Escuela Técnica Superior de Ingenieros Informáticos

Universidad Politécnica de Madrid

Sistema evolutivo híbrido para la construcción de Redes de Neuronas

Trabajo Fin de Máster

Máster Universitario en Inteligencia Artificial

AUTOR: Carlos Vázquez Losada
TUTOR: Daniel Manrique Gamo

20 de julio de 2019

Agradecimientos

Deseo expresar mi agradecimiento, en primer lugar, a Daniel, un investigador incansable y un profesor excepcional. Gracias por dedicarme tu tiempo y adaptarte a mis horarios, y gracias por despertar en mi el interés en la Computación Evolutiva con tus clases y con tu implicación.

A mis padres, Isabel Losada y Fco. Javier Vázquez, por su amor, comprensión y por su apoyo incondicional en todo aquello que me he propuesto.

A Cristina, espectadora de mis éxios y mi acompañante. Siempre sabes que decir y cómo complementarme. Te quiero.

Resumen

Este Trabajo de Fin de Máster consiste en la construcción de un sistema evolutivo para la generación de Redes de Neuronas Artificiales. Concretamente, se estudia si es posible entrenar parcialmente las Redes de Neuronas Artificiales, en lugar de un entrenamiento completo, para el cálculo del grado de adaptación.

La Programación Genética es una técnica evolutiva que se utiliza en problemas de optimización cuyas soluciones son programas informáticos. La Programación Genética Guiada por Gramáticas extiende las posibilidades de la Programación Genética tradicional con la introducción de las gramáticas, que permiten crear individuos sintácticamente válidos.

Una Gramática Libre de Contexto permite definir arquitecturas de Redes de Neuronas Artificiales válidas dado cualquier número de neuronas en la capa de entrada y en la capa de salida. El resultado de la ejecución del programa genético devuelve la arquitectura de red que mejor se adapte al problema dado. El grado de adaptación de una arquitectura neuronal se obtiene en base a su error en el proceso de entrenamiento. Este trabajo aborda la posibilidad de que este entrenamiento se realice de forma parcial con el fin de disminuir considerablemente la carga computacional de la técnica evolutiva utilizada.

Summary

This Master Thesis Dissertation consists in a research about how appropiate is to use Evolutionary Computation for creating Artificial Neural Networks. A technique of Evolutionary Computation is used for this purpose; Grammar-Guided Genetic Programming. It will be used for looking the best Artificial Neural Network for a concrete dataset, by checking both partially and fully trained networks.

Genetic Programming is an evolutionary technique (it is inspired by biology as Evolutionary Computation does) and it is used for solving optimization problems whose solution is a computer program. The Grammar-Guided Genetic Programming extends Genetic Programming by adding grammars, that allow to create valid individuals.

These grammars allow creating Neural Network architectures that are all valid, given any number of neurons in both input and output layers. The result of an execution would be the architecture that best fixes (in terms of accuracy) the given data.

1. Introducción

La optimización matemática estudia un tipo concreto de problemas donde se desea elegir el mejor de entre un conjunto de elementos. El problema clásico (Boyd and Vandenberghe, 2004) consiste en maximizar o minimizar una función objetivo que represente o mida la calidad de las decisiones. Además, esta función está sujeta a un conjunto de restricciones que acotan el espacio de soluciones.

Para resolver estos problemas, existen algoritmos de optimización, métodos iterativos y heurísticas. Uno de los primeros algoritmos de optimización es el algoritmo de Simplex (Dantzig, 1990). Los métodos iterativos buscan la convergencia hacia una solución determinada. Un ejemplo es el Método de Newton (Nocedal et al., 1999). Las heurísticas (Polya, 1945), en cambio, aproximan la solución.

La Computación Evolutiva es una familia de heurísticas inspiradas en la propia evolución biológica de los seres vivos para la resolución de problemas de búsqueda y optimización. En concreto, la Programación Genética (Koza, 1992) surge por la necesidad de extender la optimización para involucrar programas informáticos. En la Computación Evolutiva, se evolucionan poblaciones de individuos que codifican soluciones candidatas para un problema. Como un individuo codifica una solución, hay individuos mejores que otras. Esta caracterización se determina según el grado de adaptación del mismo, de tal forma que a mayor grado de adaptación, mejor solución al problema.

La Programación Genética Guiada por Gramáticas (Whigham, 1995) surge con la finalidad de resolver el problema de la Progamación Genética de creación de individuos no válidos mediante la adición de Gramáticas Libres de Contexto. Estas gramáticas permiten la generación de individuos como lo haría la Programación Genética, pero asegurando que todos los individuos son sintácticamente válidos, ahorrando tiempo y esfuerzo de computación en la reparación y/o creación de otros nuevos como sucede sin el uso de la gramática.

Las Redes de Neuronas Artificiales son técnicas de Aprendizaje Automático (Samuel, 1959) inspiradas en el sistema nervioso de los seres vivos. El objetivo de este modelo es la resolución de problemas mediante el aprendizaje. Son especialmente utilizadas en los ámbitos de Visión Artificial (Roberts, 1965) y en el Reconocimiento de Voz (Waibel et al., 1989), aunque también son ampliamente usadas para problemas de clasificación a partir de conjuntos de datos, que es en lo que se centra este trabajo.

La Computación Evolutiva y las Redes de Neuronas Artificiales, ambas inspiradas en la biología, no son opuestas sino que se complementan. La Computación Evolutiva permite resolver problemas de optimización y búsqueda, mientras que las Redes de Neuronas Artificiales son indicadas para el Aprendizaje Automático. La

Programación Genética y la Programación Genética Guiada por Gramáticas son una alternativa a la búsqueda de la mejor Red de Neuronas Artificiales, ya que la selección de la arquitectura se basa tradicionalmente en la experiencia previa, y en la preba y el error.

El objetivo principal de este trabajo es la construcción de Redes de Neuronas Artificiales con Programación Genética Guiada por Gramáticas. En este proceso, el entrenamiento es necesario para obtener el grado de adaptación de cada red candidata. Dado que el entrenamiento es costoso, se plantea la posibilidad de entrenar a los individuos de forma parcial. Se establecen varios subobjetivos:

1. Implementar un Programa Genético Guiado por Gramáticas que construya Redes de Neuronas y donde el grado de adaptación de los individuos que representan redes neuronales se calcula en base a su error en el proceso de entrenamiento.

2. Establecer, para cada problema, dos modos de entrenamiento: uno parcial y otro total, con las mismas características de ejecución.

3. Realizar estudios comparativos de ambos modelos evolutivos: con entrenamiento parcial y total.

Este documento se estructura en siete secciones después de esta introducción. Las dos primeras se centran en la *Computación Evolutiva* y en las *Redes de Neuronas Artificiales*, respectivamente. La tercera sección trata sobre la *construcción de Redes de Neuronas*, seguida del *planteamiento del problema*, en la que se describe con profundidad el propósito de este trabajo y la necesidad real de él. A esta sección le sigue la *solución propuesta*, que contiene la metodología y el procedimiento seguido, exponiendo los *resultados* obtenidos en la séptima sección. Finalmente, el trabajo finaliza con las *conclusiones y líneas futuras* propuestas.

2. Computación Evolutiva

La Computación Evolutiva comprende un conjunto de técnicas para la resolución de problemas de búsqueda y optimización inspiradas en la evolución biológica de los seres vivos. En este capítulo se ofrece, en primer lugar, un resumen de la *historia* de la Computación Evolutiva. Seguidamente, un apartado de *funcionamiento general* sobre esta rama de estudio y que finaliza hablando sobre la *Programación Genética* y la *Programación Genética Guiada por Gramáticas*.

2.1. Historia

La Computación Evolutiva surge con los trabajos de Box (Box, 1957), Friedberg (Friedberg, 1958, 1959) y Bremermann (Bremermann, 1962). Sin embargo, no se consiguieron grandes avances debido a la pobre metodología todavía sin desarrollar y a las limitaciones computacionales de la época.

Algunos años después surgen los primeros desarrollos metodológicos en una década de destacable logro científico. El trabajo de Fogel (Fogel et al., 1966) sienta las bases de la Programación Evolutiva (*evolutionary programming*) y el de Holland (Holland, 1967) las de los Algoritmos Genéticos (*genetic algorithms*). También, en esa misma época, las Estrategias Evolutivas (*evolution strategies*) fueron introducidas por Rechenberg (Rechenberg, 1965) y Schwefel (Schwefel, 1965).

Posteriormente, en los años 80, los avances computacionales permitieron aplicar las técnicas evolutivas descubiertas para resolver problemas de optimización del mundo real. En esa misma década, los estudios de Cramer (Cramer, 1985) y Koza (Koza, 1988) desembocaron en la aparición de una nueva técnica perteneciente a la Computación Evolutiva: la Programación Genética (*genetic programming*). Pocos años después, la Programación Genética contaba con más de 10.000 artículos publicados (Hu et al., 2014), convirtiéndose en una heurística destacada en el ámbito académico y empresarial por versatilidad. También se sucedieron una gran cantidad de conferencias internacionales y talleres centrados en aspectos teóricos de los Algoritmos Genéticos (Grefenstette, 1985, 1987; Schaffer, 1989), entre muchos otros. Estos años destacaron también por la aparición de otras técnicas de Computación Evolutiva como la Vida Artificial (*artificial life*) abreviada muy comúnmente como *A-Life* (Langton, 1986), los Sistemas Inmunitarios Artificiales (*artificial immune systems*) (Farmer et al., 1986), la Inteligencia de Enjambre (*swarm intelligence*) (Beni and Wang, 1989) y los Algoritmos Meméticos (Moscato, 1989).

Hacia 1990, era innegable admitir que parte de la comunidad científica del mundo ponía sus ojos en la Computación Evolutiva. A las conferencias anteriormente

expuestas le siguieron las cuatro conferencias de IEEE sobre Computación Evolutiva, que asentaron esta rama de estudio como una de las articulaciones de la Inteligencia Artificial y herramienta indispensable para la resolución de problemas de optimización y búsqueda en el ámbito académico y empresarial. La conferencia de Programación Genética (Koza et al., 1996) tuvo gran éxito y aceptación, a la cual le siguieron otras conferencias sobre el mismo ámbito, como la EuroGP (Banzhaf, 1998). Destacaron también conferencias sobre desarrollos metodológicos sobre los Algoritmos Genéticos (Rawlins, 1991; Whitley, 1993; Vose, 1995), que se consolidó como la técnica de Computación Evolutiva más destacable. Esta década se cierra con la llegada de nuevas técnicas de Computación Evolutiva: los Algoritmos Culturales (*cultural algorithms*) (Reynolds, 1994), la Evolución Diferencial (*differential evolution*) (Storn, 1996; Storn and Price, 1997) y la Evolución Gramatical (*grammatical evolution*) (Ryan et al., 1998), además de la aparición de una variante de la Programación Genética, la Programación Genética Guiada por Gramáticas (Whigham, 1995)

A estas casi cuatro décadas de logro científico le siguieron dos más de un impecable desarrollo metodológico. En estos años destacaron los estudios sobre los Algoritmos Genéticos (Deb et al., 2002; Hassan et al., 2005; Pezzella et al., 2008), entre muchos otros, además de la aparición de nuevas heurísticas como la Búsqueda Armónica (*harmony search*) (Geem et al., 2001) o los Algoritmos Genéticos Basados en Humanos (*human based genetic algorithms*) (Kosorukoff, 2001). Otras técnicas evolutivas también tuvieron su amplio desarrollo metodológico y crecimiento teórico (Couchet and Manrique, 2006; De Araujo and Tavares, 2014; Beni, 2004).

2.2. Funcionamiento general

La Computación Evolutiva se fundamenta en la evolución biológica y cuenta con un conjunto de heurísticas para la resolución de problemas de búsqueda y optimización. La evolución biológica se basa en la Teoría de la Evolución de Darwin (Darwin, 1959), donde los individuos son los protagonistas. Estos individuos viven en poblaciones, de tal forma que se produce una evolución conjunta de la población mediante la aplicación de operadores evolutivos tales como la selección, el reemplazo, el cruce y la mutación.

Para la Computación Evolutiva, los individuos representan las soluciones candidatas al problema de búsqueda u optimización dado. La población se compone del conjunto de individuos sobre la que se aplican los operadores evolutivos (referidos en este ámbito como operadores genéticos) anteriormente mencionados, con el objetivo de que se produzca una mejora y un acercamiento gradual hacia la solución óptima en cada generación. Teóricamente, se espera que tras un número finito de generaciones, la población haya convergido hacia la solución óptima al problema dado. En la práctica, puede que sea conveniente limitar este número de generaciones por las capacidades computacionales limitadas que hoy en día existen, y la población llegue hasta un cierto grado de optimalidad.

Este proceso se lleva a cabo conformando el Ciclo Evolutivo (Koza, 1992) que se repite hasta que finalmente se cumple la condición de parada.

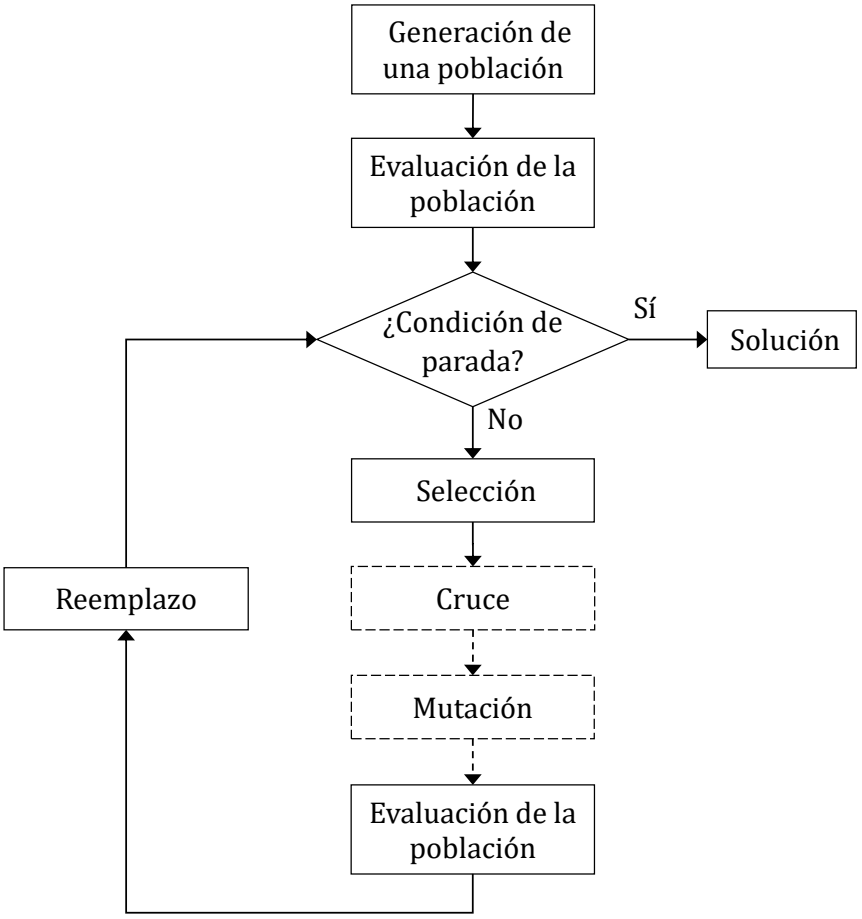


Fig. 1: Ciclo Evolutivo que rige el desarrollo de los algoritmos en la Computación Evolutiva.

En la Fig. 1 se puede apreciar este Ciclo Evolutivo. Todo comienza con una población inicial, que se corresponde con un conjunto de individuos de tamaño α perteneciente a un espacio de búsqueda E, que codifica soluciones de un espacio S. Un individuo codifica una solución mediante un alfabeto A y una función de codificación Ω. La elección del alfabeto A es fundamental, ya que de él depende que haya individuos del espacio S que no tengan representación en el espacio E, y si es una solución potencialmente buena (incluyendo la óptima), entonces el proceso evolutivo nunca la encuentra.

Tras la creación de la población inicial (a partir del espacio de búsqueda E), se evalúa, por medio de una función de evaluación (*fitness function*), a sus individuos para determinar su grado de adaptación, es decir, la calidad de la solución que representa para el problema en cuestión. Para este proceso de evaluación, es necesario decodificar a los individuos, aplicando la función Ω.

Tras el proceso de generación de la población inicial y su evaluación, se determina si se cumple alguna condición de parada y en caso contrario, continúa. Algunas condiciones de parada pueden ser:

1. Algún individuo de la población representa la solución óptima al problema.
2. La población ha convergido (a una solución óptima o no).
3. Se ha alcanzado el número máximo de generaciones.

En el primer caso, la ejecución finaliza devolviendo el individuo óptimo y, por tanto, encontrando la solución. Si la población converge sin haber encontrado antes un individuo óptimo, entonces finaliza devolviendo cualesquiera de los individuos de la misma. Si no se cumple ninguna de estas dos condiciones, entonces se comprueba si se han superado el número de generaciones máximas determinadas por el usuario. En caso afirmativo, finaliza devolviendo el mejor individuo de la población. En el caso de que no se cumpla ninguna condición de parada, el algoritmo continúa.

Después se aplican los operadores genéticos de selección (*selection*), cruce (*crossover*) y mutación (*mutation*) sobre los individuos. Tras aplicar estos tres operadores, se evalúan los nuevos individuos, se aplica el reemplazo y se comprueba si se cumple alguna condición de parada. El proceso se repite hasta que se cumpla alguna condición de parada.

Los principales operadores genéticos son:

- Selección. Es el operador encargado de elegir a los individuos de la población que se van a cruzar para generar otros nuevos. El fundamento radica que en la naturaleza, aquellos individuos mejor adaptados tienen una probabilidad de reproducción mayor frente aquellos que no lo estén. Por tanto, una implementación tradicional consiste en seleccionar con mayor probabilidad a los mejor adaptados, pero hay otras elitistas que seleccionan solo a los π mejor adaptados.
- Cruce. Los individuos seleccionados por el operador anterior acceden al *mating pool*. Estos individuos reciben el nombre de progenitores (*progenitors*), que se cruzan con cierta probabilidad para formar su descendencia (*offspring*). Normalmente, los progenitores se cruzan por parejas, conformando 2 descendientes. Este operador se puede aplicar solo sobre los individuos seleccionados de la población y permite generar otros individuos nuevos.
- Mutación. Con cierta probabilidad (normalmente muy baja), un individuo de la población experimenta cambios aleatorios en su genoma. Esto permite aumentar la diversidad de la población al generar otra posible solución potencial.
- Reemplazo. Este operador selecciona a los α individuos que formarán parte de la siguiente generación si no se cumple alguna condición de parada después

de aplicar los operadores anteriores. Hay muchas políticas de reemplazo, pero el fin común es producir la mejora gradual de los individuos generación a generación y alcanzar la convergencia.

Se puede apreciar que los operadores genéticos resultan indispensables para el desarrollo y mejora de la población y su llegada hacia la solución óptima. En concreto, el cruce es el máximo responsable del proceso evolutivo, ya que es el encargado de generar nueva descendencia, tratando de seleccionar los mejores genes de cada progenitor.

Según los operadores que se implementen, la técnica evolutiva tiende a explorar o explotar más el espacio de búsqueda. La exploración consiste en muestrear regiones desconocidas en el espacio de búsqueda donde pueden encontrarse las soluciones óptimas. La explotación trata de mejorar al individuo más destacado, acortando la búsqueda de la solución óptima en una búsqueda local. Por tanto, estos operadores (y en especial el operador de cruce) se deben comportar de forma explotativa en las primeras generaciones, ya que hay una gran diversidad y de otra forma se produce una convergencia muy lenta. Sin embargo, a medida que la población va convergiendo hacia una solución, es necesario que los operadores exploren, es decir, tengan la posibilidad de dirigirse hacia otras regiones de búsqueda para evitar la caida en óptimos no globales.

2.3. Programación Genética

La Programación Genética (Cramer, 1985; Koza, 1988) es una heurística de la familia de la Computación Evolutiva que permite resolver problemas de búsqueda y optimización cuya solución es un programa informático. La Programación Genética es similar a los Algoritmos Genéticos (Holland, 1967), donde el espacio de soluciones está compuesto por programas informáticos.

2.3.1. Codificación y representación de los individuos

Los individuos en la Programación Genética se representan mediante árboles, formados por operadores o funciones y operandos. Los operadores son los nodos intermedios, mientras que los operandos son los nodos hoja. Esta representación permite la codificación de programas informáticos. En la Fig. 2 se ejemplifican dos individuos en forma de árbol y su codificación.

La creación de los individuos debe cumplir dos propiedades. La primera de ellas se refiere a la propiedad del cierre (*closure property*), que evita la creación de individuos sintácticamente no válidos, estableciendo que la salida de una función debe poder servir de entrada para otra. La segunda propiedad es la explosión de código

(*code bloat*) (Tackett, 1995), que recoge el crecimiento desmesurado del código sin mejora de la adaptación, y, por lo tanto, debe evitarse en la medida de lo posible. La explosión de código propicia el aparecimiento de intrones (*introns*), que son fragmentos de código totalmente inútiles y que no favorecen la adaptación y que se traduce en una convergencia lenta y una demora en la evaluación de los individuos. El alfabeto A y la función de codificación Ω debe buscar en la medida de lo posible la biyección entre el espacio de soluciones y el espacio de búsqueda y que sea completo, es decir, que se puedan codificar todas las soluciones.

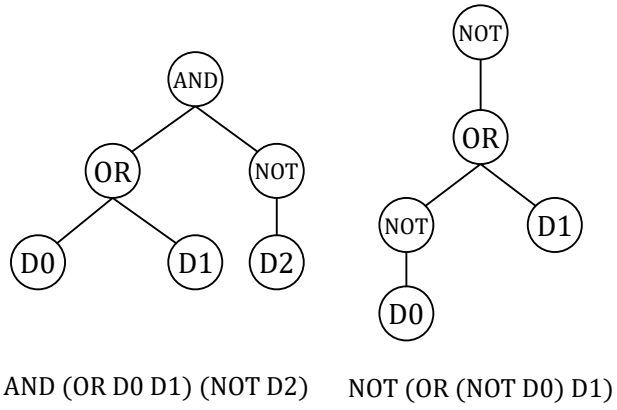


Fig. 2: Individuos de un programa genético codificados en forma de árbol y su correspondiente decodificación.

2.3.2. Generación de la población inicial

El programa genético debe partir de una población de individuos inicial, que es un subconjunto del espacio de búsqueda. La metodología de creación de individuos en Programación Genética suele ser aleatoria.

Si se generan individuos de forma aleatoria, hay una gran probabilidad de que no se cumpla la propiedad del cierre y que haya explosión de código. Por tanto, es necesario un proceso de reparación de individuos. Normalmente, se basan en la sustitución de nodos mal colocados por otros correctos de forma aleatoria, como se muestra en la Fig. 3.

Este proceso de reparación muchas veces es más lento que la generación de un nuevo individuo, por lo que en la práctica los individuos inválidos se descartan y se sustituyen por otros, que si son nuevamente no válidos se descartan, repitiendo ese proceso hasta que se alcanza una población del tamaño deseado, con todos sus individuos siendo sintácticamente válidos.

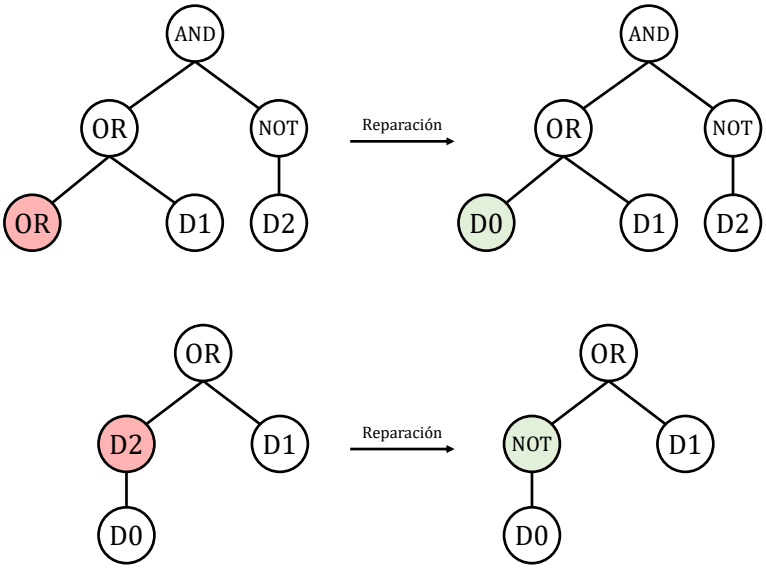


Fig. 3: Ejemplo de aplicación de la función de reparación de individuos inválidos.

2.3.3. Operadores

La Programación Genética cuenta con cuatro operadores: selección, cruce, mutación y reemplazo. Aquellos que no modifican a los individuos son el de selección y el de reemplazo, por lo que funcionan de forma análoga a como hacen en otras heurísticas como en los Algoritmos Genéticos o en la Evolución Gramatical, por lo que solo se explica el operador de cruce y el de mutación.

2.3.2.1. Cruce

El operador de cruce se aplica con cierta probabilidad sobre los individuos que han sido seleccionados y que se encuentran en el mating pool, por tanto, no se aplica en todos los individuos. El cruce debe asegurar que se cumple la propiedad del cierre y que no haya explosión de código.

Operador de Koza

El operador de cruce de Koza (Koza, 1992) selecciona un nodo de cada padre de forma aleatoria y se intercambian los subárboles. Este operador no asegura que se cumpla la propiedad del cierre, además, como en el ejemplo mostrado en la Fig. 4, se corre el riesgo de que se produzcan intrones como consecuencia de la explosión de código, ya que no cuenta con un mecanismo de control de profundidad.

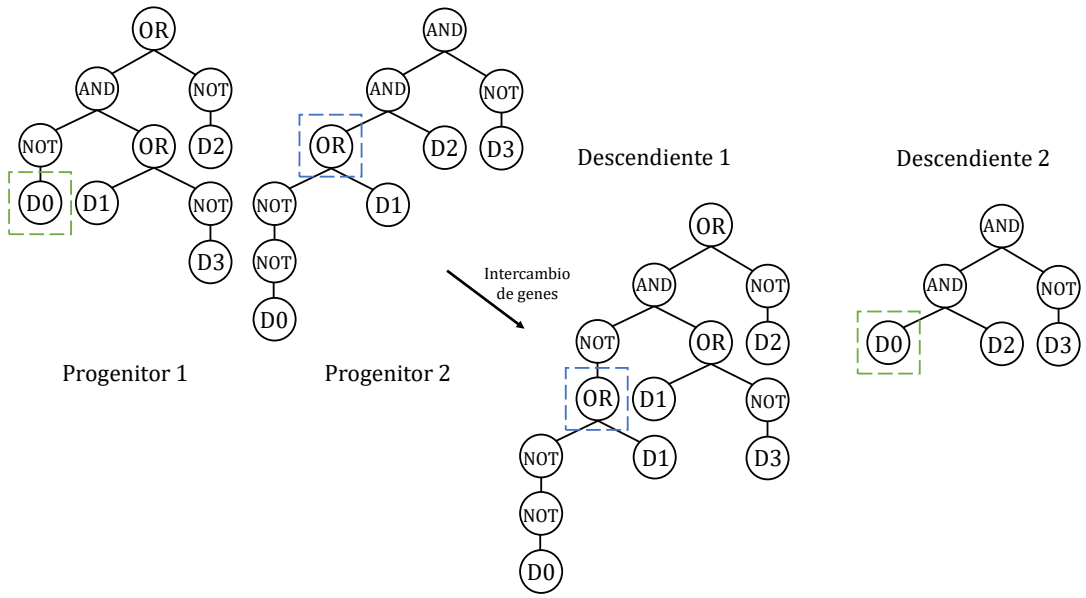


Fig. 4: Ejemplo de aplicación del operador de cruce de Koza.

Operador basado en un punto

El operador de cruce basado en un punto (Poli and Langdon, 1997a, 1997b, 1998) de la Programación Genética se inspira en el de los Algoritmos Genéticos (Goldberg, 1989). Este operador selecciona el nodo de cruce después de un proceso de alineamiento (*alignment*) entre los padres, donde se seleccionan los nodos potenciales para realizar el intercambio.

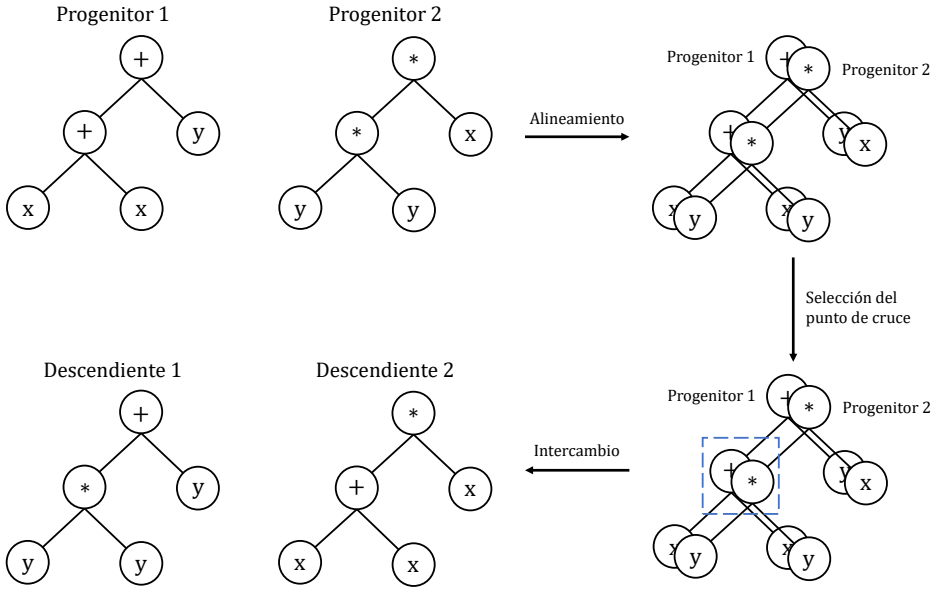


Fig. 5: Ejemplo de aplicación del operador de cruce basado en un punto.

Este operador selecciona un nodo de forma aleatoria de entre los coincidentes entre los dos padres. El problema que tiene este operador es que si la estructura de los padres es totalmente distinta, entonces el único símbolo coincidente entre los padres será el símbolo raíz, produciendo entonces una descendencia idéntica a los progenitores, por lo que se puede dar una convergencia prematura a un óptimo local.

2.3.2.2. Mutación

El operador de mutación se aplica con una muy baja probabilidad a los individuos seleccionados y a su descendencia. Debido a la baja probabilidad, puede que haya generaciones en las que no haya mutación. De hecho, hay programas genéticos que no utilizan este operador. Este operador selecciona un nodo de forma aleatoria en el individuo seleccionado, y a continuación, genera un subárbol a partir de ese nodo seleccionado de forma análoga a como se inicializan las poblaciones.

2.4. Programación Genética Guiada por Gramáticas

La Programación Genética Guiada por Gramáticas (Whigham, 1995) surge por las dificultades encontradas en el desarrollo de los programas genéticos en relación con la propiedad del cierre. En concreto, la aleatoriedad en la creación de individuos resulta una dificultad cuando los problemas son grandes y la cantidad de símbolos es elevada, pues hay más probabilidades de generar un individuo no válido (Vanneschi et al., 2010). Además, la función de reparación o la sustitución de individuos no válidos por otros bien formados aumenta la complejidad computacional.

La Programación Genética Guiada por Gramáticas evita los problemas de la creación de individuos no válidos gracias a la utilización de Gramáticas Libres de Contexto (*context-free grammars*) (Chomsky, 1959), que permiten que las soluciones que codifican los individuos formen parte del espacio de soluciones aunque sean generados de forma aleatoria.

2.4.1. Gramáticas Libres de Contexto

Una Gramática Libre de Contexto G está definida por una 4-tupla (Hopcroft and Ullman, 1979) de la forma:

$$G = (\Sigma_N, \Sigma_T, S, P), \tag{1}$$

donde Σ_N se corresponde con el conjunto de símbolos no terminales (variables), Σ_T con el conjunto de símbolos terminales (constantes), S es el axioma o símbolo inicial, del que derivan las reglas que forman al individuo y P es el conjunto de reglas de producción.

Una regla de producción (*production rule*) de P tienen la forma $\alpha \to \beta$, donde α es un símbolo no terminal, $\alpha \in \Sigma_N$, y β es una cadena de variables o terminales con $\beta \in (\Sigma_N \cup \Sigma_T)^*$.

Una derivación de la gramática G es un conjunto de reglas de producción $\alpha \to^* s$, $s \in \Sigma_T^*$, donde s es palabra, que se corresponde con la decodificación del individuo en el espacio de soluciones.

La Programación Genética Guiada por Gramáticas asegura que los individuos generados son siempre válidos porque siguen la sintaxis definida en la gramática. Sin embargo, pueden existir reglas de producción recursivas, por lo que puede haber explosión de código.

Otra característica de estas gramáticas es que puede haber ambigüedad, es decir, puede haber dos individuos distintos pero que deriven la misma palabra. Por ejemplo, la siguiente gramática es ambigua:

$$
\begin{aligned}
G &= (\Sigma_N, \Sigma_T, S, P) \\
\Sigma_N &= \{S, E, F, N\} \\
\Sigma_T &= \{0,1,2,3,4,5,6,7,8,9,+,-,=\} \\
P &= \{ \\
&\quad S ::= E = N \\
&\quad E ::= E + E | E - E | F + E | F - E | N \\
&\quad F ::= N \\
&\quad N ::= 0|1|2|3|4|5|6|7|8|9 \\
&\}
\end{aligned}
\tag{2}
$$

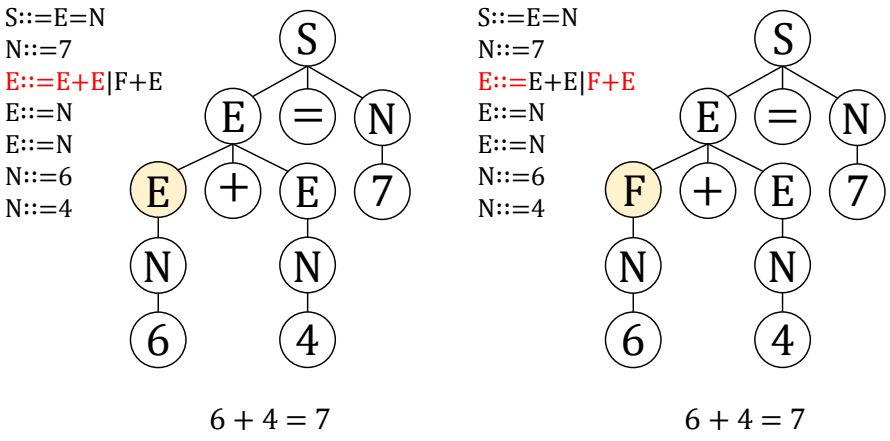


Fig. 6: Individuos conformados por una gramática redundante.

La Fig. 6 muestra dos individuos diferentes que representan la misma palabra.

Se puede comprobar que los símbolos no terminales F y E son equivalentes, ya que de ellos derivan reglas idénticas.

2.4.2. Codificación y representación de los individuos

Los individuos son árboles de derivación. Estos árboles son creados empleando una Gramática Libre de Contexto. La decodificación de estos individuos se obtiene leyendo los nodos terminales de izquierda a derecha, sin incluir los nodos interiores.

Una gramática es recursiva cuando tiene producciones recursivas. Por ejemplo, la gramática mostrada en la Eq. 2 es recursiva, ya que tres de las cuatro producciones del nodo no terminal E son recursivas. En tal caso, dependiendo del algoritmo de generación de individuos que se utilice, se pueden producir o no intrones.

2.4.3. Generación de la población inicial

Para la creación de individuos para la población inicial, hay dos técnicas: una aleatoria y otra controlada. La primera de ellas, utiliza la Gramática Libre de Contexto y genera derivaciones de forma aleatoria, siguiendo las reglas de producción. A pesar de que genera individuos sintácticamente válidos y es simple de aplicar, es incapaz de controlar la profundidad, facilitando la aparición de la explosión de código.

El segundo método es una variante del primero, ya que los individuos también se generan de forma aleatoria, pero controlando en todo momento la profundidad de la derivación, no permitiendo en ningún caso que se sobrepase la profundidad máxima. Para ello, el algoritmo conoce en todo momento la profundidad actual del individuo que está generando y busca reglas de producción que mantengan la profundidad del individuo dentro del límite de profundidad (García-Arnau et al., 2007).

2.4.4. Operadores

Los operadores de la Programación Genética de cruce y mutación son también útiles en la Programación Genética Guiada por Gramáticas, al igual que los operadores comunes de selección y de reemplazo. Con la adición de las gramáticas, existen varias implementaciones nuevas sobre el operador de cruce.

2.4.3.1. Cruce

Si bien los operadores de cruce de la Programación Genética pueden utilizarse en la Programación Genética Guiada por Gramáticas, los individuos generados con

ellos pueden ser no válidos y no cumplirán, por tanto, con la propiedad de cierre. El ejemplo más evidente se hace palpable con el operador de cruce de Koza (Koza, 1992), que genera puntos de cruce de forma aleatoria y que no tiene en cuenta las reglas de producción, tal y como se muestra en la Fig. 7 con la gramática mostrada en la Eq. 2.

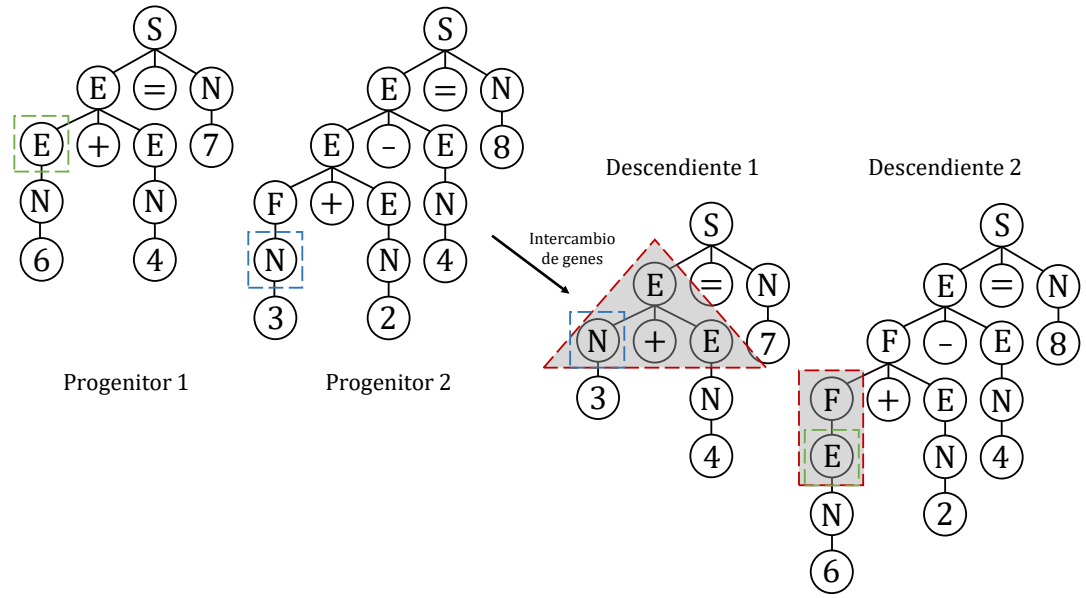


Fig. 7: Errores (en rojo) producidos en el cruce de dos individuos generados con gramáticas al aplicar el operador de Koza.

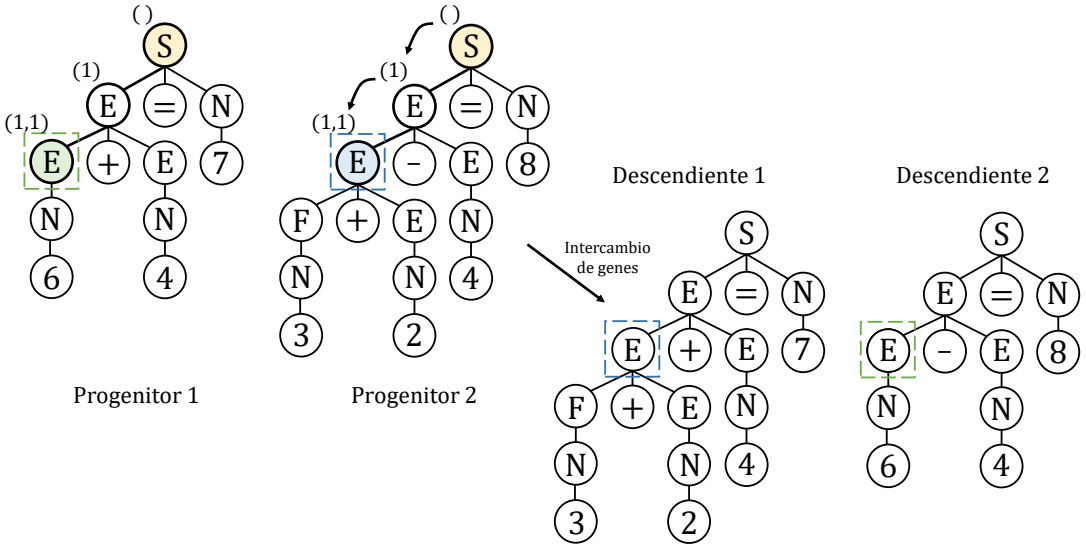


Fig. 8: Ejemplo de aplicación del operador de cruce preservador del contexto.

Operador de cruce preservador del contexto

El operador de cruce preservador del contexto (*context preserving crossover*) (D'haeseleer, 1994) restringe el cruce sólo entre subárboles que tienen localizaciones similares. Estas localizaciones vienen dadas por las coordenadas, reduciendo el exceso de exploración del operador de Koza. Estas coordenadas se definen como el camino entre la raíz y el nodo elegido. Este cruce sólo selecciona un nodo aleatorio: el del primer progenitor. A continuación, extrae las coordenadas de ese nodo aleatorio y las busca en el segundo progenitor. Si esas coordenadas existen en el segundo progenitor, entonces cruza los árboles. En caso contrario, selecciona otro nodo aleatorio del primer padre, realizando ese proceso repetidamente hasta que se puede efectuar el cruce. En la Fig. 8 se puede observar el funcionamiento de este cruce.

El problema que tiene es que es útil para individuos que son estructuralmente similares, es decir, que hay individuos que no se pueden cruzar, por lo que presenta serios problemas de exploración. Además, tiene amplia capacidad de explotación, por lo que, si bien es útil en fases tempranas del programa genético, puede producir una convergencia prematura y caídas en óptimos locales. Tampoco limita la profundidad de los árboles, por lo que propicia la aparición de intrones.

Operador de Whigham

El operador de Whigham (Whigham, 1995) busca establecer un equilibrio entre exploración y explotación. No utiliza las coordenadas, sino que selecciona un símbolo no terminal (exceptuando el axioma) del primer progenitor, y, a continuación, de los símbolos no terminales del segundo progenitor, selecciona uno aleatorio de entre aquellos que son iguales al nodo del primer progenitor. En la Fig. 9 se puede apreciar cómo funciona este operador. La única desventaja es que no controla la profundidad de los descendientes.

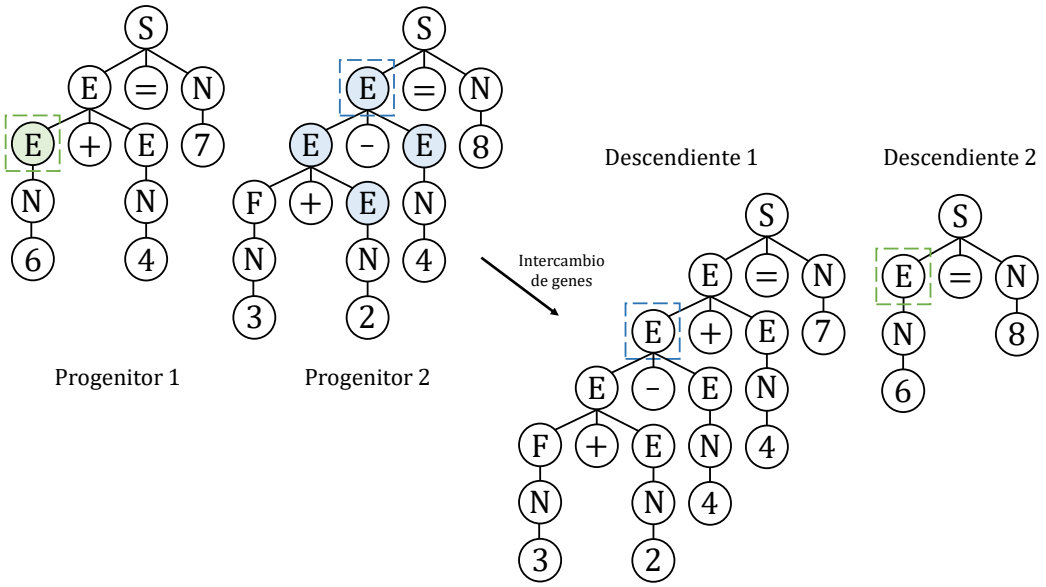


Fig. 9: Ejemplo de aplicación del operador de cruce de Whigham.

3. Redes de Neuronas Artificiales

Las Redes de Neuronas Artificiales son un modelo matemático inspirado en cómo el sistema nervioso procesa la información. El sistema nervioso está compuesto por un gran número de unidades de procesamiento de información denominadas neuronas (*neurons*). En este capítulo se ofrece, en primer lugar, un resumen de la *historia* de las Redes de Neuronas Artificiales. Seguidamente, un apartado de *funcionamiento general* sobre esta rama de estudio y que finaliza hablando sobre la *construcción de redes de neuronas*.

3.1. Historia

Los primeros trabajos sobre Redes de Neuronas Artificiales fueron las células de McCulloch y Pitts (McCulloch and Pitts, 1943) que no eran redes ni tenían capacidad de aprendizaje. Pronto se verían reforzadas con las aportaciones de Hebb (Hebb, 1949) y Joseph Erlanger (1944). Sus descubrimientos sobre el aprendizaje humano y las fibras de conexión entre neuronas, respectivamente, fueron clave en los siguientes años de desarrollo de las Redes de Neuronas Artificiales. Pocos años después aparece el primer desarrollo metodológico basado en Redes de Neuronas Artificiales: el Perceptrón (*perceptron*) (Rosenblatt, 1958, 1962), que incorpora un algoritmo de aprendizaje.

Los años 60 destacó el trabajo de Widrow y Hoff (Widrow and Hoff, 1960): un sistema con un método de aprendizaje alternativo al Perceptrón llamado ADALINE (ADAptive LInear Element). Dos años después, estos científicos desarrollarían una mejora adaptativa sobre su sistema (Widrow and Hoff, 1962). Esta década finaliza con la publicación del artículo *Perceptrons* por Minsky y Papert (Minsky and Papert, 1969), en el que además muestran las limitaciones de los perceptrones como que no pueden resolver problemas no lineales como el problema del XOR.

A raíz de esta última publicación, el interés sobre el estudio de estos modelos de aprendizaje se perdió, hasta tal punto que pasarían más de 10 años hasta la aparición de nuevos trabajos. Este silencio sería roto por Klopf (Klopf, 1972), quien formuló un mecanismo de aprendizaje adaptativo. También, apareció la primera publicación sobre una red multicapa para interpretar caracteres manuscritos: las Redes de Neuronas Artificiales Convolucionales (Fukushima, 1975). Este trabajo consigue despertar el interés de algunos investigadores, pero sin mucho éxito.

Los años 80 y 90 fueron proganizados, en primer lugar, por la publicación del mapa (o red) de Kohonen (*Kohonen map*) (Kohonen, 1982). En segundo lugar, se publicó una metodología sobre de Redes de Neuronas Artificiales Recurrentes (Hop-

field, 1982), que tuvo un efecto contrario a la publicación de Minsky y Papert, persuadiendo a cientos de investigadores, que recobraron el interés en la investigación en este ámbito. La aparición, algunos años más tarde, del famoso método de aprendizaje de retropropagación del grediente (*back-propagation learning method*) (Rumelhart et al., 1986), también contribuye al despertar del interés.

3.2. Funcionamiento general

Las Redes de Neuronas Artificiales son un modelo de procesamiento inspirado en cómo los sistemas nerviosos en la biología procesan información. En términos simples, una Red de Neuronas Artificiales es un modelo matemático del cerebro que es utilizado para procesar relaciones no lineales entre las entradas y las salidas en paralelo tal y como el cerebro humano hace todo el tiempo.

En Fig. 10 se puede apreciar la arquitectura de un Perceptrón con una neurona artificial en la salida. Esta neurona recibe un conjunto determinado de entradas (en este caso, n), con sus respectivos pesos sinápticos w_{ij}. A continuación, calcula el potencial sináptico como la suma de los productos entre las entradas por sus respectivos pesos. A este potencial se le aplica una función de activación, relativa a cada neurona, que permite devolver una salida y, en función del estado de activación.

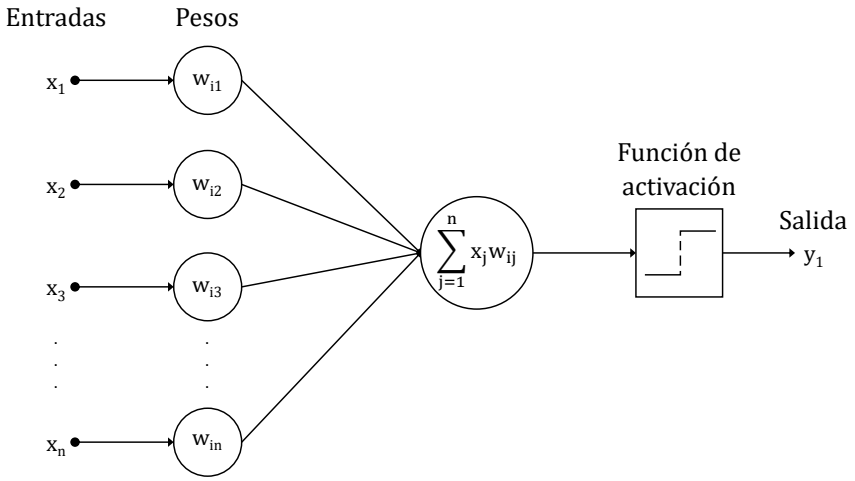


Fig. 10: Arquitectura básica de un Perceptrón.

Actualmente, la arquitectura de una red general no es tan simple como la mostrada en la Fig. 10. Una Red de Neuronas Artificiales suele componerse por, al menos, una capa oculta. Cuando una red tiene más de tres capas ocultas, se denomina Red de Neuronas Artificiales Profunda (Aizenberg et al., 2000). En Fig. 11 se muestra una arquitectura neuronal 2 capas ocultas. Además, esta red es alimentada hacia delante; modelo utilizado en este trabajo.

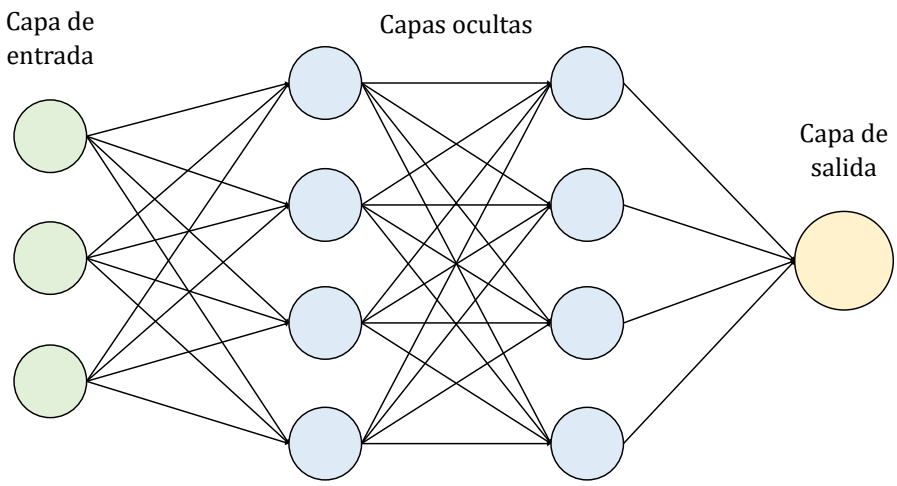
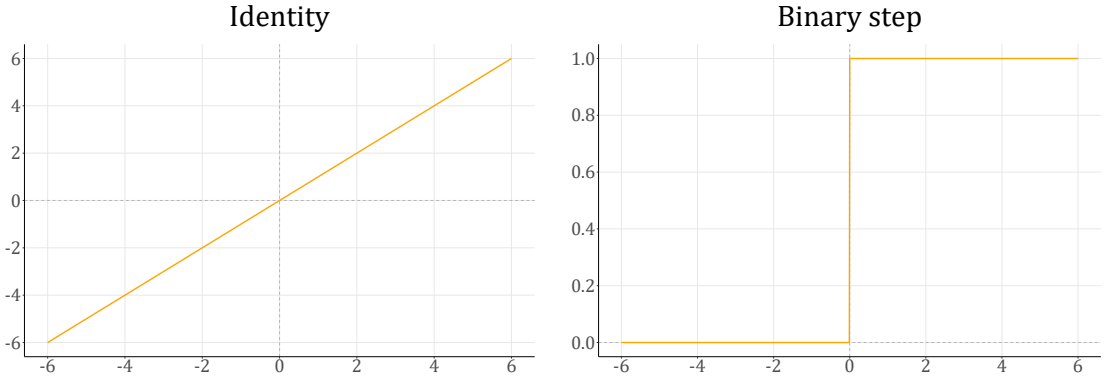


Fig. 11: Distintas capas de una Red de Neuronas Artificiales.

En la capa de entrada, las neuronas reciben datos o señales procedentes del entorno. Las neuronas de la capa de salida proporcionan la respuesta de la red a los estímulos de la entrada, aplicando una transformación que se lleva a cabo en la red. Entre la capa de entrada y la de salida están las capas ocultas (*hidden layers*), que no reciben ni suministran informacion al entorno, sino que la modifican ya que llevan a cabo el procesamiento interno de la red.

El proceso que sigue una Red de Neuronas Artificiales al actualizar los pesos se denomina aprendizaje. En el aprendizaje supervisado, la red conoce la salida correcta ante cada entrada, por lo que la red intentará minimizar el error entre el valor que predice la red y su valor deseado. En cambio, en el aprendizaje no supervisado, la red recibe un conjunto de patrones de los que no conoce la respuesta deseada. En este caso, extrae patrones y relaciones entre los datos.

Cada neurona tiene asociada una función de activación. Existen muchas funciones, cada una con sus ventajas y desventajas, y adecuadas para ser utilizadas según ciertos criterios. Además, algunas de estas funciones tienen limitaciones, como la lineal, que no permite utilizar la propagación hacia atrás para entrenar el modelo, ya que su derivada es una constante y no tiene relación con la entrada. En la Fig. 12 se muestran las funciones de activación más destacadas.



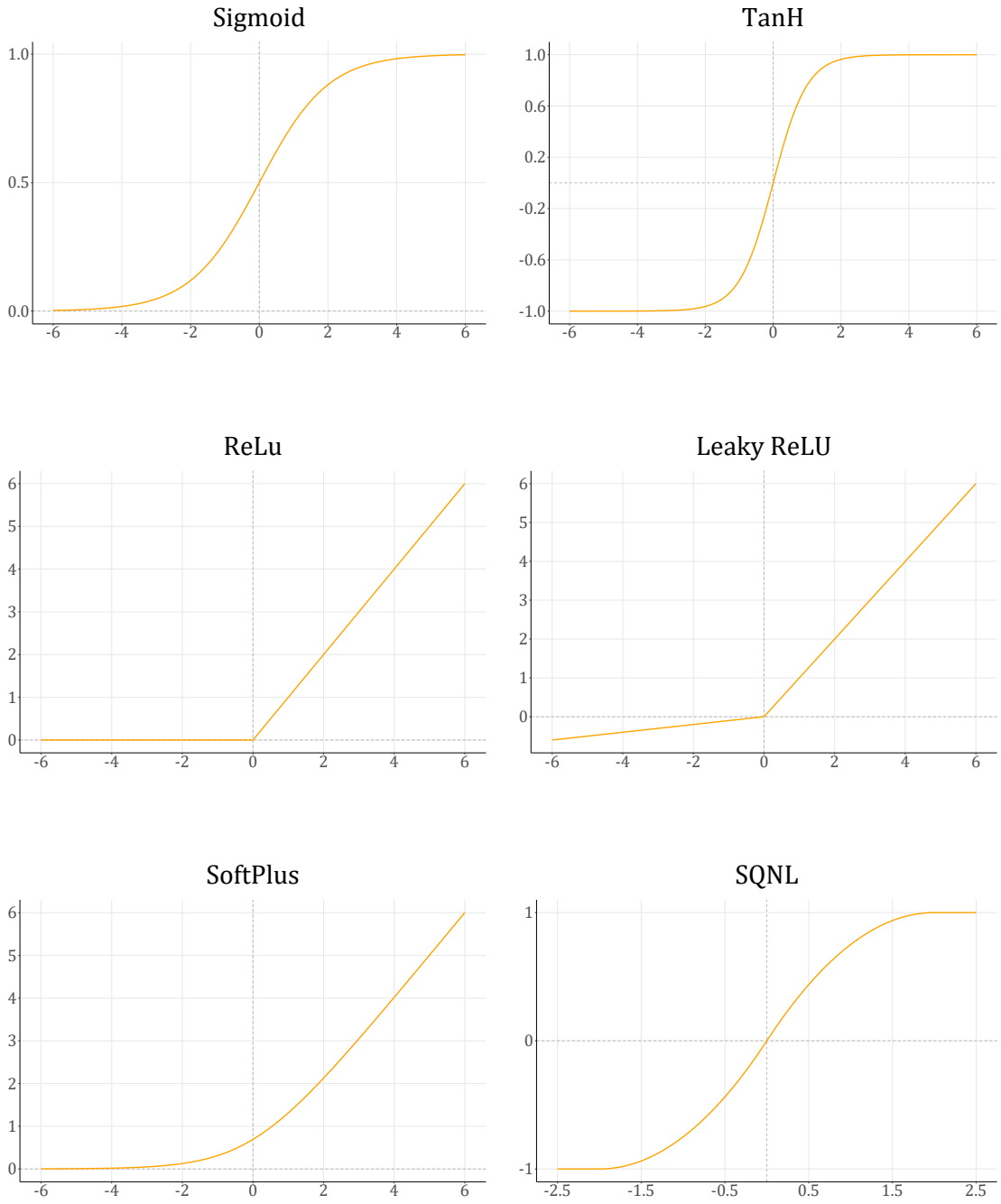


Fig. 12: Gráficas de las funciones de activación más utilizadas.

El proceso de entrenamiento de una red consiste en la actualización de los pesos entre las conexiones de las neuronas que la forman. El entrenamiento comienza tras asignar estos pesos iniciales de forma aleatoria. Este proceso varía según sea un entrenamiento supervisado o no supervisado.

En el entrenamiento supervisado, la red conoce tanto las entradas como las salidas. La red procesa las entradas y compara las salidas predichas con las reales. Los errores se propagan hacia atrás en la red (*backpropagation*), provocando la actualización de los pesos de sus conexiones. La red utiliza el conjunto de entrenamiento

durante todo el proceso, que tiene una duración limitada, medida en términos de sobreajuste (*overfitting*) o en épocas (*epochs*).

En el entrenamiento no supervisado, la red recibe la entrada, pero no la salida deseada. En el proceso, la red debe considerar cómo agrupar estos datos.

3.3. Deep Learning

El Aprendizaje Profundo (*deep learning*) es un subcampo del Aprendizaje Automático (*machine learning*) y basado en Redes de Neuronas Artificiales Profundas (*deep artificial neural networks*). Una red es profunda si tiene más de una capa oculta.

Estos modelos se entrenan con una gran cantidad de datos etiquetados, sin necesidad de una extracción manual de características. En Fig. 13 se muestra una arquitectura neuronal profunda.

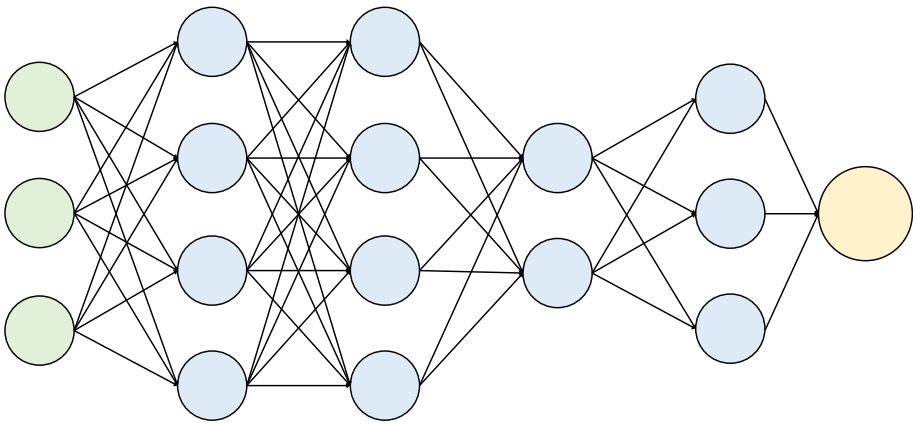


Fig. 13: Ejemplo de Red de Neuronas Artificiales Profunda.

Uno de los usos más destacados del Deep Learning es lo relativo a clasificación de imágenes con Redes de Neuronas Convolucionales (Fukushima, 1975). La diferencia más notable entre este tipo de aprendizaje y el Aprendizaje Autónomo reside en que en el segundo tipo es necesario un proceso de extracción manual de las características relevantes de las imágenes, a partir de las cuales se entrena un modelo que aprende de ellas. En el Deep Learning, las características las aprende el modelo, provisto de datos (imágenes) sin procesar.

En la Fig. 14, en el primer gráfico, se muestra un ejemplo de cómo funciona el Aprendizaje Automático. En el segundo, el funcionamiento del Deep Learning.

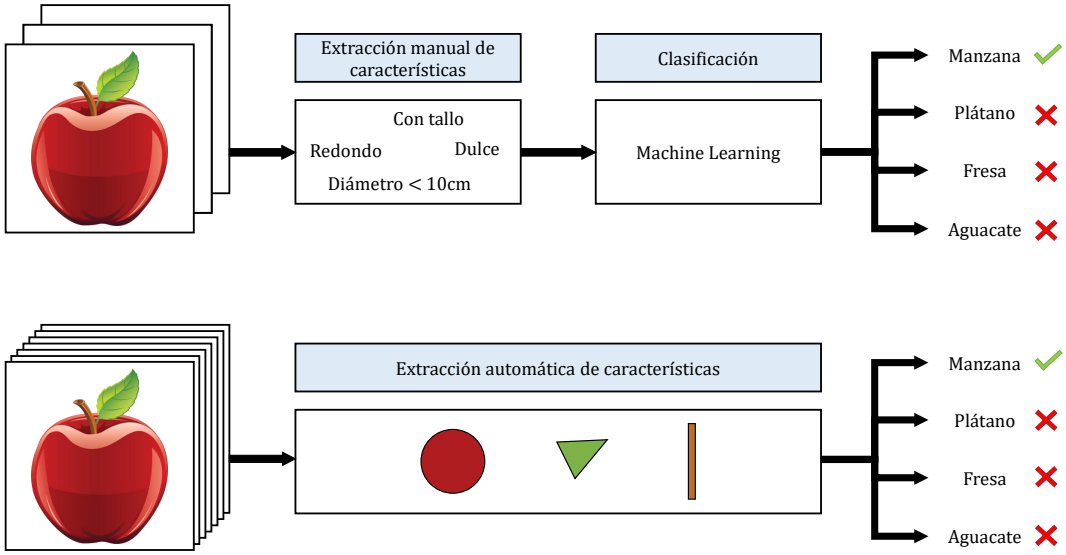


Fig. 14: Ejemplo de Red de Neuronas Artificiales Profunda.

4. Construcción de Redes de Neuronas

Para la construcción de Redes de Neuronas Artificiales se ha utilizado la Programación Genética Guiada por Gramáticas, siguiendo el trabajo de Manrique (Manrique et al., 2018).

El autor propone una gramática para generar arquitecturas neuronales que son sintácticamente válidas. Por tanto, los individuos no serán redes como tal, sino que simbolizarán la estructura de una red. La gramática en cuestión se muestra en la Eq. 3, que recibe dos parámetros: las neuronas de entrada (I) y las neuronas de salida (O).

$$
\begin{aligned}
G_{IO} &= (S, \Sigma_N, \Sigma_T, P_{IO}) \\
\Sigma_N &= \{S, A, H, Z, N\} \\
\Sigma_T &= \{n, /\} \\
P_{IO} &= \{ \\
&\quad S ::= AH/Z \\
&\quad A ::= n^I \\
&\quad H ::= HH|/N \\
&\quad Z ::= n^O \\
&\quad N ::= nN|n \\
&\}
\end{aligned}
\tag{3}
$$

La gramática mostrada codifica todas las arquitecturas válidas, es ambigua y semánticamente redundante y comienza generando pequeñas redes. Es ambigua porque distintas derivaciones pueden codificar la misma arquitectura neuronal, lo que permite encontrar más rápidamente la solución adecuada. Esta gramática produce redes totalmente conectadas y, por tanto, habrá conexiones solo entre cada capa con su posterior, excepto la capa de salida, que no se conecta con ninguna otra. Además, la gramática beneficia a las arquitecturas de red pequeñas, produciendo con menor probabilidad arquitecturas más grandes.

En la gramática, hay dos símbolos terminales: n, que se refiere a una neurona, y /, que denota el separador entre capas. Hay un total de cinco símbolos no terminales, de los cuales se mantiene el axioma, S y símbolo tradicional de inicio en todas las Gramáticas Libres de Contexto usadas en la Programación Genética Guiada por Gramáticas. A simboliza la capa de entrada, formada por n^I neuronas, es decir, con tantas neuronas como entradas a la red hay. H crea recursivamente capas ocultas o inicia la creación de neuronas en la capa oculta. Un individuo al menos tendrá una capa oculta. Z simboliza la capa de salida, formada por n^O neuronas, es decir, con

tantas neuronas como salidas de la red hay. N crea recursivamente cualquier número de neuronas en una capa.

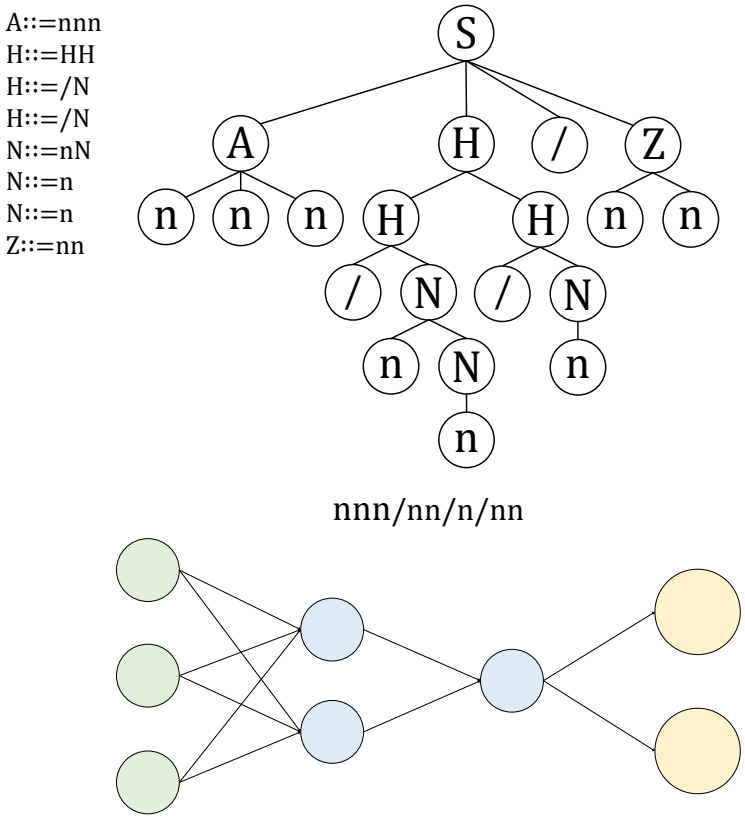


Fig. 15: Individuo de ejemplo generado por la Eq. 3 y su decodificación en la arquitectura neuronal correspondiente.

En la Fig. 15 se muestra un individuo que se ha generado con la gramática en Eq. 3 y sus reglas de producción en orden de aplicación, donde la cantidad de neuronas de entradas es $I = 3$ y la cantidad de neuronas de salida es $O = 2$.

Esta gramática permite, por tanto, la construcción de arquitecturas de redes neuronales con varias capas ocultas. Como estas redes están totalmente conectadas, crea palabras fáciles de interpretar y de decodificar, sin preocuparse sobre las conexiones entre las capas y entre las neuronas.

Sin embargo, la red neuronal resultante debe estar completamente conectada y no permiten la creación de estructuras neuronales más complejas, por ejemplo, eliminando conexiones innecesarias o redundantes entre neuronas. Además, otra limitación es que no permite la contemplación de hiperparámetros como la función de activación por capa, y debe ser impuesta a priori por el usuario, sin poder formar parte del problema de optimización.

La limitación en cuanto a las funciones de activación fue vista por el autor de la gramática, que en el mismo paper propone una segunda aproximación que permite

incluir estas funciones al problema de optimización. La gramática en cuestión se muestra en Eq. 4.

$$
\begin{aligned}
G_{IO\mathbb{F}} &= (S, \Sigma_N, \Sigma_T, P_{IO\mathbb{F}}) \\
\Sigma_N &= \{S, A, H, Z, N, F\} \\
\Sigma_T &= \{n, f_1, f_2, ..., f_\Phi\} \\
P_{IO\mathbb{F}} &= \{ \\
&\quad S ::= AHFZ \\
&\quad A ::= n^I \\
&\quad H ::= HH|FN \\
&\quad Z ::= n^O \\
&\quad N ::= nN|n \\
&\quad F ::= f_1|f_2|...|f_\Phi \\
&\}
\end{aligned}
\tag{4}
$$

La gramática anterior tiene las mismas ventajas que la mostrada en Eq. 3 pero sustituye el separador de capa por una función de activación. En este trabajo no se ha utilizado esta gramática actualizada, ya que sólo se ha enfocado el problema en problemas de clasficiación.

5. Planteamiento del problema

La Programación Genética permite la resolución de problemas de búsqueda y de optimización donde las soluciones son programas informáticos. Por tanto, cada uno de los individuos que forme parte del problema es un programa informático. La ampliación de esta técnica para soportar Gramáticas Libres de Contexto ha permitido no sólo aumentar la complejidad de los problemas que puede resolver esta técnica, sino que también ha permitido no perder tiempo y esfuerzo computacional en reparar individuos no válidos sintácticamente.

La existencia de una gramática que permite generar Redes de Neuronas Artificiales resuelve el problema de encontrar la mejor arquitectura que mejor se adapte a un problema determinado. Anteriormente, este problema se abordaba con el conocimiento experto, no siempre acertado, ya que una arquitectura neuronal depende no sólo del número de instancias, sino de las variables predictoras y de las de clase.

En el proceso evolutivo, cada individuo tiene que ser entrenado para la obtención de su fitness. Como cada individuo codifica una arquitectura de red, cada individuo se transforma en una Red de Neuronas Artificial, que se entrena. El resultado de este entrenamiento es el fitness del individuo. Este proceso de entrenamiento se utiliza para cada nuevo individuo de la población.

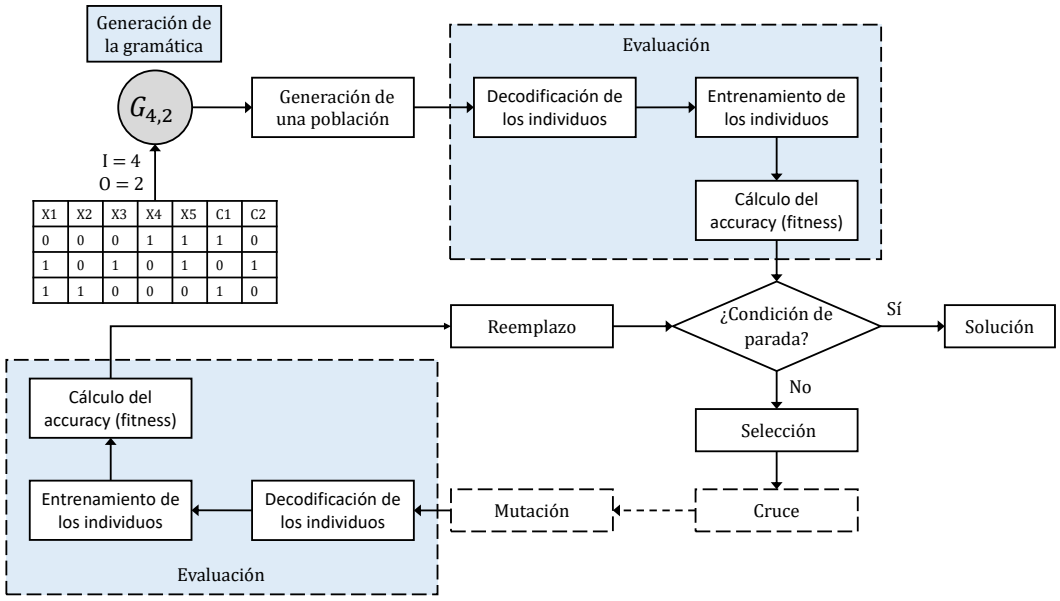


Fig. 16: Proceso evolutivo idéntico al de la Fig. 1 pero aplicado al ámbito de creación de Redes de Neuronas Artificiales.

Sin embargo, para problemas pequeños, el entrenamiento de estas redes es prácti-

camente instantáneo. A medida que los datos adquieren mayor dimensión, la demora en el entrenamiento comienza a ser significativa. Este aspecto hace que esta metodología pierda efectividad cuando realmente es más necesaria; en el ámbito del Deep Learning, donde hay una gran cantidad de información y las arquitecturas neuronales deben ser profundas para entrenar bien con ella.

La única forma de reducir la complejidad temporal que este problema atañe es la parada prematura del entrenamiento de los individuos a la hora de calcular su fitness.

Hay varios elementos que son clave en el entrenamiento de las Redes de Neuronas Artificiales:

- Tamaño del lote (*batch size*). El entrenamiento por lotes es muy común en las redes. En esta metodología, un lote es un subconjunto de los datos de entrenamiento que recorre la red modificando sus pesos.
- Época (*epoch*). Cuando todos los lotes sobre los que se ha dividido el conjunto de datos han pasado por la red, se cumple una época.
- Iteración (*iteration*). Número de lotes necesarios para completar una época. Por ejemplo, si la red quiere entrenar con un conjunto de datos que tiene 20000 instancias y se establece un tamaño del lote de 128 instancias, entonces habrá 157 iteraciones en cada época.

Hay varias formas de reducir la complejidad temporal del problema: una de ellas, es aumentar el tamaño del lote, reduciendo así las iteraciones necesarias para cumplir una época. Sin embargo, una máquina que no tenga acceso a una tarjeta gráfica sobre la que realizar los cálculos, no notará la diferencia prácticamente. Otra alternativa es disminuir el número de épocas. De esta forma, los individuos se entrenan por igual independientemente de si la máquina tuviera tarjeta gráfica o no.

Por tanto,

6. Solución propuesta

7. Resultados

8. Conclusiones y líneas futuras

Bibliografía

Aizenberg, I., Aizenberg, N.N. and Vandewalle, J.P.L. (2000). *Multi-Valued and Universal Binary Neurons: Theory, Learning and Applications.*

Banzhaf, W., Poli, R., Schoenauer, M. and Fogarty, T.C. (1998). *Genetic Programming.*

Beni, G. and Wang, J. (1989). Swarm intelligence in cellular robotic systems. *Proceedings5 of the NATO Advance Workshop on Robots and Biological Systems.* 102:703-712.

Beni, G. (2004). From Swarm Intelligence to Swarm Robotics. *International Workshop on Swarm Robotics.* 3342:1-9.

Boyd, S. and Vandenberghe, L. (2004). Convex Optimization. *Cambridge University Press 2004*: 129.

Box, G.E.P. (1957). Evolutionary operation: A method for increasing industrial productivity. *Journal of the Royal Statistical Society. Series C.* 6(2):81-101.

Box, G.E.P. and Draper, N.P. (1969). *Evolutionary Operation. A method for Increasing Industrial Productivity.*

Bremermann, H.J. (1962). Optimization through evolution and recombination. *Self-Organizing Systems 1962*: 93-106.

Chomsky, N. (1959). On Certain Formal Properties of Grammars. *Information and Control.* 2(2):137-167.

Couchet, J. and Manrique, D. (2006). Crossover and mutation operations for grammar-guided genetic programming. *Soft Computing.* 11(10):943-955.

Couchet, J., Manrique, D. and Porras, L. (2007). Grammar-Guided Neural Architecture Evolution. *Bio-inspired Modeling of Cognitive Tasks*: 438-446.

Cramer, N.L. (1985). A representation for the Adaptive Generation of Simple Sequential Programs. *Proceedings of the First International Conference on Genetic Algorithms and the Applications*: 183-187.

D'haeseleer, P. (1994). Context preserving crossover in genetic programming. *Proceedings of the First IEEE Conference on Evolutionary Computation.* 1:256-261.

Dantzig, G.B. (1990). Origins of the simplex method. *A history of scientific computing*, pages 141-151.

Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection, or Preservation of Favoured Races in the Struggle for Life.*

De Araujo, A.F. and Tavares, J.M.R.S. (2014). An Artificial Life Model for Image Enhancement. *15th International Conference on Experimental Mechanics.* 41(13):5892-5906.

Deb, K., Agarwal, S., Pratap, A. and Meyarivan, T. (2002). A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation.* 6(2):128-197.

Farmer, J.D., Packard, N.H. and Perelson, A.S. (1986). The Immune System, Adaptation, and Machine Learning. *Physica D: Nonlinear Phenomena.* 22(1): 187-204.

Fogel, L.J., Owens, A.J. and Walsh, M.J. (1966). *Artificial Intelligence through Simulated Evolution.*

Fraser, A.S. (1957). Simulation of Genetic Systems by Automatic Digital Computers. *Australian journal of biological sciences.* 10:484-499.

Friedberg, R.M. (1958). A learning machine: part I. *IBM Journal of Research and Development.* 2(1):2-13.

Friedberg, R.M., Dunham, B. and North, J.H. (1959). A learning machine: part II. *IBM Journal of Research and Development.* 3(3):282-287.

Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics.* 20(3):121-136.

García-Arnau, M., Manrique, D., Ríos, J. and Rodríguez-Patón, A. (2007). Initialization method for grammar-guided genetic programming. *Research and Development in Intelligent Systems XXIII*: 32-44.

Geem, Z.W., Kim, J.H. and Loganathan, G.V. (2001) A New Heuristic Optimization Algorithm: Harmony Search. *Simulation*, 76(2):60-68.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimization & Machine Learning.*

Grefenstette, J.J. (1985). *Proceedings of the First International Conference on Genetic Algorithms and the Applications* (Hillsdale, NJ: Lawrence Erlbaum).

Grefenstette, J.J. (1987). *Proceedings of the Second International Conference on Genetic Algorithms and the Applications* (Hillsdale, NJ: Lawrence Erlbaum).

Hassan, R., Cohanim, B., de Weck, O. and Venter, G. (2005). A comparison of particle swarm optimization and the genetic algorithm. *46th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference.*

Hebb, D.O. (1949). *The Organization of Behavior.* (New York: Wiley).

Holland, J.H. (1962). Nonlinear environments permitting efficient adaptation. *Computer and Information Sciences II*: 147-164.

Hopcroft, J.E. and Ullman, J.D. (1979). Context-Free Grammars. *Introduction to Automata Theory Languages and Computation*: 77-106.

Hopfield, J.J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences of the United States of America.* 79(8):2554-2558.

Hu, T., Banzhaf, W. and Moore, J.H. (2014). The effects of recombination of phenotypic exploration and robustness in evolution. *Artificial Life.* 20(4):457-470.

Klopf, A.H. (1972). Brain Function and Adaptive Systems - A Heterostatic Theory. *Air Force Cambridge Research Laboratories.*

Kohonen, T. (1982). Self-Organized Formation of Topologically Correct Feature Maps. *Biological Cybernetics.* 43:59-69.

Koza, J.R. (1988). Non-Linear Genetic Algorithms for Solving Problems. *University States Patent 4935877.*

Koza, J.R.(1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (Cambridge, MA: MIT Press).

Koza, J.R., Goldberg, D.E., Fogel, D.B. and Riolo, R.L. (1996). *Genetic Programming 1996* (Cambridge, MA: MIT Press).

Kosorukoff, A. (2001). Human based genetic algorithm. *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace*: 3464-3469.

Langton, C.G. (1986). Studying artificial life with cellular automata. *Physica D: Nonlinear Phenomena.* 22(1):120-149.

Manrique, D., Barrios, D.R., Delgado, G.M. (2018). Multilayered neural architectures evolution for computing sequences of orthogonal polynomials. *Annals of Mathematics and Artificial Intelligence.* 84(3):161-184.

McCulloch, W.S. and Pitts, W.H. (1943) A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics.* 5:115-133.

Minsky, M.L. and Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry* (Cambridge: MIT Press).

Moscato, P. (1989). On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts - Towards Memetic Algorithms. *Caltech Concurrent Computation Program*: 158-179.

Nocedal, J. and Wright, S.J. (1999). Numerical optimization. *Springer-Verlag.*

Pezzella, F., Morganti, G. and Ciaschetti, G. (2008). A genetic algorithm for the Flexible Job-shop Scheduling Problem. *Computers & Operations Research.* 35(10):3202-3212.

Poli, R. and Langdon, W.B. (1997a). An experimental analysis of schema creation, propagation and disruption in genetic programming. *Proc. of ICGA'97.*

Poli, R. and Langdon, W.B. (1997b). A new schema theory for genetic programming with one-point crossover and point mutation. *Proc. of the Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing.*

Poli, R. and Langdon, W.B. (1998). A review of theoretical and experimental results on schemata in genetic programming. *Proceedings of the First European Workshop on Genetic Programming.*

Polya, G. (1945). How to Solve It. *Princeton University Press 1945.*

Rawlins, G.J.E. (1991). *Foundations of Genetic Algorithms* (San Mateo, CA: Morgan Kaufmann).

Rechenberg, I. (1965). Cybernetic Solution Path of an Experimental Problem. *Royal Aircraft Establishment Library Translation 1122.*

Reynolds, R.G. (1994). An Introduction to Cultural Algorithms. *Proceedings of the 3rd Annual Conference on Evolutionary Programming*: 131-139.

Roberts, L. (1965). Machine perception of 3D solids. *Optical and Electro-Optical Information Processing.* 9:159-197.

Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. *Psychological Review.* 65(6):386-408.

Rosenblatt, F. (1962). *Principles of Neurodynamics* (Washington: Spartan Books).

Rumelhart, D.E., Hinton, G.E. and Williams, R.J. (1986). Learning internal representations by error propagation. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* 1:318-362.

Rumelhart, D.E., Hinton, G.E. and Williams, R.J. (1986). Learning representations by back-propagating errors. *Nature.* 323:533–536.

Ryan, C., Collins, J.J. and O'Neil, M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Proceedings of the First European Workshop on Genetic Programming*: 83-96.

Samuel, A.L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development.* 3(3):210-229.

Schaffer, J.D. (1989). *Proceedings of the Third International Conference on Genetic Algoritms and the Applications* (San Mateo, CA: Morgan Kaufmann).

Schwefel, H-P. (1965). Kybernetische Evolution als Strategie der experimentellen Forschung in der Strömungstechnik.

Storn, R. (1996). On the usage of differential evolution for function optimization. *Biennial Conference of the North American Fuzzy Information Processing Society*: 519-523.

Storn, R. and Price, K. (1997). Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization.* 11(4):341-359.

Tackett, W.A. (1995). Greedy Recombination and Genetic Search on the Space of Computer Programs. *Foundations of Genetic Algorithms III* (San Mateo, CA: Morgan Kaufmann).

Vanneschi, L., Gustafson, S. and Banzhaf, W. (2010). Open issues in Genetic programming. *Genetic Programming and Evolvable Machines.* 11:339-363.

Vose, M.D. and Whitley, L.D. (1995). *Foundations of Genetic Algorithms 3* (San Francisco, CA: Morgan Kaufmann).

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K. and Lang, K.J. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing.* 37(3):328-339.

Whigham, P.A. (1995). Grammatically-based genetic programming. *Proceedings of the workshop on genetic programming: from theory to real-world applications*: 33-41.

Whitley, L.D. (1993). *Foundations of Genetic Algorithms 2* (San Mateo, CA: Morgan Kaufmann).

Widrow, B. and Hoff, M. (1960). Adaptive Switching Circuits. *International Journal of Communications, Network and System Sciences.* 5:96-104.

Widrow, B. and Hoff, M. (1962). Associative Storage and Retrieval of Digital Information in Networks of Adaptive "Neurons". *Biological Prototypes and Synthetic Systems.* 1:160-160.