Neuroevolución concurrente para el juego de Flappy Bird

Barrero, P., Figueroa, R., Hernández, A., and Padilla, A.
Computación Concurrente - IIMAS
Universidad Nacional Autónoma de México
(Dated: 10 de Febrero del 2021)

Abstract

En este trabajo se presentan los conceptos básicos de los algoritmos bio-inspirados y evolutivos, de los cuales se explican los algoritmos de colonia de hormigas, de banco de peces y de neurevolución así como algunas de sus aplicaciones. Posteriormente se escoge el algoritmo de neuroevolución para aplicarlo al juego de Flappy Bird. Con la aplicación del algoritmo de neuroevolución se realiza su implementación de manera paralela para mejorar su rendimiento y se evalúa la mejora en speed-up comparando la implementación concurrente y la secuencial.

I. INTRODUCCIÓN

No son pocas las dificultades que se presentan en problemas de optimización de gran escala, los gradientes por descenso pueden estancarse en mínimos locales, los espacios de búsqueda pueden ser demasiado grandes para explorarlos exhaustivamente entre muchas otras posibles complicaciones. Por estas razones es que se han desarrollado soluciones que utilizan algoritmos inspirados en los seres vivos y su comportamiento, estos algoritmos imitan algunas de las características de los seres vivos como pueden ser el aprendizaje, la adaptación y la evolución [1].

En general los Algoritmos Evolutivos y Bioinspirados (AEB) pueden tratar de replicar tanto el comportamiento, individual y en colectividad de lo seres vivos, como cualidades físicas y genéticas como puede ser la evolución o las producción de feromonas.

Entre los algoritmos bioinspirados destaca como los científicos de la computación han utilizado el comportamiento de las colonias de abejas, las parvadas de pájaros, los bancos de peces, entre otras colectividades animales para resolver otros tipos de problemas. Este tipo de fenómenos exhiben características como la escalabilidad, la autonomía, la adaptación, el paralelismo y la tolerancia a las fallas que son de gran valor en las soluciones que se proponen con este tipo de algoritmos. Este tipo de algoritmos ha dado origen a la llamada inteligencia de colmena que es un subcampo de la inteligencia artificial [2].

Los algoritmos genéticos son una herramienta que puede aplicarse al aprendizaje automático para encontrar buenas soluciones a problemas que tienen una gran cantidad de posibles soluciones. Esta metaheurística bioinspirada, toma conceptos de los procesos biológicos al generar candidatos a soluciones, los cuales evalúa respecto al resultado deseado y va refinando las mejores soluciones [3].

La idea fundamental es los algoritmos evolutivos es mantener un conjunto de candidatos que representen una posible solución al problema. Estos se mezclan y compiten entre sí, siguiendo el principio de selección natural. Por tanto los individuos más 'aptos' tienen mayor longevidad y por ende mayor probabilidad de generar nuevos candidatos, simulando la reproducción [4].

El rendimiento de las estrategia evolutivas depende de la selección de parámetros así como la evolución de los mismos. Los operadores más importantes para este conjunto de técnicas son la selección, mutación, recombinación y elitismo, siendo el segundo de los mencionados el operador primario en un algoritmo evolutivo, a diferencia de los algoritmos genéticos. Cabe señalar que algunos parámetros pueden estar fijos durante la evolución del algoritmo, a estos se les nombra parámetros exógenos, y por otro lado están los parámetros endógenos que están codificados dentro del mismo individuo [5].

II. ALGORITMOS EVOLUTIVOS Y BIOINSPIRADOS

A. Colonia de hormigas

Los algoritmos de colonias de hormigas o ACO, *Ant Colony Optimization*, están basados en la forma en que las hormigas se organizan para recolectar comida, las cuales dejan caminos de feromonas detrás suyo. Cuando las trabajadoras encuentran comida, estas regresan al nido, reforzando el camino marcado. En caso de que haya dos caminos hacia la comida, las hormigas seleccionaran de manera aleatoria la ruta, sin embargo, al pasar el tiempo, el camino más corto será reforzado, figura 1. Asimismo, otras hormigas se verán más atraídas por las trayectorias que tengan rastros más fuertes de feromonas, mediante este método, la colonia encontrará rápidamente recursos y una gran cantidad de sus trabajadoras llegarán a estos por los caminos más eficientes [6].

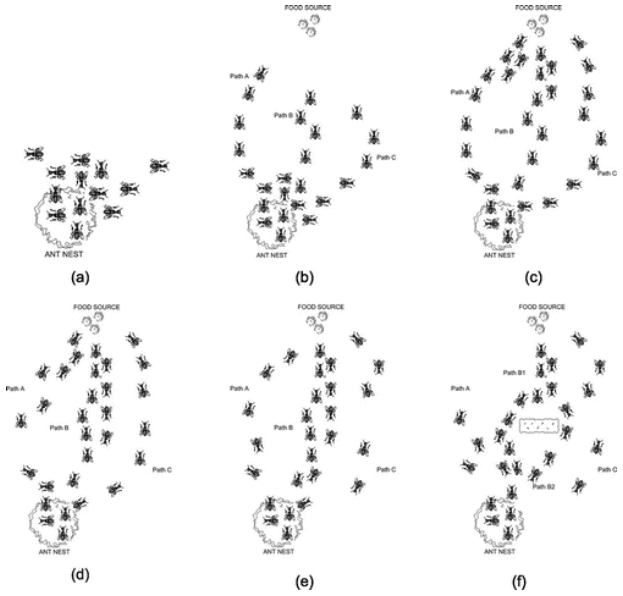


Figura 1: Comportamiento de una colonia de hormigas buscando comida. a) Nido de hormigas b) Las hormigas empiezan a dirigirse a la fuente de alimentos por varios caminos c) Las hormigas llegan a la comida, dejando su rastro de feromonas en el camino d) En el camino más corto existió una mayor concentración de feromonas por lo que algunas hormigas empiezan a preferir ese camino e) Gran parte de las hormigas se van por el camino óptimo f) Las trabajadoras se adaptan a los obstáculos [7].

1. Funcionamiento

La idea general del algoritmo consiste de varias hormigas artificiales que trazarán un camino en un espacio de soluciones en cada iteración, este espacio está compuesto por nodos que representan las soluciones y aristas entre los nodos, dirigidos hacia el objetivo. Las hormigas no pueden visitar de nuevo un nodo, y en cada paso el camino se selecciona a través de un mecanismo estocástico que se encuentra sesgado por una variable que representa a la feromona, la cual ésta asociada a cada arista y que las hormigas pueden leer [8].

```
procedure ACO algorithm for combinatorial optimization problems
    Initialization
    while (termination condition not met) do
        ConstructAntSolutions
        ApplyLocalSearch        % optional
        UpdatePheromones
    end
end ACO algorithm for combinatorial optimization problems
```

Figura 2: Estructura general del algoritmo ACO [8]

En la figura 2 se muestra un esquema general de la metaheurística de la colonia de hormigas como aplicación para problemas estocásticos de combinatoria. Después de inicializar los parámetros y los caminos de feromonas, el ciclo principal se mantiene en los tres pasos esenciales. Primero m hormigas construyen soluciones a la instancia del problema que estemos considerando. Una vez completadas las soluciones, estas deberían mejorarse y finalmente, antes de iniciar la siguiente iteración, los caminos de feromonas se modifican con tal de que reflejen las experiencias de búsqueda de las hormigas [8].

2. Casos de uso

Dada su naturaleza, las estrategias y algoritmos inspirados han encontrado mucha aplicaciones en problemas de combinatoria discreta [9].

- **El problema del agente viajero.**
  EL problema del agente viajero, TSP por sus siglas en inglés, es uno que ha generado mayor cantidad de investigación al tratarse de un problema NP-duro. El TSP también ha jugado un rol importante en el desarrollo de la investigación en ACO, pues el primer algoritmo de colonia de hormigas, llamado *Ant SYstem* fue empleado primero en el problema del agente viajero [8, 10]. En ese primer trabajo, aparte de resultar un algoritmo 'intuitivo' que se adaptaba naturalmente al problema, demostró resultados positivos, sin embargo, resultaron ser inferiores a los algoritmos en un estado del arte de la época. Sin embargo esto dio pauta al desarrollo de algoritmos ACO, con algunos encontrando aplicaciones de la vida real como vigilancia de la vida silvestre [11].

- **Problema de enrutamiento de vehículos.**
  EL problema de enrutamiento de vehículos es uno que está relacionado con el problema del agente viajero. Este consiste en que dado un conjunto de

vehículos y un conjunto de locaciones, se busca un camino que recorra todas las locaciones con un costo mínimo [12].

- **Detección de bordes en imágenes.** Hay artículos donde se ha propuesto el uso de algoritmos de búsqueda de colonia de hormigas, siendo que la imagen se modelo como una gráfica donde cada nodo representaba un pixel. Los experimentos realizados sugirieron la efectividad del uso de esta estrategia [13].

B. Algoritmo de Banco de Peces

El algoritmo de banco de peces toma su inspiración del comportamiento de los pescados, donde cada pescado busca comida por su cuenta de manera individual mientras que cada pez puede comunicarse con otros peces para llegar a una optimización global en la obtención de alimentos. De esta manera se puede construir un algoritmo que explore un espacio de soluciones D dimensional, con N peces. Cada uno de estos pesos tiene una ubicación en el espacio $X = (x_1, x_2, x_3, \cdots, x_D)$, de tal forma que se puede definir una función de aptitud $Y_i = f(X_i)$ que nos representa que tan buen acceso a alimento tiene algún pez dado. Cada pez tiene cierto rango visual y puede comportarse en fases de caza, banco y de seguimiento [2, 14].

1. Funcionamiento

El algoritmo comienza colocando a los N peces en posiciones aleatorias y estos empiezan en la fase de caza. En la fase de caza cada pez selecciona una posición aleatoria X_j en su rango visual y si esta posición aleatoria contiene más alimento $Y_j > Y_i$, entonces el pez se mueve un paso en esta dirección. Esta fase puede describirse como:

$$X_i^{t+1} = \begin{cases} X_i^t + \textbf{paso} \times \frac{X_j - X_i^t}{||X_j - X_i^t||} & Y_j > Y_i \\ \text{movimiento aleatorio} & \text{en otro caso} \end{cases}$$

Donde X_i^{t+1} es la posición al siguiente paso y X_i^t es la posición actual de algún pez dado.

Posteriormente los peces entran en fase de banco, donde evalúan si el punto central c de los peces que están en su alcance visual tiene una mayor concentración de comida que la que tiene él, para esto verifica si $Y_c/N_{\text{vecinos}} > \delta Y_i$, donde δ es un factor de multitud. De tal forma que este comportamiento se puede definir como:

$$X_i^{t+1} = \begin{cases} X_i^t + \textbf{paso} \times \frac{X_c - X_i^t}{||X_c - X_i^t||} & \frac{Y_c}{N_{\text{vecinos}}} > \delta Y_i \\ \text{fase de caza} & \text{en otro caso} \end{cases}$$

Finalmente los peces también pueden comportarse en fase de seguimiento, en la cual se dirigen hacia la posición con mejor cantidad de alimento si puede conseguir si el alimento por pez tiene un mejor rendimiento que el dado por su factor de multitud, es decir, si $Y_{\text{mejor}}/N_{\text{vecinos}} > \delta Y_i$. Con esto en mente esta fase puede describirse como:

$$X_i^{t+1} = \begin{cases} X_i^t + \textbf{paso} \times \frac{X_{\text{mejor}} - X_i^t}{||X_c - X_i^t||} & \frac{Y_{\text{mejor}}}{N_{\text{vecinos}}} > \delta Y_i \\ \text{fase de caza} & \text{en otro caso} \end{cases}$$

Con este algoritmo inspirado en bancos de peces, sus comportamientos individuales y grupales, se puede explorar eficazmente el espacio de soluciones de un problema complejo.

2. Casos de uso

Algunas aplicaciones del algoritmo de banco de peso pueden ser [14]:

- **Planificar trabajo de múltiples robots:** El algoritmo de banco de peces ha sido usado para minimizar la desviación entre el uso de recursos y los niveles objetivo de un grupo de varios robots autónomos. Los resultados de este estudio mostraron buenos resultados, convergencia rápida a la solución, insensibilidad a las condiciones iniciales y una implementación sencilla [15].
- **Optimizar los parámetros de controladores PID:** Los controladores PID son ampliamente usados en la industria y aplicaciones tecnológicas, sin embargo, encontrar los parámetros óptimos para este tipo de controladores no es tarea fácil, por lo que entre las heurísticas que se han utilizado para encontrarlas el utilizar el algoritmo de banco de peces [16].

C. Algoritmo de Neuroevolución.

Los algoritmos de neuroevolución forman parte de la familia de los algoritmos genéticos (AG). En este tipo de AG busca minimizar la función de costo de una red neuronal por métodos alternativos al descenso por gradiente. Para hacer esto se utiliza aprendizaje por refuerzo. Una ventaja de este tipo de algoritmos es que solo requieren de una métrica de rendimiento para cada uno de los miembros de la población que se está simulando [17].

1. Funcionamiento

Un algoritmo genético evoluciona una población P de N individuos. Estos individuos en el caso de la neuroevolución están caracterizados por los pesos de su red neuronal θ, en el contexto de los algoritmos genéticos θ es el genotipo (cromosomas) de cada individuo. El conjunto de cromosomas de cada individuo evoluciona, por

lo que θ_i indica los pesos de la generación i, para cada generación podemos obtener una función de aptitud (fitness) $F(\theta_i)$ la cual indica que tan buen rendimiento tiene un individuo de la generación i para una tarea dada.

Existen varias técnicas de realizar la evolución cromosómica entre generaciones [17, 18], entre ellas se encuentra la evolución entre generaciones que consiste en hacer que los T individuos con mayor aptitud de una generación dada sean los que den origen (sean padres) a los individuos de la generación siguiente. Para generar un hijo se toma a un padre de los T individuos seleccionados en la generación de manera aleatoria y se les aplica a sus cromosomas una función de deriva genética que consiste en agregar ruido gaussiano a sus cromosomas o un subconjunto de estos, de tal forma que los cromosomas del hijo se vuelve $\theta_{i+1} = \theta_i + \sigma\epsilon$, donde σ es un parámetro que se puede determinar de manera empírica mientras que $\epsilon = N(0, I)$ tiene distribución normal, con I la desviación estándar de cada mutación. Otra manera de crear mutaciones es remplazando un subconjunto de los cromosomas con nuevos valores aleatorios. Este proceso se repite para una cantidad deseada $N-u$ de individuos. La siguiente generación puede tener u individuos que tengan exactamente los mismos cromosomas de los mejores individuos de la generación anterior en un proceso llamado elitismo.

Para realizar la cruza entre individuos también existen varias opciones entre las que se encuentra la cruza ingenua lineal, la cual consiste que en lugar de tomar un único padre, podemos tomar dos (a y b) y generar los cromosomas de sus hijos como $\theta_{i+1} = (1-t)\theta_a + t\theta_b$ con t un parámetro que determina que tanto se influirá en la cruza cada padre. Este tipo de cruza se considera no segura y suele llevar a descendencia dañada [19], sin embargo, podría ser útil en redes pequeñas. Otra opción de cruza es simplemente hacer una elección aleatoria de los pesos de los padres de tal forma que los hijos contienen una fracción de los cromosomas de cada padre [18], en este tipo de cruza existen diferentes variantes dependiendo como se eligen los cromosomas de cada padre.

En cuanto a los métodos de selección destacan el método del torneo, en el cual se escogen aleatoriamente k individuos del total de individuos de la población, los cuales serán los concursantes del torneo. Con base a la función de aptitud de cada individuo escogido se seleccionan m individuos con mejor aptitud los cuales son los que ganan el torneo [20]. Otra manera de realizar selección es el método de la ruleta o el método proporcional, en el cual se escogen aleatoriamente k individuos, la suma total de la función de aptitud de estos es sum_{apt}, de tal forma que se les asigna una probabilidad $w_i = \frac{apt_i}{sum_{apt}}$ al $i$ésimo individuo escogido. Con estas probabilidades se construye un segmento de linea con longitudes consecutivas igual a las w_i, de tal forma que se elige posteriormente un número aleatorio entre 0 y 1 y dependiendo de donde caiga este número aleatorio es el individuo que se selecciona para reproducirse [21]. La selección por ruleta se ilustra en la figura 3.

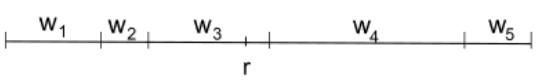


Figura 3: Segmento de linea construido con las probabilidades de selección de cada individuo escogido y el valor r generado aleatoriamente para determinar cual de ellos seleccionar [21]

2. Casos de uso

Algunas aplicaciones notables del algoritmo de neuroevolución son:

- **Simular la dinámica de la caminata bipeda:** Con el algoritmo de neuroevolución se ha logrado entrenar redes neuronales para que puedan controlar a un modelo humanoide de tal manera que este camine de manera veloz sin que pierda el equilibrio [22]. Este tipo de aplicación se ilustra en la figura 4.
- **Conducción autónoma:** Con neuroevolución también se han logrado entrenar redes neuronales para manejar de manera autónoma vehículos automotores [23]. La conducción autónoma es una rama de la investigación científica que ha aumentado de popularidad y una aplicación ingenieril cada vez más tangible e inminente.

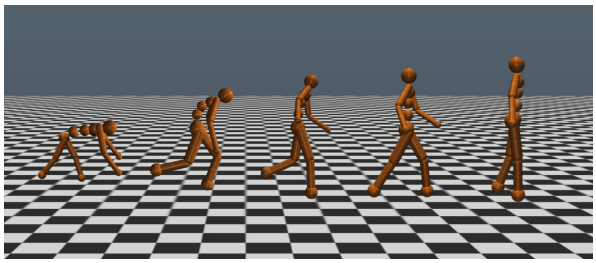


Figura 4: El algoritmo de neuroevolución ha sido usado para enseñarle modelos humanoides a caminar [24]

D. Redes Neuronales

Una red neuronal artificial son un modelo computacional que ha ido evolucionando a través de la historia, y que en los últimos años ha obtenido mucha popularidad debido a la cantidad de problemas que pueden resolver.

Una red neuronal artificial consta de una serie de capaz, cada capa consta de una serie de neuronas y una función de activación. Cada neurona posee un real. La idea de una red es convertir una serie de entradas en una salida.

1. Operaciones genéticas sobre redes neuronales

Codificación y decodificación

La codificación es una forma en representar redes neuronales complejas como un vector unidimensional, para facilitar operaciones como el intercambio o actualización de pesos.

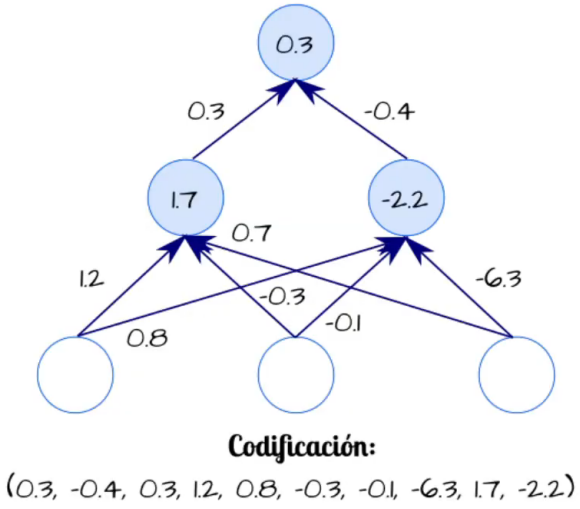


Figura 5: Ejemplo de codificación de una red neuronal [25]

Si las capas de una red neuronal están representadas como un conjunto de pesos en la forma de una matriz bidimensional, y un conjunto de *bias* como un vector de una dimensión, una algoritmo de codificación fácil de implementar es el siguiente:

1. Volvemos todas las matrices de pesos vectores unidimensionales (por ejemplo, concatenando fila por fila, o columna por columna).

2. Concatenamos cada vector de peso con su correspondiente vector de *bias*.

3. Concatenamos todos los vectores resultantes, empezando por la entrada de la red neuronal.

La operación inversa a la codificación se conoce como **decodificación**. Naturalmente, para poder decodificar un cromosoma, se necesitan saber las propiedades del individuo (en el caso del ejemplo anterior el tamaño de cada una de las capas).

Mutación

En esta operación se tiene como entrada un cromosoma y una salida que es un cromosoma también. Para cada entrada de un cromosoma se le suma un pequeño ruido con probabilidad p. En el siguiente ejemplo aplicamos una mutación con $p = 0.1$.

- *Mutación*

(0.3	-0.4	1.2	0.8	-0.3	-0.1	0.7	-6.3)
(0.3	-0.4	1.2	-2.2	-0.3	-0.1	1.2	-6.3)

Figura 6: Ejemplo de una mutación en una red neuronal [25]

Cruza

En esta operación se tienen dos cromosomas A, B del mismo tamaño como entrada a los que les denota como padres y como salida uno o varios cromosomas C. Existen varias cruzas, como por ejemplo:

1. Cruza común
   En esta operación tenemos dos cromosomas A, B como entrada y un cromosoma de salida C, $C[i]$ valdrá lo siguiente:

$$C[i] = \begin{cases} A[i] & \text{si } U[i] < p \\ B[i] & \text{en otro caso} \end{cases}$$

   donde p es la probabilidad de elegir una característica (peso/sesgo) del padre A, y $U[i]$ es una variable aleatoria uniforme entre 0 y 1. Cuando $p = 0.5$ se le conoce como **Cruza Uniforme**.

   En la figura 7 se puede ver un ejemplo de la cruza común.

- *Cruza* (*Crossover*)

(0.3	-0.4	1.2	0.8	-0.3	-0.1	0.7	-6.3)
(-0.7	-0.9	1.3	0.4	1.8	2.1	-0.2	-1.2)
(-0.7	-0.4	1.3	0.4	1.8	2.1	0.7	-6.3)

Figura 7: Ejemplo de una cruza entre dos codificaciones de redes neuronales [25]

2. Cruza de un solo punto
   En esta operación tenemos dos cromosomas A, B como entrada y dos cromosomas de salida C_1, C_2. Tomamos un punto i aleatorio uniforme en los padres, al cual llamaremos punto cruza. En un hijo tomamos la información del padre del elemento 0 hasta el i y del padre B del $i+1$ hasta el el ultimo, en el otro hijo invertimos el orden de los padres.

   En la figura 8 se puede ver un ejemplo de la cruza de un solo punto.

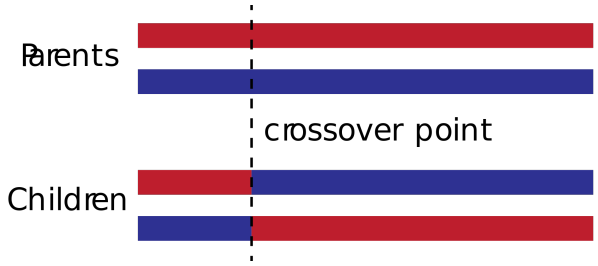


Figura 8: Cruza de un solo punto [26]

3. Cruza de n puntos
   Similar a de un solo punto se seleccionan n puntos y se hace una intercalación de elementos entre cada padre en los dos hijos.

   En la figura 8 se puede ver un ejemplo de la cruza de dos puntos.

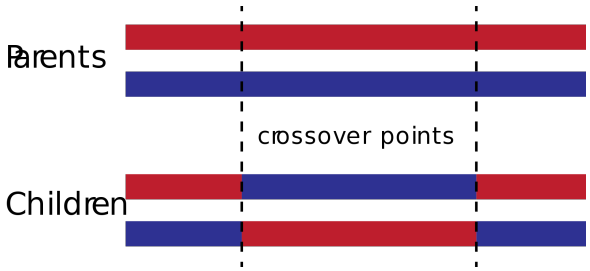


Figura 9: Cruza de dos puntos [26]

III. COMPARACIÓN ENTRE AEB INVESTIGADOS

Las estrategias de optimización de colonias de hormiga son bastante prometedoras cuando son aplicadas a la optimización de un solo objetivo, aunque en ocasiones ha sido superado por algoritmos genéticos más comunes o estrategias de búsqueda más simples [27]. Los algoritmos de colonias de hormigas sufren del mismo problema que otros algoritmos genéticos y bio inspirados. Son eficaces, siempre y cuando se tenga suficiente poder de cómputo o tiempo disponible para resolver un problema dado. Un número suficiente de iteraciones deben cumplirse para la convergencia y el algoritmo puede ser extremadamente complejo de ejecutar [6].

Como se mencionó anteriormente las ventajas más notables del algoritmo de banco de peces es que generalmente converge rápidamente a la solución óptima, que no suele ser sensible a las condiciones iniciales y que es fácil de implementar [15]. Entre las desventajas del algoritmo del banco de peces se encuentra que da malas soluciones a problemas no lineales y multimodales, que si los peces se mueven en dirección aleatoria no se aprovechará tiempo de cómputo para aproximar la solución de manera eficiente y que demasiados movimientos del banco pueden aumentar la complejidad del algoritmo y por lo tanto ralentizar la convergencia a la solución [28].

Una de las grandes ventajas de los algoritmos de neuroevolución es que pueden ser aplicados para entrenar redes neuronales profundas, por lo cual es factible resolver problemas complejos con soluciones no lineales [17, 24]. Otra ventaja es que pueden entrenar redes neuronales más rápidamente que los métodos puramente matemáticos. Entre las desventajas está que debido a que en cada generación no se espera a que todos los individuos converjan hacia la solución óptima y a que las mutaciones son aleatorias en su naturaleza se puede desperdiciar tiempo de cómputo.

Notemos que los algoritmos de neuroevolución sí son capaces de resolver problemas no lineales y que pueden ser utilizados utilizan tecnología que se espera tenga un gran impacto en la realidad inmediata de la sociedad como podría ser la conducción autónoma [24, 29]. Además, los algoritmos de neuroevolución pueden ser aplicados a videojuegos logrando en algunos casos superar el rendimiento de los humanos [30]. Por estas razones es que elegimos trabajar con el algoritmo de neuroevolución en este proyecto y en particular aplicarlo a una versión del videojuego Flappy Bird.

IV. APLICACIÓN DE NEUROEVOLUCIÓN A FLAPPY BIRD

A. Descripción del juego

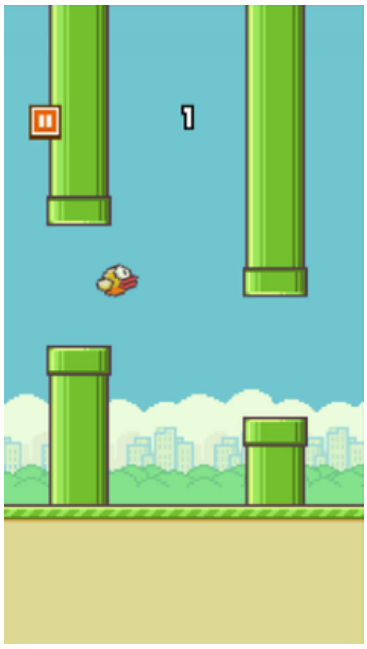


Figura 10: Interfaz del juego de Flappy Bird.

Flappy Bird (figura 10) es un juego para dispositivos

móviles desarrollado por el artista y programador de videojuegos vietnamita Dong Nguyen, bajo su compañía de desarrollo de juegos dotGears. El jugador controla un pájaro que se desplaza hacia la derecha y que adicionalmente va cayendo debido a la gravedad. Periódicamente, el pájaro se va encontrando con pares de tuberías verdes con una apertura en el medio; el objetivo del juego es pasar por dicha apertura, evitando tocar las tuberías o el suelo. Para lograr esto, el usuario puede indicar al pájaro si aletear sus alas o no, lo cual inmediatamente le da una velocidad vertical positiva, que rápidamente empieza a decaer debido a la gravedad.

Debido a que la altura de la apertura de las tuberías cambia, el decidir cuándo aletear es un problema difícil que depende de diversos factores. Por lo tanto, se presta para ser resuelto utilizando algoritmos genéticos.

B. Solución con redes neuronales y algoritmos evolutivos

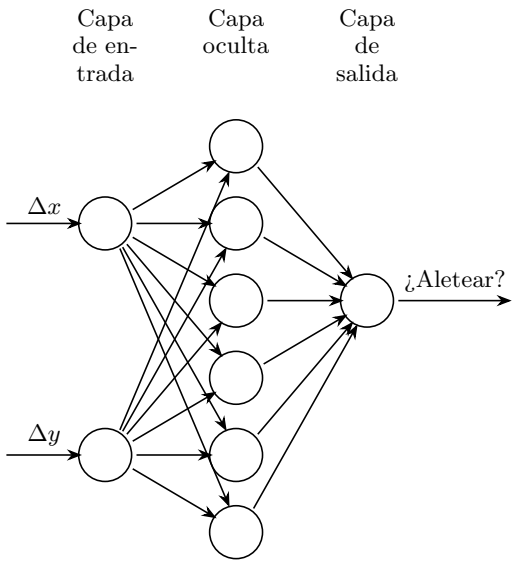


Figura 11: Diagrama de la red neuronal. Δx y Δy representan la distancia horizontal y vertical del pájaro a la apertura, respectivamente.

Para modelar la inteligencia artifical de un pájaro, utilizamos una red neuronal con una capa de entrada, una capa oculta, y una capa de salida. La descripción de cada una de estas capas es la siguiente:

- Capa de entrada: Dos neuronas con función de activación sigmoide, que toman como argumento la distancia vertical y horizontal del pájaro al centro de la apertura entre las tuberías.
- Capa oculta: Seis neuronas con función de activación sigmoide, completamente conectadas a la capa anterior.
- Capa de salida: Una neurona conectada a la capa anterior. Regresa verdadero si su valor es mayor a 0.5, y falso en otro caso.

El diagrama puede verse en la figura 11.

Por otro lado, para entrenar nuestro algoritmo, seguimos los siguientes pasos:

1. Creamos una población inicial de n pájaros con una red neuronal artificial aleatoria para cada pájaro
2. Dejamos que cada pájaro juegue.
3. Terminamos la simulación cuando todos los pájaros estén muertos, o se alcance el tiempo máximo.
4. Producimos la siguiente generación.

Evidentemente, el paso más importante de este método es la adecuada selección de los individuos para obtener la siguiente generación. Como se mencionó en secciones anteriores, cada nueva población está compuesta de tres grupos distintos:

- E: individuos élite, que pasan su genoma sin ningún cambio.
- H: hijos de dos individuos de la población anterior.
- N: individuos "normales", seleccionados aleatoriamente de la generación anterior.

Para escoger los pares de padres de los individuos de la clase H, utilizamos dos algoritmos diferentes de selección

- Selección aleatoria basada en *fitness* (ruleta)
- Selección por torneo

Adicionalmente, mutamos los individuos de las clases H y N escogiendo pesos aleatorios y sumándoles ruido gaussiano de la forma $\mathbf{\Phi}(0, 1)$. Para determinar cuántos pesos mutamos, definimos una tasa de mutación m, la cual determinará la fracción total del genoma a mutar. Es importante notar que m **no** representa el porcentaje de individuos; por ejemplo, si $m = 0.01$ significa que el 1 % del genoma colectivo sufrirá mutaciones, antes de ser distribuido a cada uno de los individuos. Por lo tanto, puede haber algunos pájaros que después de nacer no muten en lo absoluto, y otros que lo hagan bastante.

V. IMPLEMENTACIÓN

A. Elementos

En esta sección damos una breve descripción de las partes más importantes de la implementación en Python de nuestro sistema. Invitamos a revisar el documento disponible en el repositorio del proyecto si se desea una documentación detallada.

1. Redes

Cada capa de las redes neuronales es un objeto de tipo `Layer`, con los siguientes atributos:

- `W`: Arreglo de tamaño $m \times n$, donde m es el número de neuronas en la capa anterior, y n el de la capa actual. Este representa los pesos de la capa.
- `b`: Vector de tamaño n, equivalente a el *bias* de cada neurona.
- `f`: Función de activación de todas las neuronas de la capa.

Para evaluar un vector de entrada v en alguna capa, simplemente calculamos $\texttt{W} \cdot v + \texttt{b}$, y aplicamos `f` a cada uno de los elementos del vector resultante.

Una red neuronal es simplemente un objeto de tipo `NeuralNet` con dos atributos: `layers`, una lista de objetos de tipo `Layer`, y `f`, la función de activación de la última capa.

Para evaluar un vector a través de esta red, lo evaluamos en cada una de sus capas de manera secuencial, y aplicamos `f` al último resultado. Recordemos que en nuestro caso la última capa consiste de una sola neurona, y `f` es la función definida por partes:

$$f(x) = \begin{cases} \text{Verdadero} & x < 0.5 \\ \text{Falso} & x \geq 0.5 \end{cases}$$

Un par de métodos asociados a la clase `NeuralNet` y que usaremos de manera extensa son `encode` y `decode`. Como indican sus nombres, estos se encargan de codificar y decodificar las propiedades de la red, usando el método descrito en el apartado II D 1.

2. Pájaros

Cada pájaro está descrito por un objeto de clase `Bird`. Aparte de guardar los parámetros físicos necesarios para su simulación (posición, velocidad, aceleración de la gravedad, etc.), tiene los siguientes campos:

- `alive`: Si está vivo o no.
- `fitness`: Aptitud actual; cuando el pájaro nace es cero, y va aumentando conforme el pájaro se mueve en el tiempo (más detalles en una sección posterior).
- `network`: Objeto de tipo `NeuralNet`.

El pájaro decide si aletear o no con base en la salida de su red.

3. Tuberías

Las tuberías son representadas con la clase `Pipe`. Para simplificar las entradas a la red neuronal de los pájaros, hacemos que las tuberías sean las que se desplazan horizontalmente, mientras que los pájaros están fijos en $x = 0$. Por lo tanto, cada tubería tiene como atributos su posición y velocidad horizontales.

4. Mundo

Representamos el mundo en el que yacen los pájaros y las tuberías con un objeto de tipo `World`. Este se encarga de mover a todos los elementos en cada instante de tiempo, y de revisar si algún pájaro chocó contra una tubería o contra el suelo, matándolo en caso afirmativo. Adicionalmente, como puede haber múltiples tuberías en pantalla, y dado que estas son destruidas y creadas cuando se salen de los límites del mundo, el objeto `World` informa a los pájaros cuál es la tubería más cercana, para que puedan decidir si aletear o no. Finalmente, después de haber actualizado todos los objetos, recompensa a los pájaros que siguen vivos sumando al atributo `fitness` de cada uno un valor r_a, con una bonificación adicional r_p si lograron pasar una tubería.

B. Simulación

La evolución de una generación de individuos puede describirse de la siguiente manera:

Algoritmo 1 Bucle principal de un mundo.

```
mientras algún pájaro esté vivo o n < pasos_máximos hacer
    mover_pájaros()
    mover_tuberías()
    para p ∈ pájaros hacer
        para t ∈ tuberías hacer
            si colisión(p,t) entonces ▷ Si p y t chocan
                p.alive ← Falso ▷ Matamos al pájaro
            fin si
        fin para
    fin para
    para p ∈ pájaros hacer ▷ Recompensas
        si p está vivo entonces
            p.fitness ← p.fitness + r_a
            si p pasó tubería entonces
                p.fitness ← p.fitness + r_p
            fin si
        fin si
    fin para
    n ← n + 1
fin mientras
```

Donde:

- n: Contador de pasos. El algoritmo acaba cuando todos los pájaros mueren o n es mayor a

`pasos_máximos`.

- `colisión`(p, t): Función que revisa si el pájaro p y la tubería t chocaron. Como el primero es un círculo y el segundo un rectángulo, podemos usar geometría para determinar esto.
- r_a: Recompensa por mantenerse vivo una unidad de tiempo.
- r_p: Recompensa por cruzar una tubería.

Al final de la simulación, cada pájaro tendrá en su atributo `fitness` la *fitness* que alcanzó, con la cual podemos crear una nueva generación.

C. Algoritmo genético

Como mencionamos anteriormente, para producir una nueva generación, debemos de seleccionar tres grupos de individuos diferentes: élites (E), normales (N) e hijos (H). A continuación, describimos cómo escoger cada uno:

- Élites: Dado que por definición los individuos de este grupo son los mejores de la generación, simplemente ordenamos los pájaros por *fitness*, y escogemos los E mejores.
- Normales: Tan sólo tomamos N pájaros de manera aleatoria.
- : Para escoger a cada par de padres, utilizamos uno de los métodos de selección posibles: aleatoria basada en *fitness* (también conocida como ruleta) y por torneos. Escogemos $H/2$ pares diferentes (ya que bajo *crossover* de un solo punto cada par de padres produce un hijo). Adicionalmente, permitimos que los padres estén presentes en más de una pareja.

  Una vez hecho esto, hacemos *crossover* sobre cada pareja.

D. Concurrencia

Naturalmente, el simular muchas generaciones de manera secuencial es un proceso muy tardado, ya que hay que actualizar cada objeto en el mundo. Sin embargo, gracias a la manera en que estructuramos nuestro sistema, el diseñar una implementación paralela es una tarea relativamente sencilla.

Si queremos simular una generación con n pájaros, y asumiendo que nuestro sistema puede ejecutar p procesos en paralelo, tenemos dos opciones:

1. Creamos p mundos, y en cada uno de ellos cargamos $\lfloor n/p \rfloor$ pájaros. Posteriormente, asignamos cada mundo a un proceso diferente y los ejecutamos paralelamente.
2. Creamos n mundos, y en cada uno ponemos un solo pájaro. Ejecutamos los primeros p mundos de manera paralela, luego los siguientes p, y así sucesivamente hasta agotar los mundos disponibles.

En la siguiente sección veremos la diferencia entre estas dos implementaciones, y cuál es más adecuada para nuestro problema.

Afortunadamente, ambos métodos pueden implementarse utilizando la estructura de datos `Pool` del módulo `multiprocessing`. Para hacerlo, recolectamos todos los mundos a simular en una lista (llámese `worlds`) y ejecutamos `pool.map(worlds, play)`, donde `play` es un método de la clase `World` que simula el mundo siguiendo los pasos descritos en el algoritmo 1 y regresa los pájaros una vez terminado.

El objeto `pool` es una mezcla entre los conceptos familiares de cola y *mutex*. Como primer paso, asigna a cada proceso uno de los objetos a analizar (en este caso, nuestros mundos), después de lo cual los procesos ejecutan la función indicada (`play`) sobre ellos. Una vez que todos los procesos hayan terminado (indicado por un *mutex*), colecta los resultados y vuelve a asignar a cada proceso un objeto nuevo.

Otro beneficio importante de usar `Pool` es que el algoritmo puede ser escalado a más pájaros o procesos sin necesidad de hacer ningún cambio en el código.

VI. RESULTADOS

A. Método de evaluación

Antes que nada, hay que determinar el método de evaluación a usar. Dado que procesos como selección o mutación son inherentemente aleatorios (así como las poblaciones iniciales), existe la posibilidad de que algunos individuos de las primeras generaciones presenten un *fitness* más alto de lo esperado. Esto nos impide considerar el *fitness* máximo de una generación como métrica, ya que puede no ser representativo de la población completa. Adicionalmente, dado que estos individuos sobrevivirán mucho más que sus hermanos (al tener un *fitness* mayor), el tiempo de ejecución de una generación entera tampoco es una buena medida de eficiencia.

A pesar de estas diferencias aleatorias, entre más vayan evolucionando las poblaciones, más irá aumentando el *fitness* **promedio** de cada generación, ya que los individuos menos aptos serán reemplazados por unos más especializados. Por lo tanto, proponemos usar como métrica de evaluación el tiempo que tarda en alcanzar una población un cierto *fitness*. De esta manera, garantizamos que se estén considerando las propiedades de todos los individuos.

B. Parámetros predeterminados

En los apartados siguientes, empezaremos haciendo una comparación entre distintas implementaciones concurrentes de nuestro algoritmo, para lo cual necesitamos ejecutarlo. Por lo tanto, establecemos los siguientes parámetros base:

- Total de pájaros: 40
- Número de procesos (a menos que se especifique lo contrario): 4
- Elitismo : 10 %
- Tasa de *crossover*: 80 %
- Tasa de mutación: 3 %
- Método de seleción: Torneo
- Participantes en torneo: 8

Incluimos estos parámetros aquí para tener un estándar para comparar los distintos resultados.

C. Distribución de trabajo

Comparamos los dos métodos de simulación mencionados en el apartado V D; como recordatorio (tomando nuestra población de 40 pájaros y 4 procesos) estos son:

- Método 1: Crear 4 mundos de 10 pájaros cada uno, y ejecutar cada mundo de manera paralela.
- Método 2: Crear 40 mundos de 1 pájaro cada uno y ejecutar mundos de cuatro en cuatro de manera paralela.

Los resultados pueden verse en la figura 12.

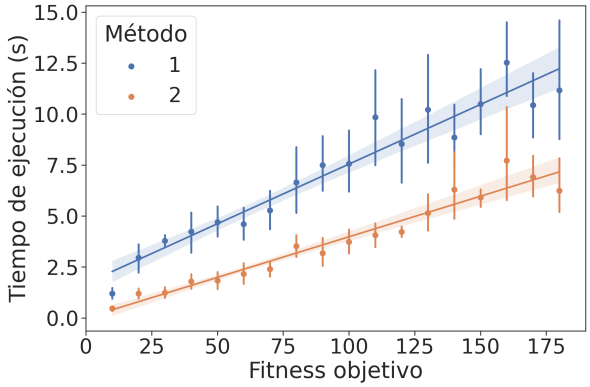
Como puede notarse, el método dos es mucho más rápido que el uno, con una tasa de aceleración de hasta 4 veces en comparación. Esto se debe a que los procesos del método 1 deben de esperar hasta que todos los pájaros de los demás procesos acaben de ejecutarse. Por lo tanto, si un proceso termina muy rápido (debido a que sus pájaros tenían una *fitness* muy baja) tiene que estar en espera de los demás procesos. Esto no ocurre con el segundo método, ya que al ser mundos con un sólo pájaro, el tiempo que están en espera los procesos es mucho menor.

Es interesante ver que la tasa de aceleración decrece conforme va aumentando el *fitness* objetivo. Podemos explicar esto recordando que un *fitness* promedio mayor inherentemente requiere de más generaciones, lo cual a su vez implica que cada vez habrá menos diferencias entre cada uno de los mundos asignados a los distintos procesos. Por lo tanto, cada proceso pasará menos tiempo en espera de que acaben los demás, ya que sus poblaciones serán aproximadamente equivalentes. A pesar de esto, el segundo método tiene un desempeño hasta dos veces mejor que el primero, por lo cual será el que utilizaremos en todos los análisis posteriores.

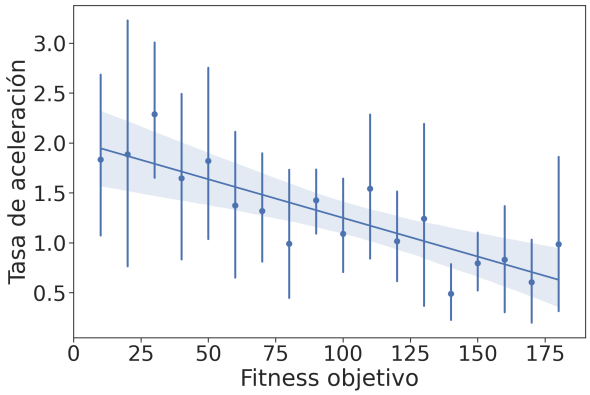
D. Concurrencia

Una vez determinado la mejor forma de distribución de trabajo, el siguiente paso lógico es observar qué ocurre si utilizamos un número distinto de procesos. Los resultados para un proceso secuencial y cuatro procesos ejecutados de manera concurrente pueden verse en la figura 13.

Si bien para las primeras generaciones prácticamente no hay diferencia (ya que para este punto los tiempos de ejecución son muy pequeños para aprovecharse de la concurrencia) conforme va incrementando el objetivo se va apreciando una diferencia cada vez más significativa entre los tiempos. El hecho de que la recta en la figura 13b sea creciente sirve para confirmar este hecho, lo cual nos dice que nuestra implementación efectivamente se beneficia de la ejecución paralela, con tasas de aceleración de casi el 100 % para los *fitness* objetivos más grandes.

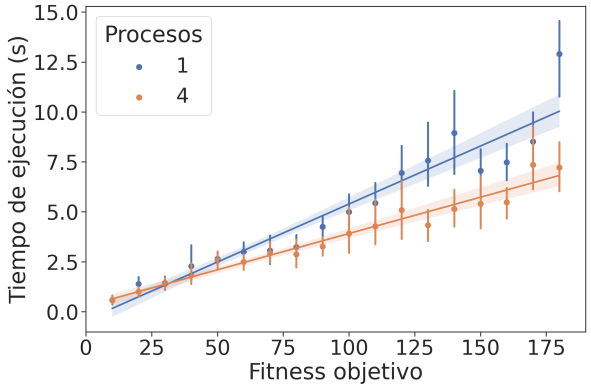


(a) Tiempos de ejecución.

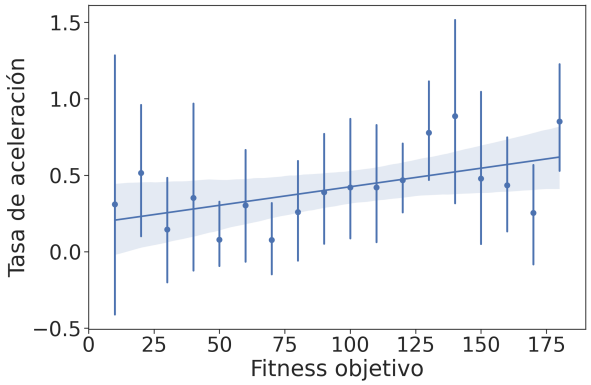


(b) Tasa de aceleración ($\frac{\text{tiempo método 1}}{\text{tiempo método 2}} - 1$)

Figura 12: Tiempos de ejecución y tasas de aceleración promediadas sobre $n = 7$ ejecuciones para distintos métodos. Se muestran ajustes lineales con intervalos de confianza del 95 %.



(a) Tiempos de ejecución.



(b) Tasa de aceleración ($\frac{\text{tiempo 1 proceso}}{\text{tiempo 4 procesos}} - 1$)

Figura 13: Tiempos de ejecución y tasas de aceleración promediadas sobre $n = 7$ ejecuciones para distinto número de procesos.

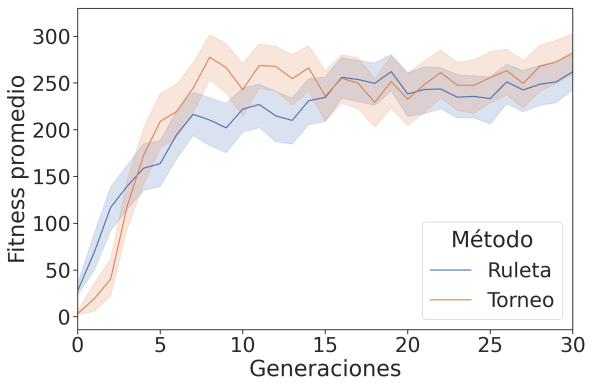


Figura 14: Fitness por generación para ambos métodos de selección.

E. Métodos de selección

Como mencionamos previamente, tenemos la posibilidad de usar dos métodos diferentes al momento de seleccionar los padres. En la figura 14, representamos el *fitness* promedio de cada generación con ambos métodos.

Podemos notar que el método de torneo converge un poco más rápido que el de ruleta. Adicionalmente, presenta "picos" menos pronunciados en su curva. Estos picos se deben a ruido estocástico producido por mutaciones y reproducción, y son fundamentales para mantener una población diversa. Sin embargo, no deben de ser muy grandes, ya que es posible que el *fitness* decrezca de manera inesperada. El algoritmo de torneo evita este fenómeno comparando múltiples individuos entre sí, lo cual hace que sea menos probable escoger uno con un *fitness* bajo. En el apartado siguiente, discutimos los efectos que tiene el número de individuos sobre el desempeño de las poblaciones.

F. Participantes

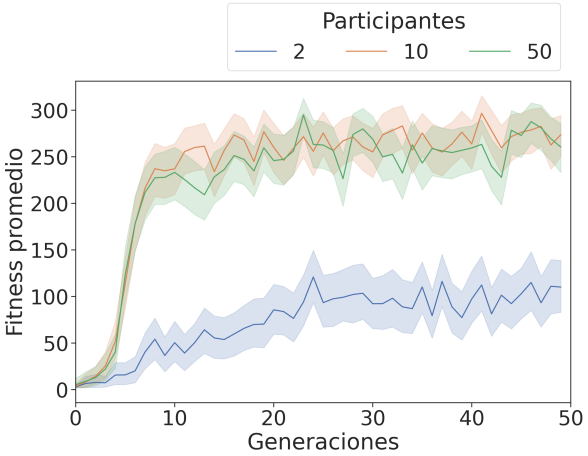


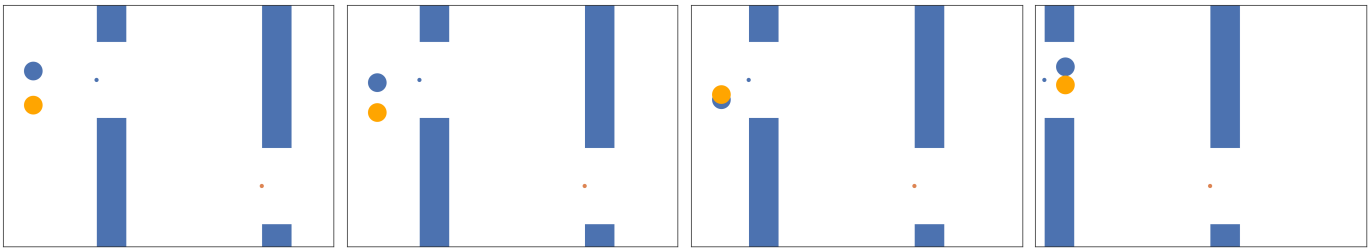
Figura 15: Fitness por generación para distinta cantidad de participantes..

Para el método de selección por torneo, el número de participantes nos ayuda a ajustar la llamada *presión de selección* [31]. La razón es que con torneos más grandes, los individuos débiles tienen menos probabilidad de ser seleccionados, ya que si un individuo débil es escogido para participar en un torneo, es más probable que un individuo fuerte se encuentre en el mismo torneo.

El encontrar un balance adecuado entre individuos fuertes y débiles es vital para que nuestras poblaciones evolucionen de la manera deseada: si el torneo es muy pequeño, se colarán muchos individuos débiles. Por otro lado, si es muy grande los individuos fuertes siempre dominarán, y no habrá suficiente variabilidad genética.

Por lo tanto, realizamos simulaciones para distinto número de participantes, y calculamos el *fitness* promedio para cada generación. Los resultados pueden verse en la figura 15.

Como era de esperarse, el *fitness* para el torneo con tan sólo 2 participantes es muy bajo, ya que prácticamente no hay presión de selección. Por otro lado, si bien los torneos de 10 y 50 individuos presentan resultados aproximada-



(a) Cuadro 0: Posición inicial. (b) Cuadro 7: La neurona de naranja se activa y hace que aleteé. (c) Cuadro 12: La neurona de azul se activa. (d) Cuadro 28: Ambos pasan el obstáculo

Figura 16: Cuadros selectos de una población experta de dos pájaros, representados por círculos de colores. Los puntos en las aperturas de las tuberías (rectángulos) representan la ubicación que usan los pájaros para calcular su distancia.

mente equivalentes, el último sufre de más decrementos bruscos en el *fitness* promedio en comparación al primero. Esto es ocasionado por una falta de diversidad en la población, lo cual es causado a su vez por una presión de selección demasiado grande. Por lo tanto, nos quedamos con un parámetro de 10 participantes por torneo.

G. Población de ejemplo

Con los parámetros obtenidos anteriormente, entrenamos de manera eficiente a una población de pájaros. Posteriormente, creamos un nuevo mundo con los dos mejores candidatos, y observamos su evolución conforme pasa el tiempo. Renderizaciones de algunas interacciones importantes pueden verse en la figura 16.

Como punto final, notamos que esta población se mantuvo viva por más de 20 minutos, y solo dejaron de jugar porque suspendimos manualmente la ejecución.

VII. CONCLUSIÓN

En este trabajo se hace un breve repaso de los algoritmos bio-inspirados y evolutivos, de los cuales de tres se discute su funcionamiento y algunos de sus casos de uso. Entre estos tres se hace un breve comparativo para justificar la elección del algoritmo de neuroevolución para la elaboración de este proyecto. Para introducir más al lector en el algoritmo de neuroevolución se explican los conceptos básicos de las redes neuronales y como estos se desarrollan con los algoritmos evolutivos.

Nuestro proyecto comprueba que mediante la implementación de un algoritmo de neuroevolución de forma paralela podemos disminuir el tiempo de ejecución del mismo programa de forma secuencial.

De acuerdo a nuestras gráficas (Figura 13) podemos ver que que en la búsqueda de una aptitud de 25 o superior podemos hacerlo en menos tiempo de forma paralela (con 4 procesos) que de forma secuencial (usando 1 proceso).

Por otro lado, mostramos que es posible entrenar redes neuronales con un método distinto al convencional (gradiente de descenso) utilizando el algoritmo de neuroevolución. Esto se puede comprobar mediante las gráficas de la Figura 12 y 13 puesto que el fitness es cada vez mayor, asimismo se puede ver de forma experimental visualizando una generación avanzada, puesto que un pajaro puede jugar correctamente como se muestra en la figura 16.

[1] K. Krishnanand, S. K. Nayak, B. K. Panigrahi, and P. K. Rout, "Comparative study of five bio-inspired evolutionary optimization techniques," in *2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*, pp. 1231–1236, IEEE, 2009.

[2] A. Darwish, "Bio-inspired computing: Algorithms review, deep analysis, and the scope of applications," *Future Computing and Informatics Journal*, vol. 3, no. 2, pp. 231–246, 2018.

[3] C. Sheppard, *Genetic algorithms with python*. Smashwords Edition, 2017.

[4] A. M. Andaluz, "Algoritmos evolutivos y algoritmos genéticos,"

[5] E. V. Cuevas Jiménez, J. V. Osuna Enciso, D. A. Oliva Navarro, and M. A. Díaz Cortés, *Optimización. Algoritmos Programados con MATLAB*. Alfaomega Grupo Editor, S.A. de C.V. México, 2016.

[6] X. Fan, W. Sayers, S. Zhang, Z. Han, L. Ren, and H. Chizari, "Review and classification of bio-inspired algorithms and their applications," *Journal of Bionic Engineering*, vol. 17, pp. 611–631, 2020.

[7] S. Christodoulou, "Scheduling resource-constrained projects with ant colony optimization artificial agents," *Journal of computing in civil engineering*, vol. 24, no. 1, pp. 45–55, 2010.

[8] M. Dorigo and T. Stützle, "Ant colony optimization: overview and recent advances," *Handbook of metaheuristics*, pp. 311–351, 2019.

[9] M. Dorigo and T. Stützle, *The Ant Colony Optimization Metaheuristic*, pp. 25–64. 2004.

[10] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, 1996.

[11] S. Chowdhury, M. Marufuzzaman, H. Tunc, L. Bian, and W. Bullington, "A modified ant colony optimization algorithm to solve a dynamic traveling salesman problem: A case study with drones for wildlife surveillance," *Journal of Computational Design and Engineering*, vol. 6, no. 3, pp. 368–386, 2019.

[12] I. S. Bonilha, M. Mavrovouniotis, F. M. Müller, G. Ellinas, and M. Polycarpou, "Ant colony optimization with heuristic repair for the dynamic vehicle routing problem," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 313–320, IEEE, 2020.

[13] H. Nezamabadi-Pour, S. Saryazdi, and E. Rashedi, "Edge detection using ant algorithms," *Soft Computing*, vol. 10, no. 7, pp. 623–628, 2006.

[14] M. Neshat, A. Adeli, G. Sepidnam, M. Sargolzaei, and A. N. Toosi, "A review of artificial fish swarm optimization methods and applications.," *International Journal on Smart Sensing & Intelligent Systems*, vol. 5, no. 1, 2012.

[15] W. Tian and J. Liu, "An improved artificial fish swarm algorithm for multi robot task scheduling," in *2009 Fifth International Conference on Natural Computation*, vol. 4, pp. 127–130, IEEE, 2009.

[16] Y. Luo, W. Wei, and S. xin Wang, "Optimization of pid controller parameters based on an improved artificial fish swarm algorithm," in *Third International Workshop on Advanced Computational Intelligence*, pp. 328–332, IEEE, 2010.

[17] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv preprint arXiv:1712.06567*, 2017.

[18] D. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms," 1989.

[19] T. Uriot and D. Izzo, "Safe crossover of neural networks through neuron alignment," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pp. 435–443, 2020.

[20] T. Blickle and L. Thiele, "A comparison of selection schemes used in evolutionary algorithms," *Evolutionary Computation*, vol. 4, no. 4, pp. 361–394, 1996.

[21] A. Lipowski and D. Lipowska, "Roulette-wheel selection via stochastic acceptance," *Physica A: Statistical Mechanics and its Applications*, vol. 391, no. 6, pp. 2193–2196, 2012.

[22] B. Jackson and A. Channon, "Neuroevolution of humanoids that walk further and faster with robust gaits," in *Artificial Life Conference Proceedings*, pp. 543–550, MIT Press, 2019.

[23] G. Sainath, S. Vignesh, S. Siddarth, and G. Suganya, "Application of neuroevolution in autonomous cars,"

[24] K. O. Stanley and J. Clune, "Welcoming the era of deep neuroevolution," *Uber Engineering*, 2017. Revisado el 08 de enero del 2021 en `https://eng.uber.com/deep-neuroevolution/`.

[25] V. E. Arriola-Rios, "Clase de redes neuronales," 2020. Consultado el 10 de febrero del 2020 en `https://www.youtube.com/watch?v=lzubLqAXKZU&list=PLdFhsX0YpsnuJziN1v9DpHkmIT_iPCjxp&ab_channel=VeronicaE.Arriola-Rios`.

[26] "Crossover (genetic algorithm)," Consultado el 10 de febrero del 2020 en `https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)`.

[27] M. López-Ibánez and T. Stützle, "Automatic configuration of multi-objective aco algorithms," in *International Conference on Swarm Intelligence*, pp. 95–106, Springer, 2010.

[28] Z. Peng, K. Dong, H. Yin, and Y. Bai, "Modification of fish swarm algorithm based on levy flight and firefly behavior," *Computational intelligence and neuroscience*, vol. 2018, 2018.

[29] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*, vol. 1. MIT press Cambridge, 2016.

[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[31] B. Miller and D. Goldberg, "Genetic algorithms, tournament selection, and the effects of noise," *Complex Syst.*, vol. 9, 1995.