de generaciones y número de episodios por individuo. Finalmente, se comentan las limitaciones del enfoque, qué tan sensibles son los resultados a los hiperparámetros y algunas ideas para trabajo futuro.**

## I. Introducción

Entrenar agentes que jueguen videojuegos clásicos se ha vuelto una forma común de probar ideas de aprendizaje por refuerzo y métodos evolutivos **sutton2018**. En este trabajo se aborda el problema de controlar la barra de un juego tipo Arkanoid, donde el objetivo es destruir la mayor cantidad de ladrillos posible evitando que la pelota caiga fuera de la pantalla.

En lugar de usar redes neuronales profundas o algoritmos de gradiente, se opta por un Algoritmo Genético (AG) clásico **eiben2003**, implementado en JavaScript y ejecutado en el navegador. Cada individuo de la población representa una política determinista que toma las observaciones continuas del juego y las convierte en una acción discreta.

**Problema.** Dado un entorno tipo Arkanoid con filas de ladrillos, una barra controlable y una o varias bolas, se busca encontrar una política que maximice una función de *fitness* que combina: (i) ladrillos destruidos, (ii) puntuación acumulada, (iii) vidas restantes y (iv) pasos sobrevividos.

**Juego elegido.** El entorno implementado es una variante de Arkanoid con:

- Una cuadrícula de ladrillos (filas $\times$ columnas).
- Una barra horizontal móvil en la parte inferior de la pantalla.
- Una pelota principal con velocidad normalizada y, opcionalmente, multibola.
- Dos tipos de *power-ups*: extensión de la barra y multibola, con un máximo de dos *power-ups* activos por partida.

**Contribución principal.** De forma resumida, este trabajo:

- Diseña y prueba una política lineal sencilla (8 características más un umbral de inacción) entrenada con AG en el navegador.
- Define una función de *fitness* adaptada al juego Arkanoid, que trata de equilibrar progreso en el nivel y supervivencia.
- Propone un protocolo experimental reproducible (semillas, configuración en JSON y *logs*) y estudia qué pasa al cambiar hiperparámetros clave como el tamaño de población, las generaciones y los episodios.

## II. Metodología

### II-A. Entorno y observaciones

El juego Arkanoid se configura con una serie de parámetros fijos: ancho y alto de la pantalla, número de filas y columnas de ladrillos, tamaño de la barra, cantidad de vidas, velocidad de la pelota y detalles de los *power-ups*.

En cada paso de la simulación, el agente no ve toda la información cruda del juego, sino un resumen en forma de vector de 8 características normalizadas:

1. Posición $x$ de la pelota principal, normalizada por el ancho de la pantalla.
2. Posición $y$ de la pelota, normalizada por la altura.
3. Componente horizontal de la velocidad de la pelota $v_x$, normalizada.
4. Componente vertical de la velocidad $v_y$, normalizada.
5. Posición $x$ de la barra, normalizada por el ancho.
6. Posición $x$ del ladrillo vivo más cercano (normalizada).
7. Posición $y$ del ladrillo vivo más cercano (normalizada).
8. Distancia normalizada entre la pelota y ese ladrillo cercano.

La idea es que, con estas ocho variables, el agente tenga suficiente contexto para tomar decisiones razonables sin necesidad de observar todo el mapa de ladrillos.

### II-B. Codificación del agente (genes)

Cada individuo del AG representa una política determinista muy simple. La política está definida por:

- Un vector de pesos $\mathbf{w} \in \mathbb{R}^8$.
- Un parámetro escalar de *deadzone* $\delta \geq 0$.

Dada una observación $\mathbf{x} \in \mathbb{R}^8$, la política calcula primero una combinación lineal:

$$y = \sum_{i=1}^{8} w_i x_i.$$

Luego, a partir de ese valor escalar $y$, decide la acción:

$$a = \begin{cases} 0 & \text{si } |y| \leq \delta \quad \text{(no mover la barra)} \\ +1 & \text{si } y > \delta \quad \text{(mover a la derecha)} \\ -1 & \text{si } y < -\delta \quad \text{(mover a la izquierda).} \end{cases}$$

En otras palabras, el genotipo del individuo está formado por los 8 pesos y el umbral de inacción $\delta$. En la población inicial, los pesos se generan de forma uniforme en el intervalo $[-2, 2]$ y $\delta$ se toma de forma uniforme en $[0, 0{,}3]$.

### *II-C. Operadores genéticos*

*II-C1. Selección:* Para la selección de padres se utiliza selección por torneo de tamaño $k = 3$. El procedimiento es el siguiente: para cada padre que se quiere generar, se escogen al azar $k$ individuos de la población y se queda con el que tenga mayor *fitness*.

Este esquema es sencillo de implementar, se adapta bien a paralelización y la presión selectiva se controla cambiando el valor de $k$: con $k$ más grande, se favorecen más a los mejores individuos.

*II-C2. Cruce:* El cruce se hace con un solo punto sobre el vector de pesos:

- Se elige al azar un punto de cruce $p \in \{1, \ldots, 7\}$.
- A partir de ese índice $p$, se intercambian los pesos entre los dos padres para formar los hijos.

Para el parámetro de *deadzone* $\delta$, en lugar de partirlo, se hace algo más simple: cada hijo copia el $\delta$ de uno de los padres con probabilidad $0{,}5$ (como si se lanzara una moneda).

Con probabilidad $p_{\text{cross}}$ se aplica este cruce; si no, los padres pasan directamente a la siguiente generación sin recombinarse.

*II-C3. Mutación:* La mutación se usa para que la población no se quede estancada en una sola zona del espacio de soluciones. En este proyecto, la mutación se aplica directamente sobre los pesos y sobre la *deadzone*.

La idea es la siguiente: para cada individuo, se recorre cada peso $w_i$ y, con una probabilidad $p_{\text{mut}}$, se le suma un pequeño ruido aleatorio. Ese ruido se genera usando una distribución normal (o gaussiana) centrada en cero. En fórmula:

$$w_i^{\text{nuevo}} = w_i^{\text{actual}} + n,$$

donde $n$ es un número aleatorio normalmente pequeño (a veces positivo, a veces negativo). Así, la mayoría de las mutaciones generan cambios suaves en la política, y solo de vez en cuando aparece un cambio más grande. La *deadzone* $\delta$ se muta de forma análoga.

Después de mutar, se aplica un recorte (*clamp*) para que los valores no se salgan de los rangos definidos para el experimento (por ejemplo, mantener los pesos en $[-2, 2]$ y la *deadzone* en un rango pequeño positivo). En los experimentos se usaron desviaciones estándar moderadas para estos ruidos, de manera que la mutación ayude a explorar el espacio de soluciones, pero sin destruir del todo las buenas políticas que ya se han encontrado.

En los experimentos principales se fijan $\sigma_w = 0{,}2$ y $\sigma_\delta = 0{,}05$.

### *II-D. Función de* fitness *y recompensas*

El *fitness* de un individuo no se calcula con una sola partida, sino promediando varias. Para cada episodio (partida), se simula el juego hasta un máximo de $T$ pasos o hasta que el agente se queda sin vidas. Durante la simulación se acumulan:

- Número de ladrillos destruidos en ese episodio.
- Puntuación interna del juego.
- Vidas restantes al final de la partida.
- Número de pasos (frames) que sobrevivió el agente.

Sea $d$ el número de ladrillos destruidos, $B$ el número total de ladrillos del nivel, $R$ la puntuación acumulada, $L$ las vidas restantes, $S$ los pasos jugados y $p = d/B$ el porcentaje de ladrillos destruidos. Con eso, la función de *fitness* por episodio es:

$$f_{\text{ep}} = 30d + 5R + 10L + 50p + \alpha S,$$

donde $\alpha = 0{,}05$ controla cuánto peso se le da al tiempo de supervivencia.

Si el agente destruye todos los ladrillos ($d = B$), se le da además una bonificación extra:

$$f_{\text{ep}} \leftarrow f_{\text{ep}} + 2000.$$

El *fitness* total del individuo se obtiene promediando sobre $E$ episodios:

$$F(\text{ind}) = \frac{1}{E} \sum_{e=1}^{E} f_{\text{ep}}^{(e)}.$$

De esta forma, un individuo es bueno si tiende a destruir muchos ladrillos, hacer buena puntuación, conservar vidas y durar bastante en la partida, y aún mejor si logra limpiar el nivel completo.

### *II-E. Pseudocódigo de evaluación y bucle del AG*

*II-E1. Evaluación del* fitness*:* En el pseudocódigo de la Figura 1 se muestra, a nivel alto, cómo se evalúa a un individuo:

```
func EVALUAR_INDIVIDUO(policy, cfg, baseSeed, E, T):
    totalFitness <- 0
    for ep in 0..E-1:
        seed <- baseSeed + 1000 * ep
        env <- Arkanoid(cfg, seed)
        episodeReward <- 0
        steps <- 0
        for t in 0..T-1:
            obs <- env.observe()
            a <- policy.act(obs)
            (reward, done) <- env.step(a)
            episodeReward <- episodeReward + reward
            steps <- steps + 1
            if done: break
        d <- ladrillos_destruidos(env)
        B <- total_ladrillos(env)
        L <- vidas_restantes(env)
        p <- d / B
        fitnessEp <- 30*d + 5*episodeReward
                     + 10*L + 50*p + 0.05*steps
        if d == B:
            fitnessEp <- fitnessEp + 2000
        totalFitness <- totalFitness + fitnessEp
    return totalFitness / E
```

Figura 1: Pseudocódigo de evaluación del *fitness* por individuo.

*II-E2. Bucle principal del Algoritmo Genético:* El bucle principal del AG se resume en la Figura 2. La idea general es:

- Inicializar una población de $N$ individuos.
- Evaluar su *fitness*.
- Guardar el mejor individuo visto hasta el momento.
- Generar una nueva población aplicando elitismo, selección, cruce y mutación.
- Repetir el ciclo durante $G$ generaciones.

```
func EVOLVE(cfg, N, G, k, pCross, pMut, elit, E, T, seed):
    rng <- MULBERRY32(seed)
    pop <- INIT_POPULATION(N, rng)
    globalBest <- null
    globalBestFit <- -inf

    for gen in 0..G-1:
        fits <- []
        for i in 0..N-1:
            fits[i] <- EVALUAR_INDIVIDUO(pop[i], cfg,
                                         seed + gen*1000 + i,
                                         E, T)
        (bestFit, bestIdx) <- ARGMAX(fits)
        if bestFit > globalBestFit:
            globalBestFit <- bestFit
            globalBest <- CLONAR(pop[bestIdx])

        nextPop <- ELITISMO(pop, fits, elit)
        while |nextPop| < N:
            p1 <- TOURNAMENT(pop, fits, k, rng)
            p2 <- TOURNAMENT(pop, fits, k, rng)
            (c1, c2) <- ONE_POINT_CROSS(p1, p2, pCross, rng)
            c1 <- GAUSSIAN_MUT(c1, pMut, rng)
            c2 <- GAUSSIAN_MUT(c2, pMut, rng)
            nextPop <- nextPop U {c1}
            if |nextPop| < N:
                nextPop <- nextPop U {c2}
        pop <- nextPop
    return globalBest, globalBestFit
```

Figura 2: Pseudocódigo del bucle del Algoritmo Genético.

### *II-F. Protocolo experimental y complejidad*

*II-F1. Configuración base y ablaciones:* Se definió primero una configuración base (**Exp. E0**) con:

- Tamaño de población $N = 30$.
- Número de generaciones $G = 60$.
- Torneo $k = 3$, $p_{\text{cross}} = 0{,}7$, $p_{\text{mut}} = 0{,}1$.
- Elitismo $e = 2$, episodios $E = 2$, horizonte $T = 5000$.
- Semilla global del generador pseudoaleatorio: 1234 (Mulberry32).

A partir de esta configuración base se probaron varias variaciones (ablaciones), cambiando algunos hiperparámetros:

- **E1 (principal):** $N = 50, G = 85, p_{\text{cross}} = 0{,}5, p_{\text{mut}} = 0{,}1, E = 2$.
- **E2:** $N = 85, G = 65, p_{\text{cross}} = 0{,}5, p_{\text{mut}} = 0{,}1, E = 3$.
- **E3:** $N = 75, G = 42, p_{\text{cross}} = 0{,}5, p_{\text{mut}} = 0{,}1, E = 4$.
- **E4:** $N = 100, G = 50, p_{\text{cross}} = 0{,}8, p_{\text{mut}} = 0{,}2, E = 4$.
- **E5:** $N = 100, G = 95, p_{\text{cross}} = 0{,}8, p_{\text{mut}} = 0{,}2, E = 4$.

En todas las corridas se mantuvieron constantes $T = 5000$, torneo $k = 3$, elitismo $e = 2$ y la misma semilla base 1234.

*II-F2. Complejidad temporal y cuello de botella:* Si se denota por $N$ el tamaño de población, por $G$ el número de generaciones, por $E$ el número de episodios por individuo y por $T$ el máximo de pasos por episodio, el coste dominante por generación viene dado por las simulaciones del entorno:

$$\mathcal{O}(N \cdot E \cdot T).$$

Por ejemplo, en la configuración E1 ($N = 50, E = 2, T = 5000$) se tiene:

$$50 \times 2 \times 5000 = 500{,}000 \text{ pasos de simulación por generación.}$$

Los operadores de selección, cruce y mutación son lineales en $N$ y, en la práctica, su coste es pequeño comparado con el de llamar a `env.step` tantas veces.

En los *logs* de E1 se observan tiempos por generación entre 0,15 s y 0,25 s, lo que confirma que el cuello de botella está en la simulación del juego y no tanto en la lógica del AG.

### *II-G. Reproducibilidad*

Cada corrida guarda un archivo JSON con:

- Metadatos (`name`, `created_at`, `run_id`).
- Semilla base (`seed`).
- Hiperparámetros del AG (`N`, `G`, `k`, `pCross`, `pMut`, `elit`, `episodes`, `T`).
- Cadena de agente de usuario del navegador (versión de Chrome y sistema operativo).

Por ejemplo, el archivo `corrida1.json` correspondiente a E1 resume la configuración de esa corrida, incluyendo:

- `N=50`, `G=85`, `pCross=0.5`, `pMut=0.1`, `episodes=2`, `T=5000`, `seed=1234`.
- `browser`: cadena `User-Agent` de Chrome en Windows 10.

En el archivo `best.json` se almacena, para cada corrida, el mejor individuo encontrado (sus pesos y *deadzone*), la generación donde apareció y los parámetros usados. Los *logs* de consola guardan, por generación, el mejor, promedio y peor *fitness*, el tiempo de evaluación y la fracción de ladrillos destruidos.

Para reproducir los experimentos, el procedimiento básico es:

1. Abrir la aplicación en un navegador compatible.
2. Cargar los archivos de configuración deseados.
3. Ejecutar el AG usando la semilla y parámetros correspondientes.

## III. Resultados

### *III-A. Evolución por generación*

En la configuración principal E1 ($N = 50$, $G = 85$), el *log* muestra que en la generación 0 el mejor *fitness* ronda los 913,4 y la media de la población está cerca de 494,7. A partir de ahí, generación tras generación, tanto el mejor individuo como la media van mejorando.

En las últimas generaciones de E1, los mejores individuos superan de forma repetida los 2000 puntos de *fitness*. Al llegar a la generación 84 (la última), E1 alcanza un mejor *fitness* global de 3226,67 y, en ese caso, el agente destruye los $60/60$ ladrillos del nivel, activando la bonificación de fin de nivel.

Si se calcula la media del mejor individuo por generación en E1, se obtiene un valor aproximado de $1782{,}7 \pm 447{,}8$. Esto indica que, en general, la población aprende a jugar mejor con el tiempo y que en muchas generaciones aparece al menos un individuo de muy buena calidad.

### *III-B. Comparación con la configuración base y ablaciones*

La Tabla I resume las configuraciones probadas y el mejor *fitness* global alcanzado en cada una.

Cuadro I: Resumen de configuraciones y mejor *fitness* global.

| Exp. | $N$ | $G$ | $E$ | $p_{\text{cross}}$ | $p_{\text{mut}}$ | Mejor $F$ |
|---|---|---|---|---|---|---|
| E0 (base) | 30 | 60 | 2 | 0.7 | 0.1 | 1784.00 |
| E1 (principal) | 50 | 85 | 2 | 0.5 | 0.1 | 3226.67 |
| E2 | 85 | 65 | 3 | 0.5 | 0.1 | 2167.47 |
| E3 | 75 | 42 | 4 | 0.5 | 0.1 | 2022.73 |
| E4 | 100 | 50 | 4 | 0.8 | 0.2 | 1960.97 |
| E5 | 100 | 95 | 4 | 0.8 | 0.2 | 2980.83 |

Si se toma el conjunto de mejores *fitness* $\{1784{,}00, 3226{,}67, 2167{,}47, 2022{,}73, 1960{,}97, 2980{,}83\}$, la media es aproximadamente $2357{,}1$ con una desviación estándar de unos $596{,}4$. Esto muestra que todas las configuraciones mejoran bastante sobre la base, pero que los resultados dependen bastante de los hiperparámetros.

### *III-C. Baseline y significancia práctica*

Como **baseline** se usa la configuración E0, que representa el AG "por defecto". La configuración E1 introduce dos cambios principales:

- Aumenta el tamaño de la población de 30 a 50.
- Aumenta el número de generaciones de 60 a 85, y baja la tasa de cruce a $0{,}5$.

En términos de *fitness*, E1 mejora la mejor solución encontrada en alrededor de un $80\,\%$ respecto a la base (de $1784{,}0$ a $3226{,}7$). En la práctica, esto significa que:

- El agente tiene más probabilidad de limpiar completamente el nivel.
- La forma de jugar es más estable: destruye más ladrillos por vida y aguanta más tiempo.
- En las últimas generaciones, la población suele tener varios individuos con buen desempeño, no solo uno aislado.

Las configuraciones E2–E5 muestran que:

- Aumentar mucho $N$ y también la tasa de mutación ($p_{\text{mut}} = 0{,}2$) no garantiza mejores resultados y puede hacer más lento el entrenamiento.
- Una configuración muy grande ($N = 100, G = 95$ como en E5) logra soluciones fuertes ($2980{,}83$), pero el coste de cómputo crece bastante.

En conjunto, E1 ofrece un buen equilibrio entre calidad de la solución y tiempo de entrenamiento, por lo que se presenta como la configuración recomendada.

## IV. Discusión

### *IV-A. Limitaciones*

Entre las principales limitaciones del enfoque se pueden mencionar:

- **Política lineal:** La política es únicamente lineal en las características. Esto simplifica la implementación y el análisis, pero limita lo que el agente puede aprender frente a políticas no lineales más expresivas.
- **Observación parcial:** El agente solo ve una pelota (la principal) y un ladrillo cercano. En situaciones con muchas bolas o patrones complicados de rebote, esta información puede no ser suficiente.
- **Costo computacional en el navegador:** Todo se ejecuta en un único hilo de JavaScript. Cuando se hacen muchas simulaciones seguidas, la interfaz puede sentirse menos fluida, incluso aunque se usen pequeñas pausas asíncronas.
- **Función de *fitness* manual:** Los pesos de la función de *fitness* se eligieron a mano. No se estudió de forma sistemática cómo afecta al comportamiento cambiar estos coeficientes.

### *IV-B. Amenazas a la validez*

**Validez interna:**

- Aunque se fija una semilla global, algunos detalles como la generación de *power-ups* o ciertas colisiones pueden depender de la implementación concreta y del navegador, lo que complica reproducir exactamente cada trayectoria.
- Solo se reporta un número limitado de corridas por configuración; para un análisis estadístico más sólido harían falta más repeticiones.

**Validez externa:**

- El agente se entrena y evalúa en un único nivel con parámetros fijos. Si se cambia la distribución de ladrillos o la velocidad de la pelota, el rendimiento puede variar.
- La política está ajustada a esta implementación concreta del juego y no se estudió si se puede transferir a otros juegos o variantes de Arkanoid.

### *IV-C. Sensibilidad a hiperparámetros*

Los resultados indican que el rendimiento del AG es sensible a varios hiperparámetros:

- **Tamaño de población $N$:** Poblaciones demasiado pequeñas pueden llevar a estancamiento; poblaciones muy grandes aumentan el coste de cómputo y no siempre compensan con mejor *fitness*.
- **Número de generaciones $G$:** Más generaciones permiten más exploración, pero los beneficios adicionales se reducen a partir de cierto punto.
- **Número de episodios $E$:** Aumentar $E$ hace que el *fitness* sea menos ruidoso (menos dependiente de una sola partida), pero multiplica el tiempo de evaluación.

- **Tasas de cruce y mutación:** Una mutación alta $(0{,}2)$ puede ayudar a escapar de óptimos locales, pero también puede romper soluciones buenas; una mutación más moderada $(0{,}1)$ parece funcionar mejor en este entorno.

## V. Conclusiones y trabajo futuro

En este trabajo se implementó y evaluó un agente para Arkanoid entrenado mediante un Algoritmo Genético que corre completamente en el navegador. La política se modeló como un sistema lineal con *deadzone*, lo que permite entrenar agentes que juegan razonablemente bien sin requerir demasiado cómputo.

La configuración base del AG ya consigue agentes que destruyen una cantidad considerable de ladrillos, pero los experimentos muestran que aumentar la población y el número de generaciones (configuración E1) mejora claramente el *fitness* global y la probabilidad de limpiar el nivel.

Como posibles líneas de trabajo futuro se plantean:

- Probar políticas no lineales (por ejemplo, redes neuronales pequeñas) entrenadas con AG o con otros algoritmos evolutivos.
- Ajustar automáticamente los pesos de la función de *fitness* usando técnicas de *meta-optimización* o búsqueda de hiperparámetros.
- Incorporar paralelismo en la evaluación (por ejemplo, con *Web Workers*) para acelerar el entrenamiento.
- Evaluar qué tan bien se mantiene el desempeño del agente al cambiar la disposición de ladrillos, las velocidades o las reglas del juego.

## Checklist experimental

La Tabla II resume los puntos principales del *checklist* experimental y cómo se cubren en este trabajo.

Cuadro II: Checklist mínimo de requisitos experimentales.

| Ítem | Descripción en este trabajo |
|---|---|
| Juego y reglas | Arkanoid con barra, ladrillos en rejilla, 3 vidas, velocidad fija de la pelota y 2 tipos de *power-ups* (máx. 2 por partida). |
| Codificación y operadores | Política lineal (8 pesos + *deadzone*); selección por torneo, cruce de un punto, mutación gaussiana, elitismo. |
| Recompensas y *fitness* | Función descrita en la sección de *fitness*, basada en ladrillos destruidos, puntaje, vidas, progreso y pasos. |
| Protocolo experimental | Parámetros $(N, G, E, T, k, p_{\text{cross}}, p_{\text{mut}}, e)$ y semillas descritos en la sección de metodología. |
| Baselines y ablaciones | Configuración base E0 y variantes E1–E5 con cambios en $N$, $G$, $E$ y tasas de cruce/mutación. |
| Estadísticas | Tabla con mejores *fitness*, media y desviación de los mejores valores, y discusión cualitativa de la mejora obtenida. |