

# Evolución de Redes Neuronales mediante Sistemas de Reescritura

Autores: Esteban Andrés García N° de Alumno: 2085/4  
Germán Leandro Osella Massa N° de Alumno: 2186/8  
Director: Lic. Laura Lanzarini  
Institución: Facultad de Informática de la  
Universidad Nacional de La Plata  
Fecha de  
presentación: Diciembre de 2003

TES  
03/2  
DIF-02953  
SALA



UNIVERSIDAD NACIONAL DE LA PLATA  
FACULTAD DE INFORMÁTICA  
Biblioteca  
50 y 120 La Plata  
catalogo.info.unlp.edu.ar  
biblioteca@info.unlp.edu.ar



DIF-02953

## Resumen

Las redes neuronales evolutivas son un caso particular de redes neuronales artificiales en donde los pesos de las conexiones no son determinados por un método de entrenamiento sino por la aplicación de un proceso evolutivo.

El método propuesto en esta tesis, NeSR, evoluciona tanto los pesos de conexión como la estructura de la red neuronal. Este método se basa en una codificación indirecta, es decir, no evoluciona redes neuronales sino sistemas de reescritura denominados Sistemas L. Esta representación permite construir una red neuronal, la cual será evaluada en el problema a resolver.

Este método tiene la virtud de brindar una poderosa flexibilidad en la estructura de las redes generadas a partir de estos sistemas, aunque requiere un costo de procesamiento extra en el paso de convertir un genotipo (Sistema L) en su fenotipo (Red Neuronal).

Las mediciones realizadas demuestran su capacidad para resolver distintos tipos de problemas en forma similar a otros métodos neuroevolutivos.



BIBLIOTECA  
FAC. DE INFORMÁTICA  
U.N.L.P.

DONACION..... LIUTI .....  
\$.....  
Fecha..... 17-10-07 .....  
Inv. E..... Inv. B..... 002959 .....

TES
0312



# Contenido

1. Introducción.....	1
1.1. Objetivos de la investigación realizada .....	1
1.2. Temas desarrollados en este trabajo .....	1
1.2.1. Redes neuronales artificiales .....	1
1.2.2. Algoritmos Genéticos .....	2
1.2.3. Sistemas L .....	2
2. Redes Neuronales artificiales .....	3
2.1. Introducción.....	3
2.2. Neurona artificial .....	4
2.3. Arquitectura de una red neuronal artificial.....	5
2.4. Modo de operación .....	6
2.5. Operaciones de capa .....	7
2.6. El perceptrón.....	7
2.6.1. Aprendizaje del Perceptrón .....	11
2.6.2. Regla de aprendizaje y algoritmo de entrenamiento .....	12
2.6.3. Perceptrón unicapa .....	15
2.6.4. Perceptrón multicapa .....	17
2.7. El Adaline .....	19
2.8. Back propagation.....	20
2.8.1. Estructura y funcionamiento de la red Backpropagation.....	22
2.8.2. Regla de aprendizaje Delta generalizada –GDR– .....	23
2.8.3. Actualización de pesos de la capa de salida .....	23
2.8.4. Actualización de los pesos de capas ocultas.....	24
2.8.5. Variantes del algoritmo .....	26
2.9. Tipos de métodos de entrenamiento para redes neuronales .....	26
2.9.1. Aprendizaje Supervisado por corrección de error .....	27
2.9.2. Aprendizaje Supervisado por Refuerzo .....	28
2.9.3. Aprendizaje Supervisado Estocástico.....	28
2.9.4. Aprendizaje no supervisado .....	28
2.9.5. Aprendizaje Competitivo.....	29
2.10. Consideraciones prácticas.....	29
2.10.1. Conjunto de entrenamiento.....	29
2.10.2. Dimensiones de la red.....	30
2.10.3. Pesos y parámetros de aprendizaje .....	31



2.10.4. Función de transferencia de las neuronas ocultas y de salida.....	31
2.11. Debilidades de la Backpropagation .....	32
2.12. Redes neuronales recurrentes .....	33
2.13. Redes de Jordan.....	34
2.14. Redes de Elman .....	35
2.15. Estructura de RNAs recurrentes y su entrenamiento.....	36
2.16. Aplicaciones de RNA .....	37
3. Algoritmos Evolutivos.....	39
3.1. Introducción.....	39
3.2. Métodos de búsqueda local .....	39
3.2.1. Hill Climbing.....	39
3.2.2. Simulated Annealing .....	41
3.3. Algoritmos Evolutivos.....	42
3.3.1. El Esqueleto de un Algoritmo Evolutivo.....	43
3.3.2. Distintas variantes de los Algoritmos Evolutivos .....	44
3.4. Algoritmos Genéticos.....	45
3.4.1. Codificación .....	46
3.4.1.1. Codificación mediante cadenas de bits.....	46
3.4.1.2. Codificación basada en números reales.....	48
3.4.1.3. Cromosomas de longitud variable .....	49
3.4.1.4. Otras representaciones.....	49
3.4.2. Inicialización .....	50
3.4.3. Evaluación .....	50
3.4.4. Selección.....	51
3.4.4.1. Método de la ruleta .....	51
3.4.4.2. Muestreo Universal Estocástico .....	52
3.4.4.3. Muestreo Estocástico con Reemplazo del Resto .....	52
3.4.4.4. Ranking.....	53
3.4.4.5. Torneo.....	53
3.4.4.6. Elitismo.....	53
3.4.5. Reproducción.....	54
3.4.5.1. Crossover de un punto .....	55
3.4.5.2. Crossover multipunto .....	55
3.4.5.3. Crossover uniforme .....	55
3.4.5.4. Crossover aritmético.....	56

3.4.5.5. Crossover especializados.....	56
3.4.5.6. Mutación de cadenas de bits.....	56
3.4.5.7. Mutación aritmética.....	57
3.4.6. Reemplazo .....	57
3.4.7. Condiciones de terminación .....	58
3.5. ¿Qué ventajas y desventajas tienen con respecto a otras técnicas de búsqueda?	58
3.6. Fundamentos teóricos.....	59
3.6.1. Teorema de los Esquemas o Teorema Fundamental .....	59
3.6.2. Paralelismo implícito y la Hipótesis de construcción de bloques .....	62
4. Neuroevolución .....	65
4.1. Conveniencia de la neuroevolución.....	65
4.2. Evolución de pesos de conexión.....	66
4.2.1. Entrenamiento evolutivo vs. entrenamiento basado en gradiente .....	67
4.2.2. Entrenamiento híbrido .....	68
4.3. Evolución de la arquitectura .....	69
4.4. Evolución de función de transferencia de los nodos .....	71
4.5. Evolución simultánea de arquitectura y pesos de conexión .....	71
4.6. Evolución de reglas de aprendizaje .....	72
4.7. Esquema de codificación directa .....	73
4.8. Esquema de codificación indirecta .....	74
5. Sistemas de reescritura .....	77
5.1. Gramáticas formales.....	77
5.1.1. Formalización de las gramáticas libres de contexto .....	78
5.1.2. Reescritura de gramáticas formales.....	78
5.1.3. Gramáticas sensibles al contexto.....	80
5.2. Sistemas L .....	80
5.2.1. Sistemas L paramétricos.....	82
5.2.2. Sistemas L sensibles al contexto .....	85
5.2.3. Sistemas L estocásticos .....	86
5.3. Aplicaciones de los Sistemas L .....	87
5.3.1. Interpretación de una tortuga estilo LOGO .....	87
5.3.2. Modelado de plantas.....	91
5.3.3. Generación de fractales .....	92
5.3.4. Exploración del espacio de parámetros .....	93
6. Neuroevolución de Sistemas de Reescritura .....	97

6.1. Introducción.....	97
6.2. Representación de las Redes Neuronales. ....	97
6.3. Comandos de construcción de RNA.....	98
6.3.1. Comandos de construcción de RNA en NeSR .....	100
6.3.2. Optimización de RNA .....	102
6.4. Evolucionando sistemas de reescritura.....	104
6.4.1. Restricciones impuestas a los Sistemas L.....	104
6.4.2. Crossover en NeSR.....	106
6.4.3. Mutación en NeSR .....	107
6.5. La red neuronal.....	109
7. Otros métodos de neuroevolución .....	111
7.1. Neuroevolución vía un Algoritmo Genético Simple .....	111
7.1.1. Codificación de redes neuronales .....	111
7.1.2. Neuroevolución .....	112
7.1.3. Análisis del método .....	112
7.2. Neuroevolución por reescritura de matrices.....	113
7.2.1. Reescritura de matrices.....	113
7.2.2. Evolución de topologías de redes neuronales.....	115
7.2.3. Análisis del método .....	115
7.3. Neuroevolución de topologías en aumento (NEAT) .....	116
7.3.1. Codificación genética .....	116
7.3.2. Seguimiento de genes a través de marcas históricas .....	117
7.3.3. Protección de la innovación mediante la especiación .....	121
7.3.4. Minimización de la dimensionalidad mediante el crecimiento incremental a partir de estructura mínima.....	122
7.3.5. Problemas resueltos con NEAT.....	123
7.3.6. Análisis de NEAT.....	123
8. Herramienta de Neuroevolución.....	125
8.1. Motivación.....	125
8.2. Diseño del Framework de Evolución .....	125
8.2.1. Clases principales .....	126
8.2.2. Especialización de los métodos .....	126
8.2.3. Extensiones para neuroevolución .....	130
8.2.4. Implementación de los problemas a resolver .....	130
8.2.5. Extensión para neuroevolución con cadenas de bits .....	132

8.2.6. Extensión para NEAT.....	134
8.2.7. Extensión para NeSR.....	135
8.3. Paralelización del algoritmo evolutivo.....	140
8.4. Flexibilidad del framework .....	142
8.5. Utilización de la herramienta de neuroevolución.....	142
8.5.1. Panel de experimentos disponibles.....	142
8.5.2. Ejecución de experimentos.....	143
8.5.2.1. Consola de ejecución de experimentos.....	144
8.5.3. Panel de información de experimentos.....	145
8.5.3.1. Estadísticas .....	145
8.5.3.2. Información específica de cada elemento.....	147
8.5.3.3. Mejor Individuo .....	150
9. Desempeño de NeSR .....	153
9.1. Experimentos.....	153
9.2. El problema del XOR .....	153
9.3. Balance del péndulo simple.....	154
9.4. Balance del péndulo doble.....	155
9.5. Planificación de experimentos.....	157
9.6. Configuración de NeSR en los experimentos.....	158
9.7. Resultados obtenidos .....	160
9.8. NeSR vs. Neuroevolución basada en cadenas de bits .....	165
9.9. NeSR vs. NEAT .....	168
10. Conclusiones y futuras líneas de trabajo .....	173
11. Referencias .....	177



# 1. Introducción

Esta tesis fue desarrollada en la Facultad de Informática de la Universidad Nacional de La Plata para ser presentada como trabajo de graduación para la carrera de Licenciatura en Informática.

En los primeros capítulos daremos una introducción a los conceptos básicos en el área correspondiente a la investigación realizada para luego pasar a explicar nuestra propuesta.

Finalmente, se muestran los resultados obtenidos así como diversas comparaciones con otros trabajos similares.

## 1.1. Objetivos de la investigación realizada

El estudio de diversos métodos de neuroevolución, desde los clásicos hasta los más recientes, nos ha llevado a la conclusión que muchos de ellos no aprovechan la posibilidad de generar estructuras de manera variable. Tal es el caso de SANE [93] y ESP [44], los cuales trabajan con una población de redes que poseen una estructura fija y solamente evolucionan los pesos de conexión. Otros métodos únicamente evolucionan la estructura, dejando los pesos de conexión para algún algoritmo de entrenamiento como el utilizado por la Backpropagation. Finalmente, hay métodos que prometen generar cualquier estructura, y si bien manejan redes de topología variables, en muchos casos resultan insuficientes para el problema a resolver.

El objetivo de nuestra tesis es presentar un nuevo método de neuroevolución, basado en codificación indirecta, que permite evolucionar tanto los pesos como la estructura de la red neuronal permitiendo, con la configuración adecuada, generar redes de todo tipo de topología, dejando al proceso evolutivo la selección de la red más apta para un problema dado.

## 1.2. Temas desarrollados en este trabajo

Esta tesis pretende ser un informe autocontenido del método neuroevolutivo desarrollado. Para tal fin, incluimos en la misma una descripción de los paradigmas cuya comprensión es requerida para el entendimiento de este método.

A continuación daremos una breve introducción a los temas más importantes que aquí hemos utilizado.

### 1.2.1. *Redes neuronales artificiales*

Desde que se han inventado las computadoras, el hombre siempre ha tratado de programarlas para imitar comportamiento humano inteligente. Éste no es un emprendimiento fácil ya que los programas deben poder hacer una gran cantidad de tareas para poder ser llamados inteligentes. Además el significado de la palabra inteligencia o de *comportamiento inteligente* es poco claro, debido al gran número de definiciones existentes y la subjetividad a la que queda sujeta esta definición.

Los métodos utilizados para alcanzar inteligencia artificial en los primeros días de la computación como sistemas basados en reglas, nunca alcanzaron los resultados esperados y más aún, no se ha podido construir un conjunto de reglas que brinden a estos sistemas la calificación de inteligentes. Como la ingeniería inversa probó ser

exitosa en muchas otras áreas, los investigadores han tratado de modelar el cerebro humano usando computadoras. Aunque los principales componentes del cerebro, las neuronas, son relativamente sencillos de describir, es imposible aún construir un cerebro artificial que imite al cerebro humano en todos sus detalles de complejidad. Esto es debido a la gran cantidad de neuronas involucradas y al enorme conjunto de conexiones entre dichas neuronas. Por lo tanto se han realizado una gran cantidad de simplificaciones para mantener el poder de cómputo necesario dentro de límites realistas y de esta manera acceder a la posibilidad de emular el comportamiento de una red de neuronas o red neuronal en una computadora, dando lugar a las *redes neuronales artificiales* o RNA.

Hay una cantidad de maneras de *entrenar* la red para que ésta *aprenda* un problema específico. En el método que proponemos no se utiliza más que el proceso evolutivo, emulado por *algoritmos genéticos* por computadora, como herramienta de entrenamiento de la red. Esto permite obtener RNA que puedan desempeñarse en tareas para las cuales un método de entrenamiento tradicional no sería posible de aplicar.

### **1.2.2. Algoritmos Genéticos**

La evolución Darwiniana, en la cual los organismos más aptos tienen más probabilidad de sobrevivir y reproducirse que los organismos menos aptos, es modelada por algoritmos genéticos. Una población de cadenas es manipulada, donde cada cadena puede ser vista como un cromosoma, el cual consiste en un número de genes. Esos genes son usados para codificar los parámetros de un problema al que hay que buscarle una solución. A cada cadena puede asignársele un valor de fitness, el cual indica cuán buena es una solución para el problema. Como ocurre en la selección natural y en genética, donde la chance de reproducción de un organismo (y por lo tanto de sus genes) depende de su capacidad para sobrevivir (su fitness), las cadenas utilizadas por el algoritmo se reproducen de acuerdo a su valor de fitness. En base a este valor se crea una nueva generación seleccionando y recombinando cadenas existentes mediante operadores genéticos. Los más comúnmente utilizados son: selección, crossover, y mutación. Estos son descritos en detalle en este trabajo y se brindará un tratamiento especial a los mecanismos básicos de búsqueda implementados en un algoritmo genético.

### **1.2.3. Sistemas L**

El desarrollo de seres vivos está gobernado por los genes. Cada célula viva contiene información genética (el genotipo) la cual determina la manera en que se desarrollará la forma final del organismo (el fenotipo). La información genética puede ser vista como una receta. Esta receta es utilizada no por el organismo como un todo, pero sí para cada célula individual. La forma y comportamiento de cada célula depende de la información de los genes que le dieron origen, la cual depende de los genes pasados y las influencias del ambiente en todas las células cercanas. Así, el desarrollo es gobernado por las interacciones locales entre elementos que responden a las mismas reglas globales. Para modelar este desarrollo en las plantas, el biólogo Aristid Lindenmayer desarrolló una construcción matemática llamada *Sistema L* o *L-System*. Mediante el uso de las llamadas reglas de reescritura, con un Sistema L, una cadena puede ser reescrita en otra cadena mediante una reescritura en paralelo de todos sus caracteres. La aplicación de reglas de reescritura depende de las reglas que se han aplicado en el pasado y los caracteres vecinos del carácter a ser reescrito.

## 2. Redes Neuronales artificiales

Dedicaremos este capítulo al paradigma de las redes neuronales artificiales o simplemente RNA. Iniciaremos con una introducción al tema brindando un panorama general del área. Luego nos detendremos en varias secciones a explicar la composición (neuronas, conexiones, arquitectura, etc.) y la operación de una RNA. A continuación mostraremos algunos tipos de neuronas y redes propuestas por diferentes investigadores incluyendo además el concepto de *entrenamiento*, para pasar luego a la arquitectura más popular de red neuronal: la *backpropagation* junto con su regla de aprendizaje. Incluiremos indicaciones prácticas acerca del entrenamiento y los posibles problemas en los que se puede caer con este método. Además, describiremos en que consisten los diferentes tipos de entrenamientos de RNA. Luego discutiremos acerca de las arquitecturas de las redes *recurrentes* junto con las ventajas y desventajas que éstas poseen. Finalmente brindaremos al lector un panorama de los campos de aplicación de las redes neuronales artificiales.

### 2.1. Introducción

Las redes neuronales artificiales nacen de la idea de imitar el comportamiento del cerebro humano a través de una computadora. Las computadoras de hoy en día pueden resolver complejos cálculos matemáticos a una velocidad impensada para el ser humano. Sin embargo hay muchas tareas que en el hombre resultan sencillas pero por sus características la computadora no puede realizar. El cerebro humano no necesita de un programa para su funcionamiento y se adapta de alguna manera a las situaciones del ambiente. Por el contrario una computadora requiere de indicaciones acerca de lo que debe hacer en determinadas situaciones y a menos que haya sido programada para ello, no podría inferir sobre acciones a realizar en ambientes para los que no fue preparada.

Por ejemplo, para una tarea tan sencilla para un médico como examinar una radiografía y detectar una fractura ósea o cualquier otra anomalía en fracciones de segundo, una computadora requiere un complejo programa de análisis de imágenes con el gran costo de procesamiento que trae aparejado este tipo de aplicaciones.

Un importante motivo que llevó al estudio del cerebro humano y la posibilidad de emular su comportamiento se basa en las cualidades que este tiene, las cuales son deseables para cualquier sistema de computadora, según se cita en [15]:

- Es robusto y tolerante a fallas, diariamente mueren neuronas sin afectar su desempeño.
- Es flexible, se ajusta a nuevos ambientes por aprendizaje, no hay que programarlo.
- Puede manejar información difusa o con ruido.
- Es altamente paralelo.

Aplicando conceptos de ingeniería inversa, las redes neuronales artificiales surgen como una aproximación al modelo del cerebro humano adaptado para poder ejecutarse en una computadora. Definir qué es una neurona individual y cómo funciona resulta simple. La complejidad de las tareas que el cerebro humano puede realizar tiene que ver con la gran cantidad de neuronas y la vasta red de conexiones que las comunica. Implementar una neurona artificial en una computadora es una tarea trivial, sin embargo crear una red neuronal artificial con una cantidad de neuronas y una interconexión similar o



equivalente al cerebro humano y hacer que actúe como tal, requiere de un hardware fuera de lo común.

A pesar de esto, las redes neuronales artificiales, que en un principio se utilizaron para la aproximación de funciones, se han mostrado muy útiles para resolver ciertos problemas acotados a un dominio en particular, como ser reconocimiento de voz, patrones en imágenes, escritura, filtrado de ruidos y hasta tareas de control. Reciben estímulos que representan la información de entrada (muestras de sonido, de imágenes, información sobre un determinado ambiente, etc.) y actualizan su estado de manera de mostrar una salida (clasificación de la información de entrada, determinación de llevar a cabo una acción en particular en el ambiente, etc.) en base al estímulo recibido.

A diferencia de los sistemas de cómputo tradicionales, las redes neuronales artificiales no ejecutan instrucciones sino que responden en paralelo a las entradas suministradas. Para cumplir su función correctamente deben ser entrenadas de manera de “aprender” la tarea que deben cumplir. Existen varios métodos de aprendizaje, algunos de los cuales veremos en este capítulo, pero en general se busca proporcionar una entrada a la red y modificarla en base a la respuesta obtenida. El comportamiento de la red estará basado en el conocimiento adquirido durante el entrenamiento, el cual queda representado en su topología y en los pesos de las conexiones entre neuronas.

## 2.2. Neurona artificial

Observaciones biológicas hechas por Warren McCulloch y Walter Pitts los llevaron en 1943 a presentar el primer modelo de una neurona artificial. Según Freeman [29], estos elementos individuales de cálculo que forman la mayoría de los modelos de RNA no suelen denominarse neuronas artificiales, lo más frecuente es darles el nombre de nodos, unidades o elemento de procesamiento o PE por sus siglas en inglés. En el presente trabajo utilizaremos estos términos como sinónimos.

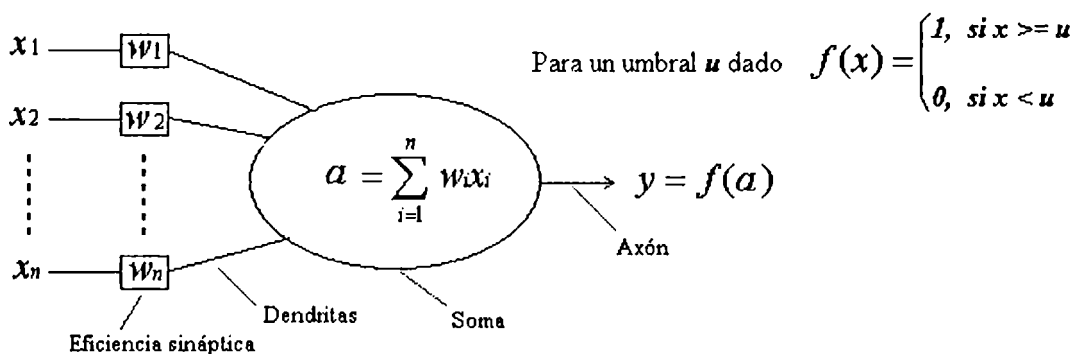


Figura 1: Modelo de neurona artificial de McCulloch y Pitts

Utilizaremos la Figura 1 para describir la neurona artificial. Como se puede observar posee un número finito de conexiones de entrada – $n$  en este caso– que se corresponden con las dendritas en el modelo biológico<sup>1</sup>, por las cuales ingresa cada una de las  $n$  componentes de un vector de entrada. Toda conexión de entrada tiene asociada una magnitud llamada peso o intensidad –eficiencia sináptica– definida por las componentes del vector de pesos. Estos pesos de conexión  $w_i$  pueden ser positivos –excitatorios– o negativos –inhibitorios–.

<sup>1</sup> En [15] puede encontrarse una descripción completa del modelo de una neurona biológica.

La neurona acumula todas las señales de entradas multiplicadas por sus pesos de conexión obteniendo el valor de activación  $-a$  en la Figura 1- al cual se aplica una función llamada de *transferencia*, en este caso la función umbral para obtener la salida del PE.

Al igual que en una neurona biológica, tenemos muchas entradas pero una única salida, la cual podrá replicarse pudiendo permitiéndole así conectarse con muchos otros PEs. No siempre es correcto asumir que los elementos de procesamiento tienen una relación uno a uno con las neuronas biológicas reales. Es conveniente pensar que un nodo representa la actividad colectiva de un grupo de neuronas. Esto nos evitará cometer el error de hablar de las redes neuronales como modelos reales del cerebro humano. En este modelo, una neurona es un elemento binario cuyo estado se manifiesta por la salida que pertenece al conjunto  $\{0,1\}$ . De esta manera podemos usar la lógica proposicional para describir la acción de ciertas redes formadas por la conexión de estas neuronas.

La aproximación anterior intenta emular el procesamiento neuronal, pero la teoría presentada por sus autores no dice nada acerca del aprendizaje (adaptación de los pesos y umbrales de los PEs y su organización para formar las redes). Sin embargo su aporte motivó que otros investigadores avanzaran en el tema.

### 2.3. Arquitectura de una red neuronal artificial

Si bien el elemento de procesamiento constituye una pieza importante en el paradigma de la computación neuronal, uno solo de ellos funcionando de manera aislada tiene un campo de aplicación muy escaso. Por ello, para ampliar la utilidad de los mismos se los combina formando un grafo con sus conexiones de entrada y de salida dando lugar a las redes neuronales.

En esta sección daremos al lector una noción acerca de la estructura de un tipo particular de RNA denominado *feedforward* pero debemos advertir que existe una variedad de formas en cuanto a la topología de redes neuronales. En secciones posteriores se verán algunas variantes de las llamadas *recurrentes*.

Volviendo a las redes *feedforward*, estas tienen una estructura similar a la que se puede apreciar en la Figura 2. Como vemos esta especie de grafo dirigido se divide en *capas*, es decir conjuntos de nodos que se encuentran en un mismo nivel de acuerdo a sus conexiones.

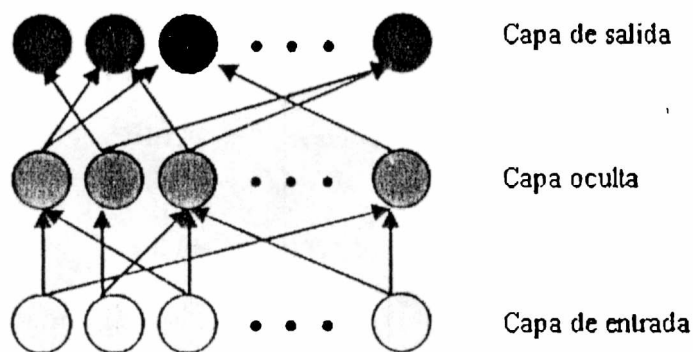


Figura 2: Disposición de las distintas capas de nodos en una red neuronal de ejemplo.

Según la Figura 2 en este tipo de redes se reconocen tres clases de capas:

- *Capa de entrada*: Conjunto de nodos cuya función es *únicamente* recibir la información de entrada a la red. Por ejemplo, si una red debe responder el resultado de la aplicación de un operador lógico a dos operandos, es de esperar que esta red disponga de dos nodos de entrada.
- *Capa de salida*: Consiste en los nodos de los cuales se obtiene la respuesta que la red genera. Siguiendo con la idea del operador lógico de aridad 2, seguramente la red tendrá una única neurona de salida, la cual al activarse se interpreta como un valor y al no activarse como otro. También puede tener dos y activar uno u otro dependiendo del valor resultante.
- *Capa oculta*: En esta capa se sitúan nodos que reciben conexiones de las entradas y se conectan hacia las salidas. Se puede aplicar esta noción para intercalar capas ocultas y así formar una red que posea más de una. Se podría decir que la capacidad de la red se encuentra codificada en esta capa basándose en su topología y valor de los pesos de conexión. La o las capas ocultas reciben la información de entrada y mediante la evaluación de la misma -más adelante veremos en que consiste esta evaluación- emiten un resultado hacia las neuronas de salida.

## 2.4. Modo de operación

Al hablar de modo de operación se está haciendo referencia a la manera en que la red neuronal procesa los estímulos externos y crea la respuesta de salida. Puede considerarse a una red neuronal como perteneciente a una de dos grandes categorías:

- **Redes estáticas**. En este tipo de red una vez establecido el valor de las entradas las salidas alcanzan un valor estacionario independientemente de las entradas en el instante anterior, y en un tiempo siempre por debajo de una determinada cota. Estas redes se pueden caracterizar estructuralmente por la inexistencia de bucles de realimentación y de elementos de retardo entre los distintos PE que las forman. Debido a su modo de funcionamiento, estas redes tienen una capacidad limitada para sintetizar funciones dependientes del tiempo en comparación con las que detallaremos en el siguiente punto.
- **Redes dinámicas**. Este tipo responde de manera diferente ante distintas secuencias de entradas, haciendo uso de manera implícita o explícita de la variable *tiempo*. Este aspecto las hace en más idóneas que las redes estáticas para la síntesis de funciones en las que aparezca de alguna manera el parámetro tiempo.

La inclusión del elemento *tiempo* se puede llevar a cabo de varias formas:

- Inclusión explícita de un *retardo temporal*, generalmente mediante la superposición de una red estática con unas funciones de retardo (habitualmente un buffer que almacena datos antiguos).
- Uso de una dinámica de realimentación o redes neuronales *recurrentes*. En este tipo de redes existen bucles entre las conexiones de los distintos elementos de proceso por lo que su representación mediante un grafo contendrá ciclos. Estas realimentaciones pueden ser de muy diferentes tipos, dándose también la posibilidad de que la salida de un elemento de proceso sea utilizada también

como una de sus propias entradas. Es posible considerar también varias opciones en cuanto a la manera en que se realiza la realimentación:

- **Realimentación de la salida:** si las salidas generadas por la red son realimentadas a las capas anteriores.
- **Realimentación del estado:** si la realimentación se produce con las salidas producidas en las capas ocultas.

Las redes con realimentación plantean problemas de convergencia y estabilidad, que son en general de difícil análisis.

También es posible caracterizar a las redes neuronales teniendo en cuenta la forma de operar a la hora de generar la salida o al actualizar los pesos. Así se tiene:

- **Operación síncrona.** Se dice que una red neuronal opera de forma síncrona cuando todos los elementos de proceso del sistema generan la salida a la vez.
- **Operación asíncrona.** Se dice que una red opera de forma asíncrona cuando los elementos de proceso que la constituyen generan la salida aleatoriamente e independientemente unos de otros. En estos casos puede añadirse a los elementos de proceso entradas de control que indiquen cuando han de ser actualizados los pesos de sus conexiones con los otros PE.

Tengamos en cuenta que en muchos modelos de redes la actualización de los pesos requiere la ejecución de la red a fin de calcular el correspondiente error. Luego el cambio en el valor de los pesos también queda influido por el modo de operación síncrona o asíncrona.

## 2.5. Operaciones de capa

Se trata de operaciones que afectan a la capa como un todo. En principio se considerarán las siguientes:

- **Normalización.** Cada elemento de proceso de la capa considerada ajusta su salida para dar un nivel constante de actividad teniendo en cuenta las salidas de todos los elementos de proceso que forman la capa.
- **Competencia.** Sólo uno o unos pocos elementos de proceso de una capa ganan y producen salida, inhibiendo la generación de las salidas en el resto.

## 2.6. El perceptrón

Como se mencionó en la sección 2.2, el modelo de nodo propuesto por McCulloch y Pitts no incluía ninguna pauta acerca del aprendizaje. El perceptrón, creado por el psicólogo Frank Rosenblatt en 1957, consistió en el primer dispositivo de aprendizaje. El término perceptrón se puede aplicar tanto a un único PE, como a una red formada por éstos. Rosenblatt creía que la conectividad existente en las redes biológicas tiene un elevado porcentaje de aleatoriedad, por lo que se oponía al análisis de McCulloch-Pitts en el cual se empleaba la lógica simbólica para analizar estructuras bastante idealizadas. Este psicólogo suponía que la herramienta de análisis más apropiada era la teoría de probabilidades.

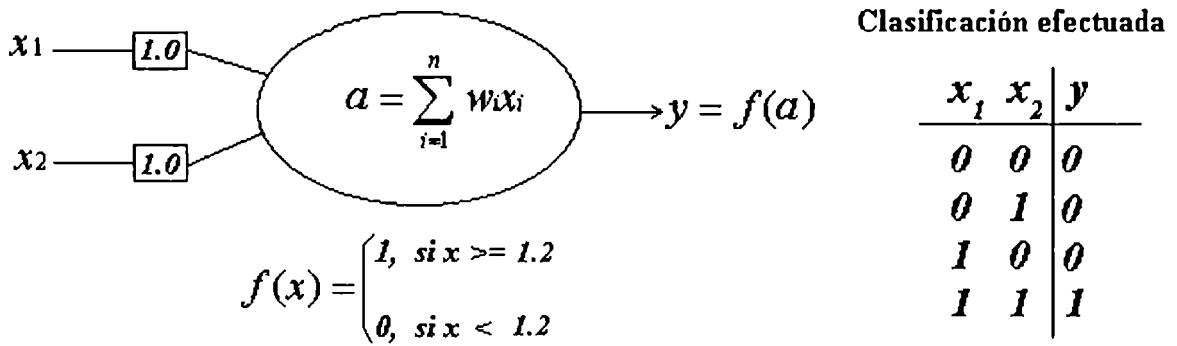


Figura 3: Definición de un Perceptrón simple. La tabla de la derecha muestra el valor de salida del Perceptrón para cada entrada binaria.

El perceptrón debía ser entrenado para conseguir la capacidad de clasificar diferentes patrones que se le presentaban. El tipo de entrenamiento se denomina *entrenamiento supervisado*. Este consiste básicamente en presentar a la RNA cada patrón de entrada con su salida esperada. Es decir para cada patrón debe conocerse la respuesta entonces la red aprenderá a generalizar este comportamiento. En base a esta respuesta se ajustan los pesos de las conexiones que influyeron en las respuestas de manera que la red origine la salida correcta para ese patrón. De esta manera se distinguen dos etapas en la formación del perceptrón. Una etapa de aprendizaje, en la cual se presentan las entradas con sus salidas deseadas a fin de acondicionar la configuración de la red para responder de forma correcta; y una etapa de operación en la cual la red responde correctamente cuando se le proporcionan nuevamente dichos patrones.

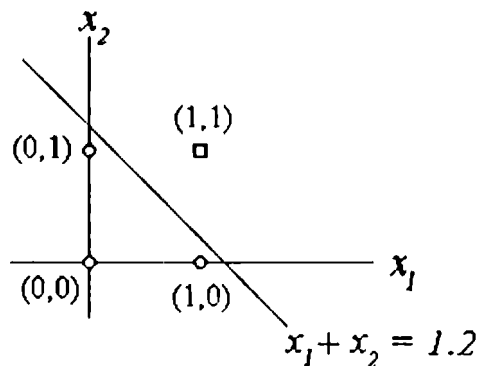


Figura 4: Interpretación geométrica de la función de clasificación que realiza el Perceptrón definido en la Figura 3. Claramente la salida de la neurona será 1 si como entrada recibe cualquier punto “por encima” o perteneciente a la recta, y 0 en caso contrario.

El primer modelo surgido del perceptrón fue el fotoperceptrón, el cual emulaba el comportamiento del ojo humano ya que respondía a señales ópticas y formaba clases de patrones en las cuales agrupaba los patrones similares.

Un perceptrón simple, es decir una red perceptrón formada por un único PE tipo perceptrón, distingue solamente dos clases de patrones dentro de un conjunto de entrada. La Figura 3 muestra un ejemplo de perceptrón y una clasificación, la cual puede decirse que consiste en la agrupación de patrones de entrada formados por dos bits de acuerdo al resultado de aplicar el operador AND entre ambos.

La clasificación realizada tiene la interpretación geométrica que muestra la Figura 4 y se deduce del siguiente razonamiento:

Podemos reescribir  $f(a)$  como:

$$f(a) = \begin{cases} 1 & \text{si } w_1 \cdot x_1 + w_2 \cdot x_2 \geq u \\ 0 & \text{si } w_1 \cdot x_1 + w_2 \cdot x_2 < u \end{cases}$$

Ahora examinemos la siguiente ecuación:

$$w_1 x_1 + w_2 x_2 = u$$

En el plano  $x_1 x_2$  representa la ecuación de una recta. En el ejemplo presentado, para  $u=1.2$ ,  $w_1=w_2=1$  la recta queda definida por la ecuación  $x_1+x_2=1.2$ .

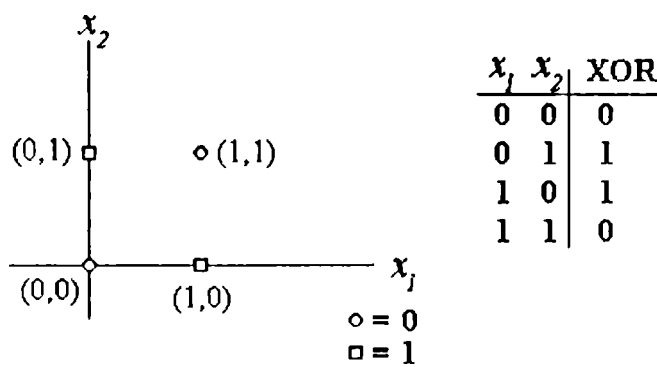
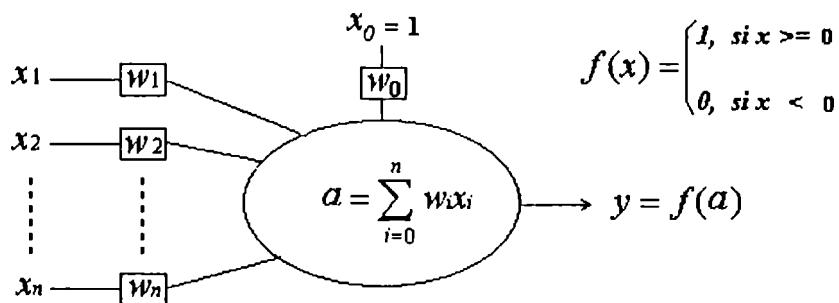


Figura 5: Disposición de los patrones y como deberían clasificarse para el problema del XOR.

De esto se desprende que un perceptrón simple puede únicamente clasificar en dos clases, patrones que pueden separarse linealmente, es decir, con un hiperplano<sup>2</sup> de dimensionalidad  $n-1$  en un espacio de patrones de  $n$  dimensiones, debe poder obtenerse la clasificación pretendida para esperar que este tipo de red la aprenda. Esta limitación no le permite resolver un problema sencillo como el del operador XOR, dado que no alcanza con una única recta en el plano para separar las dos clases de patrones. La Figura 5 muestra como debería realizarse la clasificación. Claramente se puede ver que con una única recta no es suficiente.

En 1969 se publicó el libro “Perceptrons: An Introduction to Computational Geometry” escrito por Marvin Minsky y Seymour Paper ([92]), trata en detalle esta limitación de los perceptrones poniendo en evidencia la restrictiva aplicabilidad de los mismos. Esto originó un estancamiento de más de 15 años del estudio e investigación en redes neuronales. Finalmente el problema se superó con la aparición del algoritmo de entrenamiento denominado Backpropagation.

<sup>2</sup> Un hiperplano dentro de un hiperespacio de dimension  $n$ , es un objeto de  $n-1$  dimensiones que puede descomponer dicho hiperespacio en dos regiones. Por ejemplo, una recta es un hiperplano dentro de un espacio de dos dimensiones, un plano es un hiperplano dentro de un espacio de 3 dimensiones y así siguiendo. Una adecuada colocación de los hiperplanos permiten descomponer el hiperespacio en varias regiones.



**Figura 6: Perceptrón simple con término de tendencia. Esta variación incluye la nueva entrada  $x_0$  llamada término de tendencia.**

Un aspecto muy importante que hemos dejado a un lado es el tratamiento del umbral. Si para el ejemplo se hubiese utilizado un valor de umbral de 0 en lugar de 1.2, la recta discriminante no podría nunca separar los patrones. Si se incluye el valor del umbral como una conexión de entrada más, la cual denominamos conexión de tendencia, el valor puede ser ajustado en el mismo proceso de entrenamiento. Veamos que la ecuación

$$\sum_{i=1}^n w_i x_i = u$$

Puede reescribirse como

$$\sum_{i=1}^n w_i x_i - u = 0$$

Y haciendo  $w_0 = -u$ , y  $x_0 = 1$  nos queda

$$\sum_{i=0}^n w_i x_i = 0$$

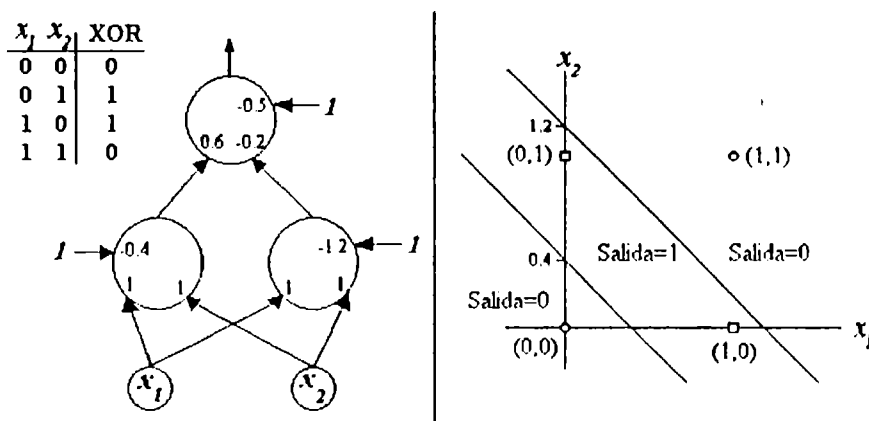
La Figura 6 muestra una nueva versión del perceptrón la cual requiere reescribir la activación ahora con  $(n+1)$  entradas, por el agregado del término de tendencia  $x_0$  cuyo valor es siempre 1. También se reescribe la función de transferencia  $f(x)$  ya que el umbral ahora es cero.

Volviendo al problema del XOR, mostraremos una manera de resolverlo utilizando una red de perceptrones o un *perceptrón multicapa* en vez de un perceptrón simple.

Como se explicó anteriormente, un perceptrón simple puede resolver sólo problemas linealmente separables ya que sólo puede separar dos regiones en el espacio de patrones debido a que puede generar un único hiperplano que lo divide de manera lineal.

Pero como ya lo mencionamos, varios hiperplanos pueden descomponer el hiperespacio en varias regiones, dependiendo su ubicación. Por ejemplo, dos rectas distintas pueden dividir un plano en 3 regiones si son paralelas o en 4 si no lo son. Con esta idea la solución al problema del XOR se obtiene combinando varios perceptrones de manera que generen varias rectas que dividen al plano. De este modo obtenemos lo que se llama un *Perceptrón Multicapa*. Dicha red se muestra la Figura 7. Como se ve esta red agrega dos nodos, cada uno de los cuales reciben los patrones de entrada y los clasifica en dos regiones. Por ejemplo, el de la izquierda con una recta  $x_2 = x_1 + 0.4$  separa el patrón de entrada (0,0) del resto mientras que el de la derecha con una recta  $x_2 = x_1 + 1.2$  separa el

patrón de entrada (1,1) del resto. Mediante el tercer nodo que integra la salida de los otros dos, los patrones que se encuentran dentro de las zonas en común que dividen las rectas son clasificados dentro de la misma clase con un valor de salida de 1. Los patrones que están en zonas disjuntas se clasifican con una respuesta de valor 0.



**Figura 7:** El problema del XOR resuelto. La red perceptrón de la izquierda es un ejemplo de cómo resolver el problema del XOR con estos PE. A la derecha se muestra la disposición de las rectas que descomponen al plano de manera de clasificar los patrones.

De todas maneras, el XOR es un problema de los denominados “de juguete” y el hecho que con una simple modificación una red de unidades perceptrones haya podido resolver un problema no linealmente separable, no significa que toda crítica realizada por Minsky y Paper se supere de esta manera. En su obra mencionan el problema del escalado, explicando que algunas redes pueden sufrir efectos secundarios indeseables cuando se expande demasiado su estructura. La ampliación de la red para la resolución del XOR no resultó en una red de gran tamaño debido justamente a que se trata de un problema que no lo requiere. La cuestión se plantea al resolver grandes problemas reales. Sin embargo hay mucha diferencia entre la potencia de cálculo de los equipos de la época de la obra de Minsky y Paper y los de hoy en día. Esto sumado al constante progreso del desarrollo del hardware permite pensar que la gran cantidad de operaciones que debe hacerse para evaluar una red neuronal de tamaño considerable no debería ser un obstáculo.

### 2.6.1. Aprendizaje del Perceptrón

Ya hemos analizado cómo se comporta un Perceptrón en la fase de operación pero no detallamos nada en absoluto sobre el entrenamiento a partir del cual este tipo de PE es capaz de aprender. El aprendizaje consiste en someter al dispositivo a una serie de muestras para que éste de manera automática vaya modificando sus pesos de conexión hasta que su comportamiento sea el deseado.

Los trabajos de Rosenblatt dieron lugar a que se demostrase un importante resultado conocido con el nombre de teorema de convergencia del Perceptrón. Este teorema afirma que para un Perceptrón simple que está aprendiendo a diferenciar tramas de dos clases diferentes, si es que la clasificación puede ser aprendida –recordemos que deben ser clases linealmente separables– una simple regla de corrección de los pesos en base a las desviaciones de las respuestas obtenidas garantiza la obtención de los pesos  $w_i$  que configuran la respuesta correcta del PE en un número finito de pasos. Este



procedimiento determina la regla de aprendizaje de la red, conocida como regla de Hebb.

La tarea consiste en determinar los pesos sinápticos para que el dispositivo represente la relación entrada/salida lo más cercano posible a la relación real entre los patrones.

### 2.6.2. Regla de aprendizaje y algoritmo de entrenamiento

El aprendizaje del Perceptrón es de tipo supervisado. Esto implica que se debe conocer la salida esperada por cada patrón de entrada a clasificar. Será necesario elegir una secuencia de ejemplos  $(\bar{x}_1, d_1), (\bar{x}_2, d_2), \dots, (\bar{x}_l, d_l)$  de pares de entrada/salida correctos. El Perceptrón a través del proceso de entrenamiento debe aprender que a cada vector de entrada  $\bar{x}_i$  le corresponde la salida  $d_i$ , con  $d_i$  perteneciente al conjunto  $\{0,1\}$ .

En el proceso de entrenamiento el Perceptrón se expone repetidamente a la secuencia de ejemplos hasta que los pesos de la red son ajustados de forma que al final del entrenamiento se obtengan las salidas esperadas para cada uno de los patrones de entrada.

El algoritmo de entrenamiento puede resumirse así:

Inicializar los pesos  $w_i$  con valores aleatorios

Repetir

Para cada par de entrenamiento  $j$  hacer

$y :=$  salida del Perceptrón para  $\bar{x}_j$

si  $y = d_j$  «no hacer nada. Clasificación correcta de  $\bar{x}_j$ » (1)

si  $(y = 0)$  and  $(d_j=1)$  «hacer  $\bar{w} := \bar{w} + \bar{x}_j$ » (2)

si  $(y = 1)$  and  $(d_j=0)$  «hacer  $\bar{w} := \bar{w} - \bar{x}_j$ » (3)

Hasta que todos los patrones hayan sido correctamente clasificados

Debe observarse que las tres expresiones anteriores (1), (2) y (3) pueden escribirse en forma compacta como  $\bar{w} := \bar{w} + (d_j - y)\bar{x}_j$ .

Esta fue la primera regla de aprendizaje que se utilizó para entrenar un Perceptrón. Más adelante se agregó un parámetro llamado *velocidad de aprendizaje*, denotado en este trabajo con la letra  $\mu$ , el cual regula el tamaño de los cambios en el vector de peso, un valor adecuado de éste evitará que el sistema oscile continuamente. Este parámetro se fija antes del entrenamiento y por lo general es un número pequeño, habitualmente su valor es menor a 0.25, de esta manera la fórmula para modificar el vector de pesos se escribe así:

$$\bar{w} = \bar{w} + \mu(d_j - y)\bar{x}_j, \mu \in (0,1)$$

Contando con este parámetro puede probarse el teorema para la regla de aprendizaje del Perceptrón que asegura la convergencia hacia la solución luego de un número finito de pasos de corrección de los pesos. Esto ocurrirá sólo si dicha solución es factible, es decir si el problema de clasificación es linealmente separable.

Para ilustrar este algoritmo de aprendizaje mostraremos un pequeño ejemplo. Entrenaremos a un Perceptrón para que aprenda la clasificación que se muestra en la Figura 8.

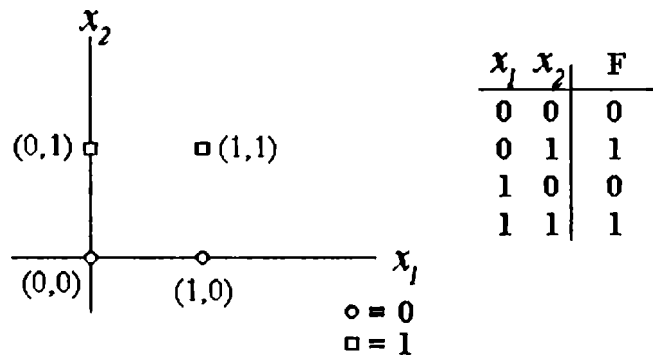


Figura 8: La clasificación especificada por F es linealmente separable, así que un Perceptrón será capaz de aprenderla en un número finito de pasos de entrenamiento

En la Tabla 1 presentamos el entrenamiento de un Perceptrón cuyo vector de pesos es inicializado en (0, 0, 0). El parámetro velocidad de aprendizaje se ha elegido  $\mu=0.5$ . La secuencia de patrones de entrenamiento será presentada tantas veces como sea necesario hasta que sea clasificada correctamente en toda su totalidad.

Paso	$x_0$	$X_1$	$X_2$	$d$	$w_0$	$w_1$	$w_2$	$Y$	Actualización de pesos		
									$W_0$	$w_1$	$w_2$
1	1	0	0	0	0	0	0	1	-0,5	0	0
2	1	0	1	1	-0,5	0	0	0	0	0	0,5
3	1	1	0	0	0	0	0,5	1	-0,5	-0,5	0,5
4	1	1	1	1	-0,5	-0,5	0,5	0	0	0	1
5	1	0	0	0	0	0	1	1	-0,5	0	1
6	1	0	1	1	-0,5	0	1	1	-0,5	0	1
7	1	1	0	0	-0,5	0	1	0	-0,5	0	1
8	1	1	1	1	-0,5	0	1	1	-0,5	0	1
9	1	0	0	0	-0,5	0	1	0	-0,5	0	1

Tabla 1: Ejemplo de tabla de entrenamiento de un perceptrón.

En el paso de entrenamiento número 5 el Perceptrón realiza la última actualización del vector de pesos alcanzando el aprendizaje buscado. No obstante sólo se puede estar seguro al comprobar que se clasificaron correctamente todos los patrones de entrada, es por ello que el entrenamiento debió proseguir hasta el paso número 9.

En las figuras mostradas a continuación (Figura 9, Figura 10, Figura 11, Figura 12 y Figura 13) se muestra la interpretación geométrica del aprendizaje en el plano  $X_0=1$ .

A partir de la quinta actualización del vector de pesos el dispositivo realiza correctamente la clasificación de todos los patrones de entrada. El entrenamiento termina en el paso número 9, una vez que se comprueba que todos los vectores de entrada han sido bien clasificados.

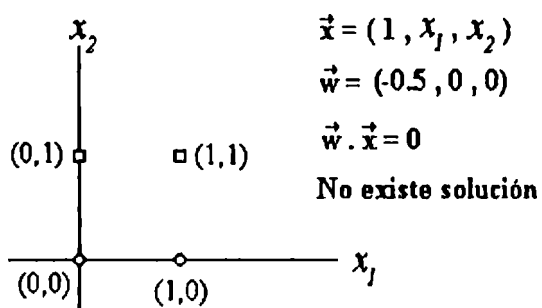


Figura 9: Primer paso de actualización en el entrenamiento de un perceptrón para el problema del XOR.

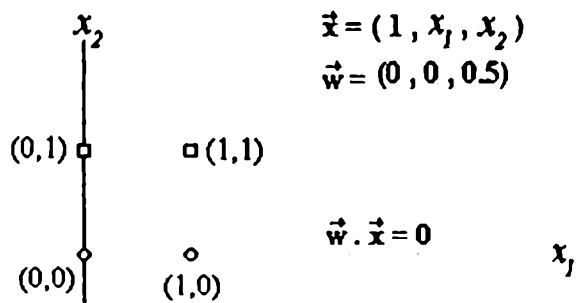


Figura 10: Segundo paso de actualización en el entrenamiento de un perceptrón para el problema del XOR.

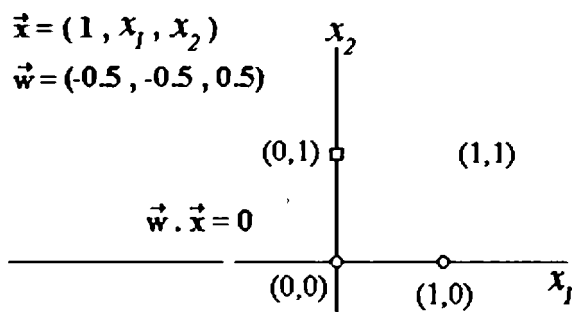


Figura 11: Tercer paso de actualización en el entrenamiento de un perceptrón para el problema del XOR.

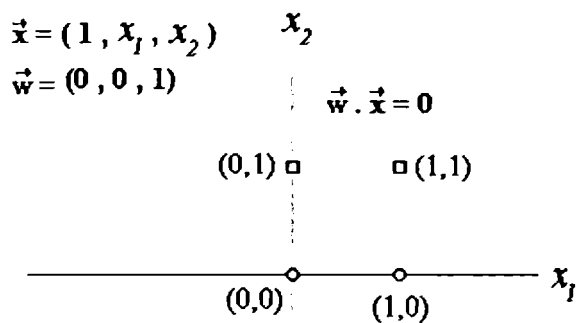


Figura 12: Cuarto paso de actualización en el entrenamiento de un perceptrón para el problema del XOR.

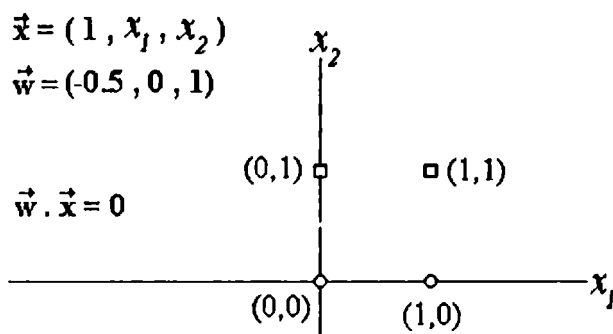


Figura 13: Quinto paso de actualización en el entrenamiento de un perceptrón para el problema del XOR.

### 2.6.3. Perceptrón unicapa

Un único Perceptrón simple tiene un alcance de aplicación demasiado limitado, ya que sólo es capaz de distinguir sólo dos grupos de patrones. Se puede aumentar su potencia utilizando varios perceptrones simples trabajando en paralelo con las mismas entradas. Esta red unicapa o *asociador lineal* y fue inventada por múltiples personas durante el período entre 1968 y 1972 [54].

Cada elemento de procesamiento –PE<sub>i</sub>– de este dispositivo es un Perceptrón simple que se encarga de aprender la correlación existente entre los patrones de entrada y una componente del vector de salida. Pero basta con que sólo un PE se enfrente a una clasificación no linealmente separable para que todo el dispositivo falle en el aprendizaje. Por ejemplo, supongamos que queremos entrenar a un Perceptrón unicapa para que aprenda la función *div2* –parte entera de dividir por dos– de números binarios de hasta tres cifras. Esto significa que la red debe aprender a realizar una clasificación en cuatro grupos o clases determinada por la Tabla 2.

Para resolver el problema debe armarse una red como la presentada en la Figura 14 compuesta por dos perceptrones simples y entrenarlos hasta lograr el comportamiento esperado. El algoritmo de entrenamiento es el mismo que vimos para el caso de un único PE y puede pensarse como dos entrenamientos independientes donde cada PE<sub>i</sub> aprenderá a correlacionar la entrada con la componente  $y_i$  del vector de salida.

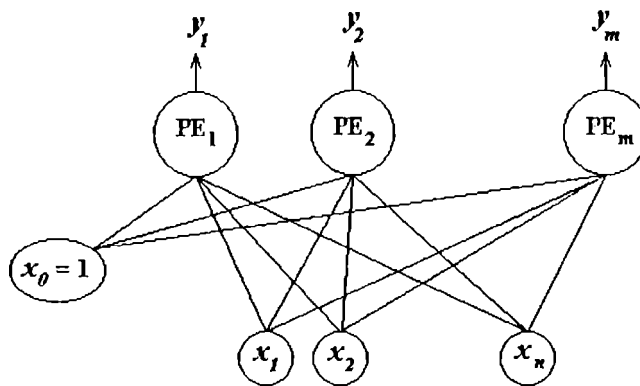


Figura 14: Perceptrón unicapa.

$\bar{x}$	$\bar{y}$
(0,0,0)	(0,0)
(0,0,1)	(0,0)
(0,1,0)	(0,1)
(0,1,1)	(0,1)
(1,0,0)	(1,0)
(1,0,1)	(1,0)
(1,1,0)	(1,1)
(1,1,1)	(1,1)

Tabla 2: Tabla de patrones y resultados esperados para la función *div2* de números de hasta tres dígitos binarios.

Perceptrón unicapa		PE <sub>1</sub>				PE <sub>2</sub>			
$\vec{X}$	$\vec{Y}$	$x_1$	$x_2$	$x_3$	$y_1$	$x_1$	$x_2$	$x_3$	$y_2$
(0,0,0)	(0,0)	0	0	0	0	0	0	0	0
(0,0,1)	(0,0)	0	0	1	0	0	0	1	0
(0,1,0)	(0,1)	0	1	0	0	0	1	0	1
(0,1,1)	(0,1)	0	1	1	0	0	1	1	1
(1,0,0)	(1,0)	1	0	0	1	1	0	0	0
(1,0,1)	(1,0)	1	0	1	1	1	0	1	0
(1,1,0)	(1,1)	1	1	0	1	1	1	0	1
(1,1,1)	(1,1)	1	1	1	1	1	1	1	1

Figura 15: El aprendizaje de la función div2 se realiza por medio del entrenamiento de dos perceptrones simples PE<sub>1</sub> y PE<sub>2</sub>.

Como mencionamos anteriormente el éxito del algoritmo de aprendizaje radica en que ambos PEs deban realizar una clasificación linealmente separable. Puede verse fácilmente que efectivamente, como lo muestran la Figura 15, la Figura 16 y la Figura 17, que esto es así.

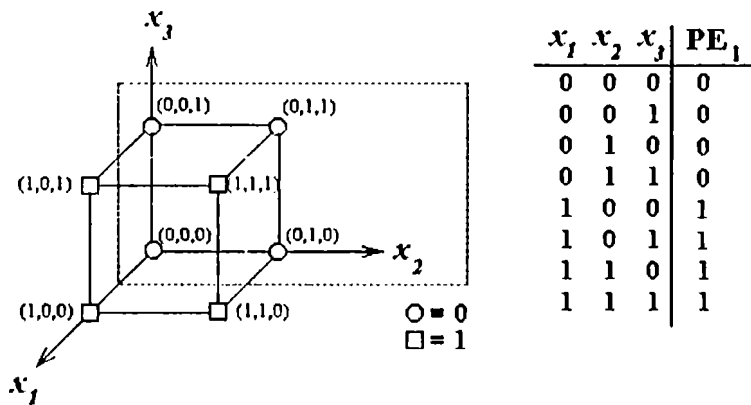


Figura 16: La clasificación será aprendida por PE<sub>1</sub> por ser linealmente separable.

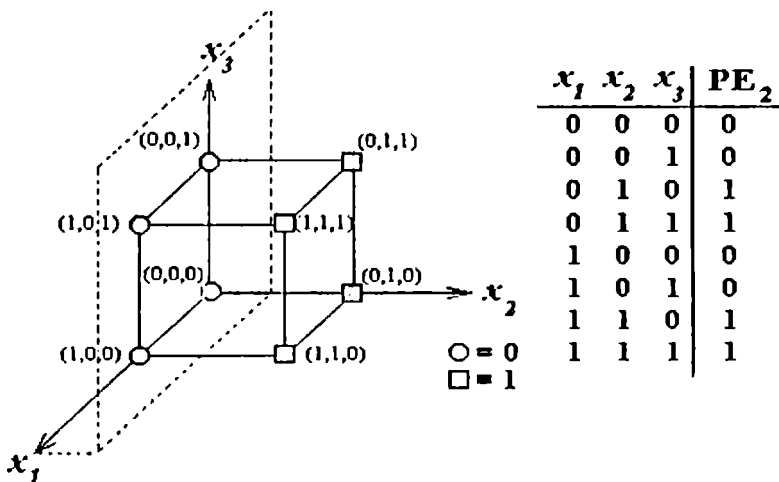


Figura 17: La clasificación será aprendida por PE<sub>2</sub> por ser linealmente separable.

Puede concluirse que el Perceptrón unicapa será entrenado con éxito en el aprendizaje de la función *div2*.

Si los patrones de entrada se encuentran en un espacio de dos dimensiones puede ser posible determinar de manera gráfica si un problema es linealmente separable o no. Sin embargo, esta visualización se dificulta cuando el conjunto de patrones de entrada es de tres dimensiones, y resulta imposible de observar gráficamente cuando los patrones de entrada son de dimensiones superiores. En este último caso se requiere plantear condiciones de desigualdad que permitan comprobar la separabilidad lineal de los patrones. Se debe resolver el sistema formado por tantas inecuaciones como patrones de entrada existan y esto se realiza con base en la ecuación de salida del Perceptrón de la siguiente manera:

$$\sum_{i=0}^n w_i x_i \geq 0, \quad \text{para aquellos patrones cuya salida deseada sea 1}$$

$$\sum_{i=0}^n w_i x_i < 0, \quad \text{para aquellos patrones cuya salida deseada sea 0}$$

#### 2.6.4. Perceptrón multicapa

Volvamos por un instante al problema de clasificación de patrones de la función XOR, que como ya se dijo, es imposible de ser realizado correctamente por un Perceptrón simple. Intentemos aproximar una solución. La Figura 5 sugiere que se podría descomponer correctamente el espacio con dos hiperplanos de manera de obtener tres regiones, como se muestra en la Figura 7. Los puntos dentro de la región acotada por los dos hiperplanos pertenecerían a una de las clases de salida y los que se encuentran fuera de ella formarían parte de la segunda clase de salida. Así que si en lugar de utilizar únicamente una neurona de salida se utilizaran dos, se obtendrían dos rectas por lo que podrían delimitarse las tres zonas necesarias. Pero para poder elegir entre una zona u otra de las tres, es necesario utilizar otra capa con una neurona cuyas entradas serán las salidas de las neuronas anteriores, por tanto se ha de utilizar una red de tres neuronas, distribuidas en dos capas para solucionar este problema.

La adición de dos unidades de capa oculta, o capa intermedia brindan a la red la flexibilidad necesaria para resolver el problema del XOR. Un Perceptrón multicapa es una red con alimentación hacia delante –feedforward–, compuesta de varias capas de neuronas entre la entrada y la salida de la misma, la cual permite separar en regiones de decisión mucho más complejas que las de dos semiplanos, como lo hace el Perceptrón de un solo nivel.

El Perceptrón simple sólo puede establecer dos regiones separadas por una frontera lineal en el espacio de entrada de los patrones. Un Perceptrón con dos capas, puede formar cualquier región convexa en este espacio. Las regiones convexas se forman mediante la intersección de las regiones que definen cada neurona de la capa oculta, cada uno de estos elementos se comporta como un Perceptrón simple, activándose su salida para los patrones de un lado del hiperplano. Si la neurona del nivel de salida implementa la función lógica AND, la región de decisión resulta ser la intersección de todos los semiplanos formados en el nivel anterior. Esta región de decisión será una región convexa con un número de lados a lo sumo igual al número de neuronas de la capa oculta.

Surge aquí la cuestión de la política de selección para las neuronas de las capas ocultas de una red multicapa, este número en general debe ser lo suficientemente grande como para que se forme una región compleja que pueda resolver el problema, sin embargo no debe ser muy grande pues la estimación de los pesos puede ser no confiable para el conjunto de los patrones de entrada disponibles.

Hasta el momento no hay un criterio establecido para determinar la configuración de la red y esto depende más bien de la experiencia del diseñador. Puede verse en la Figura 18 una cierta regla general en esta materia. Otra opción es intentar alguna forma automática para determinar este y otros parámetros de la red. La Neuroevolución –que veremos más adelante– puede constituir una solución aceptable esta problemática.

Esto último no pretende implicar que todas las críticas del Perceptrón podrían tener respuesta añadiendo capas ocultas a la estructura. Sino que las técnicas siguen avanzando hacia sistemas cuyas capacidades son cada vez mayores. De hecho el Perceptrón multicapa no se transformó en una red exitosa sino hasta la aparición de un algoritmo de aprendizaje adecuado llamado *Regla delta generalizada*, el cual aplicado a una este tipo de redes se ha denominado *Backpropagation*. Actualmente este es el algoritmo que más se emplea en la obtención de los pesos de red multicapa de perceptrones y constituye una generalización del *algoritmo de mínimos cuadrados* –introducido por la red *Adaline*–. En él se emplea una técnica de búsqueda del gradiente que minimice el error esperado entre la salida actual y la deseada.

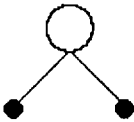

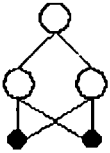

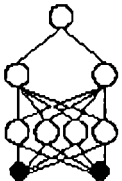

<b>Estructura</b>	<b>Tipos de regiones de decisión</b>	<b>Formas de las regiones de decisión</b>
<p>una capa</p> 	<p>medio plano limitado por un hiperplano</p>	
<p>dos capas</p> 	<p>regiones convexas o cerradas</p>	
<p>tres capas</p> 	<p>arbitraria (limitada por el número de nodos)</p>	

Figura 18: Tipos de arquitecturas de redes neuronales y tipos de regiones que cada una puede clasificar.

## 2.7. El Adaline

El Adaline fue creado como un tipo de filtro para el procesamiento de señales. En principio se trata de un único elemento de procesamiento, y como tal no constituye por sí solo una red neuronal, de igual manera que el perceptrón. Sin embargo se trata de una estructura muy importante ya que la principal diferencia respecto de su antecesor radica en que la salida no es binaria sino que puede tomar cualquier valor real. La Figura 19 muestra la estructura del adaline.

Introducido por Bernard Widrow en 1959, el adaline tuvo su propio método de entrenamiento, la *Regla Delta* o también conocida como *Regla de aprendizaje de mínimos cuadrados*, la cual será explicada más adelante.

Su aporte al filtrado de señales fue muy importante debido a su adaptabilidad a los cambios en la aplicación de los filtros. Con su método de aprendizaje es capaz de adaptarse a sí mismo a nuevas condiciones. Esto no pasaba con los elementos de filtrado tradicionales, ya que en su mayoría se trataban de circuitos RLC difícilmente modificables. Igualmente, el Adaline padece de algunas de las limitaciones del perceptrón, como por ejemplo, el no poder resolver problemas de clasificación no linealmente separables. De la misma manera, la combinación de varios elementos Adaline permite superar este problema. Así surge el *Madaline*.

El término Madaline consiste en las siglas de “muchos Adalines” y consiste en la combinación de varios de estos PE en una estructura multicapa como lo muestra la Figura 20.

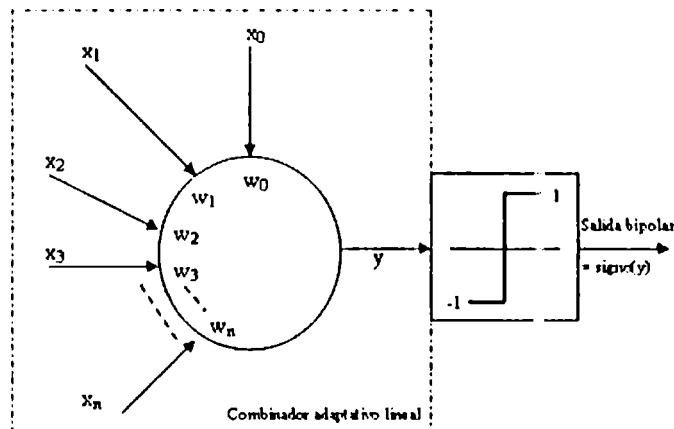


Figura 19: El adaline completo consta del combinador adaptativo lineal, que está dentro del cuadro de trazos, y de una función bipolar de salida. El combinador adaptativo lineal se asemeja al elemento de procesamiento general representado en la Figura 1.



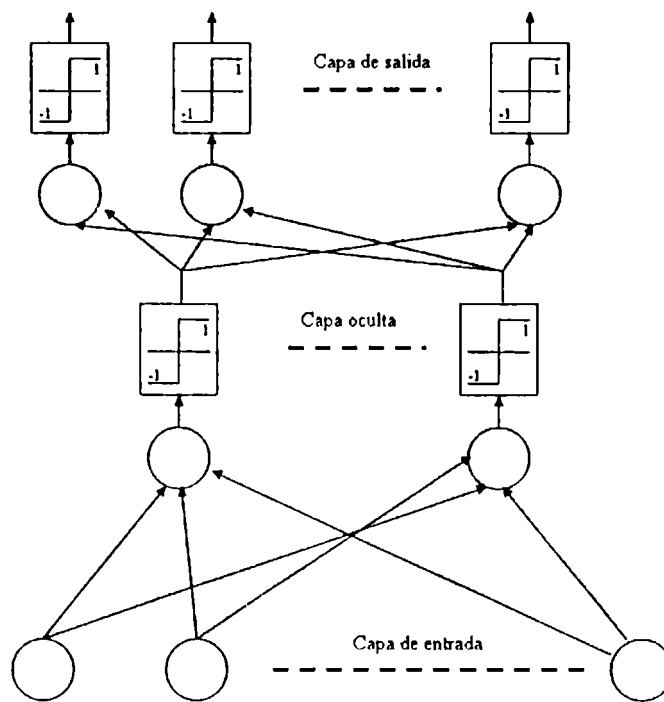


Figura 20: Madaline. Ejemplo de red neuronal formada por varias unidades Adaline.

Con el entrenamiento adecuado, una estructura de este tipo podría responder con un valor positivo en alguna de sus salidas para clasificar un patrón de entrada en una determinada categoría. Por ejemplo, puede recibir cada uno de los bits resultantes de un proceso de barrido de alguna imagen y agruparlos en distintas categorías.

## 2.8. Backpropagation

Hemos expuesto, cuando se analizó el Perceptrón de Rosenblatt, la limitación de este dispositivo para clasificar patrones que no sean linealmente separables. También se mostró cómo es posible combinar perceptrones simples en una red multicapa que resuelva satisfactoriamente uno de estos problemas como lo constituye la función XOR. Mencionamos además que mediante el agregado de capas se puede formar cualquier región arbitraria de decisión haciendo posible cualquier clasificación.

Pero hasta el momento no detallamos en absoluto algún algoritmo de aprendizaje capaz de encontrar los pesos adecuados para este tipo de redes multicapa. Por esto presentamos en este capítulo el algoritmo de entrenamiento de la RNA *Backpropagation*.

El primer algoritmo de entrenamiento para redes multicapa fue desarrollado en 1974 en un contexto general, es decir, para cualquier tipo de redes. Las redes neuronales resultaban una aplicación especial, motivo por el cual el algoritmo no fue aceptado por quienes desarrollaban redes neuronales. Recién a mediados de los años 80 cuando el algoritmo Backpropagation o algoritmo de propagación inversa fue redescubierto al mismo tiempo por varios investigadores empezó a tener más popularidad, la cual fue en aumento cuando fue incluido en el libro "Parallel Distributed Processing Group" por los psicólogos David Rumelhart y James McClelland. La publicación de este libro trajo consigo un auge en las investigaciones con redes neuronales, siendo la Backpropagation una de las redes más ampliamente empleadas, aun en nuestros días.

La mayoría de los sistemas actuales de cómputo se han diseñado para llevar a cabo funciones matemáticas y lógicas a una velocidad que resulta asombrosamente alta para el ser humano. Sin embargo esta capacidad no es lo que se necesita para reconocer patrones en entornos ruidosos tarea que, incluso dentro de un espacio de entrada relativamente pequeño, puede llegar a consumir mucho tiempo.

Lo que se necesita es un sistema de procesamiento que sea capaz de examinar todos los patrones en paralelo. Idealmente ese sistema no tendría que ser programado explícitamente, lo que haría es adaptarse a sí mismo para aprender la relación entre un conjunto de patrones dado como ejemplo y ser capaz de aplicar la misma relación a nuevos patrones de entrada –generalización–. Este sistema debe contar con la capacidad de concentrarse en las características de una entrada arbitraria que se asemeje a otros patrones vistos previamente, sin que ninguna señal de ruido lo afecte. Este sistema fue el gran aporte de la red de propagación inversa, Backpropagation.

La Backpropagation es un tipo de red con aprendizaje supervisado, que emplea un ciclo *propagación–adaptación* de dos fases. Una vez que se ha aplicado un patrón a la entrada de la red como estímulo, este se propaga desde la primera capa a través de las capas superiores de la red, hasta generar una salida. La señal obtenida se compara con la respuesta deseada y se calcula una señal de error para cada una de las neuronas de la capa de salida.

La señal de error se propaga hacia atrás (motivo por el cual se definió el nombre de Backpropagation) a partir de la capa de salida. Dicha señal retrocede a través de las capas ocultas de la red, llegando a cada neurona de manera parcial, de acuerdo a la contribución de la misma en el resultado de la salida de la red. Cuando todas las neuronas de la red hayan recibido una señal de error termina la propagación. En base a la señal de error percibida, se actualizan los pesos de conexión de cada neurona, para hacer que la red converja hacia un estado que permita clasificar correctamente todos los patrones de entrenamiento.

Este proceso tiene como efecto provocar que las neuronas de las capas intermedias se organicen a sí mismas de tal modo que cada neurona pueda estar dedicada a reconocer una determinada característica en los patrones de entrada. De esta manera cuando se le presente un patrón arbitrario de entrada que contenga ruido o que esté incompleto, las neuronas de la capa oculta de la red responderán con una salida activa si la nueva entrada contiene un patrón que se asemeje a aquella característica que las neuronas individuales hayan aprendido a reconocer durante su entrenamiento. Por otro lado, las unidades de las capas ocultas tienen una tendencia a inhibir su salida si el patrón de entrada no contiene la característica a reconocer, para la cual han sido entrenadas.

Varias investigaciones han demostrado que, durante el proceso de entrenamiento, la red Backpropagation tiende a desarrollar relaciones internas entre neuronas con el fin de organizar los datos de entrenamiento en clases. Se puede decir entonces que todas las unidades de la capa oculta de una Backpropagation son asociadas de alguna manera a características específicas del patrón de entrada como consecuencia del entrenamiento. Esto puede no resultar evidente para el observador humano, pero la red ha encontrado una representación interna que le permite generar las salidas deseadas cuando se le dan las entradas en el proceso de entrenamiento. Esta misma representación interna se puede aplicar a entradas que la red no haya visto antes, y la misma clasificará estas entradas según las características que compartan con los ejemplos de entrenamiento.

Esto habla de la importante capacidad de generalización que adquiere la red neuronal a partir de los ejemplos presentados. Esto se encuentra relacionado directamente con

nuestras capacidades cerebrales superiores, ya que para el ser humano resulta fácil reconocer a partir de pocos ejemplos una clase de objeto. Por ejemplo, nos basta con ver uno o dos automóviles para reconocer toda una clase de medios de transporte; u observar unos pocos ejemplares de perros para darnos cuenta cual es su aspecto general y estar en condiciones de distinguir entre otros ejemplos si pertenecen o no a dicha especie animal.

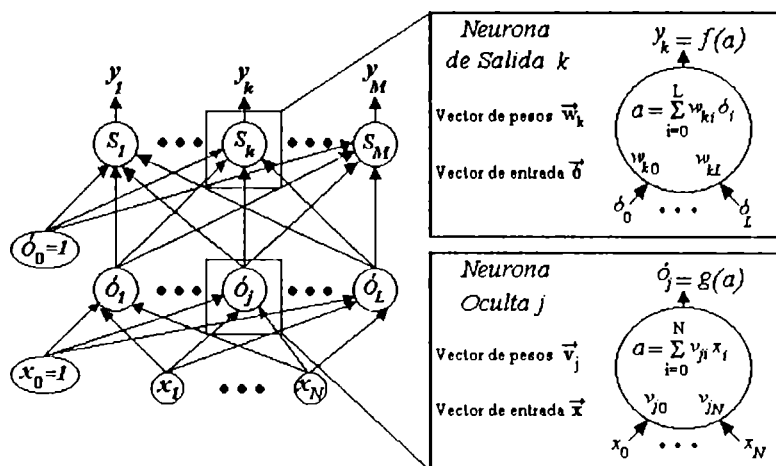


Figura 21: Arquitectura de una red Backpropagation de dos capas. Los pesos y unidades de tendencia son opcionales.

### 2.8.1. Estructura y funcionamiento de la red Backpropagation

La Backpropagation es una red formada por capas, con propagación hacia adelante (*feedforward*). Las capas se encuentran completamente interconectadas entre sí, desde la entrada hacia la salida. Por lo tanto no existen conexiones de retroalimentación, ni conexiones que salten una capa para ir directamente a otra superior.

Presentaremos la derivación de la regla de aprendizaje Delta generalizada –GDR– para una red formada por una capa oculta –entre la entrada y la capa de salida– pero se pueden admitir más.

La función de transferencia de cada PE tanto en la capa de salida como en la capa oculta puede ser cualquier función derivable. Este requisito excluye la función escalón que se utilizó en el Perceptrón pues no es derivable en todo su dominio. Las funciones más utilizadas son la función Identidad y la sigmoide logística (ver Figura 22). La selección de esta función depende de la forma en que se pretende representar los datos de salida. Por ejemplo, si se desea que las unidades de salida sean binarias, se utiliza una función sigmoide, porque esta función limita la salida y es casi biestable. En otros casos es tan aplicable una función de salida lineal como una sigmoide.

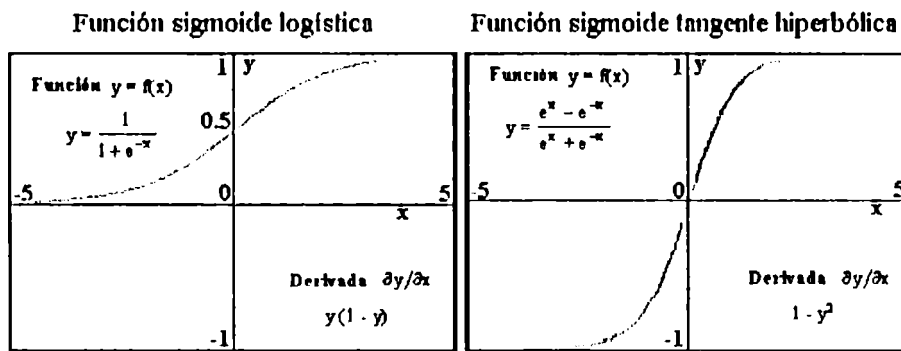


Figura 22: Funciones sigmoides

### 2.8.2. Regla de aprendizaje Delta generalizada –GDR–

Supongamos que tenemos un conjunto de pares de vectores de la forma  $(\vec{x}, \vec{d})$  que son ejemplos de una correspondencia funcional que se pretende sea aprendida por la red, con  $\vec{x} \in \mathbb{R}^N$  y  $\vec{d} \in \mathbb{R}^M$ . Derivaremos un método para entrenarla, suponiendo que los pares de vectores de entrenamiento se hayan seleccionado adecuadamente y que haya un número suficiente de ellos. Más adelante daremos precisiones sobre que significa *adecuadamente y suficiente*.

El algoritmo propiamente dicho no difiere del ya presentado para el Perceptrón sino en la forma de actualizar los pesos. Se presenta un vector de entrada, se calcula la salida, se determina el error, se corrigen los pesos de las conexiones y se vuelve a repetir el proceso hasta que el error se reduzca a un valor aceptable.

### 2.8.3. Actualización de pesos de la capa de salida

Para cada presentación de un vector de entrada se pretende minimizar el error global de la capa de salida que está formado por la suma de los errores de cada unidad de salida. Para mostrar el proceso de actualización de pesos que minimiza dicho error nos basaremos en el ejemplo de red de la Figura 21. Más precisamente la expresión del error que se desea minimizar es:

$$E = \frac{1}{2} \sum_{k=1}^M (d_k - y_k)^2$$

El factor 1/2 de la ecuación aparece por conveniencia para simplificar los cálculos. Además está claro que si minimizamos  $E$  también estamos minimizando  $\sum (d_k - y_k)^2$ .

Puede verse a  $E$  como una función de todas las componentes de todos los vectores de pesos de las neuronas de salida. Para determinar el sentido en que se deben cambiar los pesos, se calcula el valor negativo del gradiente de  $E$  respecto de todos los pesos  $w_{ki}$ .

$$\vec{\nabla} E = \left( \frac{\partial E}{\partial w_{10}}, \frac{\partial E}{\partial w_{11}}, \dots, \frac{\partial E}{\partial w_{1L}}, \frac{\partial E}{\partial w_{20}}, \frac{\partial E}{\partial w_{21}}, \dots, \frac{\partial E}{\partial w_{2L}}, \dots, \frac{\partial E}{\partial w_{M0}}, \frac{\partial E}{\partial w_{M1}}, \dots, \frac{\partial E}{\partial w_{ML}} \right)$$

Calculemos ahora la componente genérica correspondiente a la  $i$ -ésima coordenada del vector de pesos de la  $k$ -ésima neurona de salida.

$$\frac{\partial E}{\partial W_{ki}} = \frac{1}{2} 2(d_k - y_k)(-1) \frac{\partial y_k}{\partial W_{ki}}$$

$$\frac{\partial y_k}{\partial W_{ki}} = f'(a) \frac{\partial a}{\partial W_{ki}}$$

$$\frac{\partial a}{\partial W_{ki}} = \delta_i$$

$$\therefore \frac{\partial E}{\partial W_{ki}} = -(d_k - y_k) f'(a) \delta_i$$

$$\therefore -\frac{\partial E}{\partial W_{ki}} = (d_k - y_k) f'(a) \delta_i$$

El cambio de peso será en el sentido y proporcional al gradiente negativo. De esta manera los pesos de la capa de salida se actualizan según lo siguiente:

$$W_{ki}(t+1) = W_{ki}(t) + \mu(d_k - y_k) f'(a) \delta_i$$

El factor  $\mu$  se denomina parámetro de velocidad de aprendizaje, siempre es positivo y suele ser menor que 1. Más adelante se darán algunas precisiones adicionales respecto a él.

Si la función de transferencia es la Identidad, la ecuación para la actualización de pesos nos queda:

$$W_{ki}(t+1) = W_{ki}(t) + \mu(d_k - y_k) \delta_i$$

Si por el contrario utilizamos como función de transferencia la función sigmoide logística, la actualización es la siguiente:

$$W_{ki}(t+1) = W_{ki}(t) + \mu(d_k - y_k) y_k(1 - y_k) \delta_i$$

#### **2.8.4. Actualización de los pesos de capas ocultas**

Se desea repetir para la capa oculta el mismo tipo de cálculo que se ha realizado para la capa de salida. Debe notarse que la expresión del error que se utiliza para calcular la actualización de los pesos de la capa de salida también puede verse como una función de todos los pesos de las neuronas de la capa oculta.

Veamos que esto es cierto:

$$E = \frac{1}{2} \sum_{k=1}^M (d_k - y_k)^2$$

$$y_k = f(a^s)$$

$$a^s = \sum_{j=0}^L w_{kj} \acute{o}_j$$

$$\acute{o}_j = g(a)$$

$$a = \sum_{i=0}^N v_{ji} x_i$$

Así podemos reescribir  $E$  como:

$$E = \frac{1}{2} \sum_{k=1}^M (d_k - f(\sum_{j=0}^L w_{kj} g(\sum_{i=0}^N v_{ji} x_i)))^2$$

El gradiente de  $E$  respecto a los pesos de la capa oculta se calcula de la siguiente forma:

$$\nabla E = \left( \frac{\partial E}{\partial v_{10}}, \frac{\partial E}{\partial v_{11}}, \dots, \frac{\partial E}{\partial v_{1N}}, \frac{\partial E}{\partial v_{20}}, \frac{\partial E}{\partial v_{21}}, \dots, \frac{\partial E}{\partial v_{2N}}, \dots, \frac{\partial E}{\partial v_{L,0}}, \frac{\partial E}{\partial v_{L,1}}, \dots, \frac{\partial E}{\partial v_{L,N}} \right)$$

Y la componente genérica correspondiente a la  $i$ -ésima coordenada del vector de pesos de la  $j$ -ésima neurona de salida se calcula así:

$$\frac{\partial E}{\partial v_{ji}} = \frac{1}{2} \sum_{k=1}^M 2(d_k - y_k)(-1) \frac{\partial y_k}{\partial a^s} \cdot \frac{\partial a^s}{\partial \acute{o}_j} \cdot \frac{\partial \acute{o}_j}{\partial a} \cdot \frac{\partial a}{\partial v_{ji}}$$

Cada uno de estos factores puede calcularse explícitamente a partir de las ecuaciones anteriores. El resultado es el que sigue:

$$\frac{\partial E}{\partial v_{ji}} = - \sum_{k=1}^M (d_k - y_k) f'(a^s) w_{kj} g'(a) \cdot x_i$$

Se actualizan los pesos de la capa oculta proporcionalmente al valor negativo de la ecuación anterior introduciendo nuevamente el factor  $\mu$  quien una vez más representa la velocidad de aprendizaje.

$$\Delta v_{ji} = \mu g'(a) x_i \sum_{k=1}^M (d_k - y_k) f'(a^s) w_{kj}$$

Observemos que todas las actualizaciones de pesos de la capa oculta dependen de todos los términos de error  $(d_k - y_k)$  de la capa de salida. De aquí surge la noción de que los errores conocidos de la capa de salida se propagan hacia atrás, hacia la capa oculta, para determinar los cambios de pesos adecuados en esa capa. Los términos de error de las unidades ocultas deben calcularse antes que hayan sido actualizados los pesos de conexión  $w_{kj}$  con las unidades de la capa de salida.

Por lo tanto, los pesos de la capa oculta se actualizan de la manera siguiente:

$$v_{ji}(t+1) = v_{ji}(t) + \mu g'(a) x_i \sum_{k=1}^M (d_k - y_k) f'(a^s) w_{kj}$$

### 2.8.5. Variantes del algoritmo

Desde que en 1986 se presentara la regla delta generalizada, se han desarrollado diferentes variantes del algoritmo original. Estas variantes tienen por objeto acelerar el proceso de aprendizaje. A continuación, se mencionan brevemente los algoritmos más relevantes, citados en [124]

- La regla *delta-bar-delta* se basa en que cada peso tiene una tasa de aprendizaje propia, y ésta se puede ir modificando a lo largo del entrenamiento.
- Por su parte, el algoritmo QUICKPROP modifica los pesos en función del valor del gradiente actual y del gradiente pasado.
- El algoritmo de gradiente conjugado se basa en el cálculo de la segunda derivada del error con respecto a cada peso, y en obtener el cambio a realizar a partir de este valor y el de la derivada primera.
- Por último, el algoritmo RPROP –*Resilient propagation*– es un método de aprendizaje adaptativo parecido a la regla *delta-bar-delta*, donde los pesos se modifican en función del signo del gradiente y no en función de su magnitud.

## 2.9. Tipos de métodos de entrenamiento para redes neuronales

Hasta el momento hemos mencionado distintos dos tipos de elementos de procesamiento (el Perceptrón y el Adaline) y sus respectivas combinaciones para conformar estructuras que permitan resolver problemas de cierta complejidad que va más allá de la separación lineal de patrones de entrada complejas. También presentamos la noción de peso de conexión, los cuales definen lo que representa una determinada entrada a un PE. Pero hasta ahora no explicamos la manera en la que se configuran los valores de estos pesos para obtener un nodo o una red que resuelva un problema determinado. Esto es llevado a cabo a través de un proceso denominado *entrenamiento* o *aprendizaje* de la red neuronal. Este aprendizaje se plasma en la modificación de los pesos de las conexiones entre los distintos elementos que forman la red.

Existen muchos tipos de aprendizaje dependiendo del modo en que es realizado el ajuste de los pesos. En un principio, los pesos pueden ser considerados parámetros libres, aunque es posible, si se conoce información acerca de la naturaleza del problema que se va a tratar, fijar restricciones a los valores iniciales, o a los valores que puedan tomar a lo largo del proceso de aprendizaje. A continuación brindaremos una definición del mismo:

**Un conjunto de reglas bien definidas que describen el método de adaptación o modificación de los pesos de acuerdo con el entorno en el que se encuentra sumergida la red, recibe el nombre de regla de aprendizaje, y su transcripción en forma de procedimiento se denomina algoritmo de aprendizaje.**

Existe una relación muy fuerte entre la arquitectura de una red neuronal artificial y el o los algoritmos de aprendizaje que puede usar, de tal modo que diferentes arquitecturas de redes neuronales requieren diferentes algoritmos de aprendizaje.

La teoría del aprendizaje mediante ejemplos conlleva tres aspectos muy importantes a tener en cuenta: determinar la capacidad de aprendizaje, la complejidad de los ejemplos utilizados y la complejidad computacional del proceso en sí:

- **La capacidad** es un concepto relacionado con la cantidad de patrones que pueden ser almacenados y qué funciones y contornos de decisión puede sintetizar una red neuronal artificial.
- **La complejidad de los ejemplos** determina el número de los patrones de aprendizaje necesarios para entrenar la red de tal manera que quede garantizado un determinado grado de generalización. Un escaso número de patrones de aprendizaje comparado con el número de pesos (parámetros libres) puede dar lugar a problemas de sobreentrenamiento (“*overfitting*”).
- **La complejidad computacional** se refiere al tiempo requerido para que el algoritmo de aprendizaje se aproxime a la solución usando los patrones de entrenamiento. Lo más corriente es que los algoritmos de aprendizaje sean computacionalmente muy complejos.

Existen dos grandes tipos de métodos de aprendizaje: el aprendizaje supervisado y el no supervisado.

El primero es el más sencillo y consiste en la presentación de patrones de entrada junto a los patrones de salida deseados, por eso se lo llama supervisado. Dentro de este tipo de entrenamiento se pueden encontrar tres subtipos diferentes: *por Corrección de Error*, *por Refuerzo* y *Estocástico*.

Si no se le presentan a la red los patrones de salida deseados, nos encontramos ante un aprendizaje no supervisado, ya que no se le indica a la red cuáles son los resultados que debe dar, sino que se le deja seguir alguna regla de autoorganización. A continuación explicaremos más en detalle de que se trata cada uno de los tipos de método de entrenamiento nombrados.

### **2.9.1. Aprendizaje Supervisado por corrección de error**

Es considerado “el” método de aprendizaje supervisado. En mucha bibliografía sólo se usa el término aprendizaje supervisado mencionando así al aprendizaje por corrección de error.

Los ejemplos que hemos presentado, Perceptrón y Adaline, utilizan este tipo de aprendizaje. Se presentan a la red una serie de patrones de entrada junto a los patrones de salida deseados correspondientes. El aprendizaje consiste en la modificación de los pesos de las conexiones en el sentido de reducir la diferencia entre la salida obtenida y la que se desea obtener.

En los años 80 se popularizaron la arquitectura *Backpropagation* y la *regla delta* generalizada, los cuales ya han sido explicados (ver Pág. 20). La mayoría de las reglas empleadas actualmente en el aprendizaje supervisado por corrección de error son modificaciones de la regla delta generalizada.



### **2.9.2. Aprendizaje Supervisado por Refuerzo**

Este método resulta más lento que el anterior, pero tiene la ventaja de no necesitar conocer por completo el comportamiento deseado, es decir, la salida deseada exacta para cada entrada puede ser desconocida. Lo que se conoce es cómo debería ser el comportamiento de manera general ante diferentes entradas.

El supervisor debe poner en funcionamiento a la red y evaluar el éxito o fracaso de la salida. Se produce entonces una señal llamada de refuerzo que mide el buen funcionamiento del sistema. Esta señal se caracteriza por el hecho de que es menos informativa que en el caso de aprendizaje supervisado por corrección de error. Simplemente puede ser una indicación de que el dispositivo funcionó correctamente o no.

Los pesos se ajustan en base a la señal de refuerzo basándose en un mecanismo de probabilidades. Si una acción tomada por el sistema de aprendizaje es seguida por un estado satisfactorio, la tendencia del sistema a producir esa acción particular es reforzada. De no ser así, la tendencia del sistema a producir dicha acción es disminuida.

Existe bibliografía que sitúa al aprendizaje por refuerzo a medio camino entre el supervisado y el no supervisado. Más específicamente, bibliografía especializada en neuroevolución habla simplemente de aprendizaje supervisado, en alusión al aprendizaje supervisado por corrección de error, y de aprendizaje por refuerzo, para referirse al aprendizaje supervisado por refuerzo.

### **2.9.3. Aprendizaje Supervisado Estocástico**

Este tipo de aprendizaje consiste básicamente en realizar cambios aleatorios en los valores de los pesos y evaluar su efecto a partir del objetivo deseado y de distribuciones de probabilidades.

Se basa en el paradigma *Solidificación Simulada –Simulated Annealing–* que se verá en el siguiente capítulo.

El estado de mínima energía corresponde a los valores de los pesos con los que la red se ajusta al objetivo deseado, es decir que la energía representa una medida del error producido. El proceso es el siguiente:

- Se realiza un cambio aleatorio en los pesos.
- Se determina la nueva energía de la red utilizando alguna consideración sobre la distancia al objetivo deseado.
- Si la energía decrece se acepta el cambio, si no decrece se aceptaría el cambio en función de una determinada y preestablecida distribución de probabilidades.

Dos modelos de redes, descritos en [29], que han utilizado este tipo de aprendizaje son: Boltzmann Machine y Cauchy Machine.

### **2.9.4. Aprendizaje no supervisado**

Estos métodos de aprendizaje son capaces de extraer suficiente información de los datos presentados a la red neuronal como para permitir posteriormente su recuperación o clasificación de acuerdo a características similares. Extraen las propiedades estadísticas de los ejemplos de aprendizaje y los agrupa en clases de patrones similares pero sin

conocer de antemano que salida corresponderá a cada grupo o categoría de patrones de entrada.

Para el aprendizaje no supervisado no es necesario conocer la salida deseada para cada patrón de entrada, ya que el algoritmo y la regla utilizada para ajustar los pesos de conexiones permiten generar una salida consistente. Es decir, un mismo patrón de entrada o de características similares a éste, será clasificado siempre con la misma salida de la red.

En el proceso de aprendizaje existen diferentes interpretaciones que se le puede dar a la salida generada por una red utilizando este tipo de aprendizaje y que depende de la estructura y el algoritmo. Por ejemplo:

- *Grado de Familiaridad o Similitud:* Entre la información actual y la información pasada.
- *Clusterización:* Establecimiento de categorías o clases. La red se encarga de encontrar las características o propiedades propias de cada clase.
- *Codificación:* La salida se interpreta como una codificación de la entrada.
- *Mapeo de Características:* Los elementos de procesamiento de la capa de salida se disponen geoméricamente representando un mapa topográfico de las características de los datos de entrada. Es decir a entradas parecidas le corresponde la activación de neuronas próximas en la capa de salida.

Un ejemplo de red que utiliza este tipo de aprendizaje, cuya interpretación de la salida es un mapeo de características lo constituye la red SOM –Self organizing map– de Kohonen (Ver [29] Capítulo 7 y [71]).

### **2.9.5. Aprendizaje Competitivo**

En el aprendizaje competitivo, un gran número de unidades ocultas pugnan entre sí, con lo que finalmente se utiliza una sola unidad oculta para representar un patrón de entrada determinado o un conjunto de patrones similares. La unidad oculta seleccionada es aquella cuyos pesos de conexión se asemejan más al patrón de entrada. El aprendizaje consiste en reforzar las conexiones de la unidad ganadora y debilitar las otras, para que los pesos de la unidad ganadora se asemejen cada vez más al patrón de entrada.

## **2.10. Consideraciones prácticas**

### **2.10.1. Conjunto de entrenamiento**

Cuando se introdujo la regla GDR prometimos dar precisiones sobre que significa *adecuadamente* y *suficiente* con respecto a la selección de pares de vectores de entrenamiento para la Backpropagation. Lamentablemente no existe un método general para la elección de un “buen conjunto” de entrenamiento. La experiencia y un proceso de “prueba y error” nos llevan finalmente a encontrar este conjunto. Sin embargo se pueden tener en cuenta algunas consideraciones.

Frecuentemente sólo se necesita un subconjunto pequeño de los datos de entrenamiento que se disponen. Por lo general el resto se utiliza para probar la red una vez concluido la etapa de aprendizaje.

En [124] se sugiere crear tres conjuntos con los datos de entrenamiento: uno de aprendizaje, otro de validación y finalmente uno de test. La motivación es encontrar que la red adquiera la mejor capacidad de generalizar los nuevos casos.

Durante la etapa de aprendizaje de la red, los pesos son modificados de forma iterativa de acuerdo con los valores del grupo de entrenamiento. Sin embargo, cuando el número de pesos es excesivo en relación al problema, el modelo se ajusta demasiado a las particularidades irrelevantes presentes en los patrones de entrenamiento perdiendo de vista la verdadera relación entre las entradas y salidas. De esta manera la red termina siendo incapaz de generalizar. Es el problema denominado sobreajuste. Más específicamente, la red aprende “de memoria”. Podemos disminuir el número de pesos en una arquitectura Backpropagation utilizando menos unidades ocultas. También se logra disminuyendo la cantidad de neuronas de entrada y salida, pero éstas son dependientes del problema, y sólo puede conseguirse cambiando la codificación de los parámetros de entrada y de la salida.

Para evitar el problema del sobreajuste, es aconsejable utilizar el segundo grupo de datos diferentes a los de entrenamiento, el grupo de validación, que permite controlar el proceso de aprendizaje. Durante el aprendizaje la red va modificando los pesos en función de los datos de entrenamiento y de forma alternada se la alimenta con los datos de validación.

Con el grupo de validación se puede averiguar cuál es el número de pesos óptimo –y así evitar el problema del sobreajuste–, en función de la arquitectura que ha tenido la mejor ejecución con los datos de validación.

Por último, si se desea medir de una forma completamente objetiva la eficacia final del sistema construido, no deberíamos basarnos en el error que se comete ante los datos de validación, ya que de alguna forma, estos datos han participado en el proceso de entrenamiento. Se debería contar con un tercer grupo de datos independientes, el grupo de test el cual proporcionará una estimación acertada del error de generalización.

En otro sentido, Freeman y Skapura [29] aconsejan no entrenar por completo a la red con vectores de una clase, pasando después a otra clase; la red se olvidará del entrenamiento original, una estrategia de selección aleatoria del orden en que se presentarán los datos de entrada, o el intercalado entre distintas clases pueden ser dos alternativas viables.

### ***2.10.2. Dimensiones de la red***

En este aspecto se considera solamente la cantidad de capas y neuronas ocultas, ya que la cantidad de entradas y salidas está dada por la configuración del problema a resolver.

Utilizar dos capas ocultas a menudo hace que la red aprenda más deprisa, pero habitualmente con una sola alcanza. Determinar el número de unidades que hay que utilizar en la capa oculta puede resultar no muy evidente y en esto también influye la experiencia. Lo ideal es utilizar el menor número posible para ahorrar carga de procesamiento. Si la red no converge para llegar a una solución, puede requerirse el agregado de más nodos ocultos. Si converge, se puede probar con un número inferior de nodos ocultos y determinar un tamaño final basándose en el rendimiento global del sistema.

Es posible eliminar unidades ocultas que resulten superfluas. Si se examinan periódicamente los valores de los pesos de las neuronas de la capa oculta durante el

entrenamiento, a menudo se detectan pesos que cambian muy poco respecto de sus valores iniciales. Estos nodos pueden no estar participando del aprendizaje y quizá baste con un número menor de unidades ocultas.

### **2.10.3. Pesos y parámetros de aprendizaje**

Los pesos iniciales deberían ser pequeños y aleatorios, por ejemplo entre  $-0,5$  y  $+0,5$ . Más problemático suele ser la elección del parámetro de velocidad de aprendizaje,  $\mu$ , ya que tiene un efecto significativo en el rendimiento de la red. Normalmente  $\mu$  debe ser un número pequeño –del orden de  $0,05$  a  $0,25$ – para asegurar que la red llegue a asentarse en una solución. Suele ser posible decrementar el valor de  $\mu$  a medida que progresa el aprendizaje para afinar la convergencia.

Otra forma de aumentar la velocidad de convergencia consiste en utilizar una técnica llamada momento. Cuando se calcula el valor del cambio de peso, se añade una fracción del cambio anterior. Este término adicional tiende a mantener los cambios de peso en la misma dirección. Las ecuaciones de cambio de pesos de la capa de salida pasan entonces a ser:

$$W_{ki}(t+1) = W_{ki}(t) + \mu(d_k - y_k)f'(a)\delta_i + \alpha\Delta_{ki}(t-1)$$

También se utiliza una ecuación similar para la capa oculta. En la ecuación anterior,  $\alpha$  es el factor de momento, y permite filtrar las oscilaciones en la superficie del error provocadas por la tasa de aprendizaje  $\mu$  al acercarse al mínimo, y acelera considerablemente la convergencia de los pesos, ya que si en el momento  $t$  el incremento de un peso era positivo y en  $t+1$  también, entonces el descenso por la superficie de error en  $t+1$  será mayor. Sin embargo, si en  $t$  el incremento era positivo y en  $t+1$  es negativo, el paso que se da en  $t+1$  es más pequeño, lo cual es adecuado, ya que eso significa que se ha pasado por un mínimo y los pasos deben ser menores para poder alcanzarlo.

### **2.10.4. Función de transferencia de las neuronas ocultas y de salida**

El algoritmo de la *regla delta generalizada* exige que la función de transferencia sea derivable para poder obtener el error o valor delta de las neuronas ocultas y de salida. Se disponen de dos formas básicas que cumplen esta condición: la función lineal Identidad y la función sigmoide –logística o tangente hiperbólica–

Debe tenerse en cuenta que para aprovechar la capacidad de las RNA de aprender relaciones complejas o no lineales entre variables, es imprescindible la utilización de funciones no lineales al menos en las neuronas de la capa oculta [124]. Las RNA que no utilizan funciones no lineales, se limitan a solucionar tareas de aprendizaje que implican únicamente funciones lineales o problemas de clasificación que son linealmente separables. Por tanto, en general se utilizará la función sigmoide –logística o tangente hiperbólica– como función de transferencia en las neuronas de la capa oculta.

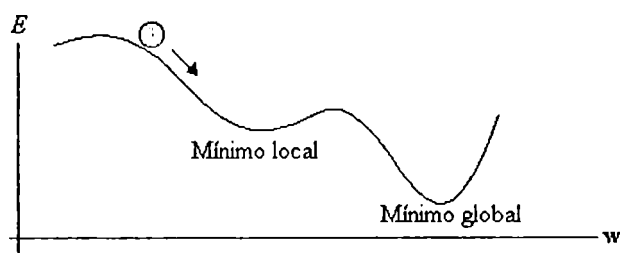
Por su parte, la elección de la función de transferencia en las neuronas de la capa de salida dependerá del tipo de tarea impuesta. En tareas de clasificación, las neuronas normalmente toman la función de transferencia sigmoide. Así, cuando se presenta un patrón que pertenece a una categoría particular, los valores de salida tienden a dar como valor 1 para la neurona de salida que representa la categoría de pertenencia del patrón, y 0 ó -1 para las otras neuronas de salida. En cambio, en tareas de predicción o

aproximación de una función, generalmente las neuronas toman la función de transferencia lineal.

## 2.11. Debilidades de la Backpropagation

A continuación detallaremos algunas debilidades o dificultades concernientes a las redes Backpropagation, muchas de ellas recopiladas en [43]:

- Los resultados dependen fuertemente de los valores iniciales aleatorios de las conexiones. Esto hace que sea conveniente entrenar varias redes con distintos valores iniciales y elegir la que mejor funcione.
- A veces se requiere mucho tiempo para obtener soluciones sencillas. Las redes más grandes necesitan más tiempo de procesamiento, pero también hacen falta más ejemplos de aprendizaje, y eso provoca un aumento mucho mayor del tiempo de aprendizaje.
- Incorporar nuevos ejemplos de aprendizaje, una vez que la red haya sido entrenada produce una “interferencia catastrófica” o empeoramiento en el rendimiento del sistema.
- Inestabilidad temporal: El uso de un coeficiente de aprendizaje elevado produce incrementos grandes en los pesos, llevando al proceso de aprendizaje a realizar oscilaciones. Esto se soluciona usando un coeficiente pequeño. Para no tener un aprendizaje muy lento es conveniente modificar dicho coeficiente adaptativamente –aumentarlo si el error global disminuye, y disminuirlo en caso contrario– como ya comentamos en un apartado anterior.
- El problema de los mínimos locales: El algoritmo de Backpropagation sigue la pendiente más pronunciada hacia abajo en la superficie del error, pero no garantiza alcanzar el mínimo global. Ver la Figura 23 para obtener una noción gráfica del problema.



**Figura 23: El problema de los mínimos locales.** Esta gráfica muestra una sección transversal de una superficie de error hipotética dentro del espacio de pesos. A partir de la posición aleatoria inicial en la superficie del error, el algoritmo nos llevará probablemente al mínimo más cercano, pero no necesariamente al más *hondo* –mínimo global.

Una vez que la red se asienta en un mínimo, sea local o global, cesa el aprendizaje. Si se alcanza un mínimo local, el error de la red puede seguir siendo alto. Afortunadamente, este problema no parece causar grandes dificultades en la práctica. De todas maneras podemos intentar solucionarlo. Una técnica que puede ayudar a no caer en mínimos locales consiste en añadir cierto nivel de ruido a las modificaciones de los pesos de las

conexiones. Otra alternativa es simplemente volver a empezar con un conjunto distinto de pesos originales. Como contra, estas medidas aumentan el tiempo de aprendizaje.

## 2.12. Redes neuronales recurrentes

Hasta aquí hemos mostrado solamente tipos de redes cuyas conexiones se dirigen desde una capa hacia la siguiente sin volver a capas anteriores ni “saltar” capas para llegar a otra. Este tipo de redes se denomina *feedforward* y es indispensable que cumplan dicha condición para poder ser entrenadas mediante el algoritmo de backpropagation como se explicó en secciones anteriores. Ahora nos dedicaremos a mostrar un tipo de RNA denominadas *recurrentes*, las cuales no cumplen con la condición de estructura de las *feedforward*. Veremos que tipo de topología poseen y que funcionalidad les brinda la misma.

Las RNAs recurrentes son básicamente sistemas dinámicos donde los estados varían de acuerdo a un estado no lineal de ecuaciones. Debido a su naturaleza dinámica, han sido aplicadas con éxito a muchos problemas, incluyendo modelado y procesamiento de señales temporales, como control adaptativo, identificación y reconocimiento del habla. Como contra del potencial y la capacidad de las redes recurrentes, el principal problema es la dificultad para entrenarlas más la complejidad y la lenta convergencia de los algoritmos de entrenamiento existentes.

Las conexiones recurrentes le permiten a la red neuronal disponer de una “memoria a corto plazo”, es decir, de cierta manera la red en el momento de su evaluación “recuerda” algo de su estado anterior. Esta característica resulta de mucha utilidad cuando tratamos problemas de control ya que mediante el agregado de conexiones recurrentes la red puede llegar a inferir estos datos para poder controlar la situación. Estas conexiones recurrentes le dan a la red la posibilidad de recordar ciertos aspectos del estado anterior (velocidades, posiciones, movimientos, etc.), los cuales influyen en su salida.

Uno de los principales inconvenientes del algoritmo estándar de backpropagation, aplicado a problemas en los cuales el tiempo es importante, es el hecho de que este algoritmo es estático, y por lo tanto independiente del tiempo. Para poder aplicar redes neuronales a problemas con series temporales aparecen las redes recurrentes que son capaces de capturar este tipo de relaciones entre las entradas. Supongamos que tenemos que construir una red que tiene que generar una señal de control que depende de una entrada externa, que es una serie temporal del tipo  $x(t)$ ,  $x(t-1)$ ,  $x(t-2)$ , etc. Si utilizamos una red multicapa como las que hemos visto hasta el momento tenemos dos posibilidades:

- Crear entradas  $x_1, x_2, \dots, x_n$  que formarán parte de los últimos  $n$  valores del vector de entrada. De este modo estamos usando una ventana de tiempo del vector de entrada que es en sí misma una entrada a la red.
- Crear entradas  $x, x', x''$ , etc. Es decir, además de introducirle a la red como entradas el propio vector de entradas  $x(t)$ , le introducimos también la primera, segunda, etc. derivadas. Obviamente el cálculo de estas derivadas debe hacerse por métodos numéricos, potencialmente sensibles al ruido.

El inconveniente que tenemos con cualquiera de estas dos opciones es que estamos aumentando la dimensionalidad de entrada de la red en un factor  $n$ , y por lo tanto,

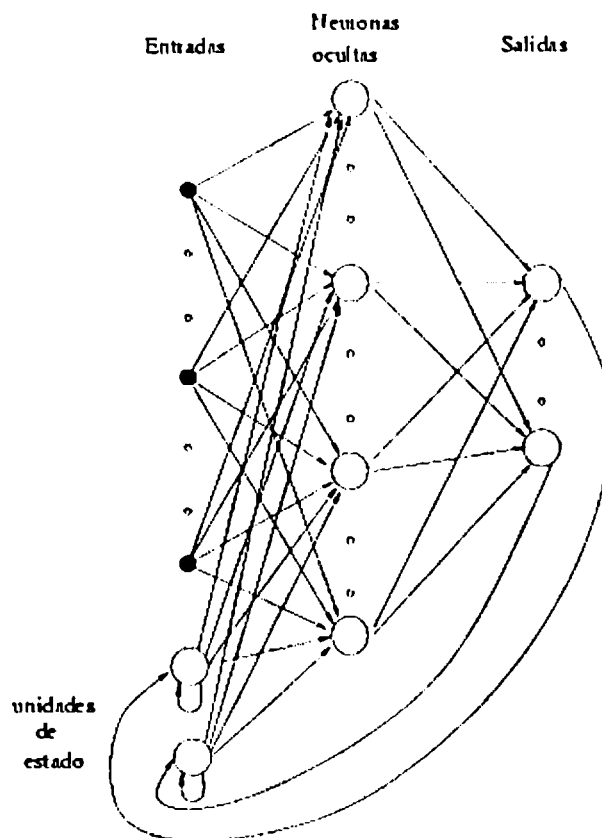
estamos construyendo cada vez redes más grandes que son más lentas y más difíciles de entrenar.

Para solucionar este tipo de problemas tenemos las redes recurrentes en las que la información, como mencionamos anteriormente, no fluye únicamente en un sentido como ocurre en las RNA feedforward usuales sino que existen ciclos en la propia estructura de la red.

Veremos a continuación dos tipos de redes neuronales parcialmente recurrentes como lo son las redes de Jordan y las redes de Elman. Ambas redes poseen conexiones recurrentes en su estructura, y por lo tanto, ante problemas como el citado no hará falta el uso de una ventana de entradas creada explícitamente para que la red aprenda la influencia de lo ocurrido en iteraciones anteriores sino que con estas redes se supone que ya aprenden por sí mismas este tipo de comportamiento.

## 2.13. Redes de Jordan

La red de Jordan fue presentada por Jordan en 1986 [65] y constituye una de las primeras redes recurrentes. En la Figura 24 podemos ver un ejemplo de estructura de una red de Jordan.



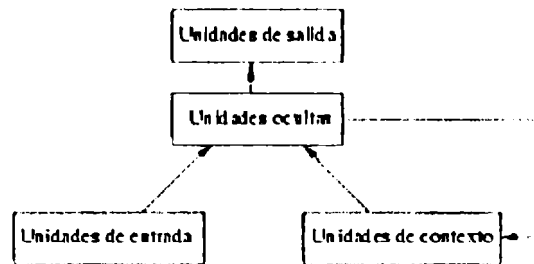
**Figura 24:** Red recurrente de Jordan. Los valores de activación de la salida son realimentados hacia la capa de entrada de la red mediante un conjunto de neuronas que se denominan unidades de estado.

En esta red los valores de activación de las salidas son realimentados hacia la capa de entrada mediante unas unidades extra que se denominan unidades (neuronas) de estado. Existirán tantas unidades de estado como neuronas de salida posea la red. Las

conexiones entre las unidades de salida y las de estado tienen un peso fijo de +1; de este modo el aprendizaje se produce sólo en las conexiones entre las neuronas de entrada y las ocultas así como en las neuronas ocultas y las unidades de salida. Por lo tanto, las reglas de aprendizaje de backpropagation ligeramente modificadas pueden ser usadas para entrenar estas redes recurrentes.

## 2.14. Redes de Elman

En esta red, introducida por Elman en 1990 ([22]), se introduce un conjunto de unidades extra denominadas unidades de contexto, que son entradas adicionales a la red cuyos valores de activación son realimentados desde las neuronas ocultas.



**Figura 25:** Esquema de una red recurrente de Elman. En esta red, los valores de activación de la capa oculta son realimentados hacia la capa de entrada mediante un conjunto de neuronas extra llamadas unidades de contexto de una red de Elman concreta, y en esta figura se puede entender más fácilmente su funcionamiento.

Esta red es muy similar a la red de Jordan excepto en el hecho de que en esta red las neuronas que realimentan son las ocultas en vez de las neuronas de salida; y además estas unidades extra de entrada a la red no tienen conexiones consigo mismas, cosa que sí ocurre en las de Jordan. En la Figura 25 podemos ver un esquema de las conexiones en una red de Elman. En esta arquitectura se utilizan tantas neuronas de contexto como neuronas tengamos en la capa oculta. También podemos tener una red Elman con más de una capa oculta, y en este caso la estructura de la red sería la misma: una neurona de contexto por cada neurona oculta cualquiera que sea la capa en la que se encuentre. Igual que ocurría en la red de Jordan, en la de Elman las neuronas ocultas están conectadas a las unidades de contexto con un peso fijo igual a +1. El aprendizaje en una red de Elman funciona de la siguiente manera:

1. Se asigna valor 0 a las unidades de contexto en el instante inicial  $t = 1$
2. Fijamos el vector de patrones  $x_t$ , realizamos los cálculos del algoritmo sin ningún bucle una vez.
3. Aplicamos el algoritmo de backpropagation
4.  $t = t + 1$ ; ir al paso 2.

Las unidades de contexto en la iteración  $t$  siempre tendrán el valor de activación de las unidades ocultas en la iteración  $t$ . En la Figura 26 podemos ver la estructura de una red de Elman concreta, y en esta figura se puede entender más fácilmente su funcionamiento.



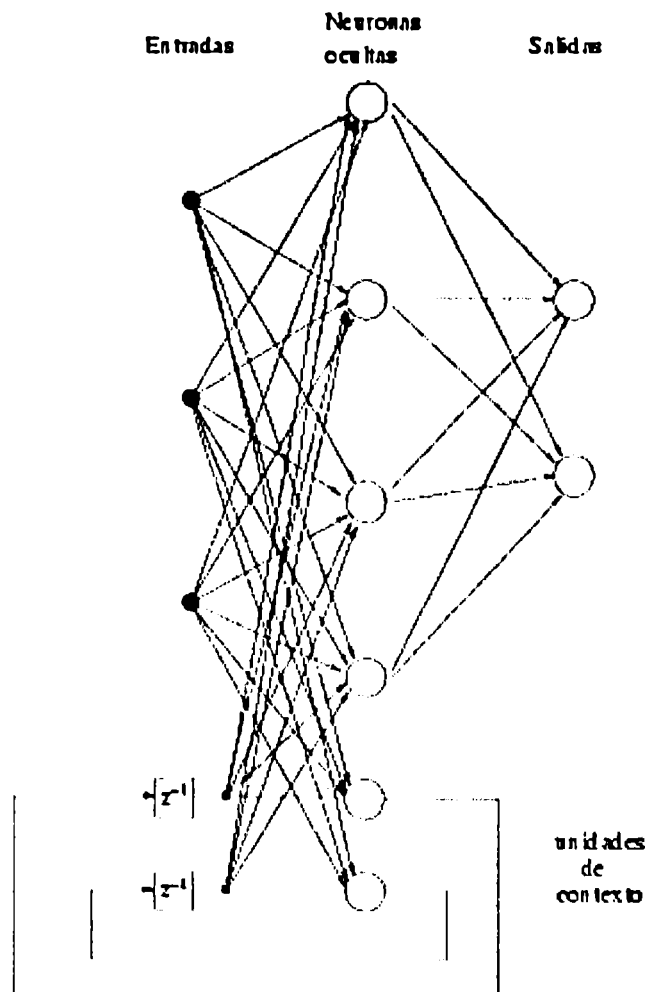


Figura 26: Red recurrente de Elman con 3 unidades de entrada, 4 ocultas, 2 salidas y 2 unidades de contexto.

## 2.15. Estructura de RNAs recurrentes y su entrenamiento

Hasta el momento hemos presentado dos tipos de redes neuronales recurrentes en las cuales las conexiones que les brindan tal característica están acotadas o limitadas de cierta manera. Por ejemplo, las redes de Elman sólo permiten conexiones recurrentes desde las unidades ocultas a las unidades de contexto, de las cuales solamente parten conexiones hacia unidades ocultas. Además el peso de estas conexiones recurrentes es fijo. Estas reglas sobre la estructura permiten enunciar un algoritmo de entrenamiento sencillo para estas redes, como mencionamos anteriormente. Puede encontrarse distintos paradigmas de entrenamiento para redes neuronales recurrentes y una unificación de los mismos en [3]. No hemos encontrado método alguno para entrenar una red recurrente con una política de conexiones libre, es decir, cualquier nodo conectado con cualquier otro nodo. Como hemos visto hasta ahora, las redes de Elman y Jordan imponen ciertas restricciones a sus conexiones lo cual las torna manejables para aplicar un algoritmo de entrenamiento. Intuitivamente puede verse que a medida que la RNA permite una mayor variedad de tipos de conexiones recurrentes, como ser de una salida a una entrada, de una neurona oculta a una entrada, de una oculta a una oculta, de una salida a

una oculta, de una salida a otra salida, etc., el algoritmo de entrenamiento correspondiente aumentará en complejidad o más aún, será imposible de llevar a cabo.

Si para el problema que buscamos resolver con una RNA nos resulta suficiente una red recurrente como las de Elman o Jordan, esto no sería un inconveniente, pero en algunas situaciones puede surgir la necesidad de otro tipo de conexiones o bien existe la posibilidad que una conexión recurrente en el lugar justo sintetice la función de varias conexiones. Obviamente estas consideraciones son muy difíciles de establecer a priori, por lo que queda librado a la experiencia del diseñador la configuración de la red para un problema determinado.

## 2.16. Aplicaciones de RNA

Las redes neuronales artificiales han sido utilizadas con éxito en una gran cantidad de aplicaciones. No es nuestra meta detenernos en cada una, sin embargo, antes de introducirnos en el próximo tema –Neuroevolución– mostraremos algunos logros obtenidos por el paradigma de las RNA:

- *Conversión de texto escrito a lenguaje hablado:* Se han desarrollado redes neuronales que transforman el texto escrito en los códigos elegidos para representar los fonemas correspondientes. Mediante la ayuda de un sintetizador se transforman los códigos en fonemas. La red aprende a hacer distinciones difíciles como pronunciar una *c* suave o fuerte según el contexto. Si bien esto se había conseguido antes, la novedad más importante reside en que no es necesario definir y programar muchas reglas complejas, pues la red extrae automáticamente el conocimiento necesario [120].
- *Aprendizaje de gramáticas:* Se han utilizado también de manera exitosa redes neuronales para aprender el pasado de los verbos ingleses. A través del entrenamiento el sistema fue mejorando y al final fue capaz de generalizar y conjugar verbos desconocidos ([117]).
- *Compresión de imágenes:* Utilizando redes neuronales se han conseguido codificar imágenes con una relación de compresión de hasta 8:1 con alta fidelidad en la reconstrucción sin tener que idear ninguna regla [16].
- *Reconocimiento de escritura manual:* Se han empleado redes neuronales por ejemplo para reconocer kanji –escritura japonesa–, eliminando la gran dificultad que presenta este lenguaje para introducirlo en la computadora. El Neocognitrón, por ejemplo, consigue un reconocimiento muy avanzado de patrones con gran capacidad de abstracción y generalización, que lo hacen capaz de reconocer patrones con distinta orientación y altos niveles de distorsión [34].
- *Problemas de combinatoria:* Las redes neuronales artificiales están ofreciendo ciertas esperanzas en el área de problemas algorítmicamente tan complejos como los NP-completos; por ejemplo el problema del viajante de comercio –Hopfield, J. & Tank, D.–
- *Reconocimiento de patrones en imágenes:* En este campo se han desarrollado numerosas aplicaciones como la clasificación de imágenes de sonar y radar, la detección de células cancerosas, lesiones neurológicas y cardíacas, prospecciones geológicas, etc. Son muy útiles para procesar imágenes de las que no se sabe bien cuales son las características esenciales o diferenciales, ya que las redes no necesitan

disponer de reglas explícitas previas para realizar la clasificación, sino que extraen el conocimiento necesario [34].

- *Visión artificial en robots industriales:* Se han utilizado redes neuronales para inspección de etiquetas, clasificación de componentes, etc. Supera a otros sistemas de visión, además minimiza los requerimientos de operadores y facilita el mantenimiento [36].
- *Predicción de señales:* Se han obtenido mejores resultados a la hora de predecir series “caóticas” usando Backpropagation que mediante métodos lineales y polinomiales ([74], [13] y [10]).
- *Filtro de ruido:* Las redes neuronales artificiales son mejores preservando la estructura profunda y el detalle que los filtros tradicionales descartan cuando eliminan el ruido. El Adaline que vimos con cierto detalle en este capítulo constituyó la primera aplicación profesional de las redes neuronales y se utilizó como filtro para eliminar ruido en las líneas telefónicas [46].
- *Modelado y predicción de indicadores económicos:* Se obtienen mejores resultados que con cualquier otro método conocido. Se ha aplicado por ejemplo a la predicción de tasas de interés, déficits comerciales, precios de stock, etc (ver [10], [14], [60], [96], [66] y [129]).
- *Servocontrol:* Compensación adaptativa de variaciones físicas en servomecanismos complicados como el control de ángulos y posiciones de los brazos de un robot [114].

Hasta aquí hemos dado una noción precisa del paradigma de las Redes Neuronales Artificiales. En el siguiente capítulo presentaremos a los *Algoritmos Evolutivos*. Una vez leído estaremos en condiciones de introducirnos en la *Neuroevolución*, área en la cual el entrenamiento de una red queda sujeto a un proceso evolutivo, permitiendo abstracción acerca de las funciones de activación de los nodos, pesos iniciales de conexiones y hasta en algunos casos de la topología necesaria para resolver un problema. En este último caso el algoritmo evolutivo seleccionará las conexiones necesarias para alcanzar el objetivo.

## 3. Algoritmos Evolutivos

### 3.1. Introducción

El objetivo de este capítulo es presentar brevemente la visión clásica de los Algoritmos Evolutivos, para luego detenernos en sus representantes más conocidos: los Algoritmos Genéticos. Informalmente, la motivación y la forma de trabajo de estos pueden verse en el siguiente ejemplo:

*«Tenemos un problema. ¡Qué mal! Bueno, vamos a intentar solucionarlo. ¿Sabemos cómo hacerlo? No. Entonces, a probar. ¿Podemos generar varias soluciones válidas a ese problema? Sí, claro que podemos, pero van a ser todas muy malas soluciones. Bueno, no importa. Las generamos: 40, 100, las que sean. ¿Alguna es mejor que otra? Todas son malas, pero unas son menos malas que otras. Tomamos las mejores, las otras las eliminamos. ¿Y ahora qué? Bueno, las podemos tomar de dos en dos y mezclarlas a ver si sale algo bueno. ¿Sí? Bueno, a veces funciona. Vamos a hacerlo otra vez. Y otra. También podemos mezclarlas de otra forma. ¡Oye, esto se estanca, ahora son todas iguales! Entonces, vamos a tomar de vez en cuando alguna de las malas, no sólo de las buenas. ¿Y si hacemos pequeños cambios al azar en pequeñas zonas de alguna solución, a ver si alguna da en el clavo? Si... ¡Caramba, esto está mejorando!»*

A lo largo de este capítulo se expondrán detalladamente tanto el funcionamiento como los distintos aspectos a considerar de los Algoritmos Genéticos. Además, por cuestiones de completitud, describiremos algunos métodos de búsqueda local que comparten algunas características con los Algoritmos Evolutivos.

### 3.2. Métodos de búsqueda local

En general, todos los métodos de búsqueda pueden dividirse, a grandes rasgos, en métodos globales y locales. Los métodos globales tratan de encontrar el máximo global de un problema, mientras que los locales se concentran en la vecindad de la solución generada inicialmente, y, por tanto, necesitan alguna técnica adicional, como comienzos múltiples, para acercarse al máximo global.

Los algoritmos de búsqueda local parten de una solución inicial, y, aplicándole operadores de variación, la van alterando; si la solución alterada es mejor que la original, se acepta, si no lo es, se vuelve a la inicial. El procedimiento se repite hasta que no se consigue mejora en la solución.

Los procedimientos de búsqueda local más usados son los basados en el gradiente: en este caso, el operador de variación selecciona una nueva solución teniendo en cuenta la derivada de la función que se quiere optimizar en el punto, tratando de ascender o descender usando el gradiente hasta llegar a un punto de inflexión donde no se puede obtener ninguna mejora adicional.

#### 3.2.1. Hill Climbing

Este método es una técnica iterativa de optimización. Parte con una solución aleatoria y en cada iteración trata de obtener una mejor solución, la cual es seleccionada dentro de un entorno definido por la solución actual. Una vez encontrada una mejor solución, ella

pasa a ser la solución actual y se vuelve a repetir el proceso hasta determinado número de veces. Si en algún momento no se obtiene una mejor solución, el algoritmo termina.

Cabe destacar que para la obtención de la siguiente solución de cada iteración no se explora exhaustivamente el entorno actual, si no que se aplican distintas heurísticas.

Debido a la forma de trabajo de éste método sólo es posible obtener óptimos locales dependientes de la solución inicial y no es posible determinar si el óptimo local encontrado es además un óptimo global.

Para incrementar las posibilidades de éxito, los métodos de *Hill Climbing* usualmente son ejecutados varias veces partiendo de diferentes soluciones iniciales.

Esta estrategia explota en todo momento la mejor solución posible, a expensas de no explorar el espacio de búsqueda.

Existen distintas versiones de este algoritmo, variando la forma en que la nueva solución es elegida.

El método de búsqueda denominado *Simple Iterated Hill Climbing (SIHC)*, explora  $N$  soluciones diferentes dentro del entorno de la solución actual, tomando la de mayor fitness y comparándola con ella. Si resulta mejor, la nueva solución pasa a tomar el lugar de la actual. Si no, el algoritmo ha alcanzado un óptimo local o global, terminando esta iteración y comenzando una nueva partiendo de otra solución aleatoria. El pseudocódigo puede observarse en el Algoritmo 1.

```
t ← 0
inicializar(mejor)
repetir
  local ← falso
  Vc ← Seleccionar al azar la solución actual
  evaluar(Vc)
  repetir
    Vs ← n Nuevas soluciones del vecindario de Vc
    evaluar(Vs)

    Vn ← Solucion con mayor valor de fitness de Vs
    si fitness(Vn) > fitness(Vc) →
      Vc ← Vn
    sino →
      local ← verdadero
  hasta local = verdadero
  t ← t + 1
  si fitness(Vc) > fitness(mejor) →
    Mejor ← Vc
hasta t = MaxIteraciones
```

**Algoritmo 1: Simple Iterated Hill Climbing (SIHC).**

Si en vez de aceptar siempre la mejor solución dentro del entorno, se optara por seleccionarla con una cierta probabilidad, al método de búsqueda se lo denomina *Stochastic Hill Climbing*. En esta versión del algoritmo de búsqueda se seleccionan, una a una, soluciones dentro del entorno, y se las acepta o no con cierta probabilidad dependiente de la diferencia de fitness que hay entre la solución seleccionada y la mejor hasta el momento. Como se ve en el Algoritmo 2, se utiliza una probabilidad  $p$ , definida en la Ecuación 1, para decidir la aceptación o no de la nueva solución. Esta probabilidad

depende de la diferencia de los fitness de las soluciones comparadas. Cuando la nueva solución tiene un mejor fitness que la actual, tiene mayor chance de ser aceptada. Sin embargo, a diferencia del SIHC, existe la posibilidad de elegir una solución con fitness menor al de la actual. Esta característica, que en principio parecería perjudicar al método, permitiría escaparse de óptimos locales, dejando siempre abierta la posibilidad de encontrar el máximo global.

La probabilidad de aceptación de una nueva solución depende además del valor de una constante  $T$ . Si  $T > 1$ , cuanto más grande sea  $T$ , todas las soluciones tienen la misma chance de ser aceptadas. En cambio si  $T < 1$ , a medida que  $T$  tiende a 0, se aceptan con mayor frecuencia solamente soluciones mejores a la actual.

```

t ← 0
Vc ← Seleccionar al azar la solución actual
evaluar(Vc)
repetir
  repetir
    Vn ← Seleccionar solución dentro del vecindario de Vc
    evaluar(Vn)
    Con probabilidad p → Vc ← Vn
  Hasta (condición de terminación)
  t ← t + 1
hasta t = MaxIteraciones
  
```

**Algoritmo 2: Stochastic Hill Climbing.**

$$p = \frac{1}{1 + e^{\frac{fitness(Vc) - fitness(Vn)}{T}}}$$

**Ecuación 1: Probabilidad de selección del Stochastic Hill Climbing.**

### **3.2.2. Simulated Annealing**

Se trata de un método de búsqueda inspirado en el proceso físico de temple que se aplica a los metales y otras sustancias. En esta técnica se eliminan casi todas las desventajas de los métodos de Hill Climbing, puesto que las soluciones ya no dependen de un punto inicial y usualmente están más cerca del óptimo. Esto se logra utilizando una probabilidad  $p$  de aceptación de la misma forma que en el Stochastic Hill Climbing, salvo que el parámetro  $T$ , denominado *temperatura*, varía a lo largo de la búsqueda. El pseudocódigo de este método se puede observar en el Algoritmo 3.

```

t ← 0
Inicializar T
Vc ← Seleccionar al azar la solución actual
evaluar(Vc)
repetir
  repetir
    Vn ← Seleccionar solución dentro del vecindario de Vc
    evaluar(Vn)
    Con probabilidad p → Vc ← Vn
  Hasta (condición de terminación)
  t ← t + 1
  T ← f(T, t)
hasta (t = MaxIteraciones) o (T ≤ MinT)

```

**Algoritmo 3: Simulated Annealing.**

Al igual que antes, cuanto menor sea el valor  $T$  de temperatura, menores serán las posibilidades de aceptación de una nueva solución. Se inicia con un valor alto de la temperatura  $T$ , el cual es disminuido en cada paso de la búsqueda, terminando cuando alcanza un valor pequeño, para el cual no se aceptan nuevas soluciones.

### 3.3. Algoritmos Evolutivos

El término algoritmo evolutivo se usa para designar a un conjunto de técnicas que basan su funcionamiento en metáforas de procesos biológicos. Puede resumirse en la siguiente definición dada por Jones en [64]:

*«El algoritmo mantiene un conjunto de soluciones potenciales a un problema. Dichas soluciones son usadas para producir nuevas soluciones potenciales mediante la aplicación de una serie de operadores. Dichos operadores actúan sobre algunas soluciones que han sido seleccionadas por su bondad con respecto al problema atacado. Este proceso se repite hasta que se alcanza un cierto criterio de terminación.»*

Esta definición puede encontrarse en la literatura expresada en un lenguaje técnico en el que se emplean términos tales como población, genes, cromosomas, etc. Dicha jerga es una reminiscencia de la ya mencionada inspiración biológica de estas técnicas, la cual da lugar a la visión tradicional que sobre su funcionamiento ha imperado durante largo tiempo; esto es, el procesamiento de esquemas y los diferentes corolarios que se extraen de éste. Dicha visión clásica será objeto de un replanteamiento surgido de la existencia demostrada de limitaciones intrínsecas al funcionamiento de dichas técnicas. Fruto de éstas, surge una visión más moderna y matemáticamente sólida de los algoritmos evolutivos.

### 3.3.1. El Esqueleto de un Algoritmo Evolutivo

Un algoritmo evolutivo es un proceso estocástico e iterativo que opera sobre un conjunto  $P$  de individuos (población), cada uno de los cuales contiene uno o más cromosomas. Dichos cromosomas permiten que cada individuo represente una posible solución al problema que se está considerando. Un proceso de codificación/decodificación permite obtener la solución a partir de los cromosomas de cada individuo. Inicialmente, esta población es generada aleatoriamente o con la ayuda de alguna heurística de construcción.

Cada uno de los individuos de la población recibe, a través de una función de aptitud (fitness), una medida de su bondad con respecto al problema que se desea resolver. Este valor es empleado por el algoritmo para guiar la búsqueda. El proceso completo está esquematizado en la Figura 27 y detallado en el Algoritmo 4.

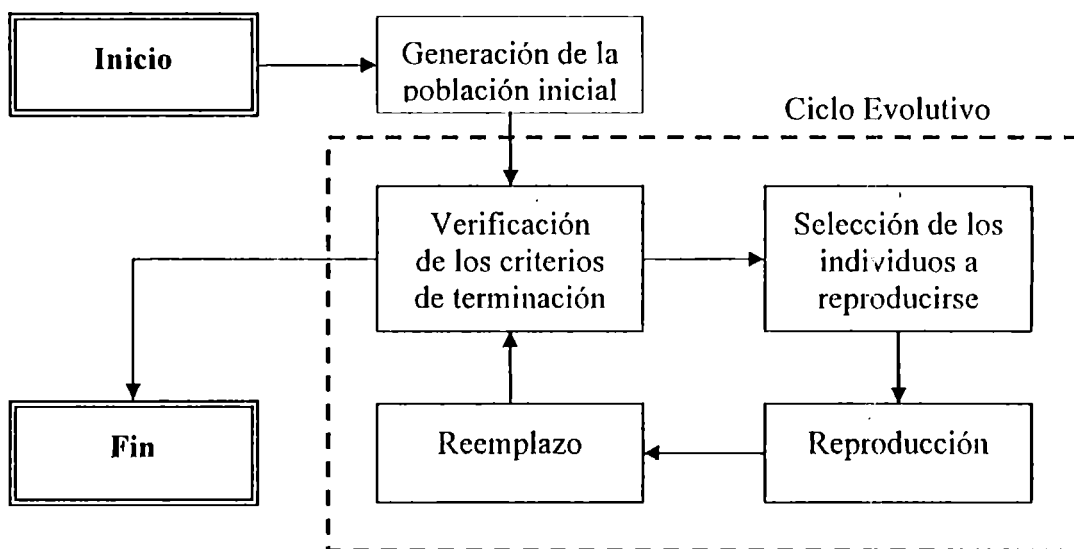


Figura 27: Esquema de un Algoritmo Evolutivo

```

Generar Población(0)
Evaluar Población(0)
t := 0

Mientras no se cumplan las condiciones de terminación
  Padres := Seleccionar(Población(t))
  Descendencia := Reproducción(Padres)
  Población(t+1) := Reemplazo(Población(t), Descendencia)
  Evaluar Población(t+1)
  t := t+1
Fin Mientras
  
```

Algoritmo 4: Pseudo código de un Algoritmo Genético

Como puede apreciarse, el algoritmo está estructurado en tres fases principales que se ejecutan de manera circular: selección, reproducción y reemplazo, las cuales se llevan a cabo de manera repetitiva. Cada una de las iteraciones del algoritmo se denomina ciclo reproductivo básico o generación. Este proceso se realiza hasta que se alcanza un determinado criterio de terminación, comúnmente, un número máximo de iteraciones.



Durante la fase de selección se crea una población temporal *Padres* en la que aquellos individuos más aptos, las mejores soluciones contenidas en la población, estarán representados un mayor número de veces que los poco aptos (principio de selección natural).

A los individuos contenidos en esta población temporal les son aplicados diferentes operadores de cambio, también denominados operadores genéticos, en la fase de reproducción. El objetivo de esta fase es producir individuos con nuevas características, idealmente mejores (principio de adaptación).

Finalmente, durante la fase de reemplazo, se substituyen individuos de la población original por los nuevos individuos creados. Este reemplazo afecta a los peores individuos y tiende a conservar los mejores (supervivencia de los más adaptados).

Obsérvese cómo este algoritmo descrito establece un compromiso entre la explotación de las buenas soluciones (fase de selección), y la exploración de nuevas zonas del espacio de búsqueda (fase de reproducción), apoyado en el hecho de que el mecanismo de reemplazo puede permitir la aceptación de nuevas soluciones que no proporcionen una mejora inmediata sobre las ya existentes.

### **3.3.2. Distintas variantes de los Algoritmos Evolutivos**

Los algoritmos evolutivos, tal como hoy los conocemos, comenzaron a existir a finales de los años 60 y principios de los 70. Durante este período y de manera prácticamente simultánea, investigadores de distintas partes del planeta comenzaron la labor de trasladar los principios de la Evolución por entonces conocidos al campo de la algoritmia, y más concretamente a tareas de búsqueda o resolución de problemas. Este origen independiente provocó el desarrollo paralelo de distintos modelos que han llegado hasta nuestros días. Dichos modelos permiten una estructuración natural de los algoritmos evolutivos dentro de tres grandes familias:

- **Programación Evolutiva (Evolutionary Programming):** Esta familia de algoritmos tiene su origen en el trabajo de Fogel [26], y ponen un especial énfasis en la adaptación de los individuos más que en la evolución del material genético de éstos. Ello implica una visión mucho más abstracta del proceso, en la cual se modifica directamente el comportamiento de los individuos en lugar de trabajar sobre sus genes. Dicho comportamiento se modela mediante estructuras de datos relativamente complejas como son los autómatas finitos. Tradicionalmente, estas técnicas emplean mecanismos de reproducción asexual y técnicas de selección mediante competición directa entre individuos. Existen desarrollos posteriores que generalizan este método para manejar problemas de optimización numérica [25].
- **Estrategias de Evolución (Evolutionsstrategie):** como su nombre originario sugiere, estas técnicas comenzaron a desarrollarse en Alemania. Su objetivo inicial era servir de herramienta para optimización de parámetros en problemas de ingeniería [111] [119]. Debido a este objetivo primordial, estas técnicas se caracterizan por manejar vectores de números reales codificados en punto flotante, aunque existen versiones de las mismas que se aplican a problemas discretos. Al igual que la programación evolutiva con la que se halla estrechamente emparentada, una estrategia de evolución basa su funcionamiento en el empleo de un operador de reproducción asexual o de mutación. Dicho operador está especialmente diseñado para trabajar con números flotantes, e incluye un complejo mecanismo de autoadaptación, en virtud del cual se consigue optimizar la dirección de la búsqueda.

- **Algoritmos Genéticos (Genetic Algorithms):** estas técnicas son probablemente el representante más conocido de los algoritmos evolutivos, y aquellas cuyo uso está más extendido. Fueron concebidas originalmente por John Holland y descritas en el ya clásico «Adaptation in Natural and Artificial Systems» [57]. Este texto ha tenido una gran trascendencia en el posterior desarrollo de estas técnicas ya que los mecanismos en él descritos han sido tomados durante largo tiempo como auténticos dogmas.

La principal característica de los algoritmos genéticos es el uso de un operador de recombinación o cruce como mecanismo principal de búsqueda. Este operador debe recombinar los cromosomas de los padres para construir descendientes que posean características de ambos. La utilidad de este operador se fundamenta en la suposición de que diferentes partes de la solución óptima pueden ser descubiertas independientemente y luego ser combinadas para formar mejores soluciones. Adicionalmente, emplean un operador de mutación cuyo uso se considera importante como responsable del mantenimiento de la diversidad en la población, aunque secundario en relación con el operador de cruce.

Estas tres familias no han permanecido aisladas y han interactuado frecuentemente, motivo por el cual las fronteras entre ellas son difusas en la actualidad. Fruto de este contacto, han aparecido nuevas variedades de algoritmos evolutivos. Resulta especialmente interesante resaltar las dos siguientes:

- **Programas de Evolución (Evolution Programs):** estas técnicas están avaladas por el trabajo de, entre otros, Michalewicz [89], englobándose bajo esa denominación a las heurísticas que, siguiendo los mismos principios de funcionamiento que los algoritmos genéticos, hacen evolucionar estructuras complejas. En la actualidad, es común emplear el término "algoritmo genético" como sinónimo de "programa de evolución", reservándose la expresión "algoritmo genético tradicional" para los algoritmos descritos en el punto anterior.

- **Programación Genética (Genetic Programming):** popularizada por Koza [72], la programación genética es otra variante de los algoritmos genéticos en la que se hace evolucionar estructuras (típicamente árboles) que representan programas de computadora. El objetivo final es el diseño automático de un programa que resuelva una tarea determinada, expresada a partir de una serie de casos de ejemplo.

De entre todos estos modelos descritos, merece la pena estudiar con algo más de detalle a los algoritmos genéticos por los motivos ya comentados: son probablemente las técnicas más extendidas y, consecuentemente, las más proclives a ser consideradas herramientas universales.

### 3.4. Algoritmos Genéticos

Los Algoritmos Genéticos son técnicas que caen dentro del esquema general mostrado en la Figura 27. Tradicionalmente, el espacio de cromosomas está constituido por todas las cadenas de una determinada longitud construidas a partir de los símbolos de un alfabeto.

La búsqueda es guiada a través de una función objetivo, la cual es específica del problema. Dicha función sirve para determinar la bondad de las soluciones que se van obteniendo. A mayor valor de bondad o *fitness*, mejor es la solución encontrada.

En las siguientes subsecciones detallaremos los distintos componentes que dan forma a un Algoritmo Genético, comentando alternativas para cada uno de ellos.

### 3.4.1. Codificación

Toda posible solución al problema que interesa resolver con un Algoritmo Genético debe poder ser representada mediante un conjunto de parámetros que se evolucionan.

En términos biológicos, al conjunto de parámetros se lo denomina *cromosoma*. Cada uno de los parámetros, que representan caracteres o rasgos por los que están formados los organismos, es un *gen*. El lugar que ocupa cada gen dentro del cromosoma se denomina *loci*. Cada carácter o gen puede manifestarse de forma diferente, es decir, puede tomar distintos valores que son denominados *alelos*. El *genotipo* contiene toda la información requerida para construir un organismo, que puede ser uno o más cromosomas en el caso de los organismos biológicos. Al organismo obtenido a partir del genotipo se lo denomina *fenotipo*. Los mismos términos suelen utilizarse en el campo de los Algoritmos Genéticos. Es necesario hacer notar que debe existir una correspondencia unívoca entre genotipo y fenotipo. Para más detalles sobre genética desde el punto de vista biológico, Corbalán en [15] dedicó todo un capítulo al tema.

Una de las tareas más importantes que tendrá el diseñador del algoritmo genético será la elección de una adecuada codificación genética que represente a las potenciales soluciones. Algunas pueden resultar más naturales para la representación, otras, más eficientes o más compactas. A continuación, se describen algunas codificaciones comúnmente utilizadas.

#### 3.4.1.1. Codificación mediante cadenas de bits

La codificación tradicionalmente utilizada para representar las posibles soluciones es la binaria. Esto se debe a que existe una demostración teórica, que afirma que el alfabeto binario es el que ofrece la mayor cantidad de esquemas por bit de información [40]. No obstante esto, a veces resulta complicado encontrar un mapeo entre el espacio de las posibles soluciones y las cadenas de bits.

En el caso de querer representar números enteros o reales, la tarea es relativamente simple de llevar a cabo. Dado un sistema binario de  $n$  bits y un intervalo  $[a, b]$ , es posible transformar cualquier cadena de bits en un número real perteneciente a dicho intervalo aplicando la Ecuación 2. En el Ejemplo 1 se puede observar una conversión de una cadena de bits a su correspondiente valor numérico dentro del intervalo  $[-5, 5]$ .

$$\text{Sea una cadena de bits } c = b_{n-1}b_{n-2}\dots b_1b_0$$
$$x = f(c) = a + \frac{(b-a)}{2^n} \sum_{i=0}^{n-1} b_i \cdot 2^i$$

Ecuación 2: Cambio de rango del sistema binario de  $n$  bits al intervalo de números reales  $[a, b]$ .

1	0	1	0	0	1	1	0	1	0	0	1	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$x = \sum_{i=0}^{n-1} b_i \cdot 2^i = 1 \cdot 2^0 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^7 + 1 \cdot 2^9 + 1 \cdot 2^{10} + 1 \cdot 2^{13} + 1 \cdot 2^{15} = 42653$$

$$x' = a + x \cdot \frac{(b-a)}{2^n} = -5 + 42653 \cdot \frac{5 - (-5)}{2^{16}} \cong 1.508$$

**Ejemplo 1: Cálculo del valor real que representa una cadena de 16 bits dentro del intervalo [-5; 5]**

El tamaño del intervalo y la precisión con la que se desea trabajar determinan la cantidad mínima de bits necesarios. Es evidente que a mayor rango del intervalo y mejor precisión, la cantidad de bits aumenta ya que se necesitan cada vez más cadenas para representar una mayor cantidad de números reales.

En la Ecuación 3 se deriva el cálculo del tamaño del cromosoma para un intervalo dado con una precisión definida. La diferencia entre dos cadenas consecutivas será la precisión con la que se quiere trabajar. A partir de ella, se obtiene cuantos bits serán necesarios. A modo de ejemplo, se muestran en la Tabla 3 diferentes configuraciones intervalo-precisión acompañadas con la cantidad de bits necesarios para alcanzar dicha representación.

$$f(x) = a + x \frac{(b-a)}{2^n}$$

$$p = f(x+1) - f(x)$$

$$p = a + (x+1) \frac{(b-a)}{2^n} - \left[ a + x \frac{(b-a)}{2^n} \right] = (x+1-x) \frac{(b-a)}{2^n} = \frac{(b-a)}{2^n}$$

$$p = \frac{(b-a)}{2^n} \Rightarrow 2^n = \frac{(b-a)}{p} \Rightarrow \log_2 2^n = \log_2 \frac{(b-a)}{p}$$

$$\therefore n = \log_2 \left( \frac{b-a}{p} \right)$$

**Ecuación 3: Cantidad de bits necesaria para representar valores reales en el intervalo [a, b] con precisión p.**

Intervalo	Precisión	Bits
[-5; 5]	0.5	5 bits
[-5; 5]	0.01	10 bits
[-100; 100]	0.01	15 bits
[-500; 500]	0.001	20 bits
[-100000; 100000]	0.0001	31 bits
$[-3.4 \times 10^{38}, 3.4 \times 10^{38}]$	$1.5 \times 10^{-45}$	278 bits

**Tabla 3: Relación entre intervalo y precisión con la cantidad de bits necesarios para representar valores reales.**

Cabe señalar que existe un problema de alta dimensionalidad al emplear esta representación. Si se tienen demasiadas variables reales, y se pretende un amplio rango y una buena precisión para cada una de ellas, entonces las cadenas binarias necesarias serán extremadamente largas y el Algoritmo Genético deberá explorar un espacio mayor en busca de buenas soluciones. Esto puede hacer lenta la convergencia del algoritmo o incluso impedirla completamente.

Recientes cuestionamientos y la evidencia empírica podrían inclinar la tendencia hacia otras representaciones, por lo menos en algunos casos concretos. El uso de un alfabeto binario puede ser desventajoso en problemas de alta dimensionalidad si se pretende trabajar con una buena precisión, lo que ocasiona cromosomas extremadamente largos y difícilmente llegue a producir resultados aceptables, a menos que se usen procedimientos y operadores especiales diseñados para el problema en cuestión [89].

### 3.4.1.2. Codificación basada en números reales

Existen pruebas empíricas que muestran que para una cantidad significativa de aplicaciones el uso directo de números reales en un cromosoma funciona mejor que la representación binaria tradicional [17], [131].

Si se observa la Tabla 3, el último ejemplo corresponde al rango y resolución mínima que posee el tipo *Single* empleado habitualmente para representar números reales de simple precisión. Dicho tipo utiliza sólo 32 bits contra los 278 necesarios para la codificación anterior. Esto se logra separando en mantisa y exponente el valor almacenado en la cadena de bits. Cabe mencionar que la ganancia en tamaño se obtiene a expensas de la resolución, ya que esta no es constante a lo largo de todo el intervalo. La representación es más precisa para los números más próximos al cero y menos para los más cercanos a los extremos del intervalo. Además, existen muchos números que no van a poder representarse exactamente, sino de forma aproximada. Esto puede apreciarse en la Figura 28.

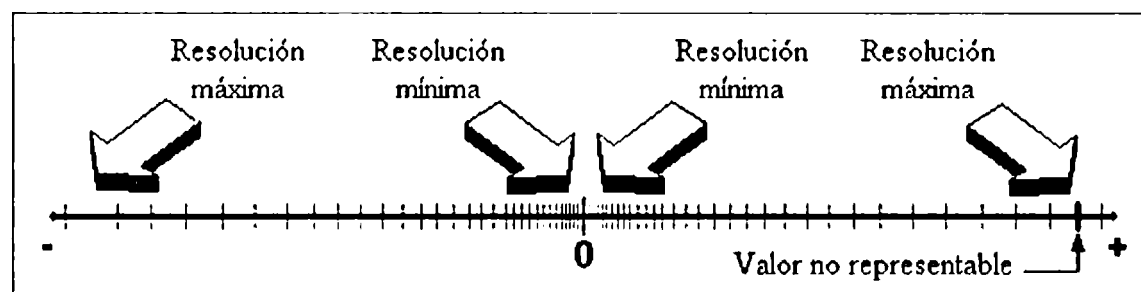


Figura 28: Esquema de la disposición de las representaciones de los números reales codificados con el tipo *Single*.

Utilizar codificación real para una cadena cromosómica ha sido común en otros tipos de Algoritmos Evolutivos [111] [119]. En esta representación, un cromosoma es un arreglo de números reales expresados en punto flotante y las soluciones deberán poder ser representadas mediante un vector de números reales.

La aplicación de operadores genéticos que trabajan sobre cadenas de bits –ver sección 3.4.5– no es conveniente para este tipo de codificación. Como los números reales están codificados en mantisa y exponente, un pequeño cambio en alguno de los bits que corresponden al exponente producirá grandes saltos en el espacio de búsqueda, mientras que perturbaciones en la mantisa pueden no cambiar de manera significativa el valor numérico codificado.

Debido a esto, se han diseñado varios operadores especiales para permitir que una codificación de este tipo siga gozando de las mismas propiedades y características estudiadas para el caso binario. Ejemplos de estos operadores serán descriptos en la sección 3.4.5.

Se han utilizado también codificaciones alternativas para las representaciones de números reales en los cromosomas. Por ejemplo, el uso de enteros para representar cada dígito decimal del valor real se ha aplicado exitosamente a varios problemas de optimización. La precisión está limitada por la longitud de la cadena, y puede incrementarse o decrementarse según se desee. Los operadores de cruce tradicionales pueden usarse directamente en esta representación, al igual que la mutación. Esta representación pretende ser un compromiso entre Algoritmos Genéticos con codificación real y una representación binaria de números reales, manteniendo lo mejor de ambos esquemas al incrementar la cardinalidad del alfabeto utilizado, pero manteniendo el uso de los operadores genéticos tradicionales casi sin cambios.

#### **3.4.1.3. Cromosomas de longitud variable**

Una posibilidad interesante a considerar es que los cromosomas no sean todos de la misma longitud, sino que ésta varíe y evolucione junto con los genes. Obviamente, esto implica una modificación en los operadores genéticos, ya que deben poder trabajar con cromosomas de distinta longitud al momento de combinarlos. Además, serán necesarios operadores que inserten material genético en los cromosomas o lo remuevan.

La longitud variable sirve para explorar un espacio mayor de soluciones, ya que cuando se utilizan cromosomas de longitud fija, existe la posibilidad de que sean excesivamente grandes o bien de tamaño insuficiente. Por ejemplo, en el diseño de redes neuronales, si bien la cantidad de neuronas de entrada y salida suelen estar fijas por la naturaleza del problema que resuelven, no ocurre lo mismo con la cantidad de neuronas ocultas. El conocimiento exacto de cuántas neuronas ocultas se necesitan generalmente está ausente, por lo que no es conveniente fijarlo de antemano. Un cromosoma de longitud variable permitirá encontrar el número adecuado de neuronas ocultas.

#### **3.4.1.4. Otras representaciones**

Muchas veces resulta natural al problema otro tipo de representaciones más apropiadas. Por ejemplo, en la Programación Genética [73] es conveniente representar los programas evolucionados como árboles. La utilización de cadenas de bits implicaría una complicación innecesaria.

Lo mismo ocurre cuando se desea codificar permutaciones. En este caso, los elementos representados en el cromosoma deben cumplir con la condición de que no se repitan, independientemente del orden en el que se presenten. Esto plantea una seria restricción para los operadores genéticos, que deben garantizar que los nuevos cromosomas que produzcan cumplan con las condiciones impuestas.

### **3.4.2. Inicialización**

Los Algoritmos Genéticos trabajan sobre una población de potenciales soluciones de un problema. Generalmente, la población inicial con la que comienza el proceso evolutivo está integrada por cromosomas generados al azar, donde cada uno de ellos está formado por genes seleccionados con probabilidad uniforme.

Una alternativa que ha sido empleada exitosamente es la de aplicar alguna técnica heurística o de optimización local para obtener esos cromosomas, en lugar de dejar que la suerte los determine. En los trabajos que existen sobre este aspecto, se constata que esta inicialización no aleatoria de la población inicial puede acelerar la convergencia del Algoritmo Genético.

Otra cuestión que puede plantearse es la relacionada con el tamaño idóneo de la población. Resulta intuitivo que las poblaciones pequeñas corren el riesgo de no cubrir adecuadamente el espacio de búsqueda, mientras que trabajar con poblaciones de gran tamaño puede acarrear problemas relacionados con un excesivo costo computacional.

El tamaño óptimo de la población es dependiente del problema, y aunque existen desarrollos teóricos para tratar de determinarlo [41], sigue siendo un parámetro crítico para el buen funcionamiento del Algoritmo Genético. Otra alternativa, planteada en [45] consiste en emplear un Meta Algoritmo Genético para evolucionar los parámetros de otro Algoritmo Genético, entre ellos, el tamaño de la población.

Normalmente se opta por utilizar una población de tamaño fijo. Sin embargo, existen esquemas que emplean poblaciones de tamaño variable [2]. En estos esquemas, la cantidad de cromosomas que componen la población aumenta o disminuye a lo largo del ciclo evolutivo. En otras palabras, se busca el tamaño óptimo de la población mediante el mismo proceso evolutivo.

### **3.4.3. Evaluación**

La selección natural opera bajo el principio de eliminar determinística o probabilísticamente a los individuos menos eficaces. Así pues, la evaluación es el elemento crítico de todo Algoritmo Evolutivo que pretenda solucionar un problema de optimización complejo; para cada solución se debe poder calcular una aptitud o fitness asociado la misma. Idealmente, cada individuo contribuye al proceso de reproducción en proporción a su correspondiente fitness. De esta forma, individuos bien adaptados, contribuyen con múltiples copias e individuos mal adaptados contribuyen con pocas o incluso ninguna copia.

El algoritmo genético únicamente maximiza, pero la minimización puede realizarse fácilmente utilizando la recíproca de la función objetivo. Una característica que debe tener esta función es la de ser capaz de "castigar" a las malas soluciones, y de "premiar" a las buenas, de forma que sean estas últimas las que se propaguen con mayor rapidez.

En muchos casos, el desarrollo de una función de evaluación involucra hacer una simulación, mientras que en otros, la función puede estar basada en el rendimiento y representar sólo una evaluación parcial del problema. Adicionalmente debe ser rápida, ya que hay que aplicarla para cada individuo de cada población en las sucesivas generaciones, por lo cual, gran parte del tiempo de ejecución de un algoritmo genético se emplea en la función de evaluación.

### 3.4.4. Selección

Es el proceso mediante el cual un individuo o cromosoma es copiado proporcionalmente a su evaluación o aptitud, formando un conjunto intermedio de individuos. Tal conjunto intermedio se convierte en una población temporal a la cual se le aplicarán los operadores genéticos. Copiar individuos de acuerdo con su aptitud significa que los de más alta evaluación poseen mayor probabilidad de tener una o más copias en la siguiente generación. Este operador está inspirado en la teoría de la selección natural darwiniana. En la Naturaleza, la aptitud de un individuo es medida por su capacidad de sobrevivir en un cierto medio ambiente.

El criterio concreto de muestreo depende del problema y del buen juicio del programador. Los más usados en la práctica son los muestreos estocásticos. Los muestreos deterministas se usan muy poco, entre otros motivos porque van en contra de la filosofía del método. A continuación detallaremos alguno de los métodos comúnmente empleados en los Algoritmos Genéticos.

#### 3.4.4.1. Método de la ruleta

Este método, descrito en [40], consiste en crear una ruleta en la que cada cromosoma tiene asignada una fracción proporcional a su aptitud. Esta se "gira" tantas veces como individuos se necesiten para la etapa de reproducción. En cada giro se selecciona el individuo correspondiente a la ranura elegida. Debido a que a los individuos más aptos se les asignó un área mayor de la ruleta, se espera que sean seleccionados más veces que los menos aptos.

Con esta metodología es necesario evitar que aparezca un súper-individuo cuyo fitness sea muy superior a todos los demás, ya que ocupará un sector muy ancho de la ruleta y será elegido muchas veces, provocando la pérdida de diversidad genética y ocasionando seguramente una convergencia prematura a una solución subóptima.

En la Tabla 4 se plantea un ejemplo de una población de 5 cromosomas con sus correspondientes valores de aptitud. Para cada uno de ellos se calcula que porcentaje representa su fitness sobre el total de la población. Con estos valores, se construye la ruleta que se presenta en la Figura 29.

Cromosoma	Aptitud	Porcentaje
1	310	28,47
2	231	21,21
3	82	7,53
4	274	25,16
5	192	17,63
<b>Total</b>	<b>1089</b>	<b>100,00</b>

Tabla 4: Valores de ejemplo para ilustrar la selección mediante ruleta.

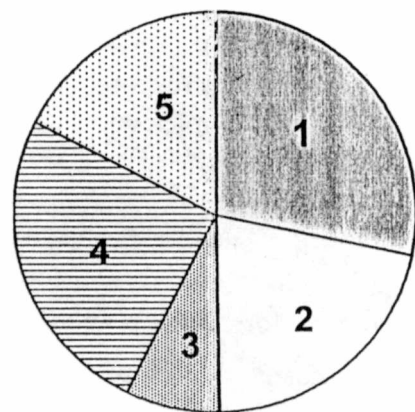


Figura 29: Ruleta que representa los valores de aptitud de la Tabla 4



### 3.4.4.2. Muestreo Universal Estocástico

Baker introdujo en [4] este método, el cual, al igual que el método anterior, construye una ruleta con ranuras de tamaño proporcional a la relación del fitness de cada individuo con respecto al total de la población. Se diferencia en que se efectúa un único giro de la ruleta y a partir del punto elegido se distribuyen uniformemente tantos marcadores como individuos se necesiten para la reproducción. Cada individuo es seleccionado tantas veces como marcadores aparezcan en su ranura. Nótese que es posible que algún individuo no resulte seleccionado por no contener ningún marcador en su ranura.

En la Figura 30 se presenta la misma ruleta mostrada para el método anterior a la cual se le agregan los marcadores distribuidos uniformemente a partir del punto seleccionado. Puede verse que el individuo 1 será elegido 4 veces mientras que el individuo 3 sólo se seleccionará una vez. También allí se puede observar que si una ranura de un individuo es de un tamaño inferior a la distancia entre dos marcadores, éste igualmente tiene probabilidad de ser seleccionado ya que éstos se distribuyen a partir de un punto inicial aleatorio. Este es el caso del individuo 3. Si el punto inicial hubiera sido diferente, podría no haber sido seleccionado.

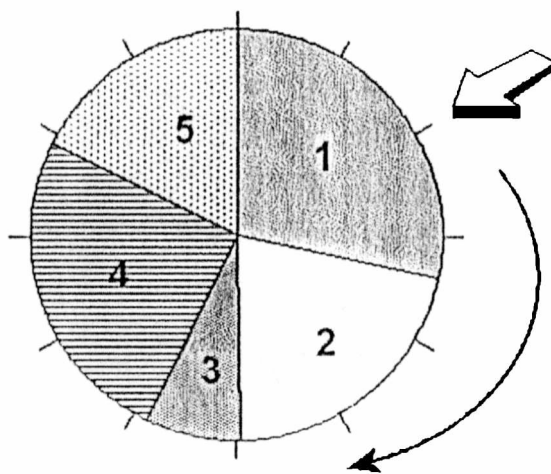


Figura 30: Ruleta empleada para el método de Muestreo Universal Estocástico.

### 3.4.4.3. Muestreo Estocástico con Reemplazo del Resto

Fue introducido por Brindle en [11] y empíricamente ha proporcionado buenos resultados. En este esquema de selección, primero se calcula para cada individuo el número esperado de ocurrencias en la nueva población. Cada individuo es seleccionado tantas veces como lo indique la parte entera del valor esperado. Todos los individuos compiten luego por el resto de los lugares de la población de acuerdo a la parte fraccionaria del número esperado de ocurrencias.

#### 3.4.4.4. Ranking

En este esquema, se construye un ranking ordenando la población de mayor a menor fitness. Los individuos son luego elegidos proporcionalmente a su posición en dicho ranking y no en función del porcentaje que representa su fitness [130].

Se obtienen  $M$  descendientes por reproducción de los  $N$  individuos mejor ranqueados. La descendencia reemplazará a los  $M$  peor calificados formando así la próxima generación (ver Figura 31).

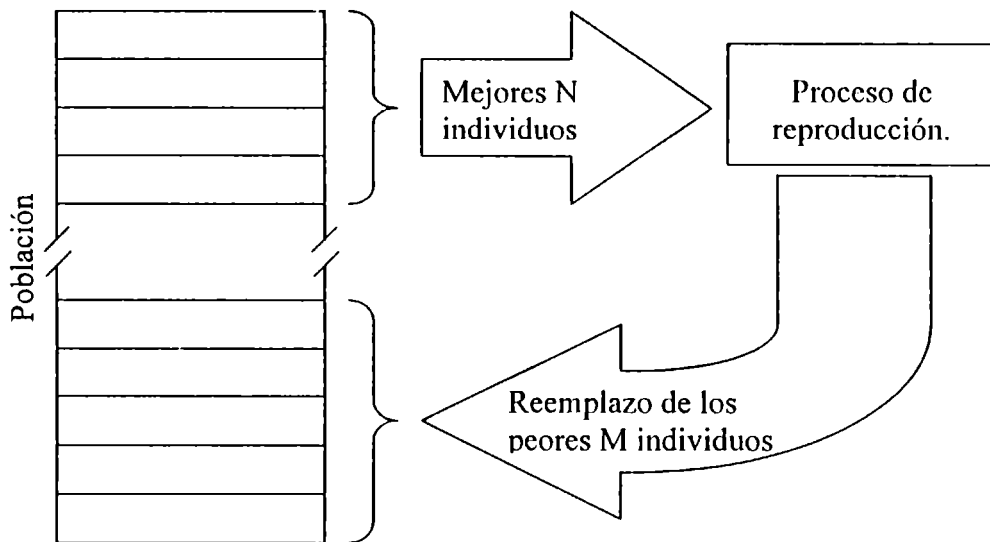


Figura 31: Método de selección por ranking.

#### 3.4.4.5. Torneo

Este método presentado en [42] consiste en mezclar la población y hacer competir los cromosomas que la integran en grupos de tamaño predefinido (normalmente en parejas). En cada uno de estos grupos se realiza un torneo del que resultará ganador aquel individuo que tenga el valor de aptitud más alto. Dicho individuo será seleccionado para la etapa reproductiva. Este proceso se repetirá tantas veces como padres sean necesarios.

Existe una versión probabilística en la cual se permite la selección de individuos sin que necesariamente sean los mejores del torneo en el que compiten.

#### 3.4.4.6. Elitismo

Consiste básicamente en realizar la etapa de selección en dos partes: Primero se forma una elite con un cierto número de miembros elegidos entre los mejores de la población actual para que pasen intactos a la siguiente generación. Luego, se aplica cualquier otro método de selección sobre el resto de la población. Esto impide que las mejores soluciones encontradas hasta el momento se pierdan.

Normalmente, el tamaño de la elite suele ser bastante pequeño, ya que se acostumbra que a los sumo uno o dos de las mejores soluciones pasen intactas a la siguiente generación.

### 3.4.5. Reproducción

Durante la fase reproductiva, los cromosomas que se seleccionaron de la población son recombinados aplicando operadores genéticos, obteniendo como resultado nuevos individuos que constituyen la descendencia de esa población.

Comúnmente, se toman parejas de cromosomas y se los combina aplicando mecanismos de *cruce* o *crossover*. A los nuevos individuos obtenidos del cruce se les aplica el operador de *mutación*. Estos dos operadores son característicos de los Algoritmos Genéticos y, a pesar de que existe una miríada de ellos, todos comparten el mismo principio fundamental: El operador de cruce se encarga de combinar el material genético de los padres para producir los hijos, mientras que el operador de mutación introduce pequeñas variaciones en los nuevos cromosomas generados. Se dice que el primer operador *explora* el material genético disponible en la población mientras que el segundo *explora* nuevas posibilidades.

Entre los operadores genéticos, el crossover es el principal, hasta el punto que se puede decir que no es un Algoritmo Genético si no lo tiene. Sin embargo, puede serlo perfectamente sin operador de mutación.

Existen otros operadores genéticos inspirados en procesos biológicos –que pueden catalogarse como otros tipos de mutaciones– denominados inversión, reordenamiento, duplicación, deleción y translocación [15]. A pesar de que ocurren en los procesos genéticos reales, estos operadores no han encontrado una aceptación popular en el campo de lo algorítmico.

La función del crossover, como se mencionó anteriormente, es el intercambio de material genético entre dos cromosomas y a veces más, como el operador *orgía* propuesto por Eiben [21] o en *MCMP* de Gallard [24].

La aplicación de este operador es aleatoria, dependiendo de cierta probabilidad denominada apropiadamente *Probabilidad de Crossover*, la cual es uno de los parámetros del Algoritmo Genético. Típicamente, se utiliza un valor alto para dicha probabilidad, de manera que casi todas las parejas de cromosomas se cruzarán, aunque habrá algunas que pasarán intactas a la siguiente generación.

En la evolución biológica, una mutación es un suceso bastante poco común ya que sucede aproximadamente una de cada mil replicaciones. En la mayoría de los casos, las mutaciones son letales, pero en promedio, contribuyen a la diversidad genética de la especie. En un Algoritmo Genético tienen el mismo papel, y la misma frecuencia, es decir, muy baja.

El operador de mutación, a diferencia del de cruce, permite introducir nueva información no presente en la población. Trabaja sobre un solo individuo, alterándolo con cierta probabilidad –denominada *Probabilidad de Mutación*, como no podía ser de otra manera–. Dependiendo del número de individuos que formen la población, puede resultar que las mutaciones sean extremadamente raras en una sola generación.

No hace falta decir que no conviene abusar de la mutación. Es cierto que es un mecanismo generador de diversidad, y, por tanto, la solución cuando un algoritmo genético está estancado, pero también es verdad que reduce el algoritmo genético a una búsqueda aleatoria si se aplica de forma indiscriminada.

A continuación, se describirán en detalle distintas variantes de los operadores de cruce y de mutación. Algunos de ellos se aplican a cadenas de bits mientras que otros, a cromosomas con codificación real.

### 3.4.5.1. Crossover de un punto

Este es uno de los casos más sencillos y se aplica a cadenas de bits. Se toman dos cromosomas, se elige aleatoriamente un punto de corte y se forman dos hijos uniendo los segmentos complementarios de ambos padres, como se muestra en la Figura 32.

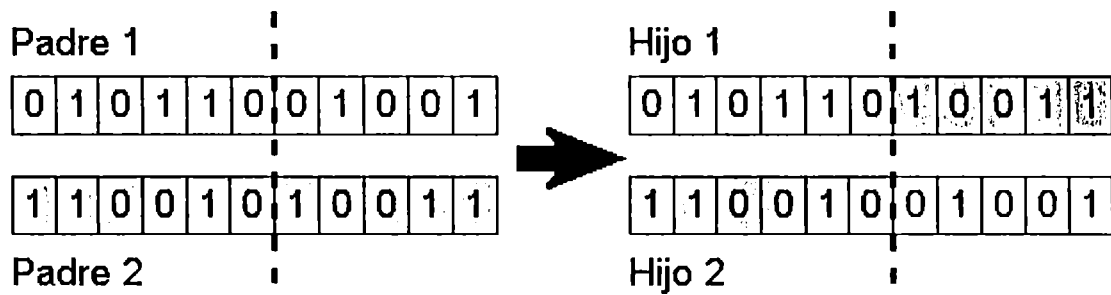


Figura 32: Posible resultado de aplicar el operador de crossover de un punto sobre un par de cromosomas.

### 3.4.5.2. Crossover multipunto

Este operador también se utiliza para combinar cadenas de bits. Se generan al azar  $n$  puntos y los cromosomas se cortan por cada uno de ellos, generando de esa manera  $n+1$  segmentos. Se numeran dichos segmentos y se obtienen dos descendientes intercambiando los segmentos pares [23].

Se ha concluido experimentalmente que el cruce basado en dos puntos representa una mejora con respecto al cruce de un punto, mientras que añadir más puntos de cruce no beneficia el comportamiento del algoritmo [75].

En la Figura 33 se ilustra la aplicación del operador de crossover de dos puntos.

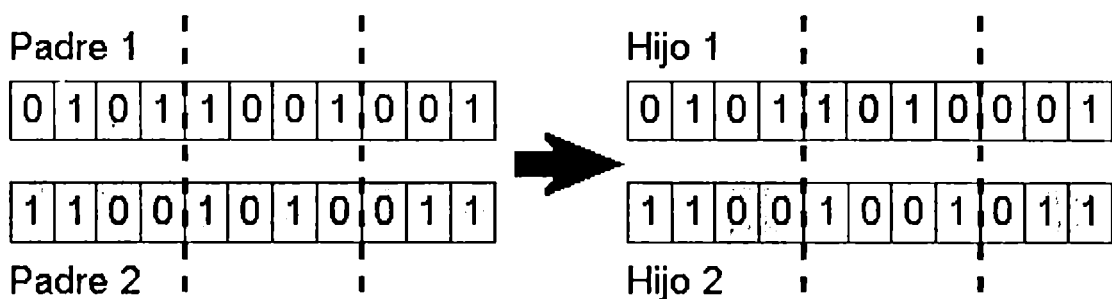


Figura 33: Posible resultado de aplicar el operador de crossover de dos puntos sobre un par de cromosomas.

### 3.4.5.3. Crossover uniforme

Este operador funciona con cadenas de bits y es una generalización del crossover de  $n$  puntos [123]. Para cada bit del primer hijo se decide, con cierta probabilidad, que padre aportará el valor que se le asignará. El segundo hijo recibirá el bit del padre no seleccionado. La aplicación de este operador es ejemplificada en la Figura 34.

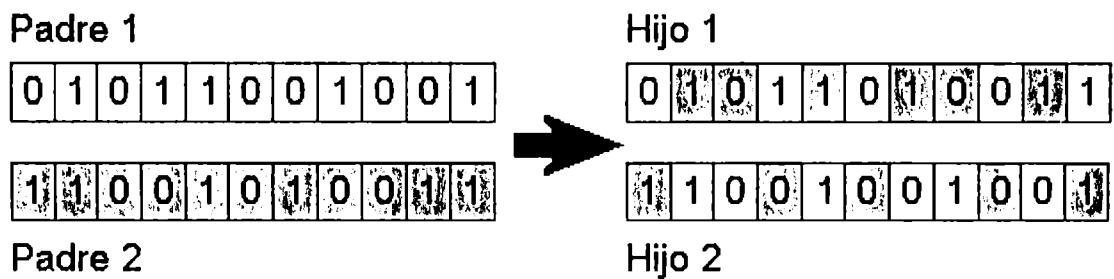


Figura 34: Posible resultado de aplicar el operador de crossover uniforme sobre un par de cromosomas.

#### 3.4.5.4. Crossover aritmético

Este operador se aplica a cromosomas que emplean una codificación basada en números reales. Se define en [90] como una combinación lineal de los dos cromosomas, de la manera indicada en la Ecuación 4. El parámetro  $\delta$  puede ser una constante, en cuyo caso este operador recibe el nombre de *Crossover Aritmético Uniforme*, o bien variar a lo largo de la evolución, llamándose al operador *Crossover Aritmético No Uniforme*. Este operador puede aplicarse a todos los elementos del cromosoma o sólo a un conjunto de elementos seleccionado.

Sean  $A = a_1 a_2 \dots a_n$  y  $B = b_1 b_2 \dots b_n$  dos cromosomas.

$C = \text{CrossoverAritmético}(A, B) = c_1 c_2 \dots c_n$

donde  $c_i = \delta \cdot a_i + (1 - \delta) \cdot b_i$

Ecuación 4: Definición del crossover aritmético.

#### 3.4.5.5. Crossover especializados

En algunos problemas, aplicar alguno de los operadores de crossover descritos da lugar a cromosomas que codifican soluciones inválidas. Esto debe ser evitado, ya sea reparando los cromosomas o forzando a que el operador genere siempre soluciones válidas. Un ejemplo de estos son los operadores de crossover usados en problemas de permutaciones [37], [18], [97].

#### 3.4.5.6. Mutación de cadenas de bits

Este operador examina cada bit de un cromosoma y con cierta probabilidad, invertirá el valor de dicho bit. Nótese que la probabilidad en cuestión afecta cada bit del cromosoma, en contrapartida con la probabilidad de crossover, que involucra al cromosoma entero. Puede verse un ejemplo de este operador en la Figura 35.

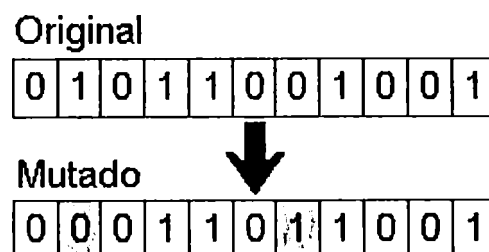


Figura 35: Posible resultado de aplicar el operador de mutación a un cromosoma binario.

### 3.4.5.7. Mutación aritmética

Se utiliza normalmente en conjunto con el crossover aritmético y, como éste, se aplica a cromosomas que emplean codificación basada en números reales.

Existen dos variantes [89]: En la primera, llamada *Mutación Uniforme*, cada elemento del cromosoma –que es un vector de valores reales– puede, con cierta probabilidad, ser reemplazado por un valor elegido al azar dentro del rango del dominio correspondiente a dicho elemento. Este operador es análogo al operador de mutación clásico para cadenas de bits.

La segunda variante, llamada *Mutación No Uniforme*, es similar a la primera salvo que el valor modificado se altera como se muestra en la Ecuación 5.

$$v' = \begin{cases} v + \Delta(t, u - v), & \text{si un dígito al azar es } 0 \\ v - \Delta(t, u - l), & \text{si un dígito al azar es } 1 \end{cases}$$

Ecuación 5: Forma de alterar un valor real del cromosoma.

En dicha ecuación,  $v$  es el valor original que contenía el cromosoma,  $v'$  es el nuevo valor que lo va a reemplazar,  $l$  y  $u$  son los límites inferior y superior del dominio del elemento modificado y  $t$  es el número de generación.

La función  $\Delta(t, y)$  devuelve un valor en el intervalo  $[0, y]$ , de manera tal que la probabilidad de que  $\Delta(t, y) \rightarrow 0$  aumenta a medida que  $t$  aumenta. Eso provoca que este operador explore uniformemente todo el espacio de soluciones al comienzo de la evolución y de forma local en las últimas etapas.

### 3.4.6. Reemplazo

Una vez obtenidos los individuos generados a partir de una determinada población mediante la aplicación de los operadores genéticos, debe seleccionarse que individuos serán reemplazados por esta descendencia, para construir así la población de la próxima generación. Supongamos que se tiene una población de  $N$  individuos y a través del proceso de selección y reproducción se generan  $n$  descendientes. El proceso de reemplazo puede hacerse de varias formas distintas:

- Los individuos de la próxima generación se forman con los  $n$  descendientes, es decir, se reemplaza completamente la población. Esto implica que  $n = N$ .
- Los  $n$  descendientes sustituyen a sus respectivos progenitores. Para mantener el tamaño de la población constante debe ocurrir que los  $n$  descendientes se hayan generado a partir de un grupo de  $n$  padres distintos.
- Los  $n$  descendientes sustituyen a aquellos miembros de la población que más se les parezcan. Esto se denomina *crowding* [19] y favorece la diversidad genética y aparición de especies.
- Se juntan los  $n$  descendientes con los  $m$  progenitores en una sola población, y en ella se seleccionan  $N$  individuos –normalmente los mejores– para formar la próxima generación, los restantes son eliminados.
- Se seleccionan  $n$  miembros de la población –generalmente los de más bajo fitness– para ser reemplazados por los  $n$  descendientes. Esto es válido sólo si  $n$  es menor que  $N$ .

- Se seleccionan los  $N$  miembros mejor ranqueados del conjunto de descendientes, luego de haberlos ordenado por su fitness, y con ellos se reemplaza a toda la población. Sólo puede aplicarse esto si  $n$  es mayor que  $N$ .

Nótese que las dos últimas alternativas constituyen la primera si  $n$  es igual a  $N$ .

Se han enumerado sólo unas cuantas técnicas de reemplazo aplicables a Algoritmos Genéticos cuya población es de tamaño constante. Existen alternativas en donde el tamaño de la población varía a lo largo de la evolución. Para estos tipos de Algoritmos Genéticos se han definido otras estrategias de reemplazo [2].

### **3.4.7. Condiciones de terminación**

La condición de terminación indica cuando el Algoritmo Genético debe detener el proceso evolutivo. Lo habitual es que sea la convergencia de la población, alcanzar un número prefijado de generaciones o, si se conoce el valor máximo posible para la función objetivo, encontrar un individuo que tenga como fitness dicho valor.

Para criterios prácticos, es muy útil la definición de convergencia introducida por De Jong en su tesis doctoral [19]. Si el Algoritmo Genético ha sido correctamente implementado, la población evolucionará a lo largo de las generaciones sucesivas de tal manera que la adaptación media extendida a todos los individuos de la población, así como la adaptación del mejor individuo, se irán incrementando hacia el óptimo global. El concepto de convergencia está relacionado con la progresión hacia la uniformidad: un gen ha convergido cuando al menos el 95 % de los individuos de la población comparten el mismo valor para dicho gen. Se dice que la población converge cuando todos los genes han convergido. Se puede generalizar dicha definición al caso en que al menos una determinada cantidad de individuos de la población hayan convergido.

Una idea interesante, planteada por Goldberg en [39] consiste en reiniciar la población cada vez que esta converja, guardando previamente las mejores soluciones encontradas hasta el momento para luego volver a introducirlas en la nueva población. Así se agrega nuevo material genético al proceso evolutivo, y con ello, diversidad a la población.

## **3.5. ¿Qué ventajas y desventajas tienen con respecto a otras técnicas de búsqueda?**

Los Algoritmos Genéticos presentan estas ventajas si se los compara con otros métodos de búsqueda tradicionales, la mayoría expuestas por Goldberg:

- Operan de forma simultánea con varias soluciones, en vez de trabajar de forma secuencial como las técnicas tradicionales.
- Cuando se usan para problemas de optimización –maximizar una función objetivo– resultan menos afectados por los máximos locales (falsas soluciones) que las técnicas tradicionales.
- Resulta sumamente fácil ejecutarlos tanto en las modernas arquitecturas masivamente paralelas como en computadoras con capacidades medias, proporcionando resultados aceptables, en cuanto a precisión y recursos empleados, para una gran cantidad de problemas difícilmente resolubles por otros métodos.

- Usan operadores probabilísticos, en vez de los típicos operadores determinísticos de las otras técnicas.

A pesar de las ventajas planteadas, también tienen sus inconvenientes:

- Pueden tardar mucho en converger, o no converger en absoluto, dependiendo de cierta medida de los parámetros que se utilicen –tamaño de la población, número de generaciones, etc. –.
- Pueden converger prematuramente debido a una serie de problemas de diversa índole, no alcanzando una solución óptima.

## 3.6. Fundamentos teóricos

### 3.6.1. Teorema de los Esquemas o Teorema Fundamental

Este teorema, desarrollado por Holland [57], proporciona el fundamento teórico de por qué funcionan los Algoritmos Genéticos. En su análisis se considera un Algoritmo Genético que emplea codificación binaria en sus cromosomas, utiliza un proceso de selección proporcional al fitness y como operadores genéticos, el crossover de un punto y la mutación que altera un sólo bit.

En dicho teorema, se define el concepto de *esquema*. Los esquemas se generan al introducir un nuevo símbolo comodín (\*) al alfabeto de los genes. De esa forma, un esquema representa todas las cadenas que coinciden con él en todos sus símbolos excepto el comodín. Por ejemplo, considerando un alfabeto binario –extendido para que incluya al símbolo comodín–  $V = \{0, 1, *\}$ , el esquema  $(1*01*)$  representa el conjunto de cadenas  $\{10010, 10011, 11010, 11011\}$ . Por otro lado, el esquema  $(0110)$  representa una sola cadena mientras que el esquema  $(****)$  representa todas las cadenas binarias de longitud 4.

Si el alfabeto, a partir del cual se definen los cromosomas, es de cardinalidad  $k$ , el correspondiente a los esquemas será de cardinalidad  $k+1$  debido al agregado del símbolo “\*”. Así, para cadenas de longitud  $L$  existirán  $k^L$  cromosomas distintos y un número mucho mayor de esquemas igual a  $(k+1)^L$ . Por ejemplo, para un alfabeto binario y cromosomas de longitud 5 tenemos:  $2^5 = 32$  cromosomas distintos y  $3^5 = 243$  esquemas distintos.

Existe una relación de “*muchos a muchos*” entre los cromosomas y esquemas. Así como un esquema representa varios cromosomas, un cromosoma está representado por varios esquemas. Considerando la cadena binaria 11111, existen  $2^5$  esquemas que lo representan, puesto que se pueden construir colocando en cada posición el bit correspondiente al cromosoma o el símbolo “\*”. En general, cualquier cadena de longitud  $L$  contiene  $2^L$  esquemas asociados que lo representan.

El *orden* de un esquema  $H$ , denotado por  $o(H)$  es simplemente el número de posiciones fijas del esquema, es decir, aquellas que no poseen el símbolo “\*”. Ejemplos:  $o(*101**1) = 4$ ,  $o(1**1101) = 5$  y  $o(******) = 0$ .

La *distancia de definición* de un esquema  $H$ , denotada por  $d(H)$ , es la diferencia entre la posición del primero y el último valor distinto de “\*”. Así,  $d(100*0**) = 4$  porque la primera posición fija es la 1 y la última la 5.  $d(***0***) = 0$  ya que tanto la primera como la última posición fija del esquema es la número 4.



El efecto de la selección sobre el número esperado de esquemas en la próxima generación es fácil de determinar bajo las hipótesis del teorema. Supóngase que  $t$  es la generación actual y  $H$  un esquema cualquiera. Denotaremos:

$m(H, t)$ : número de individuos en la generación  $t$  que son representados por  $H$ .

$f(H)$ : fitness promedio de los individuos representados por  $H$  en la generación  $t$ .

$\bar{f}$ : fitness promedio de toda la población en la generación  $t$ .

Considerando que la selección de individuos para la reproducción es proporcional al fitness, y que la descendencia se consigue simplemente como copias de dos padres –sin crossover ni mutación–, la cantidad de individuos representados por un esquema  $H$  en la próxima generación puede escribirse de la forma planteada en la Ecuación 6.

$$m(H, t + 1) = m(H, t) \cdot \frac{f(H)}{\bar{f}}$$

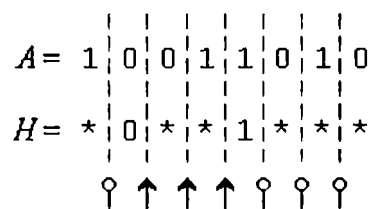
**Ecuación 6: Cantidad esperada de ocurrencias del esquema  $H$  sin crossover ni mutación.**

En otras palabras, el número de individuos asociados a un esquema crecerá de acuerdo a la proporción entre el fitness promedio del esquema y el de toda la población.

El efecto de la reproducción –sin crossover ni mutación– sobre el número de esquemas de una población a través de las generaciones es claro:

- Si el fitness promedio de un esquema es inferior al fitness promedio de la población, o sea que  $\frac{f(H)}{\bar{f}} < 1$ , este esquema tiende a desaparecer.
- Por el contrario, si  $\frac{f(H)}{\bar{f}} > 1$ , entonces el número de ocurrencia de dicho esquema aumentará.

Analicemos ahora que ocurre cuando se incluyen en el análisis a los operadores de crossover y de mutación. En este teorema se examina sólo el efecto negativo de estos operadores sobre un esquema  $H$ , puntualmente, la probabilidad de destruirlo.



**Figura 36: Posibles puntos de corte del operador de crossover de un punto.**

Para ver como afecta el crossover de un punto a los esquemas, observemos el cromosoma de longitud  $L = 8$ , junto con un esquema asociado a él, representados en la Figura 36.

Claramente, una operación de crossover sobre  $A$  no tendrá efecto sobre el esquema  $H$  si el punto de cruce elegido es la posición 1, 5, 6 ó 7 (marcados por  $\hat{\uparrow}$  en la figura). En el caso que el punto elegido sea la posición 2, 3 ó 4 (señalados con  $\uparrow$ ),  $H$  no sobrevivirá, salvo que el crossover se realice con otro cromosoma que coincida con el primero en las posiciones fijas del esquema.

Puede verse que la cantidad de puntos que producen la destrucción del esquema coincide con  $d(H)$ . En este caso, existen 3 puntos (indicados por  $d(H) = 3$ ) entre los posibles 7 puntos de cruce ( $L - 1 = 7$ ) que destruyen al esquema  $H$ .

Si el punto de cruce es elegido al azar con distribución uniforme, el esquema  $H$  será destruido con cierta probabilidad  $p_d$ , o de manera análoga, sobrevivirá con probabilidad  $p_s = 1 - p_d$ . Estas probabilidades se calculan así:

$$p_d = \frac{d(H)}{L-1} \quad p_s = 1 - p_d = 1 - \frac{d(H)}{L-1}$$

Si además se considera el efecto de aplicar el operador de crossover de manera aleatoria pero con cierta probabilidad  $p_c$ , durante la etapa reproductiva, se tiene entonces que:

$$p_d < p_c \cdot \frac{d(H)}{L-1} \quad p_s = 1 - p_d \geq 1 - p_c \cdot \frac{d(H)}{L-1}$$

Combinando este resultado con la Ecuación 6, se tiene que el número esperado de un esquema particular  $H$  en la próxima generación utilizando selección y crossover –sin mutación– será:

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \left[ 1 - p_c \cdot \frac{d(H)}{L-1} \right]$$

**Ecuación 7: Cantidad esperada de ocurrencias del esquema  $H$  aplicando crossover de un punto.**

El símbolo  $=$  a sido cambiado por  $\geq$  ya que el segundo término de la derecha representa una probabilidad, no un valor determinístico.

La Ecuación 7 nos indica que la cantidad de esquemas  $H$  crecerá o decaerá de acuerdo a dos cosas: Primero, si el fitness del esquema está más arriba o abajo del promedio de toda la población, y segundo, si el esquema posee una relativamente corta o larga distancia de definición. Claramente, el número de aquellos esquemas con ambas características –fitness promedio por encima de la media de la población y longitud de definición corta– crecerá en forma exponencial.

Ahora incluiremos en nuestro análisis el operador de mutación. La mutación se aplica independientemente sobre cada bit con probabilidad  $p_m$ . Así, la probabilidad de un alelo de no ser mutado será  $1 - p_m$ . Para que el esquema  $H$  sobreviva, es necesario que no sean mutados ninguno de los alelos del cromosoma  $A$  correspondientes a una posición fija (distinta de  $*$ ) en  $H$ . La cantidad de posiciones fijas de un esquema corresponde a  $o(H)$ , que definimos anteriormente. Por lo tanto la probabilidad de que un esquema sobreviva a la mutación será  $(1 - p_m)^{o(H)}$ , ya que la mutación se aplica independientemente a cada posición.

Así este teorema concluye que el número esperado de copias de un esquema particular  $H$  en la próxima generación, considerando los efectos de la selección, el crossover de un punto y la mutación puntual será el indicado en la Ecuación 8.

$$m(H, t + 1) \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \left[ 1 - p_c \cdot \frac{d(H)}{L-1} \right] (1 - p_m)^{o(H)}$$

**Ecuación 8: Cantidad esperada de ocurrencias del esquema  $H$  aplicando crossover de un punto y mutación.**

La conclusión final indica que *los esquemas pequeños de bajo orden cuyo fitness está por arriba del promedio reciben un incremento exponencial de representantes en las siguientes generaciones de un algoritmo genético.*

En definitiva, como un esquema representa una región en el espacio de búsqueda –un conjunto de cadenas– se ve que el algoritmo genético explota las regiones de más alto rendimiento del espacio de soluciones, dado que las sucesivas generaciones producen un número creciente de ellas. De hecho, el número de cadenas de una región o esquema dado, aumenta de forma proporcional a la estimación que se ha hecho del grado de adaptación de esa región.

### **3.6.2. Paralelismo implícito y la Hipótesis de construcción de bloques**

Cada individuo de la población pertenece a numerosos esquemas. En la sección anterior se explicó que cualquier cadena de longitud  $L$  contiene  $2^L$  esquemas asociados con él, ya que al construirlos, en cada una de las  $L$  posiciones se tienen 2 alternativas: colocar el valor correspondiente al cromosoma o el símbolo “\*”. Una población de  $n$  individuos tendrá asociada una cantidad de esquemas que varía entre  $2^L$  y  $n 2^L$ , dependiendo de la diversidad genética de sus miembros.

Si se asume que los individuos pertenecientes a un mismo esquema tienen características similares –y por ende una calidad parecida–, cuando se evalúa un individuo se obtiene información relativa a todos los individuos que comparten una pertenencia a los mismos esquemas. Bajo este supuesto, el algoritmo no procesa individuos, sino esquemas.

Sin embargo, se sabe que no todos los esquemas son procesados ya que, dependiendo de la longitud de definición y el orden, algunos tendrán una mayor probabilidad de supervivencia a los operadores de cruce y de mutación. Se ha estimado en [40] que en una generación de  $n$  individuos se procesan del orden de  $O(N^3)$  esquemas. Esta es una importante propiedad a la que Holland dio el nombre de *Paralelismo Implícito*. El Paralelismo Implícito es el que hace que un algoritmo genético que manipule una población de unas cuantas cadenas, realmente esté tomando muestras de un número de regiones o esquemas enormemente mayor. Tal paralelismo implícito –en el sentido de procesamiento paralelo–, proporciona al algoritmo genético una ventaja sobre otros métodos de búsqueda y optimización.

De acuerdo al teorema fundamental, los esquemas de corta longitud de definición, bajo orden y cuyo fitness sea mayor al promedio comenzarán a dominar rápidamente la población a medida que avance la evolución. Admitiendo entonces que juegan un papel importante en el desarrollo del algoritmo genético Goldberg prefiere darles un nombre especial: Bloques de Construcción o *Building Blocks*. De acuerdo a este autor, el funcionamiento del Algoritmo Genético será óptimo si es posible combinar estos esquemas cortos para formar soluciones cada vez mejores. A esta suposición se la denomina Hipótesis de los Bloques de Construcción o *Building Block Hypothesis*.

Así, los Algoritmos Genéticos son viables para resolver problemas cuya solución pueda formarse con bloques constructivos. Por ejemplo, si el objetivo es encontrar una cadena de  $n$  bits con mayor cantidad de 1s, el problema parece ser muy apto para un algoritmo genético, ya que los bloques constructivos son los bits con valor igual a 1.

De acuerdo con Beasley [7], una buena codificación debe animar a la formación de buenos bloques de construcción asegurando que los genes relacionados estén juntos en el cromosoma y que exista poca interacción entre genes, aunque estas dos condiciones no son siempre fáciles de conseguir.



## 4. Neuroevolución

Hasta aquí hemos visto en detalle a las redes neuronales artificiales –Pág. 3-, como funcionan y las posibilidades en cuanto a entrenamiento o aprendizaje. También presentamos a los algoritmos evolutivos en general –Pág. 39- sus variantes y aplicaciones.

Combinando estos dos conceptos nace la *Neuroevolución*. Básicamente consiste en codificar en cromosomas representaciones de redes neuronales de manera de generar poblaciones que son sometidas a una búsqueda genética orientada a establecer una red que permita resolver un determinado problema. Las RNA obtenidas mediante un método de evolución son mencionadas en buena parte de la bibliografía específica del área como *Redes Neuronales Artificiales Evolutivas* (RNAE).

En las distintas secciones de este capítulo detallaremos ventajas e inconvenientes de este paradigma, aspectos a evolucionar de las redes y los esquemas de codificación directa e indirecta.

### 4.1. Conveniencia de la neuroevolución

Las RNA son exitosas y han resuelto adecuadamente muchos problemas, pero presentan ciertas dificultades. Entre algunas otras citaremos:

- Tienen limitaciones en su capacidad de aprendizaje.
- No siempre es posible encontrar la topología más eficiente para resolver un problema.
- No es simple determinar las características adecuadas de la entrada.
- Existen muchos algoritmos de entrenamientos basados en métodos de gradiente y éste a veces es incalculable o muy costoso de hacerlo.
- Para utilizar estos algoritmos basados en el gradiente deben imponerse restricciones sobre la arquitectura de la red tales como funciones de transferencia y de error derivables.
- Para asegurar un buen desempeño, la velocidad de aprendizaje deber ser pequeña lo cual deriva en entrenamientos prolongados.
- La utilización de parámetros de velocidad de aprendizaje más grandes –para acelerar el tiempo requerido del entrenamiento– puede llevarlas a un estado de inestabilidad con oscilaciones continuas, lo que degrada el aprendizaje.
- Son proclives a ser “engañadas” cayendo en mínimos locales.
- No es fácil saber a priori –salvo recurriendo a la experiencia– cuántas capas y unidades ocultas son necesarias para resolver adecuadamente el problema.
- Generalmente poseen fuerte dependencia con los valores iniciales de los pesos escogidos aleatoriamente.
- Frecuentemente el entrenamiento requiere mucho tiempo de procesamiento.

Todas estas dificultades han llevado a los investigadores al estudio de alternativas para paliar algunos de estos inconvenientes. Así se han propuesto muchísimas estrategias para combinar el poder de las RNA con el de los algoritmos evolutivos resolviendo satisfactoriamente muchos de los inconvenientes mencionados.

Los Algoritmos Evolutivos constituyen un método de búsqueda estocástico que no tiene problemas de selección de paso, es decir parámetro de velocidad, ni de mínimos locales, y tiene la característica deseable de trabajar en paralelo. Estas cualidades hacen de este paradigma una buena opción para el entrenamiento de redes neuronales, pues al no utilizar información de gradiente, los algoritmos evolutivos son en general más rápidos, más simples de implementar y tienen menos probabilidad de caer en un mínimo local que los algoritmos de entrenamiento tradicionales.

La evolución se ha utilizado en diversas formas para conseguir establecer varios aspectos de las redes neuronales artificiales, tales como los pesos de conexión, el diseño de la arquitectura, el valor de los parámetros iniciales, las reglas de aprendizaje, etc.

Una característica distinguible de las RNAE es su capacidad de adaptabilidad a un ambiente dinámico [133]. Las RNAE pueden adaptarse a un ambiente a medida que éste va cambiando. Mientras que las RNA tienen una única forma de adaptación – aprendizaje–, las RNAE poseen dos formas de adaptación –evolución y aprendizaje– ya que se pueden evolucionar redes que luego sean sometidas a un entrenamiento tradicional. Esto sucede por ejemplo cuando se evoluciona la inicialización de los parámetros de la red, la idea es obtener la mejor inicialización para que el entrenamiento resulte más eficiente<sup>3</sup>.

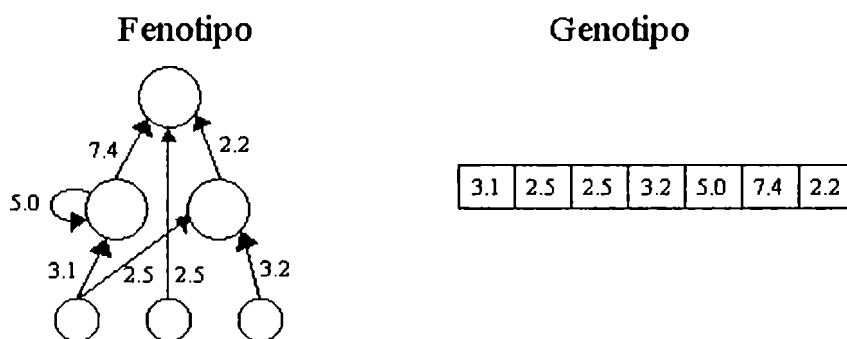
El contar con dos formas de adaptación hace que las RNAE sean mucho más efectivas y eficientes que las RNA para adaptarse a un ambiente dinámico sin intervención humana.

## 4.2. Evolución de pesos de conexión

Ya hemos mencionado los inconvenientes de la utilización del gradiente en los algoritmos de entrenamiento. Una forma de evitar estas dificultades consiste en prescindir de estos algoritmos. En su lugar, se puede recurrir a los algoritmos evolutivos para encontrar el conjunto de pesos de conexión adecuados que haga funcionar correctamente a la red en la realización de la tarea pretendida.

---

<sup>3</sup> Esto constituye una forma de hibridación de técnicas de búsqueda, donde se combinan la evolución para la tarea exploratoria y un algoritmo de aprendizaje tradicional para la tarea explotatoria.



**Figura 37: Entrenamiento Evolutivo.** Sólo se somete a proceso evolutivo el conjunto de pesos de conexión. Obsérvese que la arquitectura de la red –fijada previamente– corresponde a una topología arbitraria, el método no asume ninguna restricción sobre la misma

Para una arquitectura fija predefinida, se codifican los pesos de las conexiones en un cromosoma, se arma la población y se somete a evolución. Para calcular el fitness de los individuos se construyen las redes respectivas con los pesos de conexión determinados en los cromosomas y se evalúa la misma para obtener una calificación acerca de su desempeño. Por ejemplo, el fitness de la red podría ser una función inversamente proporcional al error cuadrático medio calculado con un conjunto de pruebas. Debe quedar claro que al evaluar la red no se realiza modificación de pesos alguna, sólo se calcula el error para poder asignarle un valor de fitness.

Debido a que sólo se somete a evolución el conjunto de pesos de conexión quedando todos los otros parámetros de la red fijos, a esta estrategia se la denomina *entrenamiento evolutivo* pues hace lo mismo que cualquier algoritmo de entrenamiento tradicional. No obstante posee algunas ventajas pues no necesita información de gradiente, no está limitado a ningún tipo de arquitectura en particular, tiene menos probabilidad de caer en un óptimo local y se puede utilizar un algoritmo genético simple sin ningún esfuerzo adicional.

En cuanto a la codificación elegida para representar en el cromosoma los valores reales de los pesos de conexión valen las mismas consideraciones que discutimos al ver los Algoritmos Genéticos en el capítulo anterior –Pág. 46–.

Si se pretende respetar el algoritmo genético simple se utilizarán cadenas binarias, así cada peso será representado por un número determinado de bits. La red neuronal quedará codificada simplemente por la concatenación de los pesos de conexión – recordemos que la arquitectura de la red es fija–. Algunos investigadores han utilizado una codificación binaria con buenos resultados. Otros han preferido utilizar un alfabeto de mayor cardinalidad proponiendo la utilización de números reales en lugar de binarios para codificar los cromosomas. El ejemplo de la Figura 37 utiliza esta última representación. Recordemos que esta discusión excede el ámbito de la Neuroevolución perteneciéndole al paradigma más amplio de la Computación Evolutiva.

#### **4.2.1. Entrenamiento evolutivo vs. entrenamiento basado en gradiente**

Ya mencionamos que el entrenamiento evolutivo es atractivo pues puede manejar el problema de la búsqueda global en superficies del error complejas, multimodales y no



diferenciables; por lo tanto, es preferible cuando la información del gradiente es inexistente, o costosa de conseguir o estimar. Existen numerosos ejemplos donde se ha utilizado una aproximación evolutiva para entrenar redes recurrentes de alto orden. Además el mismo algoritmo puede ser usado para entrenar cualquier tipo de red sin tener que considerar si ellas son feedforward, recurrentes o de alto orden.

El entrenamiento evolutivo puede ser más lento para algunos problemas en comparación con las más rápidas variantes del algoritmo Backpropagation y el algoritmo del gradiente conjugado. Sin embargo los algoritmos evolutivos son menos sensibles a las condiciones iniciales de entrenamiento. Ellos siempre buscan soluciones óptimas globales, mientras que los algoritmos de gradiente descendente pueden sólo buscar una solución óptima local en la vecindad de la solución inicial.

Existe un conjunto grande de problemas donde el entrenamiento evolutivo puede ser significativamente más rápido y confiable que el Backpropagation. Estos casos han sido estudiados y expuestos por varios investigadores entre ellos Prados, Bartlett, y Downs[133]. Sin embargo, otros resultados bastante distintos fueron presentados por Kitano[70] que encontró que el método AG-BP –una técnica híbrida que corre un Algoritmo Genético primero y luego un Backpropagation– es tan bueno como las variantes rápidas de Backpropagation para redes pequeñas, pero menos eficiente en grandes redes. Sin embargo ha habido varios otros artículos (tales como [68], [61] y [121]) que reportan excelentes resultados usando hibridación de entrenamiento evolutivo y algoritmos de gradiente descendente.

Esta discrepancia puede ser atribuida, al menos parcialmente, a que han sido comparados distintos algoritmos evolutivos y distintas variantes de Backpropagation. Pero esta diferencia también señala que no existe todavía un claro ganador en términos del mejor algoritmo de entrenamiento, sino que sus desempeños se ven afectados por la naturaleza del problema. Una característica de las RNA que puede degradar el entrenamiento evolutivo es la equivalencia funcional de distintas redes. Por ejemplo: si alteramos el orden de las neuronas de la capa oculta de una red multicapa feedforward totalmente conectada, aunque distintas, producirán los mismos resultados. A esto se lo denomina *problema de permutación* y causa el hecho indeseable que genotipos distintos produzcan fenotipos que funcionan de igual forma. Según Xin Xao en [133] el problema de permutación disminuye la eficiencia del crossover.

En un intento de reducir el impacto negativo del problema de permutación, varios investigadores han abandonado el crossover como operador genético utilizando esencialmente el operador de mutación o alguna variante –evolución primitiva–. Así se han apartado de los Algoritmos Genéticos que tienen al crossover como principal operador.

#### **4.2.2. Entrenamiento híbrido**

Para una función unimodal, es decir que posea un solo extremo –máximo o mínimo según se trate de una función de fitness o de una función de error–, la mayoría de los algoritmos evolutivos no son tan eficientes como los que usan el método del gradiente descendente, para encontrar rápidamente el punto óptimo.

Sin embargo en una función complicada, multimodal, donde existen muchos máximos o mínimos locales, los algoritmos evolutivos realizan una mejor búsqueda global. Esto es especialmente verdadero para los Algoritmos Genéticos. Por lo tanto, la eficiencia del entrenamiento evolutivo puede ser mejorada significativamente incorporando

procedimientos de búsqueda local dentro de la evolución. Así los algoritmos evolutivos encontrarán una buena región en el espacio para que un procedimiento de búsqueda local halle la mejor solución dentro de la misma. Este algoritmo local podría ser por ejemplo el algoritmo Backpropagation si la arquitectura de la red lo permite.

Muchos investigadores (como en [98], [99] y [8]) han utilizados Algoritmos Genéticos para buscar un conjunto inicial de pesos de conexión cercano al óptimo ( ). Una vez hallada esta región, un algoritmo Backpropagation completa la tarea realizando una búsqueda local desde esos pesos iniciales. Sus resultados muestran que la hibridación AG-BP es más eficiente que cualquiera de los métodos de búsqueda por separado. También se han realizado similares trabajos de evolución de pesos iniciales en redes neuronales de aprendizaje competitivo y redes de Kohonen.

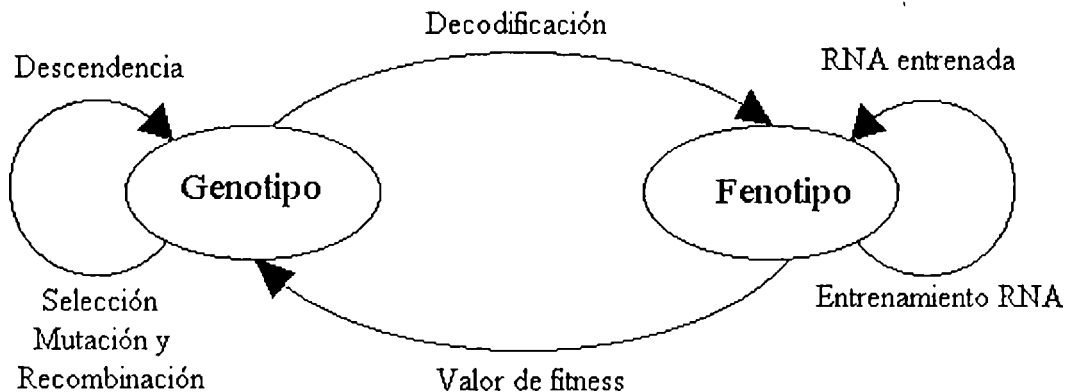


Figura 38: Esquema de un de entrenamiento híbrido.

Otra vez se está cuestionando la robustez de los Algoritmos Genéticos al presentar mejores resultados con técnicas híbridas, donde es necesario incluir información específica del problema para mejorar la solución.

### 4.3. Evolución de la arquitectura

Aquí se discute el diseño de arquitecturas de RNA. En lo que se refiere a su topología –conectividad y función de transferencia de cada nodo–.

La arquitectura es crucial puesto que posee gran impacto sobre las capacidades de procesamiento de la red. Dada una tarea de aprendizaje una red neuronal con solo unas pocas conexiones y nodos lineales puede que no sea capaz de realizarla. Por otro lado, una red con un gran número de conexiones y nodos no lineales puede sobreajustarse al ruido en los datos de entrenamiento y fallar en la adquisición de una buena capacidad de generalización.

Todavía el diseño de la arquitectura depende mucho del trabajo del humano experto y del tedioso proceso de ensayo y error. No hay una forma sistemática y automática para diseñar una arquitectura cercana a la óptima para una tarea dada. Las investigaciones sobre *algoritmos constructivos* y *destructivos* representan un esfuerzo hacia el diseño automático de arquitecturas. A grandes rasgos, un algoritmo constructivo comienza con una red minimal –mínimo número de capas ocultas, nodos y conexiones– y va agregando nuevas capas, nodos y conexiones durante el entrenamiento. Un algoritmo destructivo utiliza la estrategia opuesta, comienza con una red maximal y va borrando

capas, nodos y conexiones innecesarias durante el entrenamiento. Sin embargo estos métodos son susceptibles de “caer” en una estructura óptima local.

El diseño de arquitecturas óptimas de RNA puede formularse como un problema de búsqueda en el espacio de las arquitecturas, donde cada punto representa una arquitectura completa, y dado algún criterio para calificar su performance – menor error en el entrenamiento, menor tiempo de aprendizaje, menor complejidad de la red etc. – pueden utilizarse algoritmos evolutivos para resolver el problema.

Al igual que como ocurre con la evolución de los pesos de conexión, la evolución de arquitecturas tiene dos grandes fases: la representación del genotipo y el algoritmo elegido para llevar adelante la evolución. Uno de los puntos clave en la codificación genética es decidir cuanta información acerca de la arquitectura se codificará en el cromosoma. En un extremo, todos los detalles, es decir toda conexión y nodo de la arquitectura puede ser especificado por un cromosoma. Esta clase de representación es llamada *codificación directa*. En el otro extremo, solo los parámetros más importantes de una arquitectura, como el número de capas ocultas y nodos ocultos en cada capa son codificados. Otros detalles sobre la arquitectura son dejados para que los decida el proceso de entrenamiento. Esta clase de representación es llamada *codificación indirecta*. Para mayor detalle ver “Esquema de codificación directa” -Pág. 73- y “Esquema de codificación indirecta” -Pág. 74-.

Una vez seleccionado el esquema de codificación, la evolución de arquitecturas puede progresar siguiendo el siguiente ciclo:

1. Decodificar cada individuo de la generación corriente obteniendo así la arquitectura de la RNA –fenotipo–.
2. Entrenar varias veces cada una de las RNA con una regla de aprendizaje predefinida, comenzando desde diferentes conjuntos aleatorios iniciales de pesos de conexión.
3. Computar el fitness de acuerdo al resultado promediado de cada entrenamiento y otros criterios de performance.
4. Seleccionar los padres y generar la siguiente generación.

Notemos que la evolución de una arquitectura óptima de red neuronal artificial, debe complementarse con algún método de entrenamiento tradicional puesto que los pesos de la conexión no se someten al proceso evolutivo. Por lo tanto es un nuevo caso de hibridación, de hecho obsérvese que los pasos descritos arriba siguen el esquema general presentado en la Figura 38. Si pretendemos olvidarnos definitivamente de los entrenamientos tradicionales de RNA podemos evolucionar conjuntamente la arquitectura y los pesos de conexión –algoritmo evolutivo puro– pero esto lo veremos más adelante en este mismo capítulo.

Una investigación considerable sobre evolución de arquitecturas de RNA ha sido realizada en los años recientes. La mayoría se ha concentrado en evolucionar la topología y algunos pocos evolucionaron la función de transferencia de cada nodo.

## 4.4. Evolución de función de transferencia de los nodos

En el punto anterior la función de transferencia en cada nodo es fija y predefinida por un experto humano. Generalmente se asume que esta función es la misma para todos los nodos al menos de la misma capa.

Algunos investigadores (por ejemplo [20], [51], [132] y [108]) han realizado evoluciones de topología y función de transferencia al mismo tiempo codificándolas en los cromosomas. Se han propuesto ciertas variantes, como por ejemplo fijar dos tipos de funciones de transferencia, una sigmoide y una gaussiana, comenzar el proceso evolutivo con un determinado porcentaje de nodos con un tipo de funciones y el resto con el otro. La evolución intenta determinar el porcentaje óptimo de nodos con cada tipo de función.

## 4.5. Evolución simultánea de arquitectura y pesos de conexión

La evolución de la arquitectura de las RNA sin los pesos hace que el proceso resulte ineficiente. El problema es que para evaluar el fitness de la arquitectura conseguida se debe entrenar la red para lo cual se inicializan aleatoriamente los pesos de conexión. Pero los algoritmos de entrenamiento dependen mucho de los valores iniciales de los pesos, así un genotipo  $G_1$  puede ser de mejor calidad que  $G_2$  y sin embargo su fitness ser peor debido a los pesos iniciales elegidos durante su evaluación. Habitualmente se subsana este problema realizando varios entrenamientos con distintos valores iniciales y la medida de aptitud es tomada como un promedio de estos entrenamientos. Obviamente esto trae aparejado un mayor tiempo de procesamiento. Esta es la razón principal por la cual se han evolucionado arquitecturas de redes pequeñas con esta estrategia.

Por lo tanto podemos concluir que la evolución de arquitecturas sin información de los pesos dificulta una certera evaluación del fitness y consecuentemente la evolución se hace muy ineficiente.

Una forma obvia de solucionar este problema es evolucionar simultáneamente arquitecturas y pesos de conexión.

Una cuestión importante en la evolución de RNA es la elección de los operadores de búsqueda utilizados en el algoritmo evolutivo. Tanto crossover como mutación han sido utilizados. Sin embargo, como ya se dijo antes, el uso de crossover está cuestionado por algunos investigadores pues contradice las ideas básicas detrás de las RNA. Ya vimos en la sección 3.6.1 que este operador trabaja bien cuando existen buenos bloques constructivos<sup>4</sup>, pero esto no es claro que ocurra en las RNA puesto que ellas realizan una representación distribuida del conocimiento entre todos los pesos de la Red. La recombinación de una parte de la red con otra parte de otra es probable que destruya ambas.

Sin embargo si la red no usa una representación distribuida sino más bien localizada, como ocurre con algunos tipos de redes neuronales, entonces el crossover puede ser muy útil.

---

<sup>4</sup> Los buenos bloques constructivos son esquemas de corta longitud de definición -bajo orden- y cuyo fitness es mayor al promedio. Ver sección 3.6.2.

## 4.6. Evolución de reglas de aprendizaje

Las investigaciones en evolución de reglas de aprendizaje están aun en una temprana etapa de desarrollo[133].

Los parámetros del algoritmo de aprendizaje de la red Backpropagation –como la velocidad de aprendizaje y momento– ajustados por evolución pueden ser considerados como el primer intento de evolucionar reglas de aprendizajes. Varios autores han evolucionado estos parámetros junto con la arquitectura. Otros han evolucionado los parámetros pero con una arquitectura fija predefinida. Algunos trabajos pueden verse en [95], [12], [28] y [6]. Estos parámetros evolucionados tienden a optimizar dicha arquitectura más que a evolucionar una regla genérica de aprendizaje.

El mismo problema descrito antes con respecto a la inicialización de pesos aleatorios se repite aquí y también se resuelve con varios entrenamientos para calcular un promedio. Si la arquitectura es fija, la regla de aprendizaje evolucionará hacia la óptima para esa arquitectura. Si se desea evolucionar una regla de aprendizaje cercana a la óptima para varias arquitecturas, el fitness se basa en el promedio de entrenamientos de varias arquitecturas.

Un ciclo típico de evolución de reglas de aprendizaje para varias arquitecturas puede ser descrito así.

1. Decodificar cada regla de aprendizaje.
2. Construir un conjunto de RNA con arquitecturas y pesos iniciales aleatorios y entrenarlos usando la regla decodificada.
3. Calcular la función de fitness para cada individuo de acuerdo al promedio del resultado.
4. Seleccionar padres para generar la próxima generación.

Generalmente se asume que la regla de aprendizaje es la misma para todas las conexiones de la red y que la adaptación de los pesos depende solo de información local –valores de entrada, activación de la salida del nodo, los pesos de conexión actuales, etc. –.

Una regla de aprendizaje se considera que es una función lineal de estas variables locales y sus productos. De manera general se puede escribir como lo muestra la Ecuación 9.

$$\Delta\omega(t) = \sum_{k=1}^n \sum_{i_1, i_2, \dots, i_k=1}^n \left( \theta_{i_1 i_2 \dots i_k} \prod_{j=1}^k x_{ij}(t-1) \right)$$

**Ecuación 9: Regla de aprendizaje**

En dicha expresión  $t$  es el tiempo,  $\Delta\omega$  es el cambio de peso,  $x_1, \dots, x_n$  son variables locales, y los  $\theta_i$  son coeficientes reales que serán determinados por la evolución. En otras palabras, la evolución de reglas de aprendizaje es equivalente a evolucionar vectores de valores reales  $\theta_i$ . Diferentes  $\theta_i$  determinan diferentes reglas de aprendizaje.

Debido al largo número de posibles términos en esta ecuación la evolución se hace muy lenta y muchas veces resulta impracticable. Una cuestión importante aquí es determinar un subconjunto de términos descriptos en esta ecuación. Para citar un ejemplo Chalmers[12] define una regla de aprendizaje como una combinación lineal de 4 variables, codifica los valores reales en binario y usa arquitectura fija. Después de 1000 generaciones la evolución descubrió la bien conocida regla delta y algunas variantes. Este experimento, aunque simple, demostró el potencial de evolución de reglas de aprendizaje.

## 4.7. Esquema de codificación directa

Se han desarrollado muchos tipos distintos de codificación directa, es decir aquella que codifica exhaustivamente todos los parámetros de la red. Por ejemplo una matriz  $C^{N \times N}$  de elementos  $c_{ij}$  binarios puede representar una arquitectura con  $N$  nodos indicando ausencia o presencia de la conexión desde el nodo  $i$  al nodo  $j$  según  $c_{ij}$  contenga o no el valor nulo respectivamente. El cromosoma será la concatenación de cada fila o columnas de la matriz.

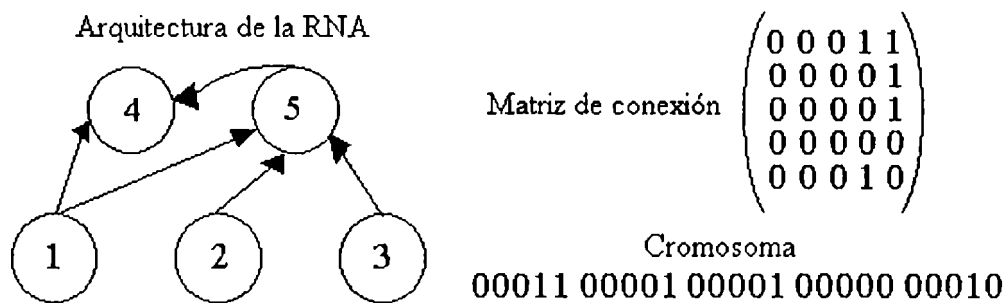


Figura 39: Codificación directa de la topología de una RNA

Generalmente se utiliza el error y el tiempo de aprendizaje para calcular la función de fitness. La medida de complejidad como el número de nodos y conexiones es también utilizada en el fitness. Schaffer[118] ha presentado un experimento con el cual muestra que una RNA diseñada por evolución tiene mejor habilidad de generalización que una entrenada con Backpropagation usando una arquitectura diseñada por el hombre.

Un problema potencial de la codificación directa es la escalabilidad. Una red grande requiere una matriz muy grande. Una forma de reducir el tamaño de la matriz es usar conocimiento del dominio para reducir el espacio de búsqueda. Por ejemplo, si se quiere evolucionar una red completamente conectada feedforward puede codificarse en el cromosoma sólo el número de capas ocultas y los nodos en cada capa oculta.

El problema de permutación existe también en la evolución de arquitecturas disminuyendo la probabilidad de producir descendencia mejor adaptada por recombinación genética. Por lo tanto algunos investigadores evitan el crossover y adoptan sólo mutación en la evolución de arquitecturas.

Sin embargo se ha mostrado que el crossover puede ser útil e importante para incrementar la eficiencia de evolución para algunos problemas. Además, Hancock<sup>180</sup>[50] sugiere que el problema de permutación puede no ser tan severo como se ha supuesto. Aparentemente este es otro caso de discusión que aún permanece abierta.

## 4.8. Esquema de codificación indirecta

La codificación indirecta ha sido utilizada por varios investigadores para reducir la longitud de los genotipos de las arquitecturas, codificando sólo algunas características de ellas en el cromosoma. Los detalles acerca de cada conexión en una RNA son o bien predefinidos de acuerdo a previo conocimiento o especificados por un conjunto determinístico de reglas de desarrollo.

Este tipo de codificación puede producir genotipos compactos pero no es muy buena para conseguir RNA compactas con buen nivel de generalización. Algunos argumentan que la codificación indirecta es más plausible que la directa en la biología, aparentemente es imposible codificar en genes cada una de las características de nuestras redes neuronales, puesto que sencillamente no hay lugar en los cromosomas para albergar tanta información.

La arquitectura de las RNA puede ser codificada en un cromosoma a través de un conjunto de parámetros tales como el número de capas ocultas, el número de nodos ocultos en cada capa, el número de conexiones entre dos capas, etc. La ventaja es que disminuye la longitud de los cromosomas, la desventaja es que también reduce el espacio de búsqueda al que podrá acceder el algoritmo. Estos recortes se realizan por un conjunto de suposiciones que deben imponerse. Por ejemplo, se puede codificar sólo el número de nodos ocultos en la capa intermedia, claro está, asumiendo que se trata de una RNA feedforward completamente conectada con una sola capa oculta. En general la representación paramétrica es conveniente cuando conocemos qué clase de arquitectura estamos tratando de encontrar, de lo contrario estaremos truncando a ciegas el espacio de búsqueda.

Otro método utilizado consiste en codificar reglas de desarrollo para construir las arquitecturas neuronales. El efecto destructivo del crossover es minimizado puesto que la representación de reglas de desarrollo es capaz de preservar los “*Building Blocks*” prometedores.

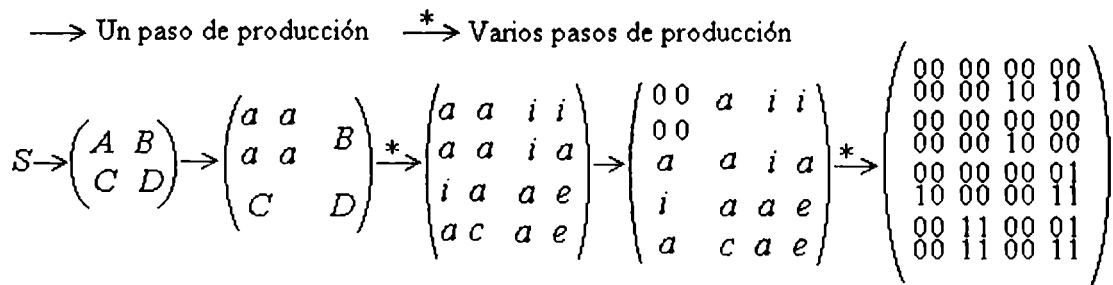
Una regla de desarrollo usualmente se describe por una ecuación recursiva o una regla de producción. Veamos un ejemplo utilizando reglas de producción.

Conjunto de reglas de desarrollo

$$\begin{aligned}
 S &\rightarrow \begin{pmatrix} A & B \\ C & D \end{pmatrix} & A &\rightarrow \begin{pmatrix} a & a \\ a & a \end{pmatrix} & B &\rightarrow \begin{pmatrix} i & i \\ i & a \end{pmatrix} \\
 C &\rightarrow \begin{pmatrix} i & a \\ a & c \end{pmatrix} & D &\rightarrow \begin{pmatrix} a & e \\ a & e \end{pmatrix} \dots \\
 a &\rightarrow \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & c &\rightarrow \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} & i &\rightarrow \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} & e &\rightarrow \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \dots
 \end{aligned}$$

**Figura 40:** Reglas de desarrollo como conjunto de reglas de producción o reescritura. Cada regla puede representarse en el cromosoma por cuatro alelos que representan la parte derecha de la misma. La parte izquierda de la regla puede representarse implícitamente por la posición dentro del cromosoma. S es el *start symbol* y por donde comienza el proceso de reescritura.

El patrón de conectividad de la arquitectura en forma de matriz es construido desde el *start symbol*, por la aplicación repetida de las reglas de desarrollos a los elementos no terminales hasta que la matriz contenga sólo elementos terminales.



**Figura 41: Desarrollo de la conectividad de la arquitectura por la aplicación sucesiva de las reglas de desarrollo de la Figura 40.**

La matriz de conexión desarrollada en la Figura 41 define una arquitectura particular de RNA.

Algunos buenos resultados con representación de reglas de desarrollo fueron reportados. Sin embargo, el método tiene varias limitaciones. A menudo necesita predefinir el número de pasos de reescritura. No permiten reglas recursivas. Un genotipo compacto no implica un fenotipo compacto es decir RNA compactas. Generalmente esta representación separa la evolución de arquitecturas de la de pesos de conexión lo que crea algunos problemas en la evolución.

Finalizamos la aproximación general al paradigma de las RNAE o Neuroevolución. Debemos mencionar que existen muchas otras estrategias neuroevolutivas que no vamos a citar en este capítulo. No obstante, en el presente trabajo, se han implementado algunas de ellas, y serán expuestas más adelante en el capítulo 7 –Pág. 111-.

Para completar las nociones básicas para la comprensión del método neuroevolutivo presentado en esta tesis, describiremos en el siguiente capítulo a los sistemas de reescritura y más en detalle los Sistemas-L, parte fundamental del desarrollo realizado.





## 5. Sistemas de reescritura

El presente capítulo está dedicado a los sistemas de reescritura. En la primer sección presentaremos las gramáticas formales, la forma en que estas se reescriben y los lenguajes asociados a las mismas. En la segunda sección describiremos en detalle a los Sistemas L, un tipo particular de gramática formal. Mostraremos las variantes que presentan los mismos y finalizaremos este capítulo brindando un conjunto diverso de aplicaciones de los mismos.

### 5.1. Gramáticas formales

Los lenguajes libres de contexto, como los conjuntos regulares, son de gran importancia práctica, sobre todo en la definición de lenguajes de programación, en la formalización del concepto de análisis de gramática, simplificación de la traducción de lenguajes de programación y en otras aplicaciones del procesamiento de cadenas. Como ejemplo, las gramáticas libres de contexto son útiles para describir expresiones aritméticas que tengan un anidamiento arbitrario de paréntesis balanceados y estructuras de bloque en los lenguajes de programación (es decir, el par begin y end como paréntesis balanceados). Como se define en [59], una gramática libre de contexto es un conjunto de variables (también conocidas como categorías no terminales o sintácticas) cada una de las cuales representa un lenguaje. Los lenguajes representados por las variables se describen de manera recursiva en términos de las mismas variables y de símbolos primitivos llamados terminales. Las reglas que relacionan a las variables se conocen como producciones. Una producción típica establece que el lenguaje asociado con una variable dada contiene cadenas que se forman mediante la concatenación de cadenas tomadas de los lenguajes representados por otras ciertas variables, posiblemente junto con algunos terminales.

La motivación original para el desarrollo de las gramáticas libres de contexto fue la necesidad de describir los lenguajes naturales. Podemos escribir reglas como las definidas en el Ejemplo 2.

<oración>	→ <frase sustantiva><frase verbal>	(a)
<frase sustantiva>	→ <frase sustantiva><adjetivo>	(b)
<frase sustantiva>	→ <sustantivo>	(c)
<sustantivo>	→ niño	(d)
<adjetivo>	→ pequeño	(e)

**Ejemplo 2: Reglas de producción para describir oraciones.**

En donde las categorías sintácticas (o variables) se describen entre los símbolos < y > y las terminales se denotan con palabras sin paréntesis como “niño” y “pequeño”.

El significado en el Ejemplo 2(a) es que una manera de formar una oración (una cadena del lenguaje de las categorías sintácticas <oración>) consiste en tomar una frase sustantiva seguida de una frase verbal. El significado del Ejemplo 2(d) es que la cadena que consiste en el símbolo terminal “niño” está en el lenguaje de la categoría sintáctica <sustantivo>. Notemos que “niño” es una cadena terminal y no una cadena de símbolos.

Por ciertas razones las gramáticas libres de contexto, en general, no se consideran adecuadas para la descripción de lenguajes naturales. En cambio, mediante la introducción de la notación conocida como *Backus-Naur Form* (BNF) que aporta pequeños cambios y abreviaturas, el uso de gramáticas libres de contexto simplificó grandemente la definición de lenguajes de programación.

### 5.1.1. Formalización de las gramáticas libres de contexto

Una gramática libre de contexto (CFG o simplemente *gramática*) se define por  $G=(V,T,P,S)$  en donde  $V$  y  $T$  son conjuntos finitos de *variables* y *terminales* respectivamente. Suponemos que  $V$  y  $T$  son disjuntos.  $P$  es un conjunto finito de producciones, donde cada producción tiene la forma  $A \rightarrow \alpha$  en donde  $A$  es una variable y  $\alpha$  es una cadena de símbolos tomados de  $(V \cup T)^*$ . Finalmente  $S$  es una variable especial conocida como el *símbolo inicial* (o *start symbol*). El Ejemplo 3 muestra la definición de una gramática.

$G = (\{E\}, \{+, *, (, ), id\}, P, E)$ , donde $P$ esta compuesto por las siguientes producciones $E \rightarrow E + E$ (a) $E \rightarrow E * E$ (b) $E \rightarrow (E)$ (c) $E \rightarrow id$ (d)
--

Ejemplo 3: Definición de una gramática.

Antes de continuar estableceremos algunas convenciones en cuanto a los símbolos utilizados para la definición de gramáticas, a saber:

1. Las letras mayúsculas  $A, B, C, D, E$  y  $S$  representan variables;  $S$  es el símbolo inicial a menos que se indique lo contrario.
2. Las letras minúsculas  $a, b, c, d, e$ , los dígitos y las cadenas en negritas son terminales.
3. Las letras mayúsculas  $X, Y$  y  $Z$  representan símbolos que pueden ser terminales o variables.
4. Las letras minúsculas  $u, v, w, x, y$  y  $z$  denotan cadenas de terminales.
5. Las letras griegas minúsculas  $\alpha, \beta$  y  $\gamma$  denotan cadenas de variables y terminales.
6. La barra vertical  $|$  es una simplificación para producciones con el mismo símbolo en su lado izquierdo. Aplicado al Ejemplo 3 obtenemos:  
 $E \rightarrow E + E | E * E | (E) | id$ .

### 5.1.2. Reescritura de gramáticas formales

Toda gramática tiene asociado un lenguaje, es decir, el conjunto de cadenas de terminales que se pueden formar aplicando producciones desde el símbolo inicial hasta obtener una secuencia de símbolos terminales, proceso que se denomina *derivación* o *reescritura*.

Sea una gramática  $G=(V,T,S,P)$  la derivación comienza a partir del símbolo definido como inicial ( $S$ ), es decir, de una producción de  $P$  que tenga en su lado izquierdo dicho símbolo.

El lado derecho de la producción pasa a ser la *cadena de reescritura*, y como paso siguiente se aplican otras producciones del conjunto  $P$  que posean en su lado izquierdo variables presentes en dicha cadena de reescritura. Entonces la aplicación de otra producción  $p$  consiste en reemplazar en la cadena la variable que coincide con el lado

izquierdo (*predecesor*) de  $p$  por el lado derecho (*sucesor*) de  $p$ . De esta manera se obtiene una nueva cadena de reescritura. El proceso continúa hasta lograr que en la cadena de reescritura sólo haya símbolos del conjunto  $T$ . Así se obtiene una palabra del lenguaje definido por  $G$ .

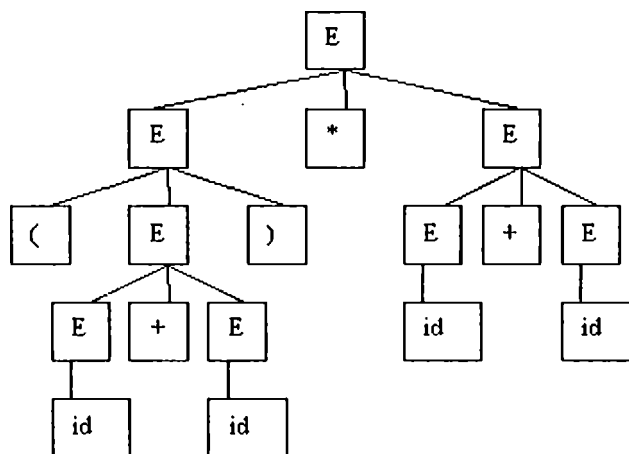
El Ejemplo 4 muestra una posible derivación de una cadena perteneciente al lenguaje descrito por la gramática del Ejemplo 3.

Iniciando con la aplicación de la producción (b) tenemos la siguiente cadena de reescritura: $E * E$	
Aplicando (c) a la primer $E$ :	$(E) * E$
Aplicando (a) a la primer $E$ :	$(E + E) * E$
Aplicando (a) a la tercer $E$ :	$(E + E) * E + E$
Aplicando (d) a todas las variables $E$ :	$(id + id) * id + id$

**Ejemplo 4: Derivación de una cadena del lenguaje especificado por la gramática del Ejemplo 3**

Como se puede ver, al estar  $G$  definida de manera recursiva hay infinitas posibilidades de derivación. El lenguaje de la gramática, es decir, el conjunto de todas las posibles cadenas de símbolos de  $T$  que una gramática  $G$  puede generar se denomina como  $L(G)$ , donde  $L(G) \subseteq T^*$ .

Las derivaciones pueden también representarse mediante lo que se conoce como *árbol de derivación*. Es útil representar las derivaciones mediante éstos árboles, que superponen una estructura sobre las palabras de un lenguaje y son de utilidad en aplicaciones tales como la compilación de lenguajes de programación. Los vértices de un árbol de derivación están etiquetados como símbolos terminales o variables de la gramática. Si un vértice interior  $n$  tiene etiqueta  $A$  y los hijos están etiquetados con  $X_1, X_2, \dots, X_n$  de izquierda a derecha, entonces,  $A \rightarrow X_1 X_2 X_n$  debe ser una producción. La Figura 42 muestra un árbol de derivación para el Ejemplo 4.



**Figura 42: Árbol de derivación. Recorriendo de izquierda a derecha las hojas de este árbol de derivación se puede obtener la cadena de símbolos terminales generada por la derivación correspondiente al Ejemplo 4.**

### 5.1.3. Gramáticas sensibles al contexto

Hasta ahora hemos presentado a las gramáticas libres de contexto, las cuales no definen *contextos* en sus predecesores. Los contextos se definen a ambos lados del sucesor por lo que en el lado izquierdo de las producciones de una gramática sensible al contexto se pueden identificar ahora tres componentes: el *contexto izquierdo*, el *predecesor estricto* (el cual siempre es una variable) y el *contexto derecho*.

Los contextos están formados por cadenas de símbolos pertenecientes a  $V \cup T$  y su semántica consiste en que en la derivación no sólo debe coincidir el predecesor estricto con alguna variable de la cadena de reescritura sino que, para poder realizar el reemplazo, deben coincidir los símbolos que aparecen a izquierda y a derecha con los definidos en los contextos correspondientes.

La utilidad del uso de contextos está en la posibilidad de preservar ciertos formatos en las cadenas de un lenguaje generado por una gramática.

En el Ejemplo 5 se ilustra una gramática sensible al contexto y como se realiza la derivación cuando se tienen contextos definidos.

Sea  $G = (V, T, S, P)$  con  $V$  y  $T$  iguales al Ejemplo 3 pero  $P$  incluye la producción:

$\langle E \rangle \rightarrow E-E \text{ (e)}$

Esta producción define en su predecesor el contexto izquierdo con el terminal  $($ , el predecesor estricto  $E$ , y el contexto derecho con el terminal  $)$ .

De esta manera nos disponemos a mostrar una posible derivación aplicando la producción (e)  
Iniciando desde la producción (b) tenemos la siguiente cadena de reescritura:  $E * E$

Aplicando (c) a la primer  $E$ :  $(E) * E$

Ahora se puede aplicar (e) a  $(E)$ :  $(E-E) * E$

Finalmente aplicamos (d) a todas las  $E$ :  $(id - id) * id$

**Ejemplo 5: Derivación con producciones sensibles al contexto**

El lector debe notar que la aplicación de la regla (e) en el Ejemplo 5 permite preservar una forma arbitraria: “una resta debe aparecer sólo entre paréntesis”.

## 5.2. Sistemas L

En 1968, Aristid Lindenmayer introdujo un formalismo llamado Sistema L o *L-System* [81] para modelar el desarrollo de organismos multicelulares simples en términos de división, crecimiento y muerte de células individuales ([81] y [80]). Este formalismo estuvo muy relacionado a autómatas abstractos y lenguajes formales, y atrajo el interés inmediato de científicos de la computación teórica [113]. El vigoroso desarrollo de la teoría matemática de los Sistemas L ([115], [55] y [112]) prosiguió con la aplicación de la teoría al modelado de plantas ([30], [31], [32], [63], [83], [105] y [104]).

El rango de aplicaciones de Sistemas L ha sido extendido a plantas de gran tamaño y complejidad de ramificación en sus estructuras, en particular plantas sin flores ([32] y [33]), descritas como una configuración de módulos en el espacio. En el contexto de los Sistemas L, el término *módulo* denota cualquier unidad discreta de construcción que se repite a medida que el organismo se desarrolla. Estos realizan una *reescritura en paralelo*, lo que significa que en cada paso se aplican todas las reglas simultáneamente, pudiendo reemplazar más de un módulo en cada derivación. Esto resulta conveniente para el objetivo inicial de estos sistemas, ya que la esencia del desarrollo a nivel modular puede ser convenientemente capturada por un sistema *paralelo de reescritura* que reemplaza un módulo *padre, madre o ancestro* por configuraciones de *hijos, hijas o módulos descendientes*. De esta manera, la reescritura en paralelo se ajusta de forma adecuada al modelado del desarrollo biológico, ya que éste toma lugar en varias partes del organismo simultáneamente.

Para el caso de las plantas, un módulo puede representar un brote, una hoja o una rama (ver Figura 43) [9], [52] y [128]. El objetivo de modelar de esta manera la formación de una planta es describirla como la integración del desarrollo de sus unidades individuales. Por ejemplo, un brote dará lugar a una rama con hojas y otro brote o bien a una hoja sola. Un Sistema L para modelar el desarrollo de esta planta se muestra en el Ejemplo 6.

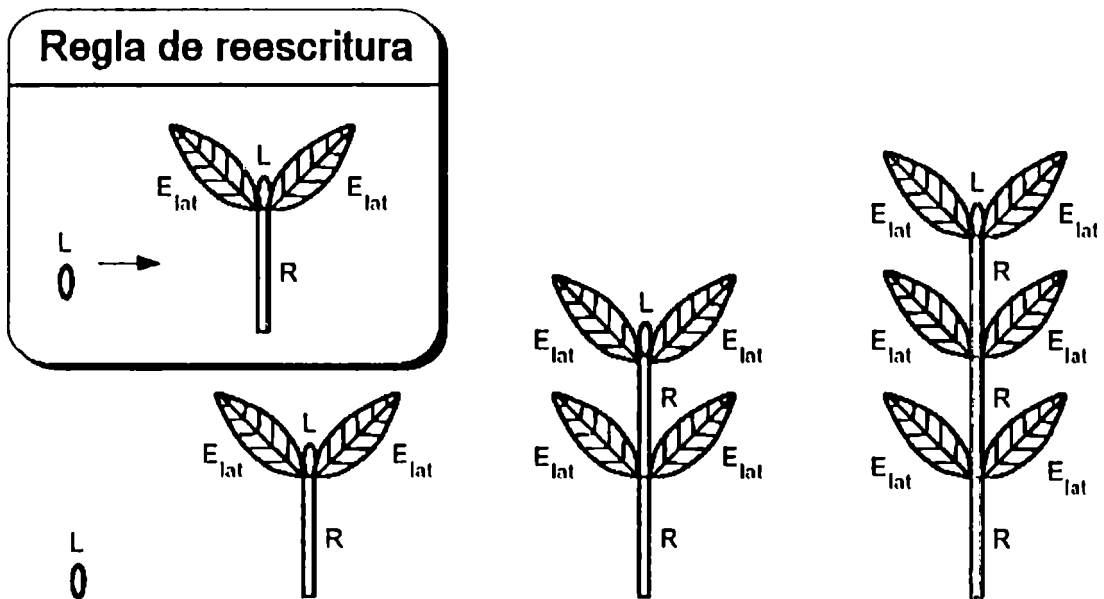


Figura 43: Ejemplo del desarrollo modular de una planta.

Como se puede notar, en cada derivación éste producirá al final de la cadena de reescritura un módulo L (que representa un brote), por lo que el crecimiento de la planta continuará indefinidamente. Debido a esto, el modelo no coincide con lo que ocurre en la realidad. Para subsanar este problema, es necesario extender la definición de un Sistema L.

$$\begin{array}{l} \omega : L \\ L \rightarrow R E_{lat} E_{lat} L \end{array}$$

Ejemplo 6: Sistema L simple para modelar el desarrollo de una planta.

### 5.2.1. Sistemas L paramétricos

Los Sistemas L paramétricos extienden el concepto básico de los Sistemas L asignando atributos numéricos a los símbolos del sistema. Esta extensión fue implementada por primera vez en la década de 1970 en el primer simulador basado en Sistemas L, llamado CELIA (acrónimo de Cellular Linear Iterative Array simulator) ([5], [56], [55] y [76]), utilizado más para programación que para construcciones teóricas. En esta tesis nos servimos como guía de la formalización introducida en [107] y [104] (ver también [47] y [106]).

Formalmente pueden definirse como una tupla ordenada de cuatro elementos  $G = \langle V, \Sigma, \omega, P \rangle$  donde:

- $V$  es el *alfabeto del sistema*, un conjunto finito de letras utilizadas para etiquetar los módulos.
- $\Sigma$  es el conjunto de *parámetros formales* que los módulos definen. Estos parámetros se reemplazan en la reescritura por su valor real.
- $\omega \in (V \times R^*)^+$  es un conjunto no vacío de palabras con parámetros llamados *axiomas*, los cuales representan un punto de partida para la reescritura.
- $P \subset (V \times \Sigma^*) \times C(\Sigma) \times (V \times E(\Sigma))^*$  es un *conjunto de producciones* finito, las cuales son utilizadas para el proceso de reescritura aplicándolas a la estructura de reescritura. El conjunto  $V \times \Sigma^*$  representa a todos los *predecesores* posibles de una producción, mientras que  $C(\Sigma) \times (V \times E(\Sigma))^*$  representa a todos los posibles *sucesores*. El símbolo  $C(\Sigma)$  representa al conjunto de las expresiones lógicas que utilizan parámetros de  $\Sigma$ . El símbolo  $E(\Sigma)$  es el conjunto de las expresiones numéricas que utilizan parámetros de  $\Sigma$ .

Los Sistemas L paramétricos operan con *palabras paramétricas*, las cuales son cadenas de módulos que consisten en *letras con parámetros* asociados. Las letras pertenecen al *alfabeto*  $V$ , y los parámetros pertenecen al conjunto de los *números reales*  $R$ . Un módulo con letra  $A \in V$  y parámetros  $a_1, a_2, \dots, a_n \in R^*$  se denota como  $A(a_1, a_2, \dots, a_n)$ . Cada módulo pertenece al conjunto  $M = V \times R^*$ , donde  $R^*$  es el conjunto de todas las secuencias finitas de parámetros. El conjunto de todas las cadenas de módulos y el conjunto de todas las cadenas no vacías se denominan como  $M^* = (V \times R^*)^*$  y  $M^+ = (V \times R^*)^+$ , respectivamente.

Los parámetros de valor real que aparecen en las palabras corresponden a *parámetros formales* usados en la especificación de las producciones de los Sistemas L. Ambos tipos de expresiones consisten en parámetros formales y constantes numéricas, combinadas usando operadores aritméticos, relacionales, lógicos y paréntesis. Los símbolos de operación y las reglas para la construcción de expresiones sintácticamente correctas son las mismas que en un lenguaje de programación como C [67]. Una sentencia lógica especificada como el string vacío, se asume que tiene valor verdadero. El conjunto de todas las expresiones con parámetros lógicos y aritméticos correctamente construidas de  $\Sigma$  son denotadas como  $C(\Sigma)$  y  $E(\Sigma)$ .

Los símbolos ‘:’ y ‘→’ son usados para separar los tres componentes de una producción: el *predecesor*, la *condición* y el *sucesor*. En el Ejemplo 7 se muestra como se escribe una regla de producción con estos símbolos.

Una producción condicional con predecesor  $A(t)$ , condición  $t > 5$  y sucesor  $B(t+1)CD(t^{0.5}, t-2)$  es escrita como:

$$A(t) : t > 5 \rightarrow B(t+1)CD(t^{0.5}, t-2)$$

**Ejemplo 7: Notación para definir una regla de producción condicional y paramétrica**

La reescritura se aplica como lo hemos explicado para el caso no paramétrico variando en criterio para determinar si una producción *coincide* o no con un módulo en la cadena de reescritura. Se produce dicha coincidencia si se encuentran las siguientes condiciones:

- la letra en el módulo y la letra en el predecesor de la producción son la misma
- el número de parámetros actuales en el módulo es igual al número de parámetros formales en el predecesor de la producción
- la condición se evalúa en *verdadero* si se verifica luego de que los parámetros formales hayan sido sustituidos por los valores de los parámetros actuales en la producción.

Otra diferencia se produce al efectuar el reemplazo del módulo a reescribir por el sucesor de la regla seleccionada ya que ahora éste contiene parámetros a los cuales debe asignárseles un valor. Los parámetros formales son sustituidos por los parámetros actuales de acuerdo a su posición. Por ejemplo, la producción del Ejemplo 7 coincide con un módulo  $A(9)$ , ya que la letra  $A$  en el módulo es la misma en el predecesor de la producción, hay un parámetro actual en el módulo  $A(9)$  y uno formal en el predecesor  $A(t)$ , y la expresión lógica  $t > 5$  es verdadera para  $t = 9$ . El resultado de la aplicación de esta producción es la palabra paramétrica  $B(10)CD(3,7)$ .

Si un módulo  $a$  produce una palabra paramétrica  $X$  como el resultado de la aplicación de una producción en un Sistema  $L G$ , escribimos que  $a \Rightarrow X$ . Dada una palabra paramétrica  $\mu = a_1, a_2, \dots, a_m$  decimos que la palabra  $v = X_1, X_2, \dots, X_m$  es derivada directamente de (o generada por)  $\mu$  y escribimos  $\mu \Rightarrow v$  si y solo si  $a_i \Rightarrow X_i$  para todo  $i = 1, 2, \dots, m$ . Una palabra paramétrica  $v$  es generada por  $G$  en una *derivación de tamaño  $n$*  si existe una secuencia de palabras  $\mu_0, \mu_1, \dots, \mu_n$  tal que  $\mu_0 = \omega$ ,  $\mu_n = v$  y  $\mu_0 \Rightarrow \mu_1 \Rightarrow \dots \Rightarrow \mu_n$ .

$\omega : B(2)A(4, 4)$   
 $p_1 : A(x, y) : y \leq 3 \rightarrow A(x*2, x+y)$   
 $p_2 : A(x, y) : y > 3 \rightarrow B(x)A(x/y, 0)$   
 $p_3 : B(x) : x < 1 \rightarrow C$   
 $p_4 : B(x) : x \geq 1 \rightarrow B(x-1)$

**Ejemplo 8: Definición del axioma y reglas reproducción para un Sistema  $L$  paramétrico.**



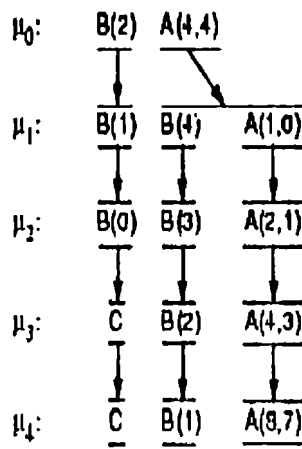


Figura 44: La secuencia inicial de cadenas generada por el Sistema L paramétrico especificado por el Ejemplo 8.

Se asume que un módulo se reemplaza a si mismo si no se encuentra una producción que coincida en el conjunto  $P$ . Las palabras obtenidas en los primeros pasos de derivación se muestran en la Figura 44.

Volvamos al primer ejemplo de modelado del desarrollo de plantas (Figura 43 y Ejemplo 6). Como habíamos mencionado en dicho ejemplo, la planta podía continuar su crecimiento indefinidamente. Con la extensión explicada en la Figura 45, es posible controlar dicho crecimiento. Como se puede ver, se obtiene una planta que termina con una única hoja en su último tramo.

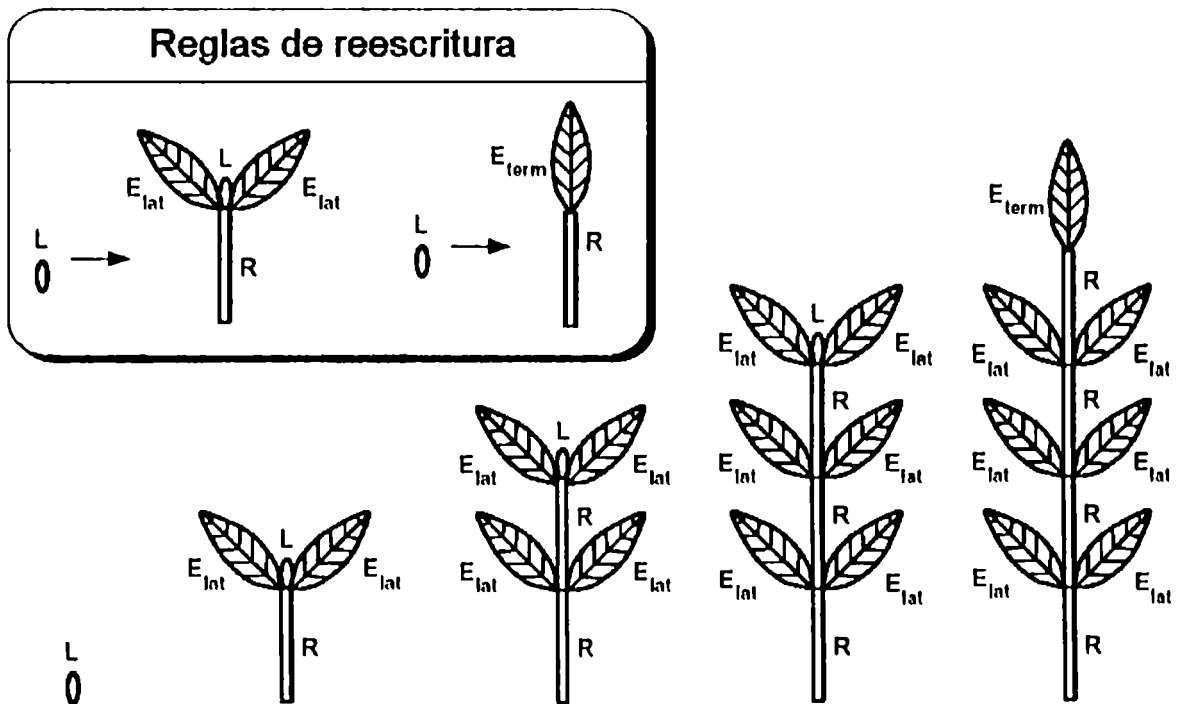


Figura 45: Otro ejemplo del desarrollo modular de una planta.

El Ejemplo 9 muestra un Sistema L paramétrico del cual se obtiene esta planta. El lector debe notar que la inclusión de parámetros permitió el diseño de una condición que hace aplicar un *caso de regla* distinto al que se aplicó anteriormente; posibilitando de esta forma la finalización del crecimiento de la planta.

$$\begin{array}{l} \omega : L(3) \\ L(n) : n > 0 \rightarrow R E_{1at} E_{1at} L(n-1) \\ L(n) : n = 0 \rightarrow R E_{term} \end{array}$$

**Ejemplo 9:** Sistema L paramétrico para modelar el desarrollo de la planta de la Figura 45.

### 5.2.2. Sistemas L sensibles al contexto

Las producciones en Sistemas L que hemos observado hasta el momento son libres de contexto, es decir, se aplican sin importar el contexto en el cual aparece el predecesor. Una extensión sensitiva al contexto es necesaria para modelar intercambio de información entre módulos vecinos. Cuando describimos de las gramáticas formales, brindamos la noción de *sensibilidad al contexto*. La diferencia con los Sistemas L radica en la posible presencia de los parámetros en los contextos. Cada componente del predecesor de la producción (el *contexto izquierdo*, el *predecesor estricto* y el *contexto derecho*) es una palabra paramétrica con letras del alfabeto  $V$  y parámetros formales del conjunto  $\Sigma$ . Cualquier parámetro formal de la misma puede aparecer en la condición y el sucesor de la producción.

$$A(x) < B(y) > C(z) : x + y + z > 10 \rightarrow E((x + y)/2) F((y + z)/2)$$

**Ejemplo 10:** Definición de una producción sensible al contexto.

El contexto izquierdo es separado del predecesor estricto por el símbolo  $<$ . Similarmente, el predecesor estricto es separado del contexto derecho por el símbolo  $>$ . El Ejemplo 11 muestra una palabra paramétrica donde la producción del Ejemplo 10 puede ser aplicada.

$$\dots A(4)B(5)C(6) \dots$$

**Ejemplo 11:** Palabra paramétrica donde puede aplicarse la regla definida en el Ejemplo 10. La regla reescribe el módulo B(5).

Esto se debe a que coincide la secuencia de letras A, B, C en la producción del Ejemplo 10 con la palabra paramétrica del Ejemplo 11, y a que existe correspondencia en las cantidades de parámetros formales y parámetros actuales. Reemplazando los parámetros formales por los actuales puede verse que la condición  $4 + 5 + 6 > 10$  es verdadera. Como resultado de la aplicación de la producción, el módulo B(5) será reemplazado por un par de módulos E(4.5)F(5.5). Naturalmente los módulos A(4) y C(6) serán reemplazados por otras producciones en el mismo paso de derivación.

Las producciones en los Sistemas L sensibles al contexto lo aplican en ambos lados del predecesor estricto. Hay un caso especial en el cual los contextos aparecen solamente de un lado del predecesor estricto de las producciones.

### 5.2.3. Sistemas L estocásticos

Si más de una producción en  $P$  coincide con un módulo, se necesita un mecanismo adicional para seleccionar que producción se aplicará a dicho módulo. Un Sistema L paramétrico  $G = \langle V, \Sigma, \omega, P \rangle$  se llama *determinístico* si y solo si por cada módulo  $A(t_1, t_2, \dots, t_n) \in V \times R^*$  el conjunto de producción incluye exactamente una producción que coincide.

En los Sistemas L *estocásticos*, la decisión se basa en factores aleatorios. En el caso más extensivo, una producción tiene el formato:

$$id : lc \langle pred \rangle rc : cond \rightarrow succ : \pi$$

Donde  $id$  es el identificador de la producción (etiqueta),  $lc$ ,  $pred$  y  $rc$  son el contexto izquierdo, predecesor estricto y contexto derecho respectivamente;  $cond$  es la condición,  $succ$  es el sucesor y  $\pi$  es una expresión aritmética que retorna un número no negativo llamado *factor de probabilidad*. Si  $P' \subseteq P$  es el conjunto de producciones que coinciden con un módulo dado  $A(t_1, t_2, \dots, t_n) \in V \times R^*$  en la cadena de reescritura, entonces la probabilidad  $prob(pk)$  de aplicar una producción particular  $pk \in P'$  queda definida por la Ecuación 10.

$$prob(pk) = \frac{\pi(pk)}{\sum_{p_i \in P'} \pi(p_i)}$$

**Ecuación 10: Probabilidad de aplicar una producción en un Sistema L estocástico.**

En general esta probabilidad no es una constante asociada con una producción pero puede depender de los valores de los parámetros en el módulo reescrito y su contexto.

$\omega : A(1)B(3)A(5)$ $p1 : A(x) \rightarrow A(x+1) : 2$ $p2 : A(x) \rightarrow B(x-1) : 3$ $p3 : A(x) : x > 3 \rightarrow C(x) : x$ $p4 : A(x) \langle B(y) \rangle A(z) : y < 4 \rightarrow B(x+z)A(y)$
---

**Ejemplo 12: Sistema L sensible al contexto estocástico y paramétrico**

En el Ejemplo 12 las producciones  $p1$ ,  $p2$  y  $p3$  reemplazan un módulo  $A(x)$  por  $A(x+1)$ ,  $B(x-1)$  o  $C(x)$ . Si el valor del parámetro  $x$  es menor o igual a 3, sólo las dos primeras producciones coinciden con  $A(x)$ . Las probabilidades de aplicar cada producción son:  $prob(p1) = 2/(2+3) = 0.4$  y  $prob(p2) = 3/(2+3) = 0.6$ . Si el parámetro  $x$  es mayor que 3, la producción  $p3$  también coincide con el módulo  $A(x)$ , y la probabilidad de aplicar cada producción depende del valor de  $x$ . Por ejemplo, si  $x$  es igual a 5, esas probabilidades son:  $prob(p1) = 2/(2+3+5) = 0.2$ ,  $prob(p2) = 3/(2+3+5) = 0.3$  y  $prob(p3) = 5/(2+3) = 0.5$ . La producción sensible al contexto  $p4$  reemplaza un módulo  $B(y)$  con contexto izquierdo

$A(x)$  y contexto derecho  $A(z)$  por un par de módulos  $B(x+z)A(y)$ . La aplicación de esta producción está “guardada” por la condición  $y < 4$ . Tomando todos estos factores en cuenta, en el Ejemplo 13 se muestra como puede ser la forma del primer paso de derivación para el Sistema L definido en el Ejemplo 12.

$$A(1)B(3)A(5) \Rightarrow A(2)B(6)A(3)C(5)$$

**Ejemplo 13:** Posible primer paso de derivación para el Sistema L del Ejemplo 12.

Se asume que, como resultado de una selección aleatoria, la producción  $p1$  fue aplicada al módulo  $A(1)$ , y la producción  $p3$  al módulo  $A(5)$ . La producción  $p4$  fue aplicada al módulo  $B(3)$ , ya que cumple con los contextos izquierdo y derecho, y la condición  $3 < 4$  es verdadera.

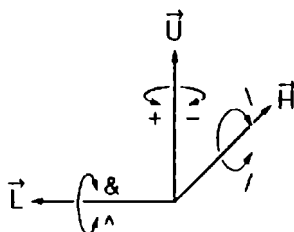
Con esto describimos de una manera detallada los conceptos de los Sistemas L necesarios para comprender el trabajo desarrollado. Si el lector lo desea puede remitirse a los artículos [78], [81], [79], [82], [84] y [85]; y libros [55], [105], [104], [112] y [115] donde encontrará más nociones acerca de la teoría de estos sistemas.

### 5.3. Aplicaciones de los Sistemas L

Las aplicaciones de los Sistemas L obtuvieron mayor atención después de 1984, cuando Smith introdujo técnicas gráficas al estado del arte para visualizar estructuras y procesos modelados. En ese entonces también se centró la atención en el fenómeno de la amplificación de la base de datos, o la posibilidad de generar estructuras complejas de conjuntos de datos compactos, lo cual es inherente en Sistemas L y forma la piedra fundamental de las aplicaciones de estos sistemas a la síntesis de imágenes.

#### 5.3.1. Interpretación de una tortuga estilo LOGO

Las cadenas generadas por Sistemas L pueden ser interpretadas geoméricamente de muchas maneras diferentes ([107] y [104]). Describiremos *la interpretación de tortuga* tipo LOGO de los Sistemas L paramétricos, introducida por Szilard y Quinton [122], y extendida por Prusinkiewicks y Hanan([48],[47]). El lector podrá encontrar además la exposición de un tutorial en [104] y resultados subsecuentes en [47].



**Figura 46:** Controles de la tortuga en tres dimensiones

$$\vec{H} \times \vec{L} = \vec{U}$$

**Ecuación 11:** Relación entre los vectores de orientación de la tortuga tipo LOGO.

$$[\vec{H}' \vec{L}' \vec{U}'] = [\vec{H} \vec{L} \vec{U}]R$$

**Ecuación 12: Ecuación de rotación de la tortuga tipo LOGO. La matriz R tiene dimensiones de 3x3 y consiste en una matriz de rotación [27].**

$$R_U(\alpha) = \begin{bmatrix} \cos \alpha & \text{sen} \alpha & 0 \\ -\text{sen} \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$R_L(\alpha) = \begin{bmatrix} \cos \alpha & 0 & -\text{sen} \alpha \\ 0 & 1 & 0 \\ \text{sen} \alpha & 0 & \cos \alpha \end{bmatrix},$$

$$R_H(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\text{sen} \alpha \\ 0 & \text{sen} \alpha & \cos \alpha \end{bmatrix}.$$

**Ecuación 13: Matrices de rotación en un para los tres vectores de la tortuga tipo LOGO.**

Después que una cadena es generada por un Sistema L, se recorre secuencialmente desde izquierda a derecha, y los símbolos consecutivos son interpretados como comandos que manejan una tortuga estilo LOGO en tres dimensiones. La tortuga es representada por su estado, el cual consiste en la *posición* y la *orientación* en el sistema de coordenadas cartesianas, como también varios valores de atributos, tal como *color* actual y *ancho de línea*. La posición es definida por un vector P, y la orientación es definida por tres vectores  $\vec{H}, \vec{L}, \vec{U}$  indicando la *cabeza* de la tortuga y las direcciones hacia la *derecha* y hacia *arriba* respectivamente (Figura 46). Estos vectores tienen longitud de una unidad, son perpendiculares uno de otro y satisfacen la Ecuación 11. Las rotaciones de la tortuga son expresadas por la Ecuación 12.

Específicamente, las rotaciones en un ángulo  $\alpha$  sobre los vectores  $\vec{H}, \vec{L}, \vec{U}$  son representadas por las matrices descriptas en la Ecuación 13.

Los cambios en el estado de la tortuga son causados por la interpretación de símbolos específicos, cada uno de los cuales puede estar seguido de parámetros. Si uno o más parámetros son presentados, el valor del primer parámetro afecta el estado de la tortuga.

Comando	Descripción
$F(s)$	Mueve hacia delante un paso de longitud $s$ dibujando una línea desde la posición de origen hasta la de destino.
$f(s)$	Mueve hacia delante un paso de longitud $s$ sin dibujar.
$@O(r)$	Dibuja una esfera de radio $r$ en la posición actual.
$+(\theta)$	Gira $\theta$ grados hacia la izquierda el ángulo respecto del eje U. La matriz de rotación es $R_U(\theta)$ .
$-(\theta)$	Gira $\theta$ grados hacia la derecha el ángulo respecto del eje U. La matriz de rotación es $R_U(-\theta)$ .
$\&(\theta)$	Gira $\theta$ grados hacia abajo el ángulo respecto del eje L. La matriz de rotación es $R_L(\theta)$ .
$\wedge(\theta)$	Gira $\theta$ grados hacia arriba el ángulo respecto del eje L. La matriz de rotación es $R_L(-\theta)$ .
$/( \theta)$	Gira $\theta$ grados hacia la izquierda el ángulo respecto del eje L. La matriz de rotación es $R_{II}(\theta)$ .
$\backslash(\theta)$	Gira $\theta$ grados hacia la derecha el ángulo respecto del eje L. La matriz de rotación es $R_{II}(-\theta)$ .
	Gira $180^\circ$ respecto del eje U. Esto equivale a $+(180)$ o $-(180)$
[	Coloca el estado actual de la tortuga (posición, orientación y atributos de dibujo) en una pila de estados (estructura LIFO).
]	Saca el primer estado al tope de la pila y lo configura como el estado actual de la tortuga. No dibuja aunque la tortuga haya cambiado su posición.
{	Comienza a guardar las posiciones subsecuentes de la tortuga como los vértices de un polígono que puede ser rellenado.
}	Rellena el polígono guardado.
~	Dibuja la superficie identificada por el símbolo que sigue después de ~ en la posición y orientación actual de la tortuga
$\#(w)$	Configura el ancho de línea como $w$ , o incrementa el ancho actual si $w$ no se especifica.
$!(w)$	Configura el ancho de línea como $w$ , o decrementa el ancho actual si $w$ no se especifica.
$;(n)$	Configura el índice del mapa de colores como $n$ o incrementa el valor del índice actual si $n$ no se especifica.
$,(n)$	Configura el índice del mapa de colores como $n$ o decrementa el valor del índice actual si $n$ no se especifica.

Tabla 5: Comandos utilizados por la tortuga tipo LOGO.

Si el símbolo no es seguido por ningún parámetro, se usarán valores por defectos especificados fuera del Sistema L. Definiendo un apropiado conjunto de comandos para la tortuga, como los que se describen en la Tabla 5, se pueden diseñar Sistemas L que hagan dibujar en un espacio de tres dimensiones.

El lector deberá tener en cuenta los conceptos expuestos en esta sección como así también una noción de los comandos de dibujo presentados ya que todos los ejemplos que mostraremos desde este punto hasta el final del capítulo se basarán en lo descrito en esta sección.

### 5.3.2. Modelado de plantas

Como mencionamos anteriormente, los Sistemas L fueron ideados para el modelado del desarrollo de organismos multicelulares. La escritura en paralelo que estos sistemas realizan se adapta a dicho mecanismo ya que un organismo desarrolla varios de sus componentes a la vez.

En este contexto un paso de derivación corresponde al paso del tiempo en un cierto intervalo. Una secuencia de estructuras obtenidas en pasos consecutivos de derivación de una *estructura inicial* predefinida o *axioma* es llamada *secuencia de desarrollo*. Ésta puede ser visto como el resultado de una *simulación discreta del paso del tiempo* dentro del contexto del desarrollo biológico.

Por ejemplo, la Figura 47 ilustra el desarrollo de una hoja estilizada incluyendo dos tipos de módulos, los *ápices* (representados por líneas delgadas) y los *nodos internos* (líneas gruesas). Un ápice ofrece una estructura que consiste en dos nodos internos, dos ápices laterales y una replica del ápice principal. Un nodo interno se alarga mediante factor de escalado constante. Contrariamente a la simplicidad de éstas reglas, se obtiene una estructura de ramificación intrincada de un único ápice en un número de pasos de derivación. De esta manera con una definición relativamente simple de un conjunto de reglas de reescritura se puede obtener una secuencia de desarrollo que representa estructuras con una trama de información más compleja y extensa.

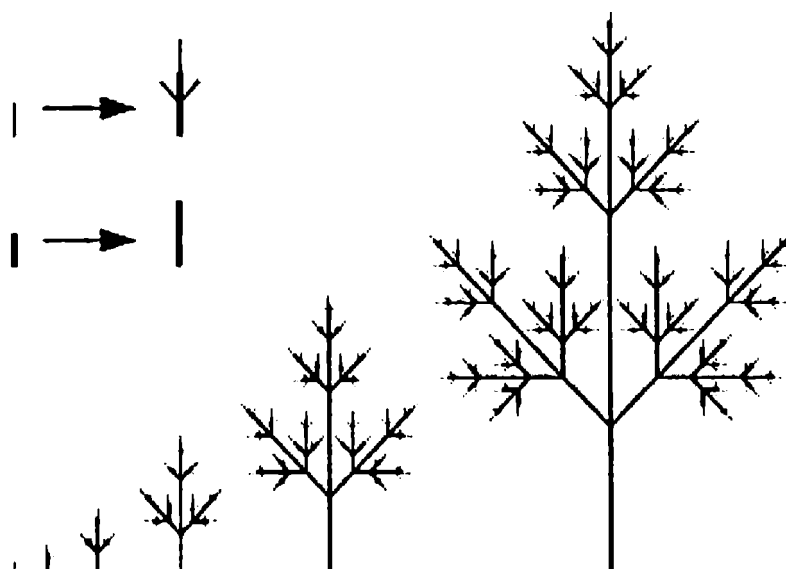


Figura 47: Modelo de desarrollo de una hoja compuesta, modelado como una configuración de ápices y nodos internos

Presentaremos ahora un Sistema L definido en el Ejemplo 14; el cual realiza una simulación de desarrollo de la estructura mostrada en la figura.

$\begin{aligned} \omega & : !(1)F(1) \\ p1 & : F(s) \rightarrow G(s)[-!(1)F(s)][+!(1)F(s)]G(s)!(1)F(s) \\ p2 & : G(s) \rightarrow G(2 * s) \\ p3 & : !(w) \rightarrow !(3) \end{aligned}$
---

**Ejemplo 14: Sistema L que genera una secuencia de desarrollo de un modelo estilizado compuesto de hojas que genera las formas de la Figura 47.**

La estructura se construye a partir de dos tipos de módulos, ápices  $F$  (representados por líneas delgadas) y nodos internos  $G$  (líneas gruesas). En ambos casos, el parámetro  $s$  determina el tamaño de la línea que representa el módulo. Un ápice construye una estructura que consiste en dos nodos internos, dos ápices laterales y una réplica del ápice principal (producción  $p1$ ). Un nodo interno se agranda mediante un factor de escala constante (producción  $p2$ ). La producción  $p3$  es utilizada para dibujar las líneas que representan a los nodos internos más anchos (3 unidades de ancho) que las que representan los ápices (una unidad). El ángulo de las ramas asociado con los símbolos  $+$  y  $-$  se sctea a  $45^\circ$  mediante una variable global, externa al Sistema L.

### 5.3.3. Generación de fractales

Éste es un ejemplo que ilustra la operación de Sistemas L libres de contexto, determinísticos y paramétricos, con la interpretación de la tortuga y su aplicación a la generación de fractales. Se pueden encontrar muchos más ejemplos en [49], [104] y [107].

$\begin{aligned} \omega & : F(1)-(120)F(1)-(120)F(1) \\ p1 & : F(s) \rightarrow F(s/3)+(60)F(s/3)-(129)F(s/3)+(60)F(s/3) \end{aligned}$
---

**Ejemplo 15: Sistema L generador de la figura “copo de nieve”**

El Ejemplo 15 muestra un Sistema L que genera la bien conocida figura de “copo de nieve” ([88]y [127]) en el cual axioma  $F(1)-(120)F(1)-(120)F(1)$  dibuja un triángulo equilátero, con lados de longitud de una unidad. La producción  $p1$  reemplaza cada segmento de línea con una figura poligonal como se muestra en la Figura 48.



**Figura 48: Descripción gráfica del reemplazo que produce la producción  $p1$  del Sistema L del Ejemplo 15**

Las producciones para los símbolos  $+$  y  $-$  no están listadas, lo que significa que los módulos correspondientes serán reemplazados por si mismos durante la derivación. . El mismo efecto se obtendría con la inclusión de producciones como las del Ejemplo 16.

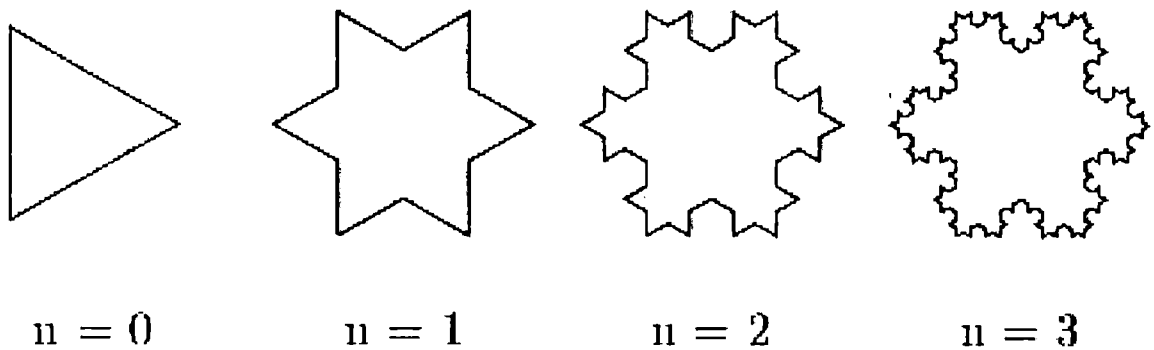


En cambio para el caso del símbolo  $F$  hay producciones listadas. Por lo tanto, cuando aparezca en la cadena de reescritura será reemplazado por el sucesor de la regla correspondiente. Una vez finalizada la reescritura se interpretará como el comando de dibujo que representa.

El axioma y las figuras obtenidas en los primeros tres pasos de derivación se muestran en la Figura 49.

$\begin{aligned} p2 &: +(a) \rightarrow +(a) \\ p3 &: -(a) \rightarrow -(a) \end{aligned}$
--

**Ejemplo 16: Producciones que no tienen efecto en la palabra de reescritura.**



**Figura 49: Axioma y primeros 3 pasos de derivación obtenidos del Sistema L del Ejemplo 15**

De esta manera podemos observar como de la aplicación repetida de una regla de producción simple se puede obtener una figura de cierta complejidad. Es decir que con esa simple regla se sintetiza la figura del copo de nieve.

### 5.3.4. Exploración del espacio de parámetros

Los Sistemas L paramétricos proveen un framework matemático conveniente para explorar el rango de formas que puede ser capturado por el mismo modelo estructural con atributos variables (constantes en las producciones). Tal exploración del espacio de parámetros es motivada en una de las más tempranas simulaciones en computadora de estructuras biológicas: los modelos de conchas de mar creado por Raup y Michelson ([109] y [110]) y los modelos de árboles propuestos por Honda [58] para estudiar factores que determinan la figura global del árbol. La exploración del espacio de parámetros puede revelar una inesperada riqueza de formas que pueden ser producidas por los modelos más simples. Por ejemplo, la Figura 50 muestra estructuras de nueve ramas generadas a partir del Sistema L descrito en el Ejemplo 17. El lector debe recordar que se utilizan los comandos de una tortuga tipo LOGO que explicamos anteriormente (ver Pág. 87).

La producción  $p1$  reemplaza un ápice  $A$  para introducir un nodo interno  $F$  y dos nuevos ápices  $A$ . Los ángulos  $\alpha_1$ ,  $\varphi_1$ ,  $\alpha_2$  y  $\varphi_2$  determinan la orientación de esos ápices con respecto al nodo interno subtendido. Los parámetros  $s$  y  $w$  especifican el tamaño del nodo interno y su ancho. Las constantes  $r1$  y  $r2$  determinan el decrecimiento gradual en el tamaño de un nodo interno que ocurre a medida que se atraviesa el árbol desde su base hacia los ápices. Las constantes  $w0$ ,  $q$  y  $e$  controlan el ancho de las ramas. El

tamaño inicial del tallo es especificado por  $w_0$  en el segundo parámetro del módulo  $A$  del axioma. Para  $e = 0.5$ , el área combinada de las ramas descendientes es igual al área de la rama madre, como postuló Leonardo da Vinci ([88], pagina 156 y [86], páginas 131-135). El valor  $q$  especifica las diferencias entre el ancho de las ramas descendientes originadas en el mismo vértice. Finalmente, la condición previene la formación de ramas con una longitud menor que el umbral min. En la Tabla 6 se muestra la relación de cada rama de la Figura 50 con el correspondiente juego de valores para las constantes del Sistema L del Ejemplo 17.

$\omega : A(100, w_0)$ $p1 : A(s, w) : s \geq \text{min} \rightarrow_i (w) F(s)$ $[+(\alpha 1) / (\varphi 1) A(s * r1, w * q^e)]$ $[+(\alpha 2) / (\varphi 2) A(s * r2, w * (1-q)^e)]$
--

Ejemplo 17: Sistema L determinístico y libre de contexto que genera las ramas de la Figura 50.

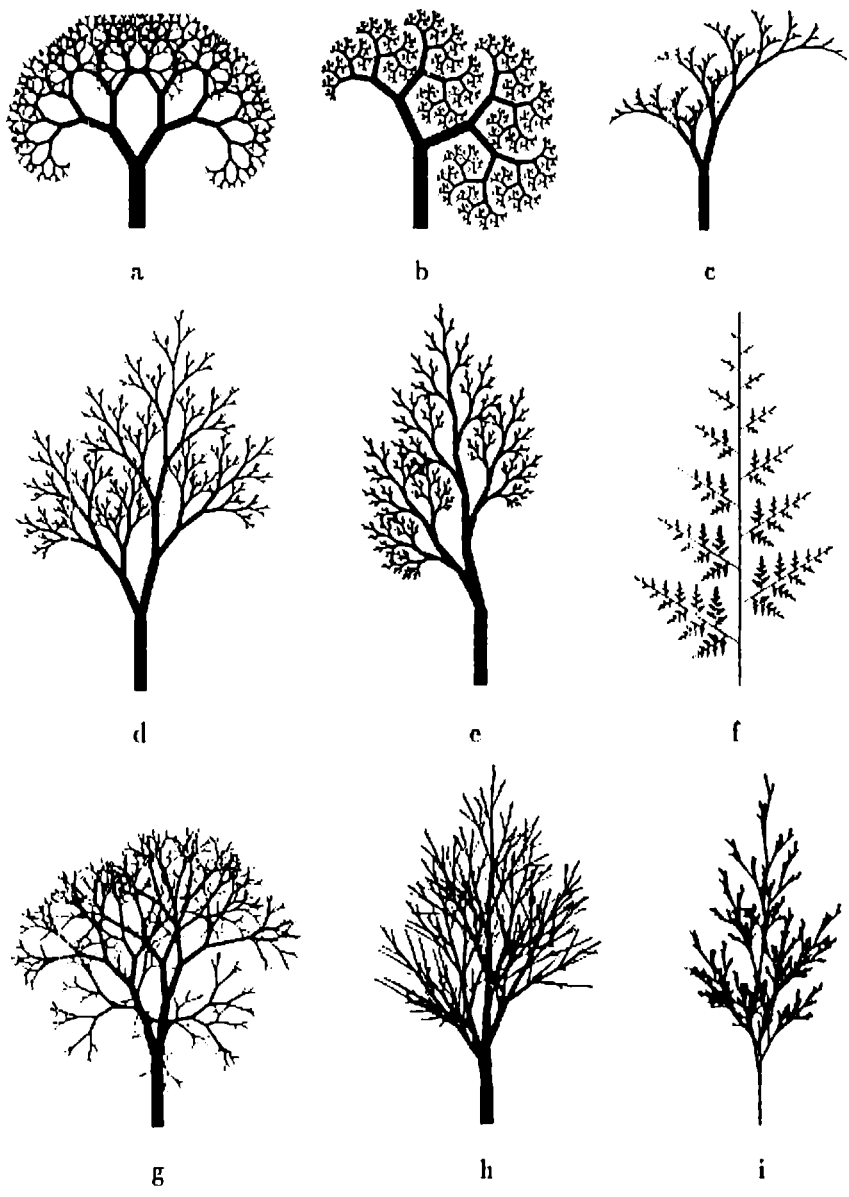


Figura 50: Estructuras de ejemplo generadas por un Sistema L paramétrico, determinístico y libre de contexto con diferentes valores de constantes.

Figura	$r_1$	$R_2$	$\alpha_1$	$\alpha_2$	$\varphi_1$	$\varphi_2$	$\omega_0$	$Q$	$e$	$min$	$n$
a	.75	.77	35	-35	0	0	30	.50	.40	0.0	10
b	.65	.71	27	-68	0	0	20	.53	.50	1.7	12
c	.50	.85	25	-15	180	0	20	.45	.50	0.5	9
d	.60	.85	25	-15	180	180	20	.45	.50	0.0	10
e	.58	.83	30	15	0	180	20	.40	.50	1.0	11
f	.92	.37	0	60	180	0	2	.50	.00	0.5	15
g	.80	.80	30	-30	137	137	30	.50	.50	0.0	10
h	.95	.75	5	-30	-90	90	40	.60	.45	25.0	12
i	.55	.95	-5	30	137	137	5	.40	.00	5.0	12

**Tabla 6: Ejemplos de valores de las constantes relacionados con la correspondiente representación gráfica en la Figura 50. La columna  $n$  indica el número de pasos de derivación. Las constantes están definidas en el Sistema L del Ejemplo 17.**

Hemos dado una noción de la teoría de los Sistemas L suficiente para el alcance de este trabajo y brindamos al lector un variado conjunto de aplicaciones de los mismos.

Terminamos así este capítulo esperando que, junto a los anteriores, se constituya en una fuente útil de información donde el lector encuentre lo necesario para hacer de este trabajo de grado una exposición autocontenida.

## 6. Neuroevolución de Sistemas de Reescritura

En este capítulo presentaremos el método de neuroevolución en el cual se centra el presente trabajo: *Neuroevolución de Sistemas de Reescritura* o NeSR [100] [101].

En las siguientes secciones detallaremos como representar redes neuronales utilizando Sistemas L y plantearemos distintas alternativas para ello. Luego, mostraremos como es posible evolucionar Sistemas L definiendo un conjunto de operadores genéticos apropiados. Terminaremos describiendo el modelo de Red Neuronal utilizado.

### 6.1. Introducción

Antes de introducirnos de lleno en el método neuroevolutivo que presenta este trabajo, asumiremos que el lector en este punto comprende los conceptos de Neuroevolución -Pág. 65-, Redes Neuronales -Pág. 3- y Sistemas L -Pág. 80-, puntales fundamentales del mismo. En líneas generales, el método combina Redes Neuronales, Evolución y Sistemas L de la siguiente manera:

Las Redes Neuronales son evolucionadas para resolver un problema dado mediante una variación de un Algoritmo Genético Simple. Este algoritmo cuenta con una población de Sistemas L, donde cada uno de ellos produce una Red Neuronal como resultado de su reescritura. El desempeño de cada Red Neuronal es evaluado en el problema que interesa resolver, obteniendo a partir de él un valor de bondad o fitness que es asignado al Sistema L que generó dicha red. De esta manera, los Sistemas L son utilizados para representar indirectamente a las Redes Neuronales evolucionadas. Los sistemas con mayor valor de fitness serán aquellos que generen las redes neuronales que mejor se comporten en el problema planteado.

### 6.2. Representación de las Redes Neuronales.

El método utiliza una codificación indirecta para representar las Redes Neuronales. Una codificación indirecta no incluye en el cromosoma todos los atributos que describen una solución sino que estos se generan o infieren a partir del cromosoma. Así se logra la síntesis de una solución a partir de una representación más compacta que si se utilizara codificación directa. En este caso, se utiliza un Sistema L para representar indirectamente a una Red Neuronal.

Para ello, se define un conjunto  $C$  de módulos terminales, llamados comandos, los cuales no aparecen en los predecesores de las reglas de los sistemas evolucionados, sólo en los sucesores. Esto provocará que dichos módulos no participen en el proceso de reescritura; solo podrán ser introducidos por la reescritura de alguno de los módulos no terminales. Como no existirá ninguna regla que los reescriba, permanecerán inalterados hasta el final del proceso de reescritura.

Luego, si se obtiene la cadena resultante de la reescritura de un sistema y se la recorre secuencialmente, tomando sólo los módulos que pertenezcan al conjunto  $C$  de los comandos, estos pueden ser interpretados como una serie de instrucciones para guiar el proceso de construcción de la red neuronal. Cada módulo representa una acción a realizar sobre la red neuronal que se está definiendo y los valores asociados a dicho módulo indican parámetros que alteran el significado de esa acción. Cualquier otro módulo no terminal que hubiera quedado luego de la reescritura no será tenido en cuenta durante la interpretación de la cadena obtenida. Al finalizar el proceso, se habrá obtenido una Red Neuronal producto de ese Sistema L.

Como hemos visto cuando describimos a los Sistemas L -Pág. 80-, existen distintos tipos de sistemas de reescritura, de acuerdo a si son determinísticos, si poseen parámetros o si son sensibles al contexto. Todos estos factores influyen en la forma en que estos sistemas se comportan. En NeSR, los Sistemas L utilizados son determinísticos, libres de contexto y paramétricos.

Se optó por Sistemas L determinísticos porque estos arrojan el mismo resultado para la misma cantidad de pasos de reescritura. Si el sistema fuera no determinístico, esto puede no ocurrir, pues algunas veces se aplicaría una regla de reescritura y algunas no. Esto rompería con la correspondencia entre el genotipo y el fenotipo, ya que un mismo cromosoma produciría diferentes redes neuronales.

La elección de Sistemas L que sean paramétricos se debe a que estos proveen de flexibilidad y variantes en las cadenas de reescritura producidas. Una misma regla con distintos parámetros puede generar diferentes reescrituras debido a la posibilidad de utilizar condiciones o guardas. Cabe resaltar que a diferencia de los Sistemas L no determinísticos, los sistemas determinísticos paramétricos generan siempre la misma respuesta para el mismo juego de parámetros.

Finalmente, se optó por Sistemas L libres de contexto porque posibilita intercambiar libremente módulos entre dos reglas que posean el mismo predecesor. Si ambas reglas tuvieran contextos diferentes, referencias a parámetros de los contextos imposibilitarían el intercambio de módulos, ya que algunos parámetros no existirían en las dos reglas.

### 6.3. Comandos de construcción de RNA

Hemos expuesto que los elementos del conjunto  $C$  de comandos son módulos terminales, donde a cada módulo terminal tiene una semántica asociada y puede pensarse en ellos como comandos que indican la manera de construir la red neuronal.

Por ejemplo, suponiendo que el conjunto  $C$  estuviera formado por una única definición de módulo de la forma  $LINK(N1, N2, W)$ , con el significado de “establecer una conexión desde la neurona  $N1$  hasta la neurona  $N2$  con peso  $W$ ”, éste alcanzaría para representar cualquier red neuronal describiendo con distintos parámetros todas las conexiones. La Figura 51 muestra un ejemplo de la obtención de una RNA a partir de un Sistema L que utiliza este comando (el cual aparece representado con el símbolo  $L$ ).

Otro conjunto de módulos puede ser utilizado para permitir la construcción de redes que cumplan determinadas restricciones (por ejemplo, que no sean recurrentes) o que posean una forma preestablecida.

Además, la semántica de los comandos puede ser extendida para que no realicen acciones que impliquen un cambio directo sobre la red neuronal, sino que alteren el estado del proceso de construcción. Por ejemplo, se puede definir un comando que marque una neurona como la neurona actual y cada vez que se ejecuta otro comando que realice una conexión, ésta se haga desde la neurona actual hacia una que se especifica en dicho comando. En este caso estamos ante comandos relativos a la neurona actual. En este contexto también puede existir una pila de estado y codificar en los comandos instrucciones para apilar y desapilar estados que pueden representar la neurona actual sobre la que se basarán los siguientes comandos. Esa es la idea de los comandos relativos. Intuitivamente podemos observar que un cambio al inicio de la secuencia puede construir una red completamente diferente a la que se construía antes de tal cambio.

En contrapartida, los comandos absolutos siempre generan lo mismo independientemente del orden en que se ejecutan. Los comandos *LINK* presentados en el ejemplo anterior son absolutos, ya que la ejecución de *LINK(0, 2, 1.5)* establecerá una conexión desde la neurona 0 hasta la 2 con peso 1.5, cualquiera fuera la posición en la que se ejecuta.

**Sistema-L**

**Inicio :** P1(1.5)

**P1(n0) :** (n0 >1) → L(3,4,n0) P2(5,-1,True) L(5,3,2)L(0,4,1)

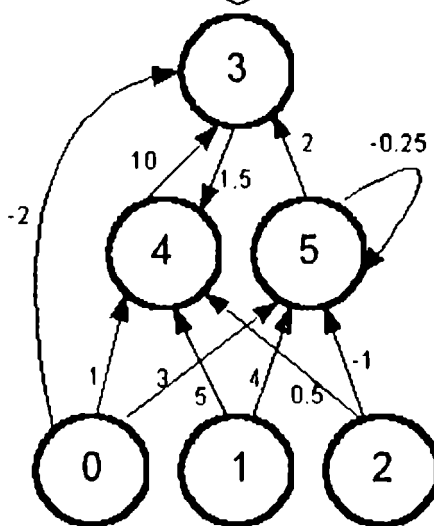
**P2(n0, n1, n2) :** (n2) → L(0,n0, n1+4)L(2,4,0.5)L(2,n0,n1)P2(1,4,False)  
(not n2) → L(n0,n1+1,n1)P3(5)L(n0,n1,n1+1)

**P3(n0) :** (True) → L(5,5,-0.25)L(0,3,-2)L(4,3,10)

Aplicación de 4 pasos de reescritura.

L(3,4,1.5)L(0,5,3)L(2,4,0.5)L(2,5,-1)L(1,5,4)L(1,4,5)  
L(5,3,2)L(5,5,-0.25)L(0,3,-2)L(4,3,10)L(0,4,1)

Interpretación de la cadena obtenida



Red neuronal representada.

Figura 51: Ejemplo de construcción de una RNA a partir de un Sistema L mediante comandos *LINK* representados por el símbolo *L*.

Si bien a primera vista parecería deseable el efecto de los comandos absolutos, luego de varios experimentos y comparaciones resultó más potente el enfoque relativo. Supongamos que se necesita al menos una neurona oculta para resolver un problema determinado. Utilizando comandos absolutos, como por ejemplo *LINK*, tienen que aparecer en la reescritura tantos comandos como entradas se utilicen para el problema y que además todas conduzcan a la misma neurona y se conecten con cada una de las entradas. Para complicarlo aún más, tiene que haber tantos comandos como salidas, de manera de conectar esta misma neurona con cada una de las salidas. En el contexto de este algoritmo evolutivo es muy poco probable que ocurra algo de esta naturaleza.

En cambio, pueden utilizarse comandos relativos a una neurona, de manera que un comando específico le indique al intérprete que esté “parado” sobre una neurona determinada. Los comandos siguientes crearán conexiones desde o hasta dicha neurona, aumentando las probabilidades de crear una neurona oculta.

Por otro lado, debido al significado que se les da a los valores de los parámetros de los comandos, es conveniente que se les asocie un tipo. Esto se debe a que para los parámetros que representan índices de neuronas no tiene sentido pensar en valores reales mientras que para los que representan pesos de conexiones esto es necesario. También se puede pensar en valores booleanos que afectan comandos o que participan en alguna condición. Por todo esto, hemos extendido los Sistemas L para que los parámetros de los módulos puedan ser de tipo Real, Entero o Booleano.

Asignarle tipos a los parámetros evita que tengan que hacerse conversiones de valores reales a valores enteros o booleanos cuando se ejecutan los comandos.

### 6.3.1. Comandos de construcción de RNA en NeSR

Antes de describir los comandos utilizados para NeSR, es necesario analizar una serie de condiciones que se establecen en el momento de la construcción de la red neuronal:

- La cantidad de entradas y salidas son conocidas a priori, ya que dependen del problema. Por ejemplo, si vamos a resolver el problema del XOR, las RNA tendrán dos entradas, el término de tendencia y una salida.
- A cada neurona se le asigna un número entero que la identifica. A la primera neurona de entrada se le asigna el número 0, la segunda el 1 y así sucesivamente.
- Los identificadores de las neuronas de salida son continuos a los de las de entradas, de manera que si la última entrada tiene identificador 3, la primera neurona de salida será la 4.
- A las neuronas ocultas se les asignan identificadores seguidos de los de la última neurona de salida.

De esta forma, a las neuronas de una RNA con 3 entradas, 1 salida y 2 ocultas se les asigna los siguientes identificadores:

Entrada 1	Entrada 2	Entrada 3	Salida 1	Ocultas 1	Ocultas 2
0	1	2	3	4	5

Antes de exponer el conjunto de comandos utilizados en NeSR, debemos aclarar que la elección del mismo responde a una serie de pruebas con distintos juegos de comandos. Este conjunto permite construir libremente cualquier estructura de RNA posible.

No obstante, una de las potencialidades del método es la de modificar total o parcialmente dicho conjunto de comandos, de manera de adaptarlo a las necesidades del problema, aplicando restricciones a las RNA generadas o facilitar la construcción de algún tipo de estructura en particular.

En NeSR se utilizó un conjunto de comandos relativos basados en una neurona actual, de la cual parten o llegan conexiones. Estos comandos se encuentran listados en la Tabla 7.

Durante el proceso de construcción, se utilizan dos variables de estado que influirán en la ejecución de los comandos. La primera es *NEURONA\_ACTUAL*, la cual contiene el identificador de la neurona sobre la que se aplican los comandos. Pueden ser identificadores tanto de neuronas ocultas como de salida. Cuando se detallan los comandos quedará claro por qué no se utilizan identificadores de neuronas de entrada. La otra variable es *NEURONAS\_OCULTAS*, en la cual el intérprete almacena la cantidad actual de neuronas ocultas que posee la red en construcción. El valor inicial de *NEURONAS\_OCULTAS* se fija de antemano como parámetro del algoritmo, pudiendo ser modificado durante el proceso de construcción de la RNA.

Comando	Significado
<b>Neuron (id)</b>	Modifica la variable <i>NEURONA_ACTUAL</i> de manera de tomar como referencia a la neurona indicada por el identificador <b>id</b> para la aplicación de los comandos siguientes. El identificador es relativo a la primera neurona de salida, pudiendo seleccionarse de esa forma neuronas ocultas y de salida.
<b>FromI (id, w)</b>	Establece una conexión con peso <b>w</b> desde la neurona de entrada indicada por ( <b>id mod CANTIDAD_ENTRADAS</b> ) hasta la indicada por <i>NEURONA_ACTUAL</i> .
<b>FromO (id, w)</b>	Establece una conexión con peso <b>w</b> desde la neurona de salida indicada por ( <b>id mod CANTIDAD_SALIDAS</b> ) hasta la indicada por <i>NEURONA_ACTUAL</i> .
<b>ToO (id, w)</b>	Establece una conexión con peso <b>w</b> desde la neurona indicada por <i>NEURONA_ACTUAL</i> hasta la neurona de salida indicada por ( <b>id mod CANTIDAD_SALIDAS</b> ).
<b>FromH (id, w)</b>	Establece una conexión con peso <b>w</b> desde la neurona oculta indicada por <b>id</b> hasta <i>NEURONA_ACTUAL</i> .
<b>ToH (id, w)</b>	Establece una conexión con peso <b>w</b> desde la neurona indicada por <i>NEURONA_ACTUAL</i> hasta la neurona oculta indicada por <b>id</b> .
<b>Rec (w)</b>	Establece una conexión con peso <b>w</b> desde la neurona indicada por <i>NEURONA_ACTUAL</i> hacia sí misma.
<b>AddHidden</b>	Incrementa el valor de la variable <i>NEURONAS_OCULTAS</i> .

Tabla 7: Conjunto de comandos de construcción de RNA utilizados en NeSR.

$$\varepsilon(w) = \begin{cases} -MaxValue, & \text{si } w \leq -MaxValue \\ w, & \text{si } |w| < MaxValue \\ MaxValue, & \text{si } w \geq MaxValue \end{cases}$$

Ecuación 14: Función de ajuste de los pesos de las conexiones.



Debido a que los valores que toman los parámetros de los comandos no están sujetos a ningún tipo de restricción, a veces es necesario realizar ajustes a dichos valores.

En el caso de los parámetros que denotan identificadores, como pueden ser expresiones negativas, se toma su valor absoluto antes de determinar a que neurona corresponde. Además, para evitar que el valor esté fuera del rango permitido, se aplica en operador `mod` para acotarlos.

Cuando alguno de los comandos intenta generar una conexión ya creada, el peso de la nueva conexión es sumado al de la conexión existente. Debido a esto, los pesos de las conexiones no están acotados a ningún rango. Para restringir el rango de los pesos, se define una constante positiva *MaxValue* para ajustarlos empleando la Ecuación 14. La aplicación de esta función evita generar conexiones con pesos fuera del rango deseado.

### **6.3.2. Optimización de RNA**

Por cuestiones de desempeño en NeSR, una vez reescrito el sistema, interpretados todos los comandos y obtenido una Red Neuronal, se procede a optimizar la estructura de dicha red generada por los Sistemas L antes de evaluarla.

La optimización consiste en eliminar neuronas y conexiones que no están en ningún camino entre una neurona de entrada y una de salida, es decir no aportan nada a la respuesta de la red. En la Figura 52 pueden verse distintos ejemplos de optimizaciones de redes neuronales.

Las neuronas sin conexiones de salida son una de las candidatas a ser eliminadas, ya que no participan en la respuesta de la red neuronal. También es posible eliminar circuitos que poseen entradas y salidas pero que finalmente no influyen sobre ninguna neurona de salida.

La importancia de realizar estas optimizaciones de estructura radica en que se reduce la cantidad de cálculos necesarios para obtener la respuesta de la red neuronal para un estímulo dado. Si consideramos que en un problema de control típico con redes neuronales recurrentes es necesario evaluar la red todo el tiempo durante la simulación, reducir la cantidad de operaciones por cada respuesta de la red se traduce en un mejor desempeño y en menor tiempo de simulación.

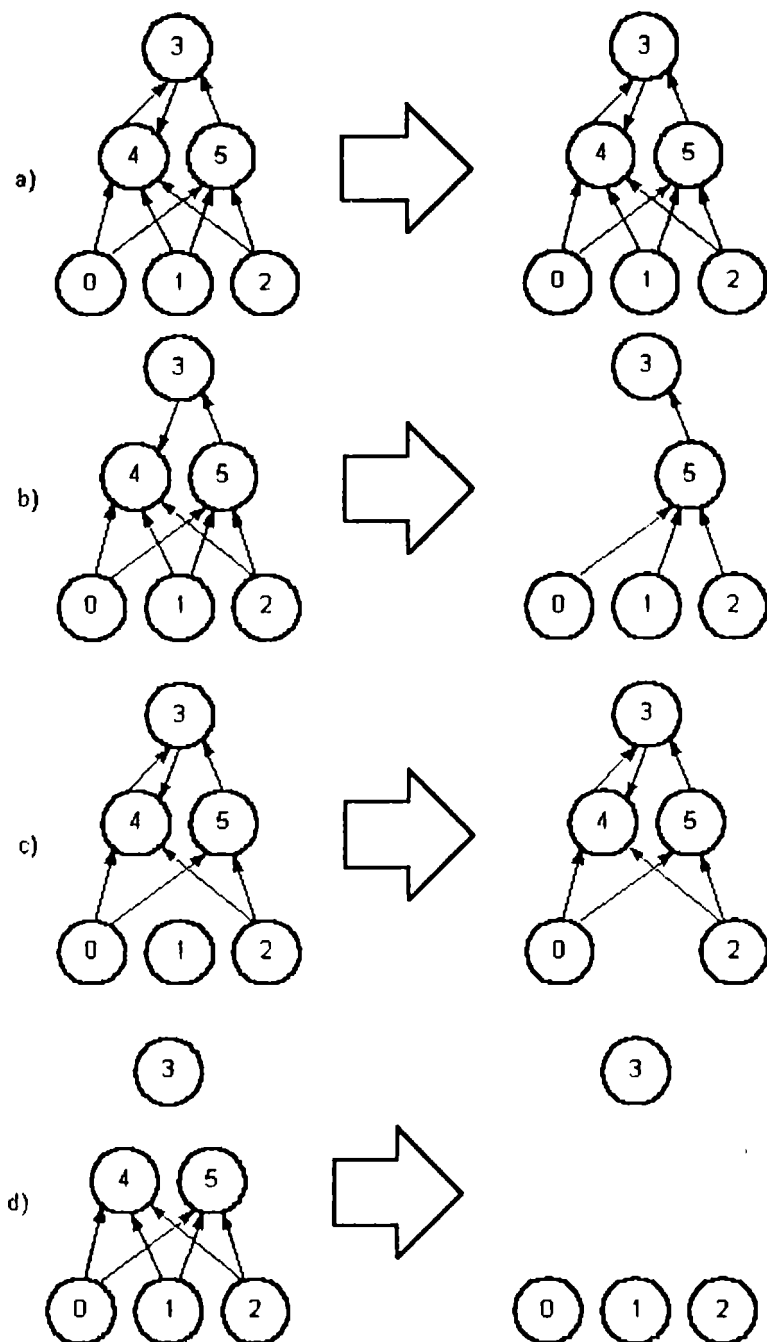


Figura 52: Optimización de estructuras de RNA. En el caso a) no hay cambios debido a que cada elemento de la red aporta algo a la salida final. En b) desaparecen neuronas y conexiones, mientras que en c) sólo neuronas. En el caso d) vemos una red que luego de su optimización queda vacía. Esta red no genera ninguna respuesta.

## 6.4. Evolucionando sistemas de reescritura

La evolución de los Sistemas L se realiza usando un algoritmo genético simple, de la forma descrita en la sección 3.4 y es aplicado a una población de cromosomas que representan Sistemas L.

El algoritmo genético se inicia con una población de Sistemas L generados al azar. Estos sistemas cumplen con una serie de restricciones impuestas a su forma. Estas restricciones serán detalladas en la sección 6.4.1 -Pág. 102-. Para la selección de los

individuos a reproducirse se utiliza el método de la ruleta. El reemplazo de la población se realiza aplicando elitismo, es decir, una determinada cantidad de los mejores individuos pasan intactos a la siguiente generación. Esta cantidad es configurable como un parámetro del algoritmo. El resto de la población es reemplazada por nuevos individuos obtenidos a partir de la aplicación de operadores genéticos sobre pares de cromosomas seleccionados dentro del total de la población.

```

T = 0
P(T) = Población inicial aleatoria de N
Evaluar P(T)
Mientras T < Máxima generación y no alcance el fitness máximo
    P(T + 1) = Población con los m mejores individuos de P(T)
    Mientras Cantidad de individuos de P(T + 1) < N
        Seleccionar Padre1 y Padre2 dentro de P(T)
        Hijo = Mutación(Crossover(Padre1, Padre2))
        Agregar Hijo a P(T + 1)
    Evaluar P(T + 1)
    T = T + 1

```

**Algoritmo 5: Pseudocódigo del algoritmo genético simple empleado en este método.**

En NeSR, se proponen dos operadores genéticos: el crossover entre dos Sistemas L, de manera de combinar las reglas que poseen el mismo predecesor, y la mutación, que realiza cambios en las reglas, agregando, reemplazando o quitando módulos, o alterando las expresiones que estos contienen.

#### **6.4.1. Restricciones impuestas a los Sistemas L**

Hemos elegido Sistemas L determinísticos, libres de contexto y paramétricos los cuales nos garantizan una reescritura idéntica para sistemas idénticos, mayores posibilidades de reescritura y variantes en los valores de los argumentos de los comandos que construirán la red neuronal. Sin embargo, resulta necesario imponer una serie de restricciones adicionales a los Sistemas L que resultan de vital importancia para el desarrollo del método. Estas restricciones son:

- **Conjunto de módulos:** El conjunto de módulos utilizado para definir los predecesores de las reglas es único y común a todos los sistemas creados. Antes de iniciar el proceso de neuroevolución, se inicializa un conjunto **P** de módulos *no terminales*  $p_1 \dots p_n$  y otro conjunto **C** de símbolos *terminales* o comandos  $c_1 \dots c_m$ . Para cada sistema de la población, los predecesores de las reglas utilizan elementos de **P** mientras que los sucesores de cada regla utilizan tanto elementos de **P** como de **C**. Al mantener un conjunto de módulos en común para cada sistema tiene sentido aplicar el operador de crossover para que combine partes de un sistema con partes de otro, como se verá cuando se detalle dicho mecanismo. Tanto el conjunto **P** como el **C** permanecen iguales durante todo el proceso de evolución.
- **Parámetros:** Cada módulo tiene un número mínimo y máximo de parámetros. Los elementos del conjunto **P** son inicializados con una cantidad aleatoria de parámetros, que se encuentra dentro de un mínimo y máximo establecidos de antemano. De esta manera si se inicializa un módulo  $P_1$  con tres parámetros, en donde el primero es un real, el segundo un entero y el tercero un booleano, esto no cambia en ningún momento y cada vez que se utilice este módulo, aparecerá con los tres parámetros de los tipos establecidos. Esto simplifica la reescritura ya que siempre que coincida un módulo con el predecesor de una regla tendrán la misma

definición. Intuitivamente se puede ver que en un crossover que combine partes de dos sistemas, mantener constantes estas definiciones le da sentido al operador, como también facilita su implementación. Los parámetros de los módulos del conjunto **C** tienen una definición fija ya que representan comandos de construcción de redes neuronales establecidos de antemano. Como no son creados aleatoriamente sino que son definidos por el programador, no hay restricciones de parámetros para éstos.

- **Cantidad de Reglas:** La cantidad de reglas de producción en cada sistema es la misma. Cuando se crea la población, cada sistema es inicializado con igual cantidad de reglas como elementos contenga **P**. Esta cantidad permanece inalterable, por lo que durante toda la búsqueda los sistemas de la población tendrán siempre una cantidad fija de reglas. Esto simplifica notablemente el diseño del operador de crossover, como explicaremos mas adelante.
- **Módulo de inicio:** Todos los sistemas comienzan su reescritura a partir del mismo módulo, pero cada sistema puede contener expresiones distintas en dicho módulo. Se establece que el primer módulo del conjunto **P** cumpla el rol de módulo de inicio para todos los sistemas de la población.
- **Pasos de reescritura:** Cada sistema se reescribirá un máximo de veces o hasta que la cadena de módulos resultante supere una determinada longitud. Estas cantidades son parámetros del algoritmo y permiten poner un límite al tamaño de los comandos generados.
- **Tamaño de las reglas:** Cada regla de cada sistema tiene un número mínimo y máximo de sucesores con sus correspondientes condiciones. Esto permite utilizar una reescritura con *guardas* y a la vez mantener los sistemas en un tamaño razonable. Esta restricción permite un intercambio de sucesores de regla entre uno y otro sistema pues todo sistema tendrá al menos una cantidad mínima para intercambiar con otro sistema.
- **Tamaño de los sucesores:** Cada sucesor de regla tiene un número mínimo y máximo de módulos. Esta es otra restricción que controla el tamaño de los sistemas.
- **Selección de la regla a reescribir:** Como hemos dicho los Sistemas **L** utilizados en NeSR son determinísticos, es decir, un mismo sistema arrojará el mismo resultado en  $n$  reescrituras independientes. La selección de la regla a aplicar en una reescritura es simple, la primer regla definida cuyo predecesor coincida con el módulo a reescribir y que además la condición evaluada resulte verdadera.

Hasta aquí presentamos las restricciones impuestas a los Sistemas **L** utilizados exponiendo las razones de las mismas. Cabe aclarar que todas las cantidades mínimas y máximas mencionadas en este detalle forman parte del conjunto de parámetros del método NeSR.

#### **6.4.2. Crossover en NeSR**

El diseño del operador de crossover para el método aquí presentado es la consecuencia de la evaluación de los resultados arrojados por distintos experimentos, lo cual nos permitió comparar entre diferentes alternativas. En todo algoritmo evolutivo el efecto del crossover puede resultar *destrutivo* si no se tiene en cuenta la forma de los cromosomas y el significado de cada gen. Este operador debería recombinar lo mejor de dos individuos y obtener uno nuevo que si bien puede no superar, al menos igualará a sus padres en solucionar el problema a resolver. Pero a veces no ocurre así. Por ejemplo

un crossover de un punto puede dividir a un buen individuo en una sección donde se codifica parte de su buen comportamiento y al combinar con otro, que no posea las mismas características en esa sección, estas no se transmitirán a la descendencia. Más claro se ve aún en el paradigma de la neuroevolución. Una RNA codificada en un cromosoma puede perder su buen comportamiento al ser dicho cromosoma dividido en una posición dentro del bloque en el que se representan pesos de conexiones importantes.

Debido a esto, el operador de crossover utilizado en NeSR fue pensado para evitar o minimizar estos efectos destructivos. Básicamente recorre las reglas de dos sistemas a combinar, intercambiando los casos de las reglas comunes a ambos. Además existe la posibilidad de intercambiar el símbolo de inicio con sus expresiones. Como se explicó anteriormente (Pág. 102), éstos tienen la misma definición para todos los sistemas de la población por lo tanto este intercambio puede verse como un cambio completo de todos los valores de los parámetros entre los símbolos de inicio de ambos padres. Decimos que el operador así definido minimiza los efectos adversos del crossover, ya que es de esperar que ambas reglas tengan un significado similar dentro de sistemas de reescritura parecidos. El hecho de intercambiar casos de reglas completos puede verse como un intercambio de bloques. Por ejemplo si un caso de regla permite construir una sección de la RNA en particular, al reemplazarse por otro puede ocurrir que en esa sección cambie la estructura o los pesos de dicha red.

```

Con probabilidad  $p_{\text{cross}}$  →
  Nuevo Sistema = Sistema vacío con el módulo inicial elegido al azar
  entre los módulos iniciales de S1 y S2
  Para cada predecesor i de S1
    Para cada regla j con predecesor i de S1
      Obtener regla j con predecesor i del S2
      Si existe →
        Nueva Regla = Regla vacía con predecesor i común a S1 y S2
        Para cada caso k de la regla j de S1 →
          Obtener caso k de la regla j de S2
          Si existe →
            Nuevo Caso = copia del caso k de la regla j de S1 o S2,
            eligiéndose el sistema en forma aleatoria
          Sino →
            Nuevo Caso = copia del caso k de la regla j de S1
          Agregar Nuevo Caso a la Nueva Regla.
        Sino →
          Nueva Regla = copia de la regla j con predecesor i de S1
        Agregar Nueva Regla al Nuevo Sistema
      Sino →
        Nuevo Sistema = copia de S1

```

**Algoritmo 6: Algoritmo de crossover entre dos sistemas de reescritura S1 y S2**

Si bien el intercambio del símbolo de inicio puede introducir un cambio con un mayor alcance que el intercambio de un caso de regla<sup>5</sup>, habrá reglas que funcionen como lo hacían anteriormente ya que provienen del mismo sistema.

<sup>5</sup> Debemos tener en cuenta que el símbolo de inicio es la raíz desde donde comienza el proceso de reescritura. Un pequeño cambio en éste puede afectar a todo el resultado que se obtenga de la derivación de un Sistema L y para NeSR en particular la RNA representada.

El Algoritmo 6 muestra el pseudocódigo que describe como funciona este operador de crossover. Dicho operador permite explorar las combinaciones de los casos de reglas disponibles. Sin embargo, no altera los sucesores de las reglas, con lo que se apunta a mantener las buenas combinaciones entre módulos dentro de una misma regla.

### 6.4.3. Mutación en NeSR

El operador de crossover presentado anteriormente produce intercambios entre casos de reglas pero no modifica su forma ni tampoco las expresiones aritméticas y lógicas contenidas tanto en los módulos como en las condiciones. Utilizando únicamente el crossover definido, la búsqueda queda limitada a la combinación de las reglas de los sistemas generados en la población inicial, sin posibilidad de explorar otras zonas del espacio de soluciones.

Para superar la pérdida de diversidad se introduce un operador de mutación que hemos creado específicamente para los Sistemas L. Dicho operador actúa modificando la estructura de las reglas e introduciendo cambios en las expresiones contenidas en módulos y condiciones.

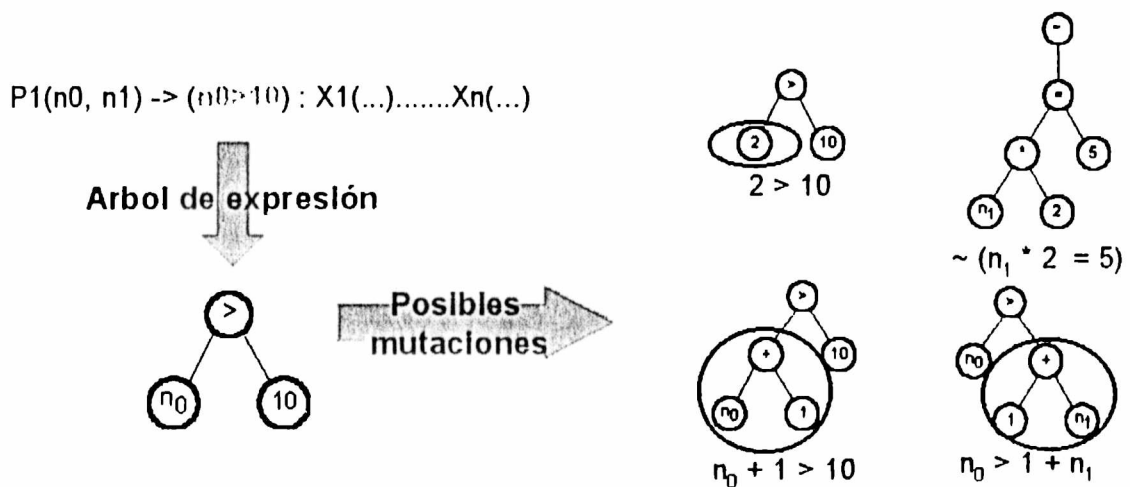


Figura 53: Ejemplo de una mutación de expresiones aplicada a una condición de un caso de regla. Notar que en los ejemplos de la derecha aparecen parámetros que no estaban originalmente en la expresión aunque efectivamente se encuentran definidos en el encabezado de la regla.

El operador de mutación definido en NeSR puede actuar en dos aspectos diferentes del sistema: estructura y expresiones.

- *Mutación de expresiones:* se altera con cierta probabilidad las expresiones del Sistema L, tanto en las condiciones de las reglas como en las expresiones contenidas en los módulos de los sucesores. El operador trabaja recorriendo cada una de las expresiones contenidas en un Sistema L y con una cierta probabilidad determina si alterarla o no. En el Algoritmo 7 se detalla este proceso. Una vez que se decide modificar una expresión, se toma la misma como un árbol, se selecciona al azar un nodo y todo el subárbol que lo tiene como raíz es reemplazado por un nuevo árbol generado al azar. En la Figura 53 se muestra gráficamente esta idea. Para crear el nuevo subárbol se tienen en cuenta todos los parámetros que define el predecesor de la regla que contiene la expresión a modificar.

un crossover de un punto puede dividir a un buen individuo en una sección donde se codifica parte de su buen comportamiento y al combinar con otro, que no posea las mismas características en esa sección, estas no se transmitirán a la descendencia. Más claro se ve aún en el paradigma de la neuroevolución. Una RNA codificada en un cromosoma puede perder su buen comportamiento al ser dicho cromosoma dividido en una posición dentro del bloque en el que se representan pesos de conexiones importantes.

Debido a esto, el operador de crossover utilizado en NeSR fue pensado para evitar o minimizar estos efectos destructivos. Básicamente recorre las reglas de dos sistemas a combinar, intercambiando los casos de las reglas comunes a ambos. Además existe la posibilidad de intercambiar el símbolo de inicio con sus expresiones. Como se explicó anteriormente (Pág. 102), éstos tienen la misma definición para todos los sistemas de la población por lo tanto este intercambio puede verse como un cambio completo de todos los valores de los parámetros entre los símbolos de inicio de ambos padres. Decimos que el operador así definido minimiza los efectos adversos del crossover, ya que es de esperar que ambas reglas tengan un significado similar dentro de sistemas de reescritura parecidos. El hecho de intercambiar casos de reglas completos puede verse como un intercambio de bloques. Por ejemplo si un caso de regla permite construir una sección de la RNA en particular, al reemplazarse por otro puede ocurrir que en esa sección cambie la estructura o los pesos de dicha red.

```

Con probabilidad  $p_{cross}$  →
  Nuevo Sistema = Sistema vacío con el módulo inicial elegido al azar
  entre los módulos iniciales de S1 y S2
  Para cada predecesor i de S1
    Para cada regla j con predecesor i de S1
      Obtener regla j con predecesor i del S2
      Si existe →
        Nueva Regla = Regla vacía con predecesor i común a S1 y S2
        Para cada caso k de la regla j de S1 →
          Obtener caso k de la regla j de S2
          Si existe →
            Nuevo Caso = copia del caso k de la regla j de S1 o S2,
            eligiéndose el sistema en forma aleatoria
          Sino →
            Nuevo Caso = copia del caso k de la regla j de S1
          Agregar Nuevo Caso a la Nueva Regla.
        Sino →
          Nueva Regla = copia de la regla j con predecesor i de S1
          Agregar Nueva Regla al Nuevo Sistema
      Sino →
        Nuevo Sistema = copia de S1
  
```

**Algoritmo 6: Algoritmo de crossover entre dos sistemas de reescritura S1 y S2**

Si bien el intercambio del símbolo de inicio puede introducir un cambio con un mayor alcance que el intercambio de un caso de regla<sup>5</sup>, habrá reglas que funcionen como lo hacían anteriormente ya que provienen del mismo sistema.

<sup>5</sup> Debemos tener en cuenta que el símbolo de inicio es la raíz desde donde comienza el proceso de reescritura. Un pequeño cambio en éste puede afectar a todo el resultado que se obtenga de la derivación de un Sistema L y para NeSR en particular la RNA representada.

**Regla a modificar**

P1(N0, N1): N0 + 1 > 3 → Neuron(N0) FromH(N0, N1) Rec(0.5) P2(N1) P1(N0-1, N1)



**- Eliminación:**

P1(N0, N1): N0 + 1 > 3 → Neuron(N0) P1(N0-1, N1)

**- Reemplazo:**

P1(N0, N1): N0 + 1 > 3 → Neuron(N0) P1(N+1, 0.2) ToO(-10, N1) P1(N0-1, N1)

**- Inserción:**

P1(N0, N1): N0 + 1 > 3 → Neuron(N0) P1(N+1, 0.2) ToO(-10, N1) FromH(N0, N1) Rec(0.5) P2(N1) P1(N0-1, N1)

Figura 54: Ejemplo de mutación de estructura en una regla de producción.

### 6.5. La red neuronal

En esta sección detallaremos el tipo de red neuronal utilizada en el método cada vez que se reescribe un Sistema L y se interpretan los comandos para la construcción de la misma. Simplemente mencionaremos particularidades. Sin embargo para obtener una visión más amplia de los aspectos generales en cuanto a RNA se puede retomar el capítulo correspondiente de este mismo trabajo (Pág.3). Ahora continuaremos suponiendo que el lector está familiarizado con el paradigma de RNA.

Una de las potencialidades de NeSR es no imponer restricciones a las estructuras de las redes, es decir, las conexiones entre neuronas son libres y puede haber conexiones entre cualquier par de nodos. Esto excluye a las conexiones desde neuronas ocultas o de salida hacia las entradas, debido a que no consideramos las entradas como neuronas en sí, sino simplemente un punto de partida para la información que es provista a la RNA. La retroalimentación puede darse libremente en todo el conjunto formado por los nodos ocultos y de salida. Decimos que las redes neuronales obtenidas son típicamente recurrentes. Por todo lo anterior, cada neurona actualiza su valor de activación de la siguiente manera:

$$x_i(t+1) = \sigma \left( \sum_{j=1}^N a_{ij} x_j(t) + \sum_{j=1}^M b_{ij} u_j(t) + c_i \right), i = 1..N$$

donde  $M$  es la cantidad de entradas y  $N$  es la suma de la cantidad de neuronas ocultas y de salida que posee la red. Los elementos  $a_{ij}$  representan los pesos de las conexiones entre las neuronas ocultas y de salida, los  $b_{ij}$  representan las conexiones desde las neuronas de entrada hacia el resto de las neuronas y los  $c_i$  representan los términos de tendencia de cada neurona. Por lo tanto, la activación de cada neurona es actualizada en función de las estradas  $u_j$  y de las activaciones  $x_j$  del estado anterior.

La función  $\sigma$  de activación de cada neurona está definida de la siguiente forma:

$$\sigma(x) = \begin{cases} -1 & , \text{si } x \leq -MaxValue \\ \frac{x}{MaxValue} & , \text{si } abs(x) < MaxValue \\ 1 & , \text{si } x \geq MaxValue \end{cases}$$

En NeSR los términos de tendencia se encuentran implementados a través de una neurona de entrada adicional, cuyo valor de activación es siempre 1.



La evaluación de la RNA se realiza en dos fases, una de propagación, en la cual el valor anterior de activación de cada neurona es colocado en las conexiones de salida de la misma y otra de evaluación propiamente dicha en la cual cada neurona toma los valores que tiene en sus conexiones de entrada y calcula el siguiente estado. Si la RNA a evaluar se trata exclusivamente de una red feedforward alcanza con realizar este proceso una única vez. Pero debido a la topología de las redes creadas es necesario establecer una repetición de este proceso una pequeña cantidad de veces antes de considerar la salida de la RNA como la respuesta obtenida. Esta cantidad depende del problema a resolver pero en nuestros experimentos nunca superó las 5 evaluaciones. De esta manera se logra que las recurrencias puedan influir en el resultado final. En los problemas de clasificación, antes de evaluar cada patrón, la RNA es inicializada con los valores de activación de las neuronas en cero. Por lo tanto en un primer paso de evaluación, las conexiones recurrentes no tienen efecto ya que toman el valor cero inicial del nodo del cual nacen. Recién en una segunda evaluación la recurrencia toma valor y comienza a influir en la salida final de la red.

Con esto damos por finalizada la descripción del método neuroevolutivo en el cual se centra la presente tesis. Nos resta en los capítulos siguientes brindarle al lector una breve noción de otros métodos que utilizamos para comparar con NeSR, así como algunos detalles de los experimentos realizados junto con un análisis de los resultados obtenidos.

## 7. Otros métodos de neuroevolución

En el presente capítulo se discutirán tres métodos de neuroevolución de distintas características. El primero de ellos se basa en un método que utiliza representación directa y sólo evoluciona los pesos de conexión. El siguiente, trabaja con cromosomas que representan sistemas de reescritura de matrices y de manera indirecta codifica solamente la estructura de la red, dejando los pesos para otro método. Por último se discutirá el método NEAT, el cual utiliza una codificación directa pero evoluciona tanto los pesos como las estructuras de las redes.

Los métodos que se explicarán a continuación fueron seleccionados por diferentes motivos. NEAT es un método novedoso que ha mostrado buenos resultados y, como NeSR, evoluciona tanto las topologías como los pesos de las conexiones. El método de reescritura de matrices fue inspirado por la misma idea de NeSR: utilizar codificación indirecta y sistemas de reescritura, aunque solamente evoluciona topologías pero no los pesos de las conexiones. El método restante es el más antiguo y el que menos cambios impone sobre un algoritmo genético simple.

Salvo el método neuroevolutivo de reescritura de matrices, los otros dos métodos se utilizaron como punto de comparación en los experimentos realizados con NeSR.

### 7.1. Neuroevolución vía un Algoritmo Genético Simple

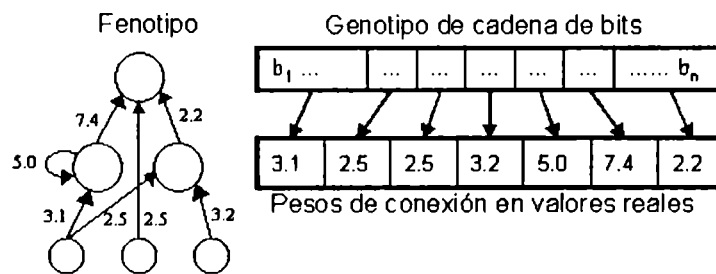
El algoritmo genético simple como el que hemos presentado en capítulos anteriores – sección 3.4- ha usado siempre cadenas binarias para codificar soluciones alternativas, usualmente llamados *cromosomas*. En tal esquema de representación, cada peso de conexión es representado por un número de bits con un determinado tamaño. Una RNA es codificada por una concatenación de todos los pesos de conexión de la red en el cromosoma.

Una heurística con respecto al orden de la concatenación es colocar juntos los pesos de conexión de los mismos nodos ocultos o de salida. Los nodos ocultos en RNAs son en esencia, extractores de rasgos y detectores. La separación aparte de las entradas de un mismo nodo oculto en la representación binaria incrementaría la dificultad de construir detectores de características de utilidad ya que deberían ser destruidos por los operadores de crossover. Generalmente es muy difícil aplicar operadores de crossover en la evolución de pesos de conexiones ya que tienden a destruir detectores de rasgos encontrados durante el proceso evolutivo.

Particularmente, la neuroevolución mediante esta codificación de cadenas binarias está clasificada dentro de los métodos de codificación directa. Un método neuroevolutivo implementado de esta forma solamente se centra en la evolución de los pesos de conexión utilizando una topología fija durante todo el proceso de simulación. Sin embargo, por utilizar representación y operadores genéticos simples, resulta eficiente en cuanto a costo de procesamiento.

#### 7.1.1. Codificación de redes neuronales

En el capítulo referido a algoritmos evolutivos se dedicó una sección a la codificación de números reales utilizando cadenas de bits –Pág. 46-.



**Figura 55: Mapeo de un genotipo representado por una cadena de bits a su fenotipo correspondiente, en este caso, una red neuronal. El genotipo representa números reales que se convierten en pesos de conexiones predeterminadas**

Una vez seleccionada la manera de codificar los valores reales que representarán las conexiones sólo se necesita determinar la cantidad de conexiones que tendrá el fenotipo resultante. Determinado este parámetro, el cromosoma tendrá un tamaño correspondiente a la cantidad total de posibles conexiones en la red por la cantidad de bits que se seleccionó para codificar cada peso de conexión, como se muestra en la Figura 55.

Cada posición en el genotipo representa un peso de una conexión determinada en el fenotipo.

El genotipo puede incluir más información, por ejemplo, conexiones de tendencia, tipo de función de activación de cada neurona, habilitación o no de una conexión agregando bits adicionales al genotipo. En el proceso de codificación del genotipo en su fenotipo se interpretarán estos bits de acuerdo a la representación seleccionada.

### 7.1.2. Neuroevolución

Para completar la descripción de este método de neuroevolución, resta utilizar un algoritmo genético simple aplicando la codificación presentada para este caso particular.

El uso de un algoritmo genético simple sobre una población de individuos codificados por cromosomas de cadenas de bits es inmediato según se explicó en secciones anteriores.

La diferencia consiste en la transformación del genotipo en el fenotipo, ya que es necesario implementar un mecanismo para transformar la cadena de bits en la correspondiente red neuronal, la cual será evaluada en el problema a resolver y a partir de esto se obtendrá el valor de fitness para el individuo que generó dicha red.

### 7.1.3. Análisis del método

Este método tiene como ventaja pocos requerimientos de procesamiento, ya que pueden aplicarse los operadores genéticos de cadenas de bits, los cuales no insumen gran tiempo de ejecución.

Puede criticarse la necesidad de fijar una estructura de red a ser evolucionada ya que, dependiendo del problema, no puede saberse a priori la estructura necesaria. Se puede correr el riesgo de seleccionar una estructura insuficiente y no resolver el problema o redundante y empobrecer la performance del método.

## 7.2. Neuroevolución por reescritura de matrices

Este método propuesto por Kitano en [69], encuadra dentro del conjunto de esquemas de codificación indirecta. Si bien este método solamente evoluciona la arquitectura de la red, puede verse como un predecesor de NeSR, ya que se basa en sistemas de reescritura, cuyo resultado se utiliza como molde para crear las redes neuronales.

### 7.2.1. Reescritura de matrices

Los sistemas de reescritura de matrices definen reglas de reescrituras donde la parte izquierda de la regla o predecesor es un símbolo y la parte derecha o sucesor es una matriz de  $n \times n$ . La matriz del lado derecho puede ser de tres tipos diferentes:

- *Matriz de no terminales definidos por mayúsculas:* se utilizan solamente para definir como se reescribe el símbolo de inicio.
- *Matriz de no terminales definidos por minúsculas:* se utilizan para definir como se reescribe un símbolo no terminal definido por mayúsculas.
- *Matriz de terminales:* contienen unos y ceros y definen presencia o ausencia de conexión. Definen como reescribir un símbolo definido por una letra minúscula.

Para explicar el proceso de reescritura de matrices nos basaremos en el siguiente ejemplo con matrices de  $2 \times 2$ :

La reescritura comienza con un símbolo de inicio S. Se definirá S de la siguiente manera:

$$S \rightarrow \begin{matrix} A & B \\ C & D \end{matrix}$$

Entonces en el primer paso de reescritura el resultado será la matriz que figura del lado derecho de la regla anterior.

Definimos las siguientes reglas para A, B, C y D:

$$\begin{aligned} A &\rightarrow \begin{matrix} a & a \\ a & a \end{matrix} \\ B &\rightarrow \begin{matrix} i & i \\ i & a \end{matrix} \\ C &\rightarrow \begin{matrix} i & a \\ a & c \end{matrix} \\ D &\rightarrow \begin{matrix} a & e \\ a & e \end{matrix} \end{aligned}$$

Como se explicó anteriormente, estas reglas definen de su lado derecho solamente símbolos no terminales definidos con letras minúsculas.

Reemplazando la matriz obtenida en el paso anterior por la matriz resultante de aplicar estas reglas de reescritura obtenemos la siguiente matriz:

```

a a i i
a a i a
i a a e
a c a e

```

Finalmente se definen reglas de reescritura para a, i, c y e:

```

a → 0 0
    0 0
i → 1 0
    0 0
c → 0 0
    1 0
e → 0 1
    0 0

```

Notar que estas reglas definen de su lado derecho solamente símbolos terminales.

Reemplazando cada símbolo de la matriz anterior de acuerdo a las nuevas reglas definidas, se obtiene:

```

0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0

```

La cual es interpretada como una matriz de conexiones donde un 1 en la celda  $(i, j)$  de la misma representa una conexión desde la neurona  $i$  hasta la neurona  $j$  de la red resultante como se muestra en la Figura 56.

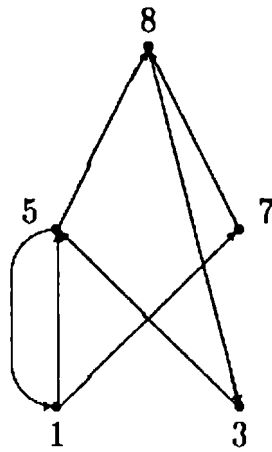


Figura 56: Estructura de red neuronal generada por un sistema de reescritura de matrices.

### 7.2.2. Evolución de topologías de redes neuronales

El tamaño del cromosoma depende de la cantidad de símbolos no terminales a utilizar y las dimensiones de las matrices de reescritura.

Las reglas que describen la reescritura de un símbolo no terminal definido con letras minúsculas en una matriz de símbolos terminales son las mismas para todos los cromosomas. Con esto se logra la reutilización de patrones de conexión.

Siguiendo con el ejemplo del sistema anterior un cromosoma que represente el sistema descrito es el siguiente:

ABCDaaaaiaaiaaacaee

El significado de cada alelo está relacionado con la posición que ocupa en el cromosoma.

Es necesario definir un número de pasos de reescritura y no se permiten reescrituras recursivas.

### 7.2.3. Análisis del método

Como se mencionó anteriormente este método sólo evoluciona arquitecturas de redes y no los pesos de conexiones. Por utilizar una codificación indirecta resulta un punto de comparación para el método presentado en este trabajo. Sin embargo, al no evolucionar una red completa, este método requiere de algún mecanismo adicional para obtener el conjunto de pesos de conexión adecuado, como por ejemplo entrenamiento por backpropagation, simulated annealing, etc.; o codificando la estructura seleccionada para cualquier otro método de neuroevolución. Como pudimos observar anteriormente (ver Pág. 71) esto trae aparejado algunos inconvenientes y requiere un considerable tiempo extra de procesamiento.

## 7.3. Neuroevolución de topologías en aumento (NEAT)

Este es un método presentado por Kenneth O. Stanley en [124] y se centra tanto en la evolución de los pesos de conexión como de las topologías de las redes neuronales. Mediante éste se intenta demostrar que si se hace correctamente, la evolución de estructuras junto con los pesos de conexión puede mejorar significativamente el desempeño de la neuroevolución. Este método llamado *Neuroevolución de topologías en aumento* está diseñado a fin de aprovechar la estructura para minimizar las dimensiones del espacio de búsqueda de los pesos de conexión. Si la estructura es evolucionada de manera tal que las topologías son minimizadas y crecen incrementalmente, se obtienen significantes progresos en la velocidad de aprendizaje resultante. El autor asegura que se obtiene una mejor eficiencia de topologías si estas se mantienen mínimas a lo largo de la evolución en lugar de minimizar las mismas al final del proceso.

Se estudió este método debido a que en cierto modo posee características comparables al método propuesto, ya que NeSR también evoluciona los pesos de conexión y las estructuras de las redes neuronales, aunque NEAT utiliza una codificación directa de la red.

### 7.3.1. Codificación genética

El esquema de codificación genética de NEAT está diseñado para permitir a los genes correspondientes ser fácilmente alineados cuando se realiza el crossover entre dos genomas. Los genomas son representaciones lineales de la conectividad de una red (Figura 57). Cada genoma incluye una lista de *genes de conexión*, cada uno de los cuales representa a dos *genes de nodos* conectados. Los genes de nodos proveen una lista de entradas, nodos ocultos y salidas que pueden ser conectados. Cada gen de conexión especifica el nodo de entrada, el nodo de salida, el peso de la conexión, un indicador de habilitación del gen y un *número de innovación*, el cual permite encontrar la correspondencia entre genes (como se explicará en secciones posteriores).

En NEAT se define un operador de mutación que puede cambiar tanto pesos de conexión como estructuras de red. Los pesos de conexión mutan como en cualquier sistema de neuroevolución, perturbando o no el peso de cada conexión en cada generación. Las mutaciones estructurales ocurren en dos formas (Figura 58). Cada mutación expande el tamaño del genoma agregando uno o varios genes. En la mutación de *agregado de conexión*, se agrega un solo gen de conexión que representa una conexión entre dos nodos previamente existentes y no conectados. En la mutación de *agregado de nodo* una conexión existente se divide en dos ubicando al nuevo nodo donde estaba ésta. La vieja conexión es deshabilitada y se agregan al genoma dos nuevas conexiones. La nueva conexión que llega al nuevo nodo recibe un peso de 1 y la nueva conexión que sale de éste recibe el mismo peso que la conexión vieja. De esta manera la mutación no afectaría demasiado el comportamiento original de la red, permitiendo a los nuevos nodos ser integrados inmediatamente, contrariamente a lo que ocurre al agregar estructuras extrañas que deben ser evolucionadas en la red posteriormente. De esta manera, debido a la clasificación por especies, la red tendrá tiempo de optimizar y hacer uso de su nueva estructura.

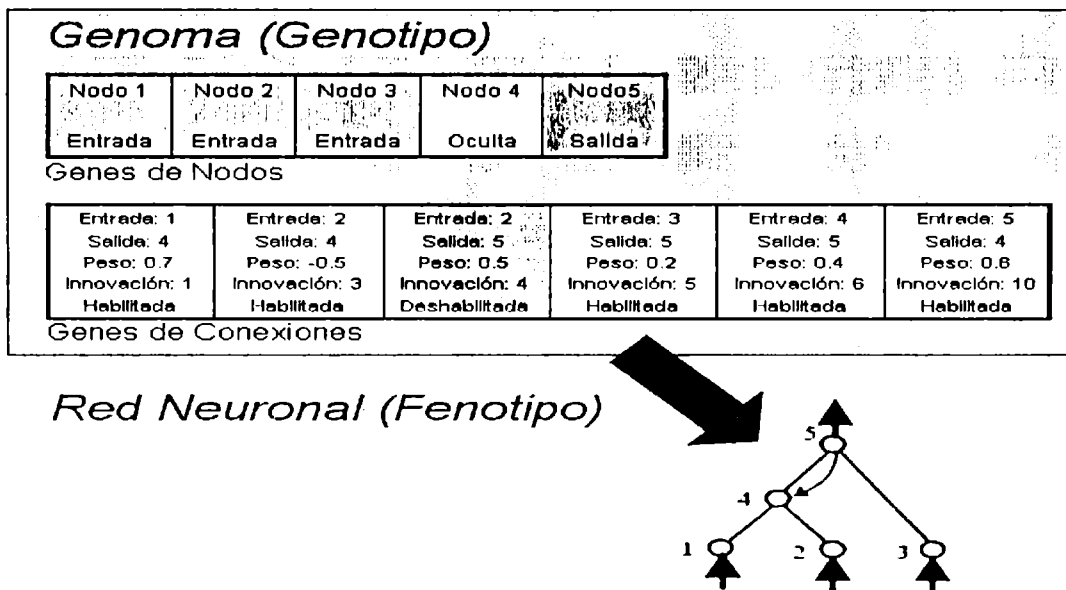


Figura 57: Un ejemplo de mapeo entre genotipo y fenotipo. Se muestra un genotipo y el fenotipo que se produce a partir de éste. Hay 3 nodos de entrada, uno oculto, uno de salida y 6 definiciones de conexión, una de las cuales es recurrente. El tercer gen está deshabilitado, por lo tanto la conexión que especifica (entre los nodos 2 y 5) no está expresada en el fenotipo.

A través de la mutación, los genomas en NEAT gradualmente se tornarán más grandes. Los genomas de longitud variable poseen a veces diferentes conexiones en las mismas posiciones.

### 7.3.2. Seguimiento de genes a través de marcas históricas

En NEAT se propone una manera de determinar cuales genes tienen el mismo origen entre distintos individuos de una población topológicamente diversa. Lo implementa introduciendo información adicional llamada *número de innovación*. Esta información es el origen histórico de cada gen. Dos genes con el mismo origen histórico pueden representar la misma estructura (aunque posiblemente con diferentes pesos), ya que han sido derivados del mismo gen ancestral en algún punto en el pasado. Por eso, todo lo que un sistema necesita hacer para saber cual gen se alinea con cual es mantener el origen histórico de cada gen en el sistema.

El seguimiento de los orígenes históricos requiere poco procesamiento. Cada vez que aparece un nuevo gen (a través de mutación estructural), un *número global de innovación* se incrementa y se asigna a ese gen. Los números de innovación, por lo tanto representan una cronología de la aparición de cada gen en el sistema. Como un ejemplo, diremos que las dos mutaciones de la Figura 58 ocurren una después de la otra. Al nuevo gen de conexión creado en la primera mutación se le asigna el número 7, y a los dos nuevos genes de conexión agregados en la mutación se les asigna los números 8 y 9. En el futuro, cada vez que esos genomas vayan a ser apareados, la descendencia heredará el mismo número de innovación en cada gen; los números de innovación nunca cambian. Por lo tanto, el origen histórico de cada gen en el sistema se conoce durante toda la evolución.



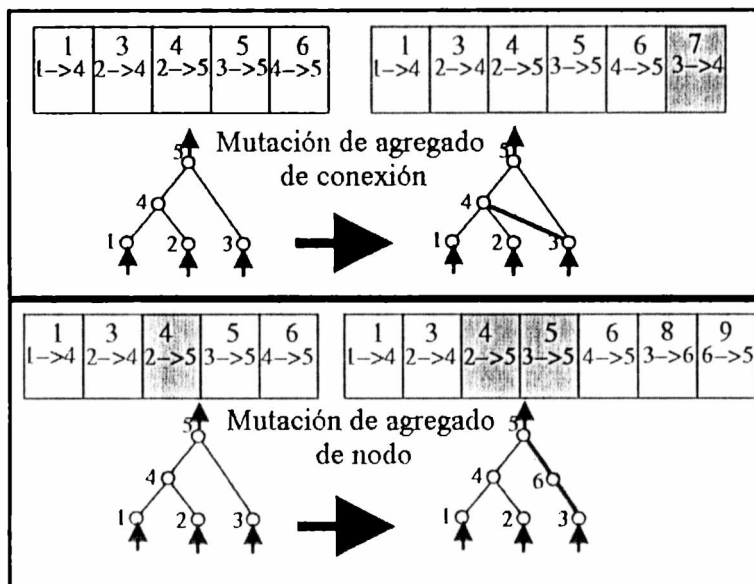


Figura 58: Los dos tipos de mutación estructural en NEAT. Ambos tipos, tanto el agregado de una conexión como el agregado de un nodo, se ilustran con los genes de conexión de una red sobre su fenotipo. El número superior en cada gen es el número de innovación. Los números de innovación son marcas históricas que identifican el ancestro histórico de cada gen. A los nuevos genes se les asigna nuevos números de manera incremental. Cuando se agrega una conexión, se agrega un nuevo gen de conexión al final del genoma y se le da el siguiente número de innovación disponible. En el agregado de nodos, el gen de conexión que se divide es deshabilitado, y se agregan dos nuevos genes al final del genoma. El nuevo nodo está entre las dos nuevas conexiones. Un nuevo gen de nodo (no mostrado en la figura) representando al que se agregó, se incorpora en el genoma también.

Un problema posible es que la misma innovación estructural recibirá diferentes números de innovación en la misma generación si esta aparece más de una vez. NEAT lo soluciona, manteniendo una lista de innovaciones que ocurrieron en la generación actual. Así es posible asegurarse que, si la misma estructura surge más de una vez en mutaciones independientes en la misma generación, cada mutación idéntica recibirá el mismo número de innovación. Por lo tanto, no existirá una explosión de números de innovación, aunque estructuras generadas en distintas generaciones pueden ser iguales pero con distinto número de innovación.

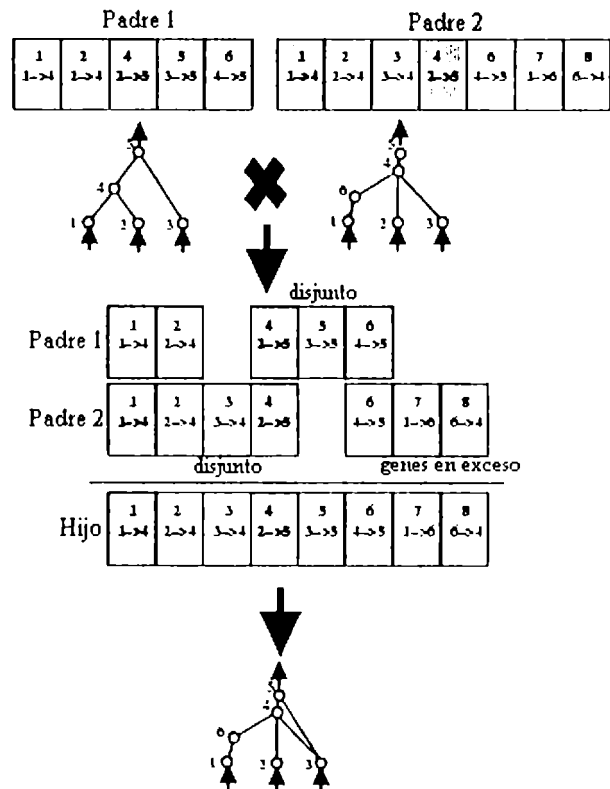
Según Stanley, las marcas históricas le proporcionan al método NEAT una nueva y poderosa capacidad de resolver el problema de tener distintas representaciones de una misma red. Pero esto no es así ya que pueden existir dos genomas distintos que dan origen a una misma red neuronal. Para el siguiente ejemplo supongamos que se codifican redes con 3 neuronas de entrada (1, 2 y 3) y una de salida (4):

Utilicemos los genomas G1 y G2 construidos como se menciona en el Ejemplo 18.

Estos genomas pueden ser generados por mutaciones independientes en una misma generación o en el transcurso de varias. Las redes que codifica cada uno no son estrictamente idénticas en cuanto a los identificadores de las neuronas, pero si lo son en términos de funcionamiento, es decir, con las mismas entradas generan salidas idénticas, como se muestra en la Figura 60.

El sistema utiliza los números de innovación para saber que genes coinciden con cuales (Figura 59). Cuando se realiza el crossover, los genes en ambos genomas con el mismo número de innovación son alineados. Esos genes son llamados genes *coincidentes*. Los genes que no coinciden son llamados genes *disjuntos* o sino genes *en exceso*,

dependiendo si están dentro o fuera del rango de los números de innovación del otro padre. Éstos representan estructura que no está presente en el otro genoma. Para componer la descendencia se eligen al azar los genes coincidentes entre ambos padres y se incluyen los genes disjuntos y en exceso del padre con mayor fitness. El autor del método infiere que las marcas históricas le permiten a NEAT realizar el crossover usando genomas lineales de tamaño variable sin la necesidad de análisis topológicos costosos.



**Figura 59:** Coincidencia de genomas de diferentes topologías de red usando números de innovación. Aunque los padres se ven diferentes entre sí, sus números de innovación (mostrados en el tope de cada gen) nos indican que genes coinciden con cuales. Aun sin ningún análisis topológico, puede crearse una nueva estructura que combina las partes coincidentes de los dos padres como también sus partes diferentes. Los genes que coinciden son heredados aleatoriamente, mientras que los genes disjuntos (los que no coinciden en el medio) y los genes en exceso (los que no coinciden en el final) son heredados del padre de mejor fitness. En este caso, se asume igual fitness, entonces los genes disjuntos y en exceso se heredan al azar.

G1									
Conexión	1 → 4	2 → 4	3 → 4	1 → 5	5 → 4	2 → 5	3 → 6	6 → 4	2 → 6
Peso	3	2	4	3	1	2	4	1	3
Habilitada	No	No	No	Si	Si	Si	Si	Si	Si
Innovación	1	2	3	4	5	6	7	8	9
G2									
Conexión	1 → 4	2 → 4	3 → 4	3 → 7	7 → 4	2 → 7	1 → 8	8 → 4	2 → 8
Peso	3	5	4	4	1	3	3	1	2
Habilitada	No	No	No	Si	Si	Si	Si	Si	Si
Innovación	1	2	3	10	11	12	13	14	15

**Ejemplo 18:** Definición de dos genomas NEAT que representan la misma red neuronal.

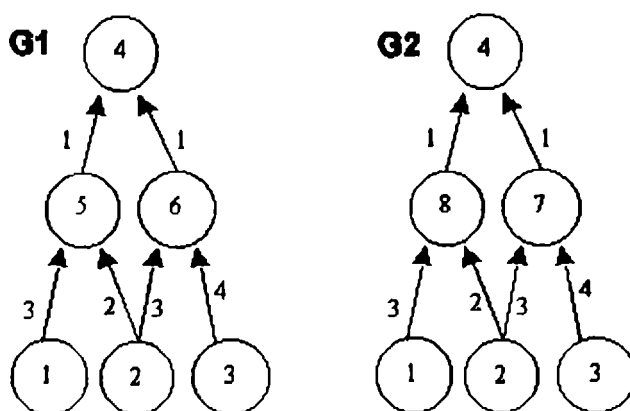


Figura 60: Redes generadas por los genomas NEAT del Ejemplo 18.

Luego de analizarlo detenidamente llegamos a la conclusión que el crossover resulta equivalente a una mutación de pesos en el padre con mejor fitness, ya que toda la estructura del genoma resultante de la aplicación de este operador es idéntica a la de uno de los padres con algunos pesos diferentes. Más adelante se comprueba que efectivamente es así ya que pruebas realizadas sobre el método muestran como puede resolver problemas sin utilizar el crossover.

Mediante el agregado de nuevos genes a la población y un crossover razonable entre genomas que representan diferentes estructuras, el sistema puede formar una población de diversas topologías. De todas maneras, tal población no puede mantener por sí misma las innovaciones topológicas. Como las estructuras más pequeñas se optimizan más rápido que las más grandes, y el agregado de nodos y conexiones por lo general decrementa inicialmente el fitness de la red, las estructuras recientes tienen poca esperanza de sobrevivir más de una generación aunque las innovaciones que representan pueden ser cruciales para resolver la tarea en el largo plazo.

### 7.3.3. Protección de la innovación mediante la especiación

La especiación de la población permite a los organismos competir dentro de sus propios nichos en lugar de hacerlo contra toda la población. De esta manera, las innovaciones topológicas son protegidas en un nuevo nicho donde las redes tengan tiempo de optimizar su estructura mediante la competencia interna. La idea es dividir la población en especies formadas por topologías similares.

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \overline{W}$$

**Ecuación 15:** Formula de la compatibilidad entre dos genomas NEAT. Combinación lineal simple de los números de genes en exceso ( $E$ ) y disjuntos ( $D$ ), como así también el peso promedio de las diferencias entre los genes que coinciden ( $\overline{W}$ ) incluyendo los genes deshabilitados. Los coeficientes  $c_1$ ,  $c_2$  y  $c_3$  permiten ajustar la importancia de los tres factores y  $N$ , número de genes en el genoma más grande, normaliza el tamaño de los genomas.

En NEAT se implementa la especiación aplicando una técnica de fitness sharing [87], basándose en el marcado histórico mencionado anteriormente.

Para poder aplicar fitness sharing, se define una función de compatibilidad entre genomas de la siguiente manera:

El número de genes en exceso y disjuntos entre un par de genomas es una medida natural de su distancia de compatibilidad. Los dos genomas más disjuntos son los que menos historia evolutiva comparten, y por lo tanto son los menos compatibles. Entonces, podemos medir la distancia de compatibilidad  $\delta$  de diferentes estructuras mediante la Ecuación 15.

La medida de distancia  $\delta$  le permite a NEAT llevar a cabo la especiación usando un umbral de compatibilidad  $\delta_c$ . Se mantiene una lista ordenada de especies. En cada generación, los genomas son secuencialmente ubicados en especies. Cada especie existente es representada por un genoma aleatorio dentro de las especies provenientes de la *generación previa*. Un genoma dado  $g$  en la generación actual es ubicado en la primera especie en la cual  $g$  es compatible con el genoma representativo de esa especie. De esta manera, las especies no se solapan unas con otras. Si  $g$  no es compatible con ninguna especie existente, se crea una nueva especie con  $g$  como representativo.

Como mecanismo de reproducción de NEAT se utiliza *fitness sharing explícito* [38], donde los organismos en las mismas especies deben compartir el fitness de su nicho. Por lo tanto, una especie no puede permitirse un crecimiento muy grande aun si muchos de sus organismos resuelven bien la tarea. Más aún, para cualquier especie es improbable que tome la población entera, lo cual es crucial para que trabaje la evolución con especiación. El fitness ajustado  $f'_i$  para un individuo  $i$  es calculado de acuerdo a su distancia  $\delta$  de cada organismo  $j$  como lo muestra la Ecuación 16.

$$f'_i = \frac{f_i}{\sum_{j=1}^n sh(\delta(i, j))}$$

**Ecuación 16:** Cálculo del fitness sharing para un individuo  $i$ . La función de sharing  $sh$  devuelve 0 cuando la distancia  $\delta(i, j)$  está sobre el umbral  $\delta_c$ , de otra forma,  $sh(\delta(i, j))$  devuelve 1.

Como se puede observar en la Ecuación 16,  $\sum_{j=1..n} sh(\delta(i, j))$  reduce el número de individuos similares a  $i$  en la misma especie. Esta reducción es natural ya que las especies se construyen por compatibilidad usando el umbral  $\delta_c$ . A todas las especies se les asigna una cantidad potencialmente diferente de organismos de la descendencia en proporción a la suma de los fitness ajustados de sus miembros  $f'_i$ . Las especies entonces se reproducen eliminando primero los miembros de más bajo desempeño de la población. La población entera es entonces reemplazada por la descendencia de los organismos restantes en cada especie.

Volvamos al Ejemplo 18 de los dos genomas G1 y G2 que representan la misma red. Estos pueden ser asignados a distintas especies aun cuando ambos representan la misma estructura. El problema de esto es que dos redes iguales al ser representadas por genomas cuya compatibilidad no se encuentra por debajo del umbral, quedan ubicadas en distintas especies. Entonces se pierde la idea de las innovaciones topológicas, ya que se espera que dos redes que provienen de distintas especies tengan una estructura diferente.

Según el autor del método el efecto de la especiación de la población es que las innovaciones topológicas sean protegidas.

### **7.3.4. Minimización de la dimensionalidad mediante el crecimiento incremental a partir de estructura mínima**

En el momento de crear la población inicial NEAT busca espacios de dimensiones minimales comenzando con una población *uniforme* sin nodos ocultos (es decir, todas las entradas conectadas directamente a las salidas). Como las nuevas estructuras son introducidas incrementalmente cuando ocurren mutaciones estructurales, y solo sobreviven las estructuras que resulten útiles durante la evaluación del fitness, Stanley asegura que los espacios de búsqueda son mínimos. Como la población comienza mínimamente, las dimensiones del espacio de búsqueda es minimizado, y NEAT siempre busca dentro de dimensiones más pequeñas que otros métodos y de neuroevolución de topología fija donde ya hay una cantidad de neuronas ocultas preestablecidas desde el comienzo de la evolución. Pero en realidad para poder decir que comienza con el mínimo espacio de búsqueda posible, el inicio debería ser con redes sin conexiones y que el propio método explore el espacio en busca de las conexiones adecuadas.

### **7.3.5. Problemas resueltos con NEAT**

Los creadores del método evaluaron el desempeño del mismo en dos problemas clásicos: el problema del XOR y el del péndulo doble sin velocidades. Con el primero intentan determinar la capacidad de NEAT de crear estructura necesaria para un problema, dado que una posible solución al problema XOR requiere de una red con una neurona oculta.

A través del segundo problema se probó la capacidad de NEAT en la resolución de problemas de control complejos.

### **7.3.6. Análisis de NEAT**

El método NEAT ha sido aplicado a la resolución del problema del péndulo doble sin velocidades y sometido por los autores a una serie de pruebas, en las cuales evalúan el método quitándole alguna de sus características. Las pruebas han sido las siguientes:

- Sin la posibilidad de crecimiento de la estructura de las redes con un 20% de efectividad en resolver el problema.
- Sin clasificación por especies con una efectividad de 75%.
- Iniciando con una población aleatoria en vez de uniforme con un 95% de efectividad.
- Sin aplicar crossover con un 100% de efectividad, situación que requiere mayor tiempo de evolución.

De estos datos se puede inferir que NEAT dejaría de resolver muchos problemas sin la mutación de estructura de las redes ya que al iniciar con una población uniforme y mínima pierde la posibilidad de introducir neuronas ocultas o conexiones recurrentes. Entonces no podría resolver un problema como el XOR. Además el hecho que el crossover no afecte en la efectividad del método no habla muy bien del mismo ya que este operador fue específicamente diseñado para este esquema.

Debido a la política de búsqueda en espacios mínimos en conjunto con la clasificación en especies, el método se parece más a una búsqueda de los pesos óptimos de

conexiones en paralelo, ya que cada especie encierra su propia topología. Aplicar crossover no hace más que una alteración de pesos en uno de los padres y al incorporar innovaciones topológicas importantes un nuevo individuo pasa a pertenecer a una especie distinta a la de sus padres y se combinará con individuos similares, intercambiando los pesos de conexión. Pero el método carece de capacidad de combinar estructuras de redes neuronales dejando la tarea de crear nuevas topologías al operador de mutación.

El diseño de éste operador no permite crear fácilmente una capa oculta ya que para agregar una neurona oculta que integre todas las entradas en una red con  $n$  neuronas de entrada y  $m$  de salida debe ocurrir una mutación de agregado de neurona entre una neurona de entrada y otra de salida; y luego  $n-1$  mutaciones que conectan esta nueva neurona con cada una de las entradas más  $m-1$  mutaciones que la conecten con la salida. En una red con solamente 5 neuronas de entrada y 3 de salida deben darse las mutaciones correctas. Esto es, deben ser 7: una para agregar la neurona, 4 para conectarla con las entradas restantes y 2 para conectarla con las salidas restantes.



## 8. Herramienta de Neuroevolución

En el presente capítulo detallaremos el ambiente de neuroevolución desarrollado para llevar a cabo experimentos con el fin de obtener resultados sobre el comportamiento de los métodos descritos anteriormente. También especificaremos aspectos relacionados con el diseño e implementación de la aplicación y como se obtuvo un framework fácilmente adaptable para desarrollar y evaluar nuevos métodos evolutivos, no sólo aplicados a redes neuronales sino a la evolución en general. Incluimos en este capítulo una descripción del diseño realizado para adaptar este modelo a una paralelización. Finalmente brindaremos al lector una breve ayuda acerca del uso de la herramienta de software *Win Evolution*, cuyo ejecutable y código acompañan a este trabajo, la cual constituye una interface gráfica que permite utilizar y poner en funcionamiento el framework en cuestión.

### 8.1. Motivación

Desde un comienzo el objetivo consistió en desarrollar una biblioteca de funciones que facilitaran la implementación de un método evolutivo, permitiendo a quien lo desarrolle enfocar la atención en el método propiamente dicho y no en aspectos relacionados con la programación del mismo. Esto aporta ventajas cuando se interesa experimentar, por ejemplo, con un nuevo operador de crossover aplicado a un Algoritmo Genético Simple, con cadenas de bits y mutación clásica. Lo ideal es implementar únicamente ese nuevo operador y combinarlo en un algoritmo previamente desarrollado y depurado. Esta idea se puede aplicar además de los operadores, al método de selección, el cromosoma, la función objetivo e incluso el mismo algoritmo genético.

Debido a la necesidad de reutilizar partes del código existentes, decidimos desarrollar nuestro software en el marco del paradigma de la programación orientada a objetos, transformando la idea inicial de una biblioteca de funciones en un **Framework de Evolución**.

Por otro lado, existen ventajas al paralelizar partes del algoritmo evolutivo, por lo que resultó útil modelar al framework pensando en una posible ejecución en una arquitectura multiprocesador. Debido a esto, fue conveniente que este desarrollo sea portable entre distintos sistemas operativos, permitiendo distribuir el procesamiento entre máquinas heterogéneas.

### 8.2. Diseño del Framework de Evolución

Como hemos mencionado anteriormente, utilizaremos la programación orientada a objetos para el desarrollo de la herramienta. Supondremos que el lector está familiarizado con los conceptos del paradigma y avanzaremos sobre la descripción de las clases que dan forma al framework.



### 8.2.1. Clases principales

En el Diagrama UML 1 hemos detallado las clases más importantes que caracterizan al desarrollo explicado en este capítulo. En éste se puede ver en detalle la clase **TEvolutionaryAlgorithm**, la cual captura el comportamiento general de un algoritmo evolutivo. El mismo actúa sobre una población modelada en **TPopulation** la cual contiene objetos que tienen el comportamiento especificado en **TEvolvable**. Esta clase abstrae todo el comportamiento de un individuo dentro de un algoritmo genético. Como tal debe ser capaz entre otras cosas de ejecutar mutaciones y combinaciones con otros individuos de la población, permitir la asignación de fitness, etc. Esta abstracción permite la implementación de diferentes alternativas en cuanto a la representación de soluciones.

En cualquier estrategia evolutiva el algoritmo puede claramente descomponerse en varias etapas bien definidas, como se vió en 3.3.1.

Estas etapas fueron modeladas en las siguientes clases:

- **TInitializationMethod**: Se encarga de la inicialización de la población del algoritmo.
- **TSelectionMethod**: Su función es capturar el criterio de selección de los individuos a reproducir para crear la nueva generación.
- **TEvaluationMethod**: Realiza la evaluación de las posibles soluciones. Esta clase interactúa con objetos de la clase **TTester** y provee una abstracción que permite modelar el problema a resolver, desacoplándolo del método de evaluación a utilizar.
- **TTerminationMethod**: Método que determina cuando el algoritmo llegó a su fin. Se evalúa en cada generación.

### 8.2.2. Especialización de los métodos

En la subsección anterior describimos cuatro componentes importantes en el algoritmo evolutivo dentro de nuestro framework. Estos imponen criterios acerca de la inicialización de la población, selección de los organismos a reproducir, evaluación de las posibles soluciones y finalización del algoritmo.

En esta sección describiremos las diferentes alternativas implementadas para cada uno de esos métodos.

El Diagrama UML 2 muestra la jerarquía de clases de **TInitializationMethod**. En esta se pueden apreciar dos especializaciones, es decir dos maneras diferentes de inicializar la población. Una inicializa la población con individuos creados completamente al azar (**TRandomInitialization**) mientras que la otra (**TUniformInitialization**) genera una población de individuos similares a un prototipo pero con algunas alteraciones.

Otro método especializado es el de selección, el cual está modelado de manera abstracta en la clase **TSelectionMethod**. La jerarquía correspondiente se puede observar en el Diagrama UML 3. Hemos implementado tres estrategias: Método de selección por rueda de ruleta (**TRWSMethod**), Stochastic Universal Sample (**TSUSMethod**) y Stochastic Remainder Sample (**TSRSMMethod**). Estas tres estrategias se encuentran explicadas en el correspondiente capítulo de evolución incluido en el presente trabajo (ver sección. 3.4.4)

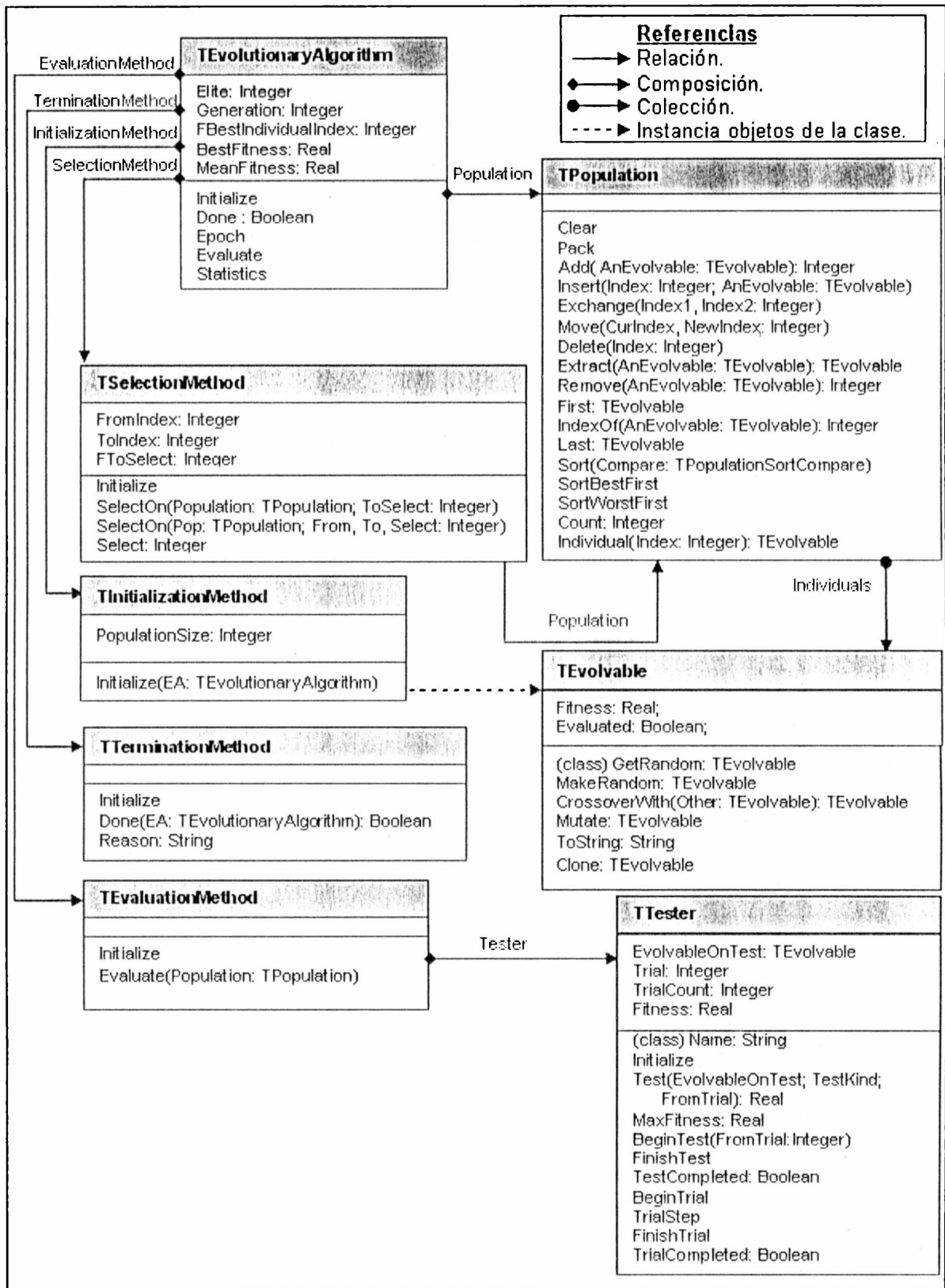


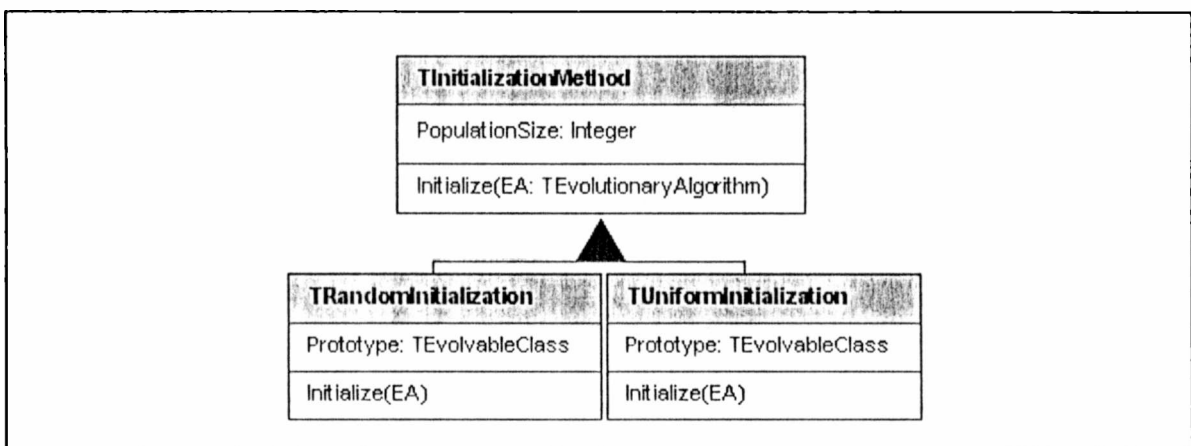
Diagrama UML 1: Clases principales del framework de evolución.

El método de evaluación (**TEvaluationMethod**) se subclasificó en dos alternativas. Una de ellas (**TSimpleEvaluator**) evalúa todos los individuos de una población en el problema a resolver de manera secuencial, es decir, uno a continuación del otro. En cambio la alternativa propuesta en **TParallelEvaluator** permite una evaluación en paralelo de varios individuos a la vez en una arquitectura multiprocesador. Fácilmente se puede ver que en evaluaciones costosas, la segunda opción es más conveniente si se dispone de la arquitectura adecuada. Más adelante, en este mismo capítulo brindaremos detalles acerca de la paralelización que llevamos a cabo. El Diagrama UML 4 muestra gráficamente la subclasificación de la jerarquía **TEvaluationMethod**.

Finalmente el método de finalización del algoritmo (**TFinalizationMethod**) ofrece varias alternativas la cuales se pueden ver en el Diagrama UML 5:

- **TMaxGenerationReached**: Utilizando este método de terminación, el algoritmo se ejecuta hasta alcanzar un determinado número de generaciones.
- **TMaxFitnessReached**: El algoritmo finalizará cuando el mejor individuo de la población se mantenga por sobre un determinado fitness una cierta cantidad de generaciones predefinida de antemano.
- **TTesterMaxFitnessReached**: Si se puede fijar un fitness máximo para un problema, la utilización de este método permitirá que el algoritmo se detenga cuando el mejor individuo se mantenga un número predeterminado de generaciones con el fitness por sobre dicha marca.
- **TCompositeMethod**: Permite una combinación de cualquiera de los tres métodos anteriores. La evaluación de la condición de terminación se realiza relacionando todos los resultados de los métodos que lo componen con el operador OR.

Hemos descrito hasta aquí las clases principales del framework de evolución en conjunto con especializaciones en los componentes en los cuales dividimos al algoritmo evolutivo. El lector puede observar que fácilmente se pueden combinar diferentes alternativas de los mismos dando origen a diferentes algoritmos evolutivos. Más aún, es sencillo probar nuevas estrategias en cada método debido a la modularidad del diseño realizado.



**Diagrama UML 2: Alternativas para el método de inicialización de un algoritmo evolutivo.**

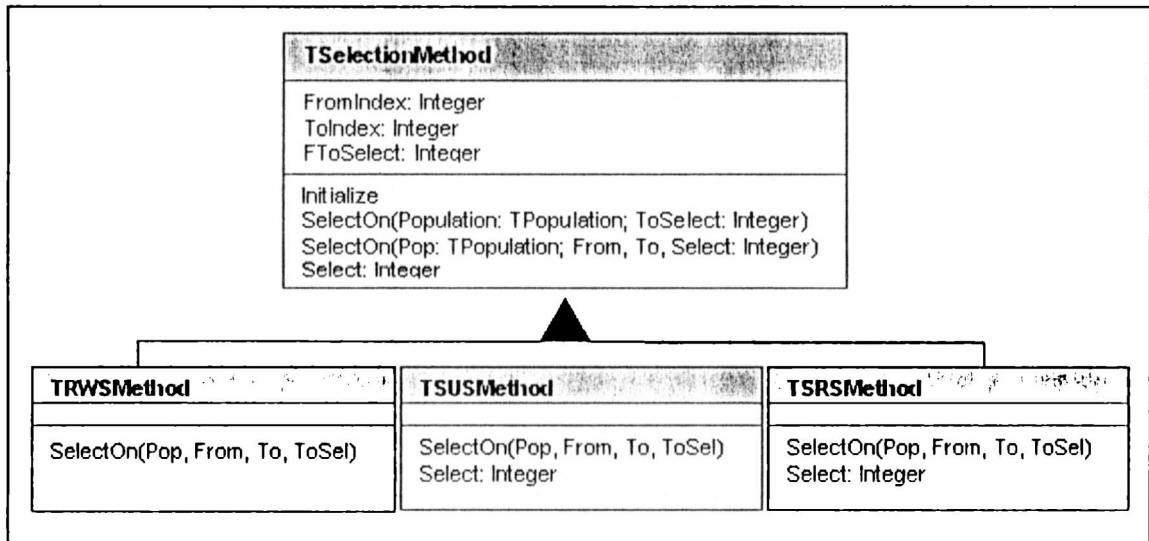


Diagrama UML 3: Diferentes estrategias de selección de individuos para la reproducción.

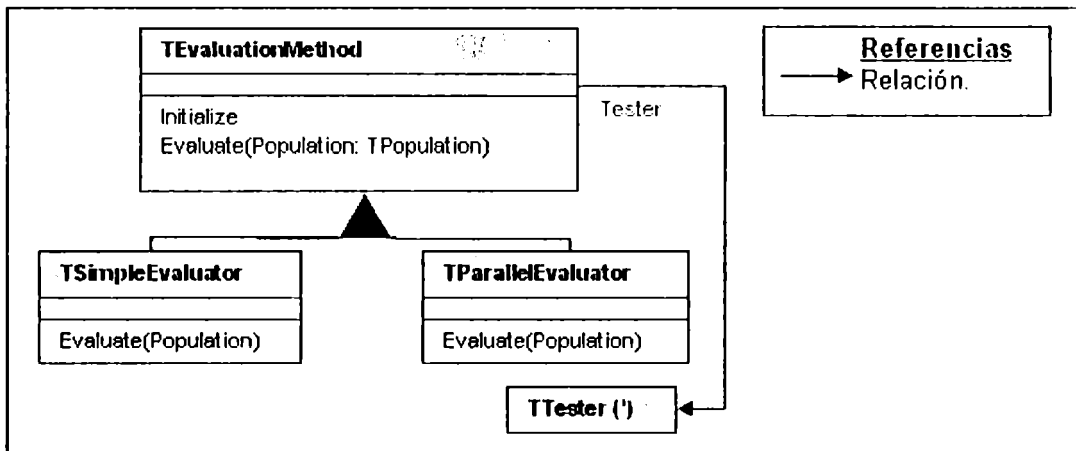


Diagrama UML 4: Especialización de la jerarquía de TEvaluationMethod.

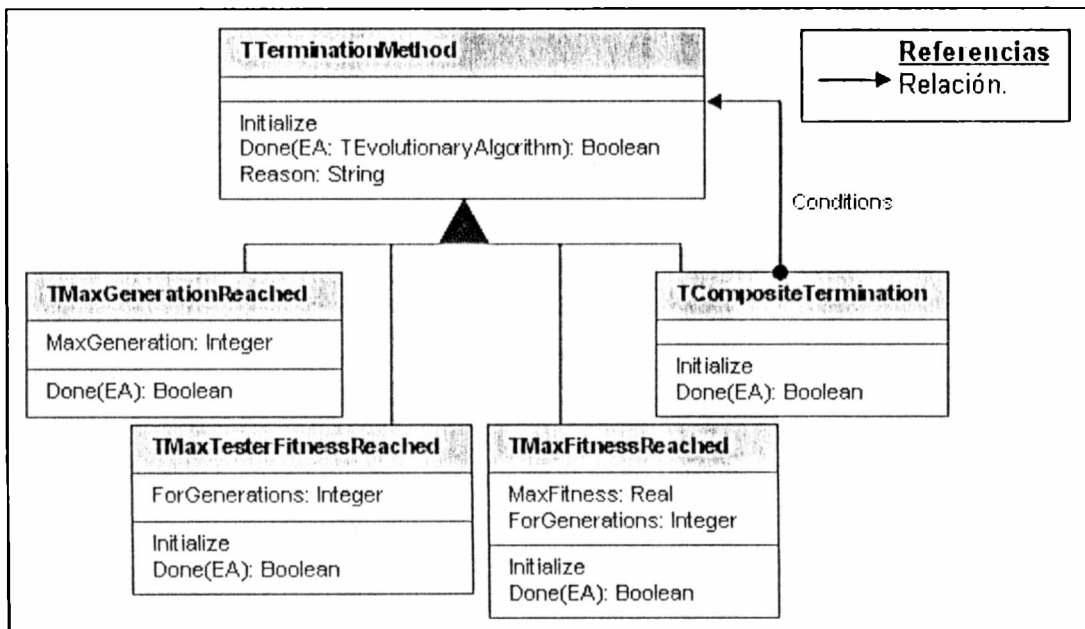


Diagrama UML 5: Criterios de terminación del algoritmo evolutivo.

### 8.2.3. Extensiones para neuroevolución

El framework presentado hasta el momento tiene características que permiten correr un algoritmo evolutivo en su aspecto más general, pero no mencionamos nada que nos lleve a ejecutar un proceso de neuroevolución.

El Diagrama UML 6 muestra como se extiende la jerarquía de **TTester** descrita en secciones anteriores para soportar la evaluación de redes neuronales. La clase **TNeuralNetTester** implementa el comportamiento de una especie de ambiente de pruebas para redes neuronales e interactúa con objetos de la clase **TNeuralNet**. Esta clase encapsula el comportamiento de una red neuronal, permitiendo setear valores en sus entradas, evaluarla y obtener valores en sus salidas. En la figura se puede observar como interactúa con las neuronas (**TNeuron**) y con las conexiones (**TLink**). De esta manera un **TNeuralNetTester** entiende que tiene que evaluar un **TNeuralNet** y por lo tanto le proporciona entradas de acuerdo al problema, invoca una evaluación de la red e interpreta la salida aplicándola al ambiente.

Como hemos visto anteriormente, el algoritmo evolutivo solo trabaja con objetos de la clase **TEvolvable**, los cuales no pertenecen a la jerarquía de **TNeuralNet**. Por lo tanto se requiere una acción intermedia implementada en la clase **TNeuralNetBuilder**. Esta acción consiste en obtener a partir de un **TEvolvable** su correspondiente **TNeuralNet**. Entonces, un **TNeuralNetTester** que por compatibilidad de interfaces con el resto del framework recibe objetos **TEvolvable** para evaluar, necesita de la colaboración de un **TNeuralNetBuilder** que ejecute el paso de la construcción del **TNeuralNet** correspondiente, sin alterar en absoluto el resto de las clases.

Esta descomposición permite que para introducir una nueva representación de RNA en un algoritmo evolutivo resulte suficiente con una clase que representa al individuo en la población y que pertenecerá a la jerarquía de **TEvolvable** y con otra que convierta la anterior en un **TNeuralNet** y que estará en la jerarquía de **TNeuralNetBuilder**.

Además, puede extenderse la jerarquía de **TTester** para implementar un tester específico de cualquier otro objeto distinto de un **TNeuralNet**.

### 8.2.4. Implementación de los problemas a resolver

Ya presentamos las clases **TTester** y su subclase **TNeuralNetTester** en forma general, sin mostrar como implementar los problemas en los que evalúan a los individuos.

La clase **TTester** descrita anteriormente tiene una variedad de métodos para ser reimplementados por sus subclases de manera de adaptarse a prácticamente cualquier tipo de problema. Estos métodos son los que interesan a la hora de agregar un nuevo problema, como ser:

- *MaxFitness*: donde se especifica si es que existe un fitness máximo para el problema. Por defecto es el valor real más grande que se puede representar.
- *BeginTest*: Ejecuta acciones antes de iniciar el test de un individuo
- *FinishTest*: Ejecuta acciones cuando finalizó el test de un individuo
- *TestCompleted*: Condición que devuelve verdadero si se completó el test. Se ejecuta después de cada intento.
- *BeginTrial*: Ejecuta acciones antes de iniciar un intento dentro del test completo del individuo.



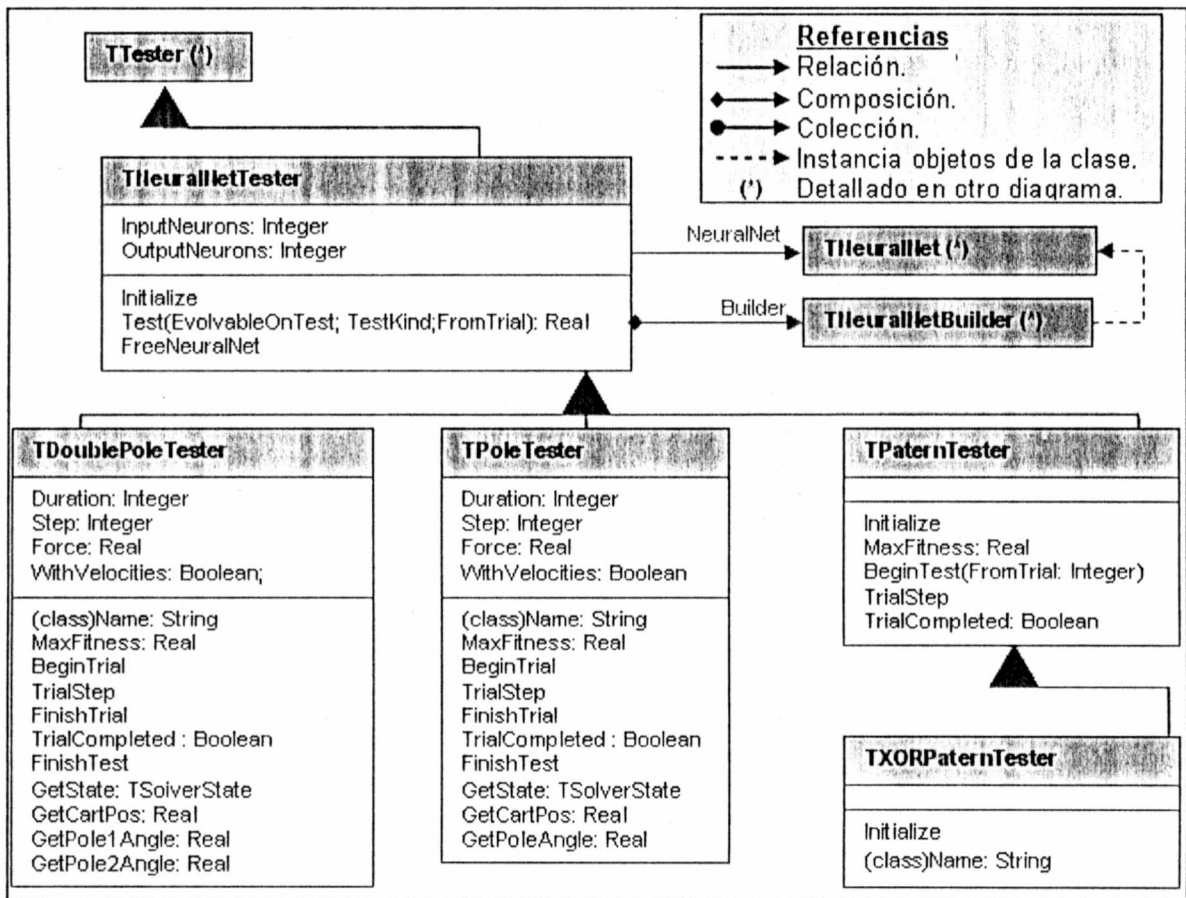


Diagrama UML 7: Detalle de la implementación de los diferentes problemas con los cuales se experimentó.

### 8.2.5. Extensión para neuroevolución con cadenas de bits

Para poder utilizar el framework que presentamos como soporte para ejecutar un método de neuroevolución basado en una representación directa mediante cadenas de bits es necesaria la implementación de tres clases.

La clase **TSimpleGA** que constituye un algoritmo evolutivo en su versión de algoritmo genético simple. Éste reemplaza de su superclase el método *Epoch*, el cual se encarga de generar la nueva población a partir de la población actual ya sometida a la evaluación.

Es necesario combinar con una población de objetos de la clase **TBitString** los cuales son una especialización de **TEvolvable** y contienen la representación del genoma de cadena de bits junto con la implementación de los operadores genéticos de crossover y mutación correspondientes.

Finalmente se completa la implementación de este método de evolución mediante el agregado al framework de una subclase de **TNeuralNetBuilder** que se encarga de obtener a partir de un objeto **TBitString** un objeto **TNeuralNet**. Esta clase es **TFromBitStringBuilder**. Todas las clases mencionadas en esta sección se encuentran detalladas en el Diagrama UML 8.

Los detalles de este método de neuroevolución mediante cadenas de bits fueron presentados en la página 109 del presente trabajo.

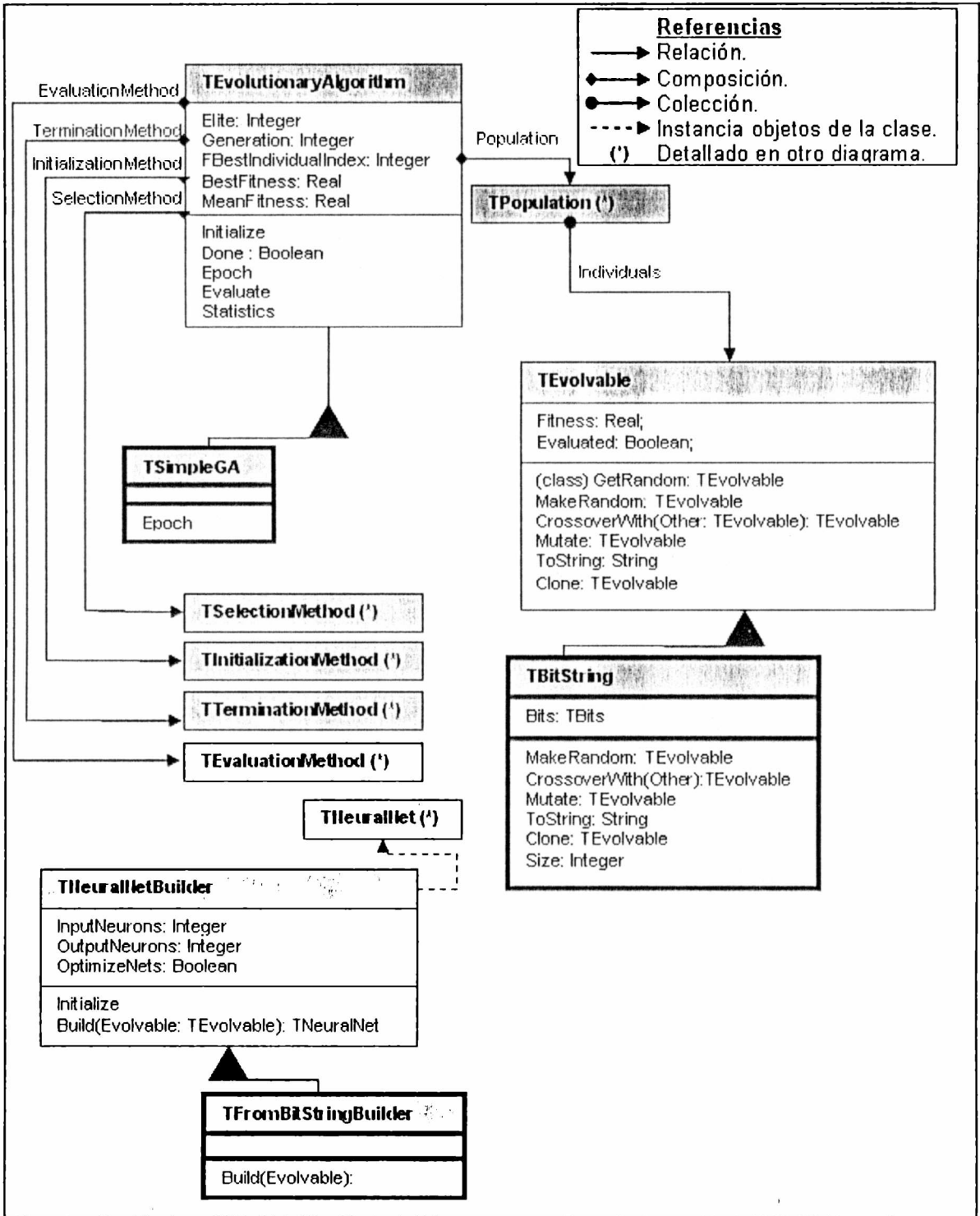


Diagrama UML 8: Las clases detalladas con líneas más gruesas corresponden a las clases necesarias para adaptar el framework a la neuroevolución mediante cadenas de bits.



## 8.2.6. Extensión para NEAT

En este caso nos encontramos ante un método que requiere una mayor cantidad de clases debido a que tiene una estrategia más compleja que en el caso anterior.

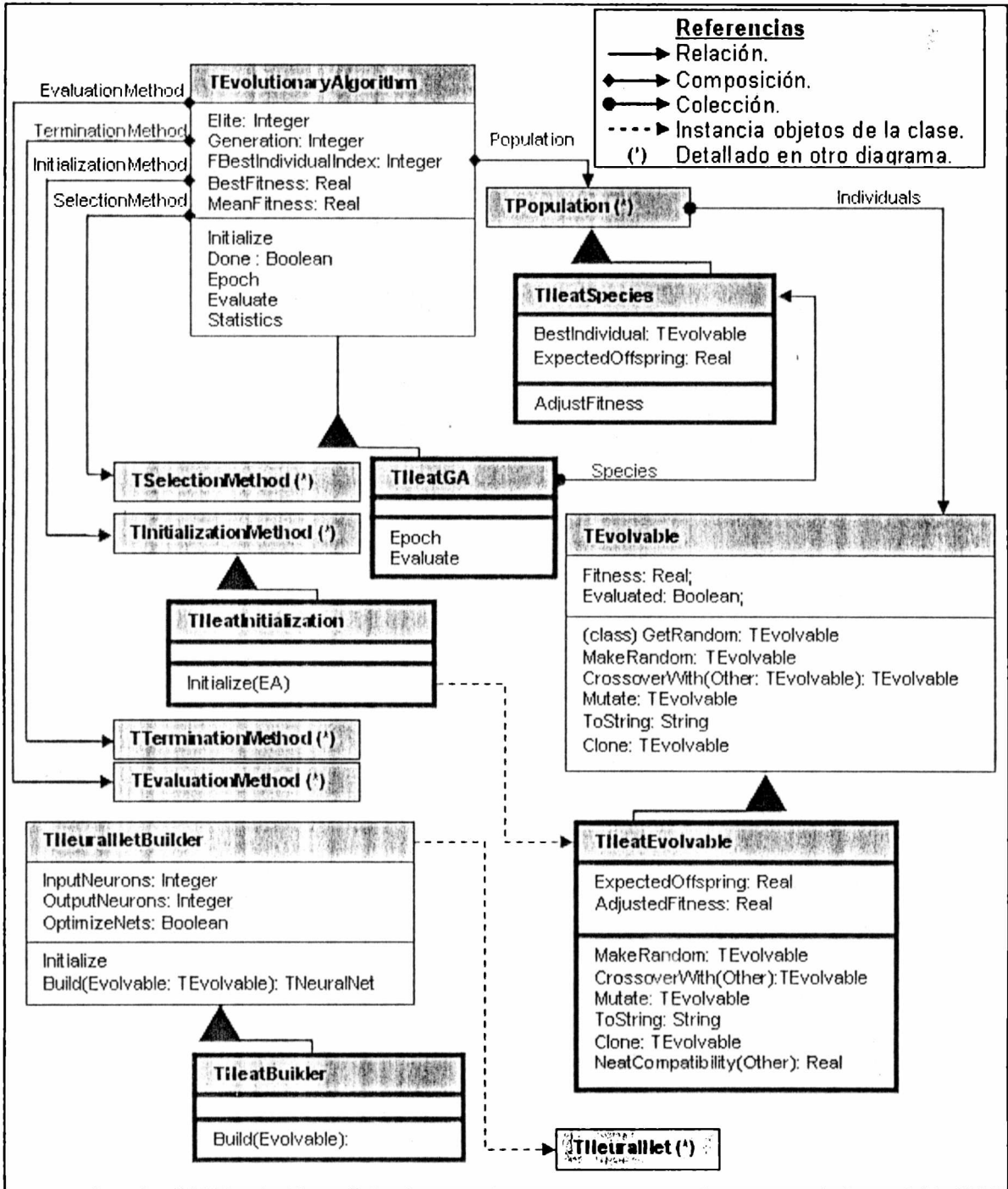


Diagrama UML 9: Las clases detalladas con líneas más gruesas corresponden a las clases necesarias para adaptar el framework al método NEAT.

Utiliza un método de niching [87] en el cual divide la población en especies a partir de una función de compatibilidad y calcula la descendencia esperada para cada especie mediante la técnica de fitness sharing.

Para implementar este método se requiere de un **TNeatGA** que extiende al algoritmo evolutivo debido a que requiere un manejo de las especies. Además se agrega una clase **TSpecies** que por ser una subpoblación tiene un comportamiento similar al de **TPopulation** con la diferencia que incluye funciones para el cálculo de la descendencia esperada y el fitness ajustado.

Además el método requiere una inicialización de la población de manera diferenciada. Dicha inicialización se implementa en la clase **TNeatInitialization**, la cual pertenece a la jerarquía de **TInitializationMethod**.

Finalmente como en el caso anterior es preciso definir la clase que hará las veces de individuo de la población y la clase que implementa el correspondiente constructor de redes neuronales. La primera es **TNeatEvolvable** y sobrescribe las operaciones de crossover y mutación agregando además métodos que permiten calcular la compatibilidad entre individuos. La segunda clase es **TNeatBuilder** y se encarga de interpretar un genoma representado en **TNeatEvolvable** y construir la RNA asociada.

El Diagrama UML 9 muestra las clases que fue necesario agregar para implementar el método NEAT el cual fue explicado anteriormente (ver Pág. 114)

### 8.2.7. Extensión para NeSR

Finalmente presentaremos las clases que fueron necesarias para poder ejecutar en el framework el método presentado en este trabajo.

Como estrategia evolutiva se utilizó la misma clase que en el método de cadenas de bits, es decir **TSimpleGA**, debido a que el método no requiere ninguna acción especial que no realice esta clase. También fue necesario definir la clase **TLSystem**, la cual representa un Sistema L de reescritura e implementa los operadores genéticos específicos de NeSR. Además incluye la capacidad de realizar el proceso de reescritura que caracteriza a estos sistemas.

Otra clase indispensable para la implementación de NeSR es **TCommandBasedNeuralNetBuilder**. Esta clase se encarga de interpretar el resultado de la reescritura de un **TLSystem** como una secuencia de comandos los cuales indican como construir la correspondiente **TNeuralNet**.

Estas clases se describen en el Diagrama UML 10 y como se podrá observar la clase **TLSystem** requiere de la colaboración de muchas otras clases.

Los objetos **TLSystem** representan Sistemas L de reescritura y como tales poseen un símbolo de inicio (*start*) y las reglas (*rules*).

La descripción del símbolo de inicio requiere un análisis de la jerarquía **TModule**. Esta jerarquía agrupa objetos que se comportan como módulos dentro de un Sistema L.

El Diagrama UML 11 muestra esta jerarquía. Se puede observar que cada módulo posee una definición (**TModuleDefinition**), la cual determina a que comando o referencia a regla de producción representa, los parámetros que dicha definición especifica, etc. Dependiendo lo que acompañe a la definición los módulos pueden ser con expresiones (**TModuleWithExpressions**) los cuales aparecen en el símbolo de inicio y en los sucesores de las reglas. Estos módulos contienen expresiones que pueden hacer referencia a los parámetros con que se define la regla que los contiene. La otra opción, siguiendo con la jerarquía son los módulos con valores (**TModuleWithValues**) los cuales son creados como resultado de la reescritura y representan los comandos con los

valores en sus parámetros listos para ser interpretados o bien referencias a reglas de producción con los valores en sus parámetros listos para ser reescritos. En el momento de la reescritura un **TModuleWithExpressions** de un sucesor es evaluado y genera el correspondiente **TModuleWithValues** en la cadena de reescritura.

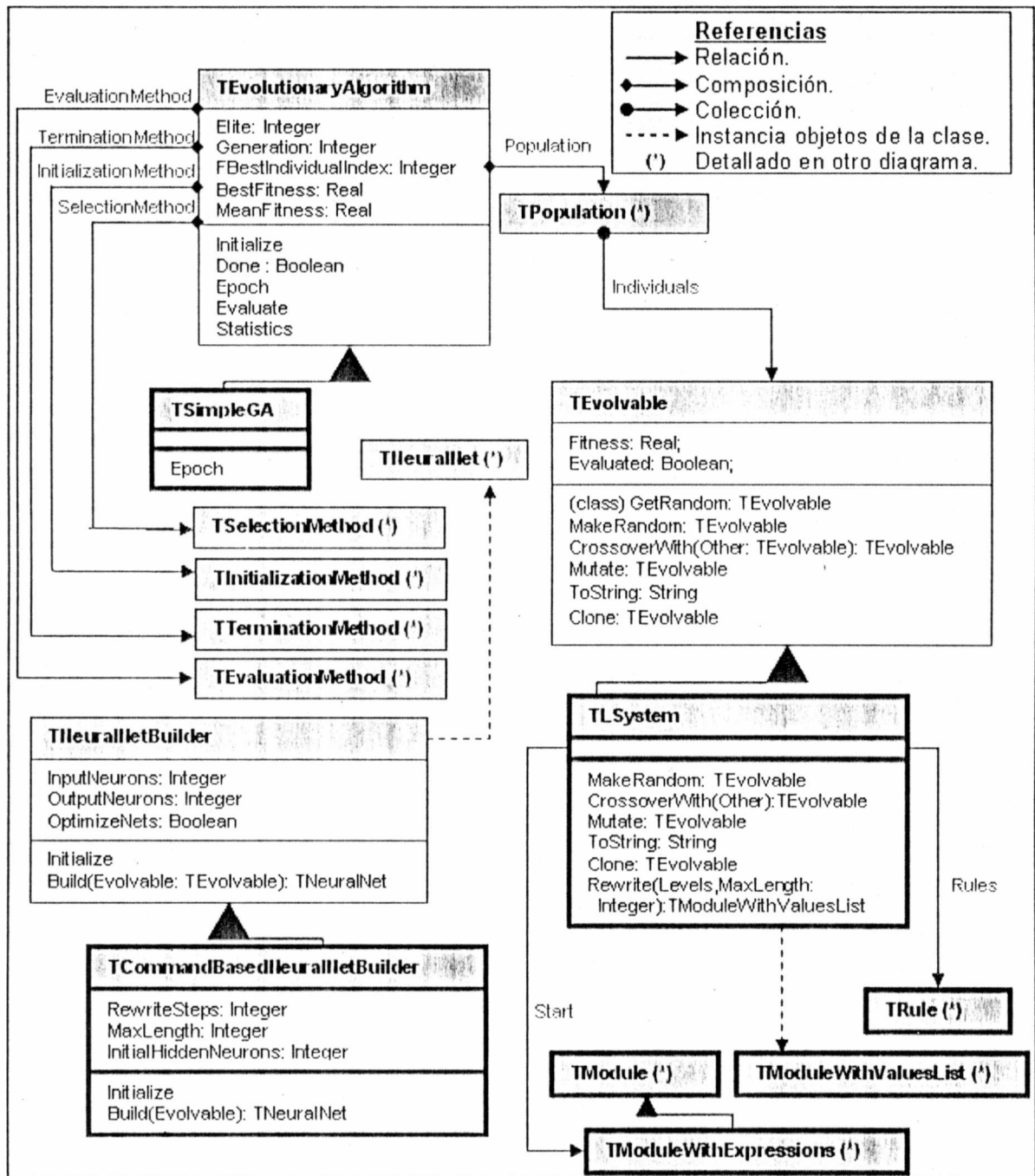
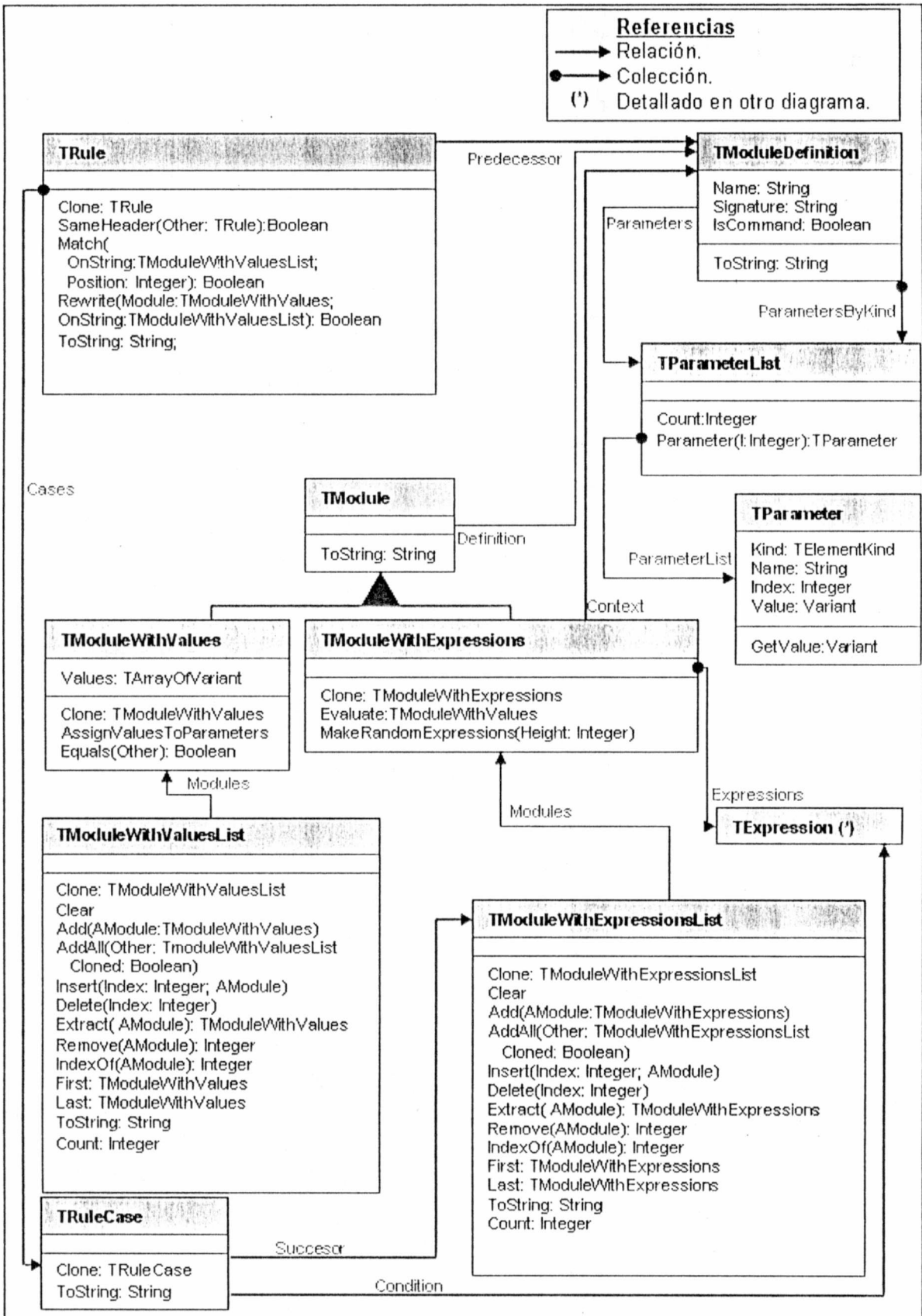


Diagrama UML 10: Las clases detalladas con líneas más gruesas corresponden a algunas de las clases necesarias para adaptar el framework al método NeSR.



**Diagrama UML 11: Clases más importantes que modelan a los Sistemas L.**

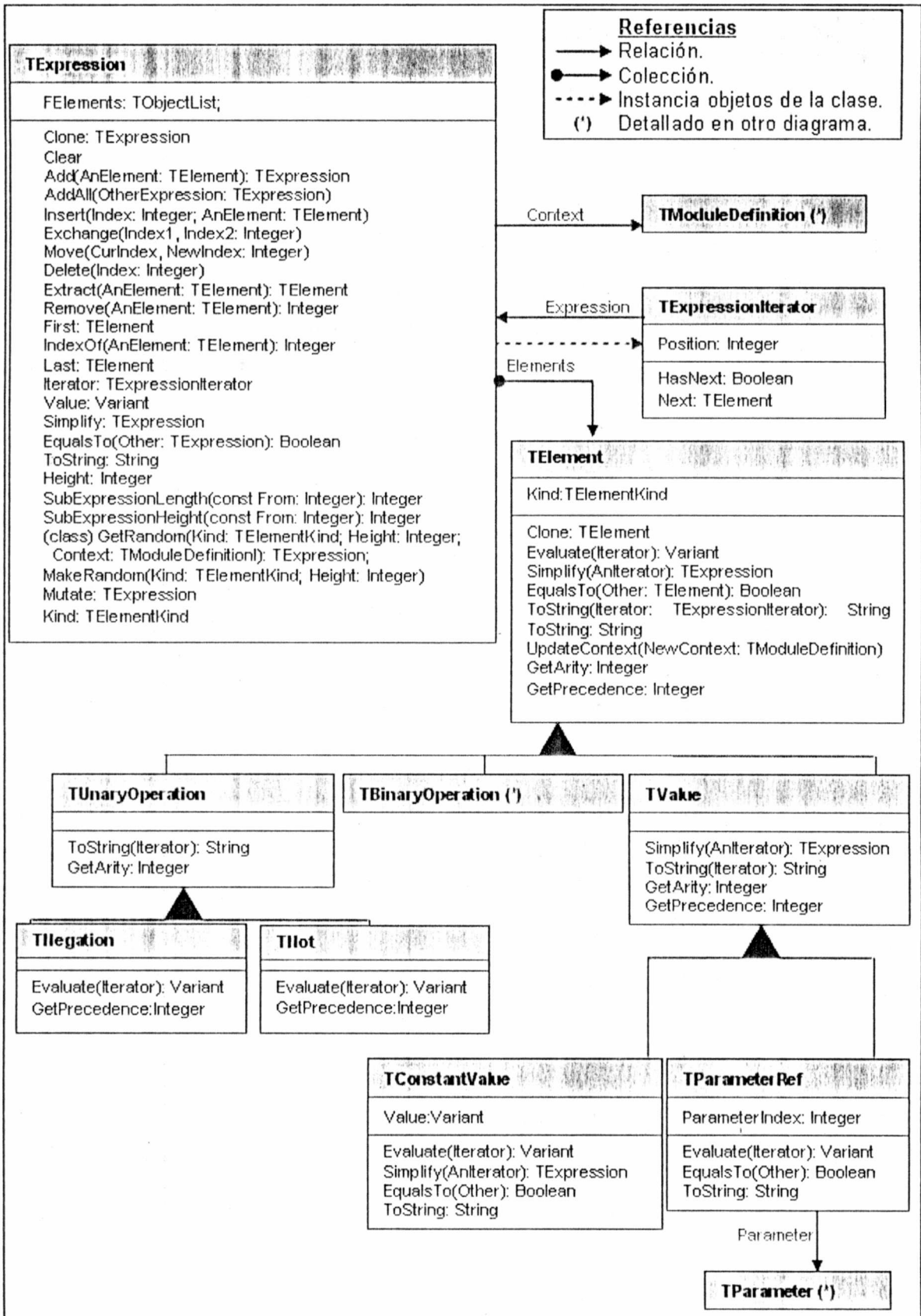


Diagrama UML 12: Detalle de la clase TExpression y los objetos con los que se relaciona.

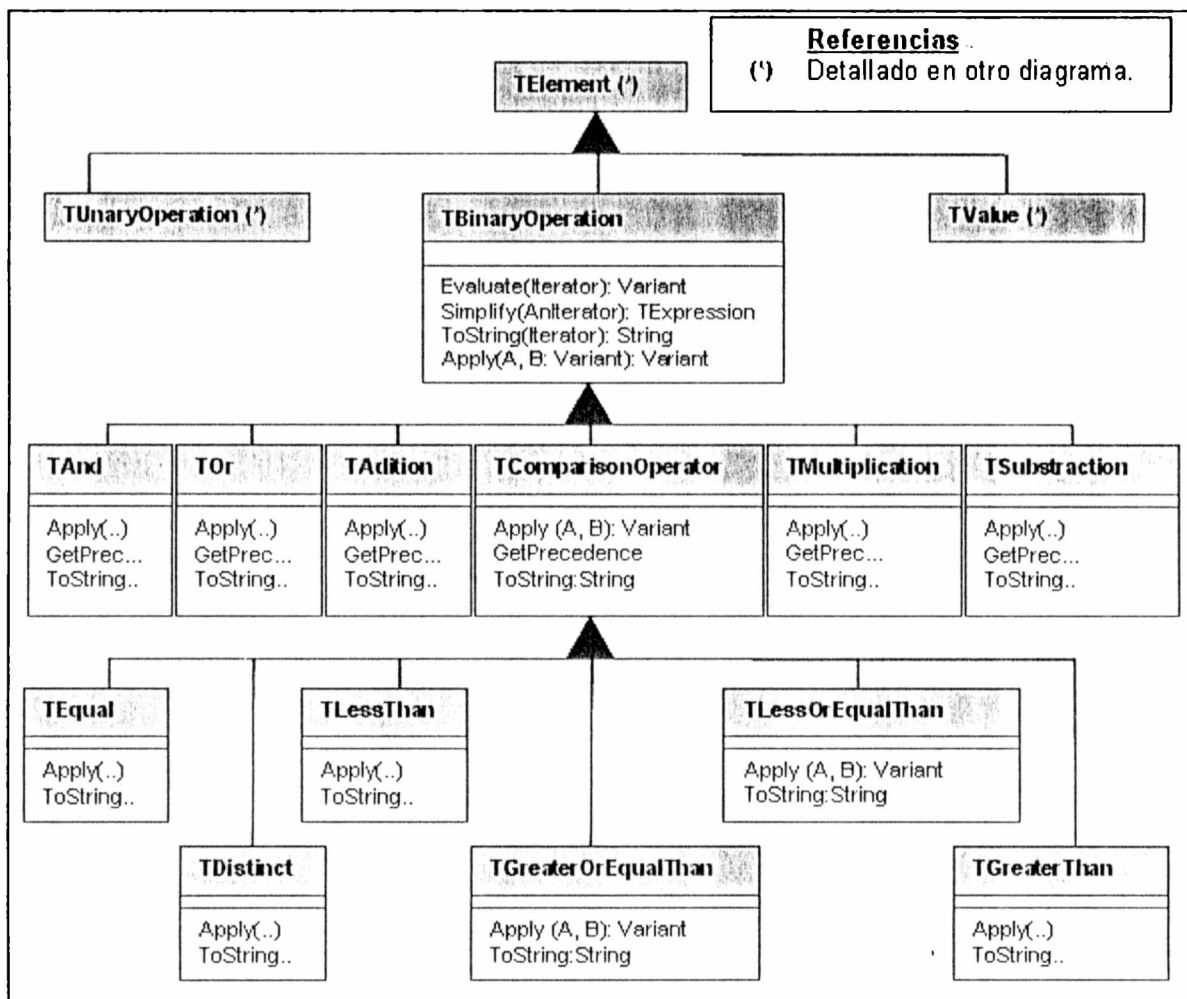


Diagrama UML 13: Especificación de los operadores binarios de expresiones.

El otro componente de **TLSystem** es una colección de objetos que se comportan como reglas de producción implementadas en la clase **TRule**. Esta clase se compone de un predecesor (*predecessor*) el cual es una definición de módulo y especifica la parte izquierda de la regla y una colección de casos de regla (*cases*) los cuales se implementan en la clase **TRuleCase**. Esta combina un objeto **TExpression** que representa la condición (*condition*) que se evalúa para determinar si se aplica o no dicho caso con un objeto **TModuleWithExpressionsList**. Este último se muestra como una lista de objetos **TModuleWithExpressions** y representa el sucesor (*successor*) del caso de regla o la parte derecha de la misma.

Finalmente debemos completar la descripción con la clase **TExpression** la cual es detallada en el Diagrama UML 12. Esta clase tiene la finalidad de representar expresiones aritméticas y lógicas mediante una colección de elementos (*elements*) los cuales son implementados por la clase **TElements** y representan los diferentes componentes de una expresión, como ser valores constantes (**TConstantValue**), referencias a parámetros de las reglas de los Sistemas L (**TParameterRef**), operadores lógicos (**TAnd**, **TOr** y **TNot**), aritméticos (**TNegation**, **TAddition**, **TSubstraction** y **TMultiplication**) y de comparación (**TEqual**, **TDistinct**, **TGreaterThan**, **TGreaterOrEqualThan**, **TLessOrEqualThan** y **TLessThan**). El detalle de los mismos se puede observar en el Diagrama UML 12 y en el Diagrama UML 13. Los

elementos son recorridos por un **TExpressionIterator** el cual, de acuerdo a la aridad y la precedencia de los operadores, determina el orden de evaluación de los mismos.

Con esto se completa la descripción de la composición de la clase **TLSystem**. El método para obtener un resultado de su reescritura (*rewrite*) el cual es invocado por el **TNeuralNetBuilder** correspondiente (**TCommandBasedNeuralNetBuilder**) y devuelve una **TModuleWithValuesList** que contiene módulos de la clase **TModuleWithValues** para luego interpretar los comandos y construir la RNA representada.

De esta manera se pudo integrar en un framework de evolución Sistemas L de reescritura que representan redes neuronales para dar forma a NeSR, sin alterar el resto del entorno y permitiendo así la posibilidad de utilizarlo para experimentar fácilmente otros métodos evolutivos.

### 8.3. Paralelización del algoritmo evolutivo

El proceso de simulación de evolución por computadora es potencialmente paralelizable. Si nos remitimos al caso de la evolución de organismos vivos en la realidad, todos los individuos se desenvuelven en su ámbito de manera completamente paralela. Esto puede ser mapeado a los algoritmos genéticos de manera de paralelizar la evaluación de cada uno de los individuos, ya que el proceso de selección y reproducción requiere un tiempo de procesamiento que puede considerarse despreciable. Esta división de trabajo tiene sentido realizarla si la tarea en la que se evalúan los individuos necesita de un costo de procesamiento considerable. Esto es debido a que no se divide el tiempo en forma lineal entre los procesadores ya que existe un costo de paralelizar que es la comunicación entre los distintos nodos de procesamiento. Por ejemplo, existen problemas de control con los cuales hemos experimentado que requieren un alto tiempo de CPU debido a que la simulación del ambiente incluye gran cantidad de cálculos matemáticos para emular el paso del tiempo.

En el ámbito en el cual desarrollamos nuestra herramienta tuvimos acceso a un *cluster* de computadoras<sup>6</sup>, con el cual se puede configurar una arquitectura multiprocesador tipo MIMD. En esta estructura se utiliza una plataforma diferente a la utilizada originalmente, por lo que el desarrollo realizado requirió de una adaptación para correr en la nueva plataforma.

Para llevar a cabo la paralelización fue necesario separar la aplicación en dos partes. Una parte que se encarga del proceso evolutivo en sí que llamaremos *módulo de evolución* del cual hay una única instancia corriendo. La otra parte es el *módulo de evaluación* del cual hay tantas instancias como procesadores disponibles para evaluar individuos.

Cuando detallamos la jerarquía de **TEvaluationMethod** (ver Diagrama UML 4) mencionamos una clase **TParallelEvaluator** la cual permitía una evaluación en paralelo pero sin brindar más detalles. Esta clase permite desacoplar la paralelización del resto del algoritmo ya que únicamente en esta clase se administra todo lo que tiene que ver con la paralelización de la evaluación, disparando los módulos evaluadores remotos y

---

<sup>6</sup> Varias arquitecturas monoprocesador independientes interconectadas mediante interface de red formando, con el software adecuado, una arquitectura multiprocesador del tipo MIMD.

maneja la comunicación con los mismos; basándose en un protocolo de comunicación entre computadoras de distintas plataformas, como PVM<sup>7</sup>, por ejemplo.

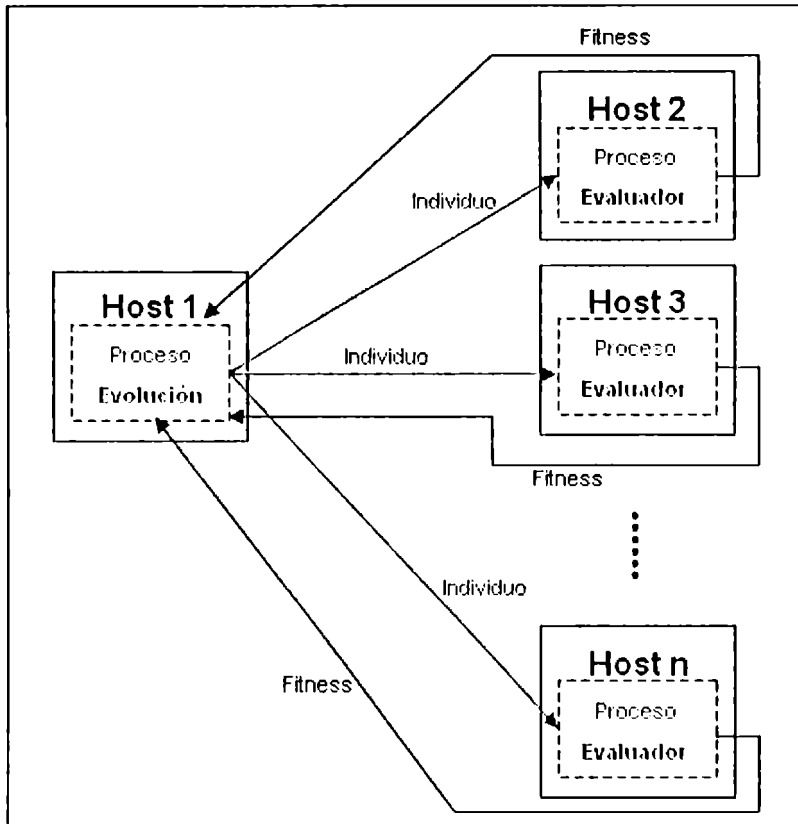


Figura 61 : Diagrama de la disposición de los procesos y la comunicación entre los mismos cuando se aplica paralelización en el framework de evolución.

El algoritmo evolutivo se ejecuta en el módulo de evolución que corre en una única computadora. En este módulo, un **TParallelEvaluator** se encarga, en el momento de la inicialización de todo el algoritmo, de disparar e inicializar mediante mensajes los procesos o módulos evaluadores uno por cada uno de los hosts restantes del cluster. Luego de la inicialización, la evolución se ejecuta en el host de evolución. Una vez realizado el proceso de reproducción le sigue la evaluación. En esa instancia, al invocar el método *Evaluate* de **TParallelEvaluator** se envían uno a uno, mediante mensajes propios del software de comunicación utilizado, los individuos a evaluar a cada uno de los procesos evaluadores ya inicializados. Cuando un evaluador finaliza su tarea con un individuo envía su fitness al módulo de evolución y queda a la espera de un nuevo individuo para evaluar. Si cuando un evaluador envía el fitness quedan individuos sin evaluar, el **TParallelEvaluator** envía un nuevo individuo para que no quede ocioso dicho evaluador. Este proceso puede verse gráficamente en la Figura 61. A medida que los evaluadores devuelven los valores de fitness obtenidos, estos son asignados a los individuos del lado del proceso de evolución. Una vez evaluada la población continúa el proceso, es decir, se determina si finalizó el algoritmo y de no ser así se repite nuevamente la tarea a partir de la reproducción.

<sup>7</sup> Parallel Virtual Machine, es un software que provee un conjunto de funciones las cuales permiten una configuración de una computadora paralela a partir de varias computadoras independientes comunicadas por red.



## 8.4. Flexibilidad del framework

La herramienta implementada aquí descrita permite una gran flexibilidad al programador en cuanto a las posibilidades de combinación para la configuración de un algoritmo evolutivo. El lector podrá observar lo sencillo que resulta la implementación de diversas estrategias dentro del entorno ya que fue diseñado de manera modular para permitir una composición de módulos independientes que van a contribuir para la ejecución del algoritmo evolutivo deseado. Igualmente ocurre con la experimentación de estrategias alternativas. Están dadas todas las condiciones para agregar nuevas opciones alterando mínimamente el resto del framework y generando el código necesario específicamente para la nueva funcionalidad.

## 8.5. Utilización de la herramienta de neuroevolución

La herramienta de software Win Evolution fue diseñada para ejecutar y visualizar resultados de procesos evolutivos implementados en un framework específico. Esta herramienta funciona únicamente en plataformas Win32.

En las líneas siguientes el lector encontrará información acerca de cómo utilizar esta aplicación e interpretar los resultados obtenidos.

### 8.5.1. Panel de experimentos disponibles

Cuando por primera vez nos encontremos ante la interface de Win Evolution veremos una imagen como la que se ve en la Figura 62. En la misma veremos una lista sobre la izquierda. Esta lista consiste en una vista en forma de árbol de los experimentos configurados que se pueden observar.

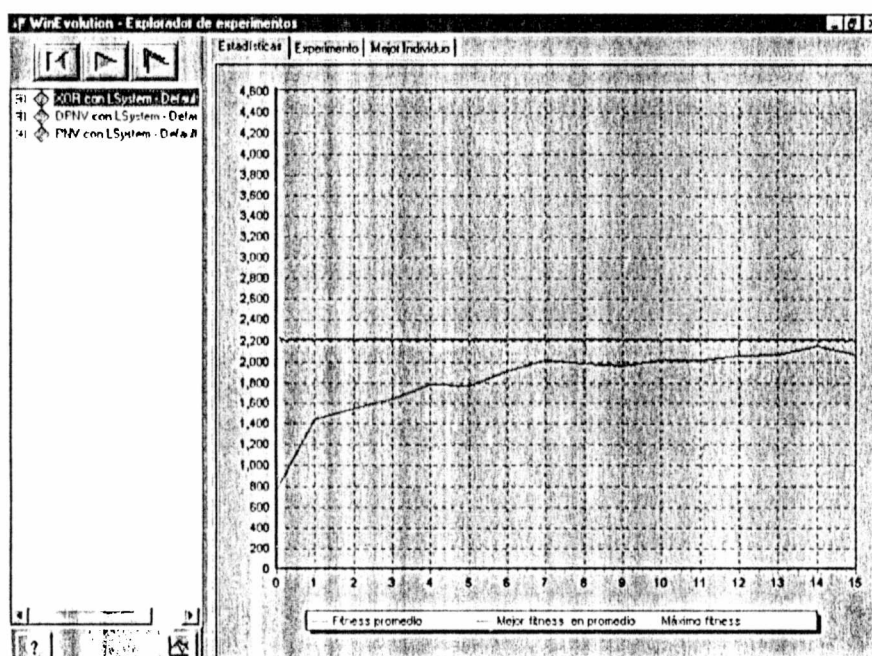


Figura 62: Vista principal de la aplicación.

Al estar organizada como un árbol, se puede explorar los contenidos de cada experimento, es decir, las corridas que se hicieron y las generaciones por corrida. En la Figura 63 se puede ver la organización de la misma.

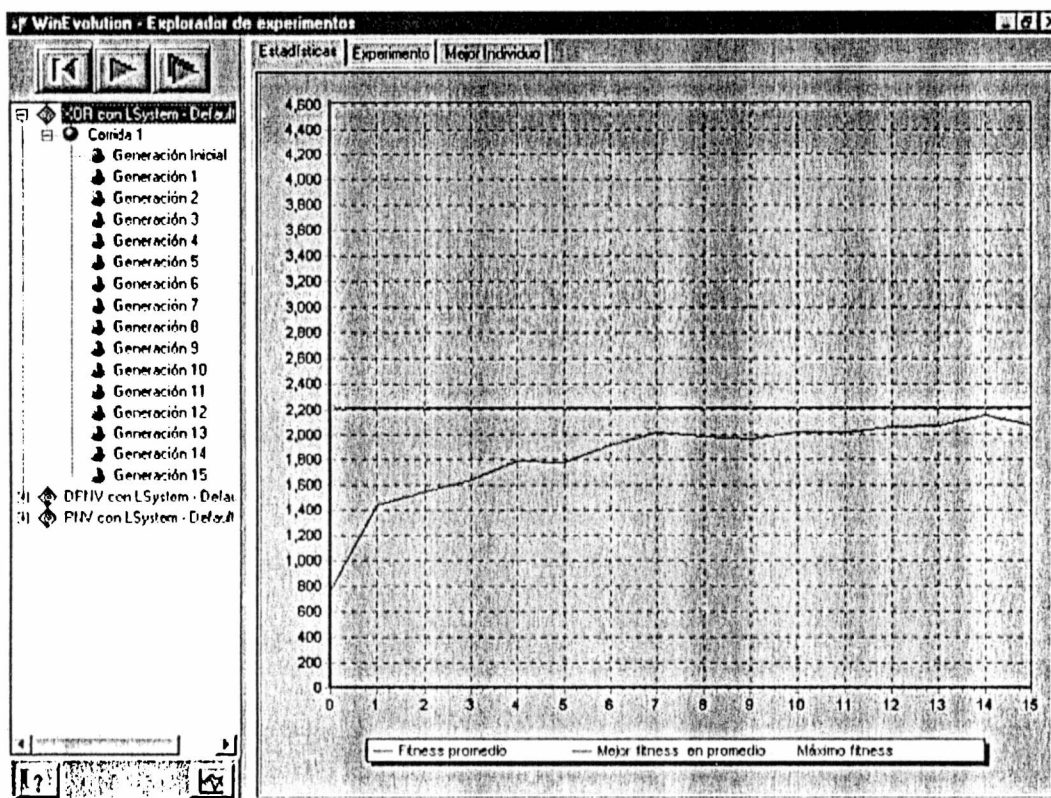


Figura 63: Vista de la aplicación con la lista de experimentos desplegada en un ítem. Notar que los iconos varían de acuerdo al tipo de elemento de la misma.

### 8.5.2. Ejecución de experimentos

En el sector superior izquierdo de la interface de la aplicación nos encontramos con tres botones cuya función es permitir al usuario el control de la ejecución de los experimentos. En detalle, cada botón cumple la siguiente función:



Continúa con la ejecución del experimento actualmente seleccionado o del cual se haya seleccionado una corrida o generación dentro de una corrida del mismo.



Reinicia la ejecución del experimento seleccionado, eliminando cualquier información de corridas anteriores.



Continúa con la ejecución de manera secuencial de todos los experimentos que muestra la lista de experimentos disponibles.

### 8.5.2.1. Consola de ejecución de experimentos

Cualquiera de los botones que se encuentran sobre el borde superior de la lista de experimentos mostrará una consola de ejecución de evoluciones como la que se muestra en la Figura 64. Esta muestra el desarrollo de la evolución en el experimento seleccionado.

Debido a que al correr experimentos, los datos acerca de su desempeño cambian, cada vez que se vuelve a la vista principal después de mostrar una consola la información acerca de los experimentos se carga nuevamente.

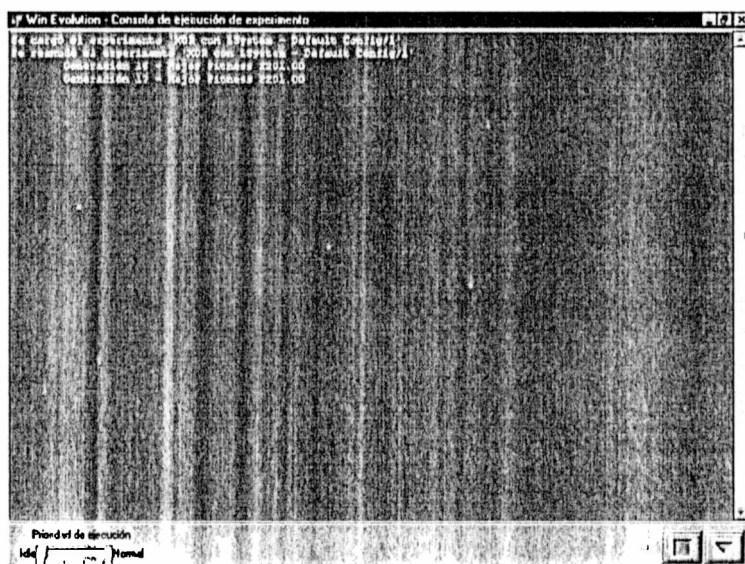


Figura 64: Consola de ejecución de evolución.

#### Información

El panel negro que ocupa gran parte de la interface es utilizado para comunicar el estado de la evolución. Proporciona datos tales como:

- Nombre del experimento.
- Número de la corrida actual del experimento.
- Mejor fitness de la última generación.
- Razón por la cual termina una corrida.

Por razones ajenas a la aplicación cuando esta es ejecutada sobre Windows 95/98/ME cada 200 líneas de texto se eliminan todas las líneas dado que luego no se muestran los nuevos reportes.

#### Operación

En la parte inferior de la consola se puede ver sobre la izquierda un control que permite configurar la prioridad de ejecución respecto de todos los procesos activos en el momento.

Sobre la parte derecha de la barra inferior se pueden ver dos botones.

Cada uno de ellos tiene la siguiente función:



Detiene la ejecución actual y cierra la consola para volver a la interface principal de la aplicación.



Minimiza la aplicación trabajando en segundo plano. Se puede volver a acceder a la misma haciendo click sobre el correspondiente icono en la barra de tareas.

### 8.5.3. Panel de información de experimentos

Sobre la derecha de la interface de Win Evolution se puede observar un panel con solapas que permite acceder a información sobre experimentos, corridas de experimentos y generaciones seleccionados de la lista.

#### 8.5.3.1. Estadísticas

Este es el primer panel de información comenzando por la izquierda (solapa “Estadísticas”) y muestra información estadística dispuesta en gráficos de línea acerca del objeto seleccionado. En la Figura 65 puede observarse dicho panel.

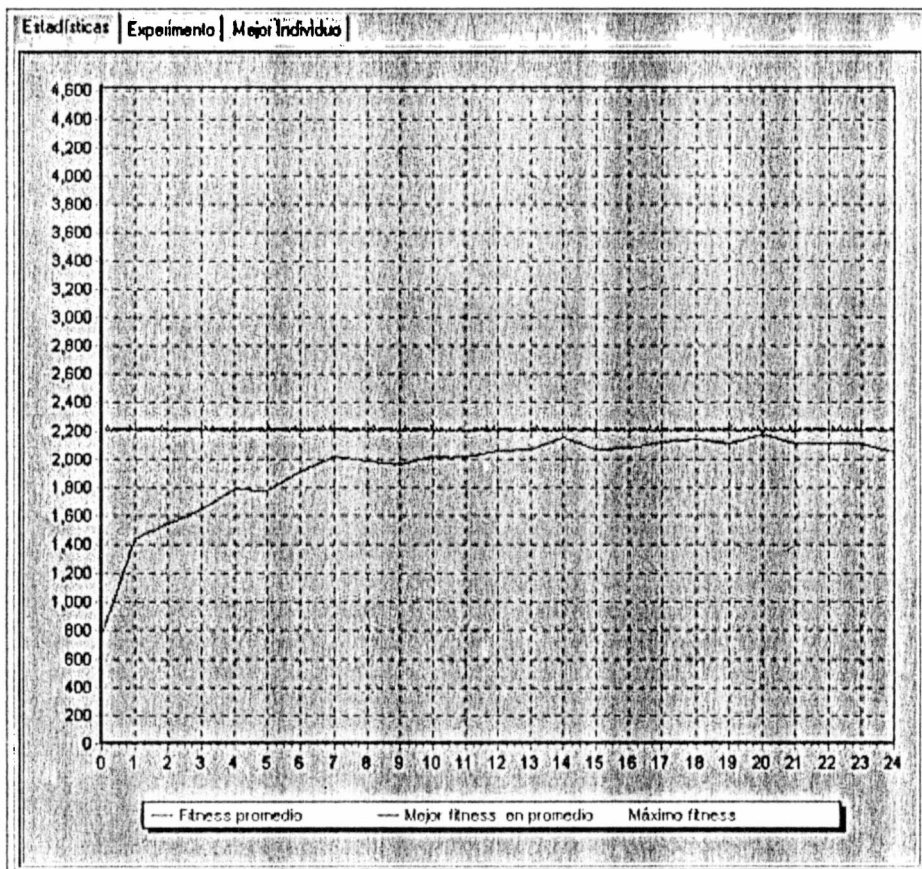


Figura 65: Información estadística.

## Interpretación

Dependiendo el objeto seleccionado muestra diferentes datos, pero en todos los casos el eje horizontal representa la generación y el vertical el valor de fitness. Para el caso de un experimento completo se puede observar la siguiente información:

- Máximo fitness en cada generación
- Fitness promedio en cada generación
- Máximo fitness en promedio para cada generación.

Para una corrida de experimento muestra los dos primeros puntos mencionados anteriormente, mientras que para una generación muestra en el número correspondiente el máximo fitness obtenido. En este último caso se agrega un gráfico más que muestra la distribución del fitness en esa generación. En la Figura 66 puede apreciarse dicho gráfico agregado.

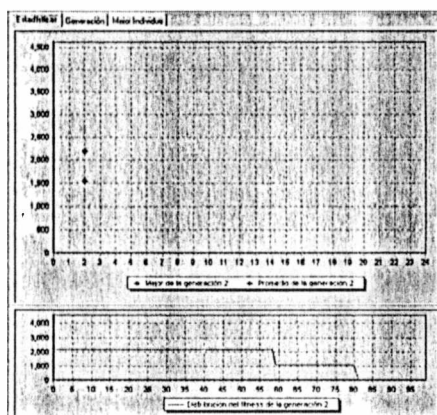



Figura 66: Panel de datos estadísticos cuando se selecciona una generación en la lista.

## Configuración

El botón  permite acceder a un diálogo de configuración que permite dejar fijas algunas series de manera de poder comparar al cambiar de selección en la lista. Esta interface se muestra en la Figura 67.

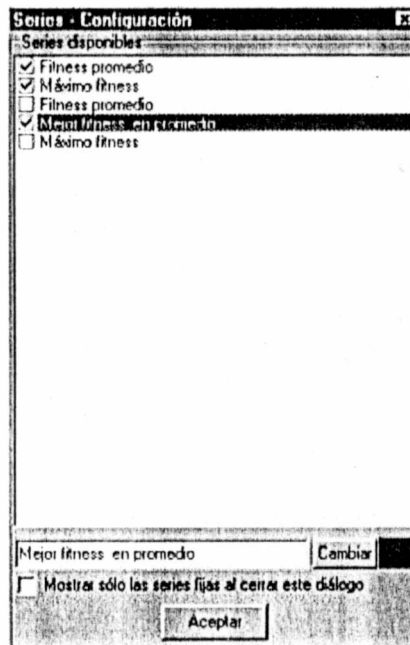


Figura 67: Diálogo de configuración de series fijas.

Cuando se marca una serie para que quede fija, el gráfico estadístico siempre se muestra aunque en la lista se seleccione cualquier otro experimento. De esta manera se puede comparar el desempeño evolutivo en un experimento contra otro.

En la ventana de configuración aparece una lista con todos los gráficos que se muestran hasta el momento: los correspondientes al elemento seleccionado actualmente y los que ya estaban fijos. Así, es posible elegir las series que el usuario desea que no se eliminen al cambiar la selección en la ventana principal. Debajo de la lista se puede cambiar el nombre y el color que la serie tiene por defecto. Finalmente, tiene una opción de “Mostrar sólo las series fijas...”, cuyo efecto es eliminar del gráfico las series que no se hayan marcado como fijas, de manera que el usuario pueda observar únicamente las que seleccionó.

### 8.5.3.2. Información específica de cada elemento

Cada tipo de elemento tiene una información específica para mostrar, es decir un experimento completo brindará datos diferentes que los de una corrida independiente del mismo o que una generación. En la segunda solapa de la ventana principal de Win Evolution se puede ver la información específica del elemento seleccionado. Según lo que se seleccione mostrará una etiqueta de “Experimento”, “Corrida de experimento” o “Generación”

### Información de todo un experimento

Cuando se selecciona todo un experimento completo importa una visión global de todas las corridas –promedios generales, efectividad, etc.– junto con una lista del estado de las corridas del mismo. El panel de la derecha, en su solapa de “Experimento” mostrará un aspecto como el de la Figura 68. En este panel se puede observar tres solapas en la zona inferior. Esta permite seleccionar entre la vista anteriormente mencionada y la posibilidad de ver y modificar archivos de configuración con formato XML.



Figura 68: Vista de la información global de todo un experimento.

Como se puede observar en la Figura 69 y la Figura 70 se muestra un editor de texto para modificar los datos de configuración. Al abandonar el editor la aplicación consultará al usuario si desea mantener las modificaciones (si las hubo).

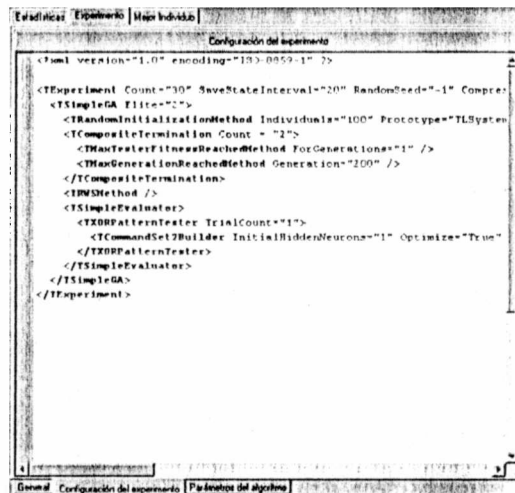


Figura 69: Vista de la configuración del experimento.

En caso que el archivo no respete con el formato preestablecido, se volverá a la versión anterior. En la primer solapa de las configuraciones (Figura 69) se encuentran los datos referentes al experimento en general, es decir, el problema que evalúa, la cantidad de corridas, el tipo de algoritmo a utilizar, etc. En la segunda solapa (Figura 70) se configuran datos específicos del algoritmo evolutivo seleccionado para el experimento en su mayoría probabilidades.

```

MaxRuleCount="10" MaxRuleCrossoverCount="
MaxRuleCount="5"
MinRuleCrossoverCount="1" MaxRuleCrossoverCount="
MinSuccessorSize="3" MaxSuccessorSize="15"
MinExpressionHeight="1" MaxExpressionHeight="4"

CrossoverProb = "0.90"
StructureCrossoverProb = "0.80"
StructureMutationProb = "0.15"
CommandSelectionProb = "0.50"
StructMutInsertProb = "0.10"
StructMutRemoveProb = "0.10"
StructMutReplaceProb = "0.40"
StructMutInsertProbByStructureProb = "0.0"
StructMutInsertMaxModuleCount = "2"
StructMutRemoveMaxModuleCount = "2"

ExpressionCrossoverProb = "0.10"
ExpressionMutationProb = "0.05"
ExpressionCombinationCrossoverProb = "0.00"
NeuronMaxValue="2"
/>

```

Figura 70: Vista de la configuración de los parámetros del algoritmo evolutivo.

### Información de una corrida de experimento

En la Figura 71 se puede observar la forma del panel que se muestra al seleccionar una corrida de experimento y la solapa etiquetada con "Corrida de experimento". Es información global acerca de la corrida seleccionada, es decir, el estado en el que se encuentra (si finalizó o no), el motivo de la finalización y la cantidad de generaciones alcanzadas en la misma.

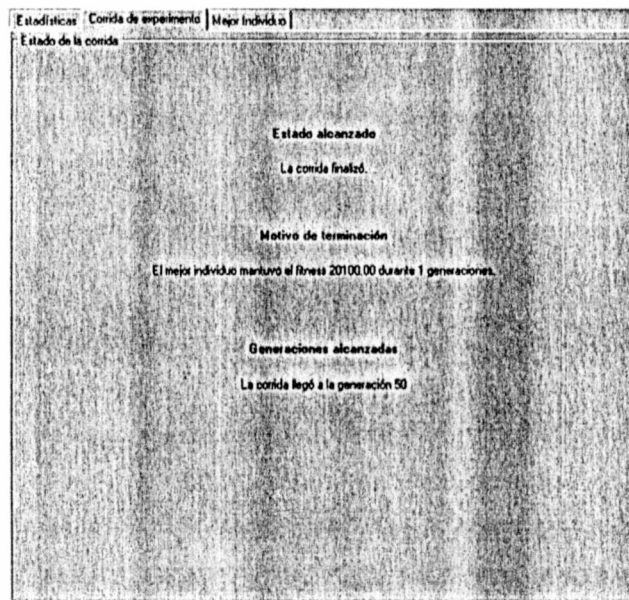


Figura 71: Datos correspondientes a una corrida de experimento.



### **Información de una generación en una corrida de experimento**

Finalmente, el otro tipo de objeto que se puede seleccionar es una generación dentro de una corrida de experimento. Al seleccionar esta clase de ítem y la solapa “Generación” se muestra un panel como el de la Figura 72. Se puede ver en la misma un gráfico donde se refleja la distribución del fitness en toda la población en esa generación en particular. Además se muestra el mejor y el fitness promedio de la misma.

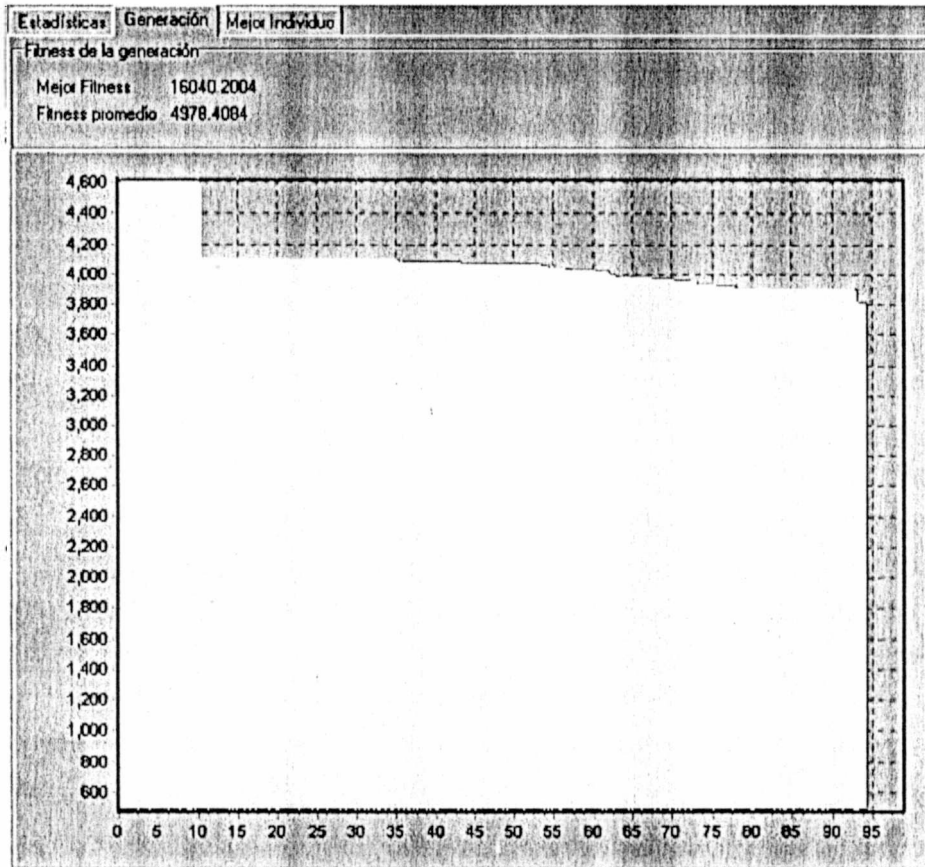


Figura 72: Información específica que se muestra para una generación.

#### **8.5.3.3. Mejor Individuo**

Independientemente del tipo de objeto seleccionado, cada uno tiene su mejor individuo, es decir el mejor de todo el experimento, el mejor de una corrida o el mejor de una generación. La tercera solapa (“Mejor Individuo”) de la interface principal de Win Evolution permite al usuario explorar la información acerca del mismo.

#### **Red Neuronal**

Esta aplicación está orientada a experimentar con métodos de neuroevolución, por lo tanto debe estar capacitada para mostrar las redes neuronales de los mejores individuos. La primera solapa inferior desde la izquierda (con etiqueta “Red Neuronal”) muestra un aspecto como el de la Figura 73. Se puede observar una sección principal en donde se muestra gráficamente la red neuronal que representa el mejor individuo del ítem seleccionado.

Sobre la derecha se encuentran las referencias que dan significado al gráfico, una tabla que muestra las conexiones y otra que muestra las neuronas de la red.

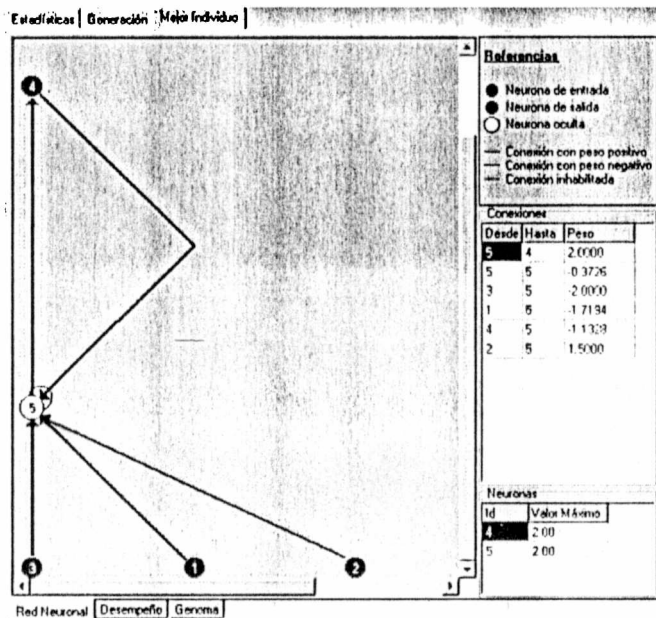


Figura 73: Vista de la red neuronal del mejor individuo del elemento seleccionado.

### **Desempeño en el problema**

La segunda solapa de la zona inferior etiquetada "Desempeño" muestra una interface que permite la evaluación del individuo en el problema configurado para el experimento seleccionado. Sobre la parte inferior izquierda se muestran dos botones, uno para iniciar la visualización (hay problemas de control que muestran una simulación) y otro para detenerla cuando el usuario desee, si la demostración no llegó a su fin. En la Figura 74 puede verse un ejemplo.

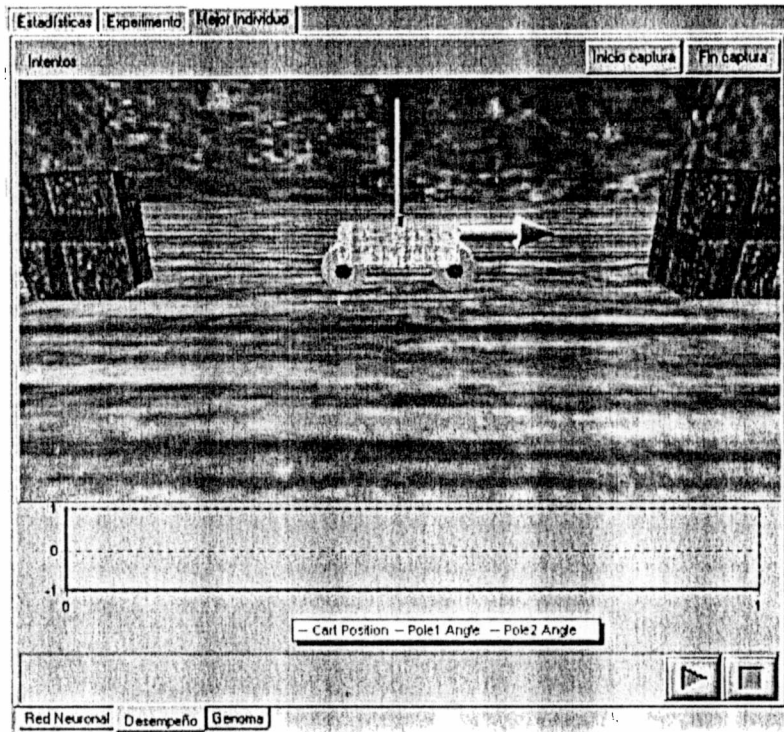


Figura 74: Vista del desempeño del mejor individuo. En este caso se muestra el desempeño de un individuo en el problema del balance del péndulo doble.

### Genoma

En la tercera solapa etiquetada “Genoma” se pueden observar datos acerca de la representación genética del mejor individuo. Se muestra en modo texto como se puede ver en la Figura 75. Este texto depende de como implementa el genoma la función de representación en una cadena de caracteres. De todas maneras la aplicación está diseñada para soportar otros tipos de visualizaciones en futuras implementaciones.

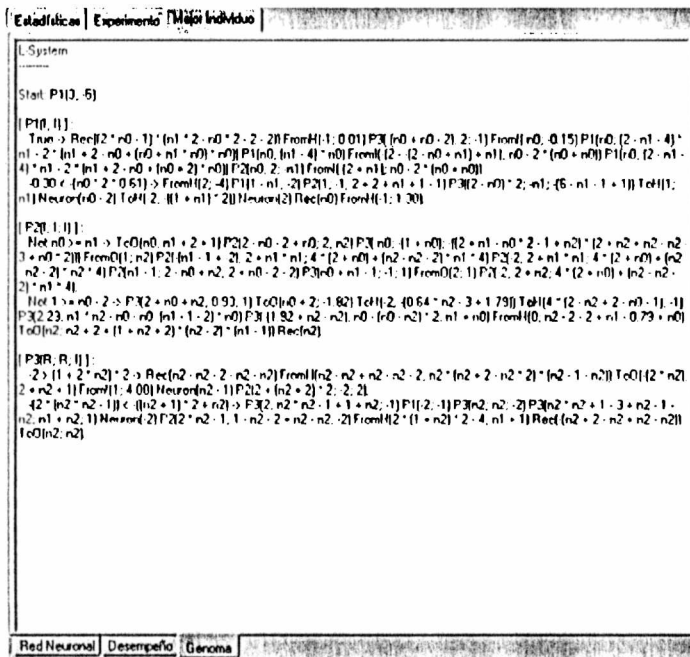


Figura 75: Información del genoma del mejor individuo.

## 9. Desempeño de NeSR

Este capítulo estará dedicado a la exposición de los resultados obtenidos acerca del desempeño de NeSR y su comparación con otros métodos neuroevolutivos. En primer lugar daremos una descripción detallada de los problemas a resolver por los métodos. Luego presentaremos la manera en que fueron planificados y configurados los experimentos detallando particularmente la configuración de NeSR utilizada. Finalmente presentaremos los resultados obtenidos realizando un análisis de los mismos acompañado con gráficos comparativos que permiten observar el desempeño de nuestro método comparado con otros métodos seleccionados.

### 9.1. Experimentos

Para comprobar la efectividad de NeSR hemos configurado bancos de prueba para tres problemas diferentes. El clásico problema de clasificación no lineal, XOR y dos problemas de control, el balance del péndulo y el balance del péndulo doble.

En las próximas secciones brindaremos un detallado informe de la simulación de estos problemas. Más adelante mostraremos resultados obtenidos y la comparación con otros dos métodos: neuroevolución mediante cadenas de bits (la descripción de este método se encuentra en la Pág. 109) y NEAT (Pág. 114).

Debemos aclarar que no hemos incluido en estos experimentos el método de neuroevolución mediante reescritura de matrices (pág 111) debido a que solamente evoluciona topologías de redes neuronales artificiales que luego deben someterse a algún método de ajuste de pesos.

### 9.2. El problema del XOR

Este problema encuadra dentro de los problemas de clasificación y es un test clásico para cualquier método de neuroevolución.

Consiste en proporcionarle a la red neuronal dos entradas que pueden ser 1 o 0 y debe devolver el resultado de aplicar el operador lógico XOR a estos valores.

Si bien se trata de un problema sencillo, su resolución implica encontrar una estructura de red adecuada que va más allá de la simple estructura que conecta todas las entradas de la red con todas sus salidas, ya que se debe generar una cierta capa oculta o una combinación adecuada de recurrencias.

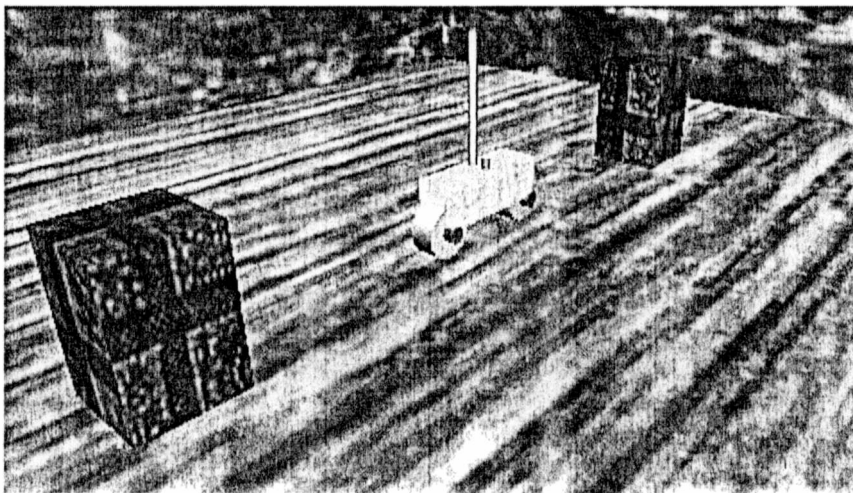
En este punto es conveniente aclarar la política de evaluación de la red en el momento de realizar la prueba. Debido al tipo de red neuronal utilizada - ver Pág. 107 - es necesario determinar la cantidad de actualizaciones que se realizan sobre los valores de entrada de cada neurona hasta considerar la salida como la respuesta de la red. Existen dos opciones: realizar actualizaciones un número fijo de veces o actualizar la red hasta que los valores de la salida lleguen a un cierto valor y se estabilicen. Nos inclinamos por la primera porque además de la simplicidad, permite limitar el tamaño de las redes: si la altura entre las entradas y la salida es mayor que la cantidad de actualizaciones, la red no generará ninguna respuesta, con lo que su fitness será bajo y será rápidamente eliminada de la población.

versión más difícil del problema previo. En este caso se cuenta con dos mástiles de diferente tamaño y peso a los cuales hay que mantener balanceados.

Como en el caso de la sección anterior, la pista por donde se mueve el carrito tiene una longitud de 4,8 mts. Los mástiles son considerados balanceados si se mantienen dentro del rango de  $[-36^\circ, 36]$ . Ahora el estado se basa en 6 variables: posición y velocidad del carrito; y las posiciones y velocidades angulares de cada uno de los dos mástiles. En la Figura 77 se muestra gráficamente el ambiente del problema.

Para calcular el estado del ambiente utilizamos el modelo matemático descrito por la Ecuación 18, el cual coincide con el utilizado en [102].

Al igual que en el caso anterior, este problema puede utilizarse brindando toda la información (velocidades y posiciones) a la red neuronal al ser evaluada, o bien informar solamente las posiciones angulares de los mástiles y la posición que ocupa el carrito dentro de la pista. De esta manera el problema se presenta de manera más difícil, dejando que el mismo proceso evolutivo genere la estructura adecuada para inferir la información faltante. Para evaluar NeSR optamos por la versión que no informa las velocidades cuando se realiza la prueba.



**Figura 77:** Esquema gráfico del entorno del problema del balance del péndulo doble.

Otra cuestión a tener en cuenta es la configuración inicial en cada prueba. Ahora, al disponer de un nuevo mástil, las posibilidades de combinaciones de los valores de comienzo se multiplican. Para realizar una comparación justa nos hemos basado en las configuraciones utilizadas en [124], donde el mástil de menor tamaño tiene una posición inicial vertical y sin velocidad angular, variando únicamente los parámetros del mástil mayor, con lo cual las 18 configuraciones iniciales mencionadas en la sección anterior tienen sentido si se aplican las variaciones al mástil de mayor tamaño.

$$\ddot{x} = \frac{F + m_1 l_1 \theta_1^2 \sin \theta_1 + \frac{3}{4} m_1 \cos \theta_1 \left( \frac{MUP \dot{\theta}_1}{m_1 l_1} + g \sin \theta_1 \right) + m_2 l_2 \theta_2^2 \sin \theta_2 + \frac{3}{4} m_2 \cos \theta_2 \left( \frac{MUP \dot{\theta}_2}{m_2 l_2} + g \sin \theta_2 \right)}{m_1 \left( 1 - \frac{3}{4} \cos^2 \theta_1 \right) + m_2 \left( 1 - \frac{3}{4} \cos^2 \theta_2 \right) + m}$$

$$\ddot{\theta}_1 = - \frac{3 \left( \ddot{x} \cos \theta_1 + g \sin \theta_1 + \frac{MUP \dot{\theta}_1}{m_1 l_1} \right)}{4 l_1}, \quad \ddot{\theta}_2 = - \frac{3 \left( \ddot{x} \cos \theta_2 + g \sin \theta_2 + \frac{MUP \dot{\theta}_2}{m_2 l_2} \right)}{4 l_2}$$

**Ecuación 18: Modelo matemático para simular el ambiente del problema del péndulo doble.**

El fitness es el promedio del total de pasos en los 18 intentos en los cuales se mantuvieron ambos mástiles dentro del rango del ángulo de inclinación establecido. También se suma una cierta cantidad por cada entrada conectada a la red, de manera que cuando los individuos no son buenos en la tarea, sobrevivan aquellos que aprovechan toda la información brindada. Esto fue establecido de esta manera dado que los individuos que no conectaban la neurona de entrada que recibe la posición del carrito con el resto de las neuronas, podían mantener el balance pero no cumplían su tarea porque no controlaban su posición y traspasaban los límites de la pista.

## 9.5. Planificación de experimentos

Cada experimento se centró en probar un método de neuroevolución en varias corridas independientes para encontrar redes neuronales que resuelvan un determinado problema. Se utilizó el framework y la aplicación descriptas en el capítulo anterior (ver Pág. 123). Además de experimentar con NeSR (Pág. 95) utilizamos dos métodos de neuroevolución explicados anteriormente, neuroevolución basada en representación de cadenas de bits (Pág. 109) y NEAT (Pág. 114), para configurar los experimentos.

El experimento con el problema del XOR fue aplicado solamente a NeSR ya que en todo nuevo método neuroevolutivo es el primer e ineludible banco de pruebas. Debido a que el método se comportó de manera satisfactoria, como se verá más adelante, se prosiguió a experimentar y compararlo en los problemas del balance del péndulo (simple y doble) contra los otros dos métodos. Para estos problemas consideramos que la versión que proporciona la información de las velocidades resulta trivial de resolver por ello decidimos no proporcionar a la red la velocidad del carrito y del o de los péndulos al momento de evaluarla obligando así a crear una estructura que estime dichos valores.

Problema	Corridas	Generaciones	Tamaño de población
XOR (sólo NeSR)	50	200	100
Bal. Péndulo	30	500	100
Bal. Péndulo Doble	30	1000	200

**Tabla 8: Configuraciones generales de los diferentes experimentos.**

Para el XOR con NeSR realizamos 50 corridas independientes del algoritmo de 200 generaciones como máximo y con una población de 100 individuos. Luego, para los tres métodos en el problema del balance del péndulo, efectuamos 30 corridas de 500 generaciones con poblaciones de 100 individuos; y en la versión de dos péndulos, 30

### 9.3. Balance del péndulo simple

Este es un problema de control que ha sido utilizado ampliamente por los investigadores para evaluar el desempeño de los métodos de neuroevolución ([1] y [91]).

El entorno del problema consiste en un carrito con una masa de 1 kg. que puede moverse hacia la izquierda o derecha dentro de una pista de 4,8 mts de longitud. En el centro del carrito se encuentra un mástil articulado con una polea que le permite inclinarse hacia la izquierda o derecha. Dicho mástil tiene una longitud de 0,5 mts. y una masa de 0,1 kg.

La red neuronal se encarga de aplicar sobre el carrito una fuerza no mayor a 10 Newtons hacia la izquierda o derecha de manera de mantener el mástil equilibrado con una variación entre  $[-12^\circ, 12^\circ]$  respecto de su posición vertical durante un total de 50.000 pasos de simulación. Cada paso de simulación equivale a 0,02 segundos, por lo tanto se debe mantener el equilibrio por algo más de 16 minutos. Además el carrito no podrá traspasar los límites de la pista. La Figura 76 muestra como se disponen los elementos en el ambiente del problema.

El estado del ambiente consiste en la información acerca de la posición del carrito, la posición angular del péndulo, velocidad del carrito y velocidad angular del mástil.

La red podría recibir los cuatro datos, pero para agregar un mayor nivel de dificultad al problema, en el momento de la evaluación de la red neuronal se le proporciona el conocimiento de las posiciones del carrito y el mástil, debiendo inferir las velocidades generando la estructura adecuada.

Para calcular el estado del ambiente utilizamos el modelo matemático descrito por la Ecuación 17 y presentado en [94]:

$$\ddot{\theta}_t = \frac{mg \sin(\theta_t) - \cos(\theta_t) [F_t + m_p l \dot{\theta}_t^2 \sin(\theta_t)]}{\frac{4}{3} ml - m_p l \cos^2(\theta_t)}, \ddot{\rho}_t = \frac{F_t + m_p l [\dot{\theta}_t^2 \sin(\theta_t) - \ddot{\theta}_t \cos(\theta_t)]}{m}$$

donde

$\rho$  = posición del carrito,  $\dot{\rho}$  = velocidad del carrito.

$\theta$  = posición angular del péndulo,  $\dot{\theta}$  = velocidad angular del péndulo.

$l$  = longitud del péndulo = 0,5 m,  $m_p$  = masa del péndulo = 0.1 kg.

$m$  = masa del péndulo y del carrito = 1.1 kg,  $F$  = magnitud de la fuerza aplicada = 10 N.

$g$  = aceleración debido a la gravedad =  $9.8 \frac{m}{seg^2}$ .

**Ecuación 17: Modelo matemático para simular el ambiente del problema del péndulo simple.**

Las variables del ambiente al inicio de la prueba pueden tener diferentes valores, lo cual tiene una gran influencia en el desempeño de la red en evaluación. Por ejemplo, si se inicia una prueba con el carrito en una posición central, sin velocidad, con un ángulo de inclinación del péndulo cercano a la posición vertical y con velocidad angular nula, le resultará mucho más fácil a la red neuronal resolver el problema que si se comienza con el carrito en una posición cercana a uno de los límites con el péndulo cayéndose para el mismo lado. Una configuración inicial aleatoria puede explorar las distintas configuraciones iniciales del problema pero no resulta equitativo ya que un individuo que fue el mejor en una generación (por haber comenzado la prueba con una configuración inicial accesible a resolver el problema) puede ser uno de los peores en la

siguiente (porque tuvo un comienzo más difícil) simplemente porque otro tuvo un comienzo más sencillo.

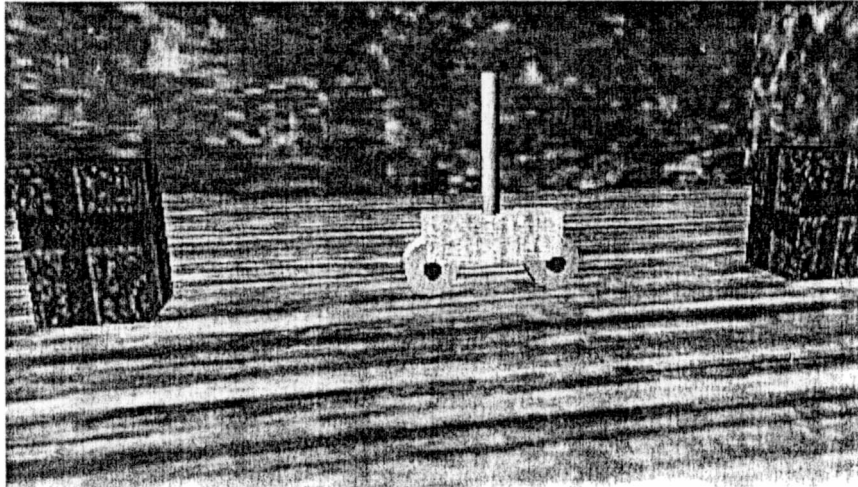


Figura 76: Esquema gráfico del entorno del problema del balance del péndulo simple

Además hay posiciones iniciales que son prácticamente imposibles de resolver como por ejemplo comenzar con el carrito sobre el límite izquierdo con el mástil cayéndose hacia la izquierda con un ángulo cercano a los  $12^\circ$  hacia la izquierda. Para salvar la situación el carrito debería aplicar una fuerza hacia la izquierda, pero traspasaría los límites de la pista.

Por todo esto decidimos configurar el test para que cada red realice 18 intentos con diferentes configuraciones preestablecidas e iguales para todos los individuos. Las configuraciones se basan en la combinación de:

- 2 posiciones distintas del carrito: una cercana al borde izquierdo y otra cercana al borde derecho.
- 3 posiciones angulares diferentes del péndulo: inclinado hacia la izquierda, con un grado de inclinación e inclinado hacia la derecha.
- 3 velocidades angulares: cayendo hacia la izquierda, hacia la derecha y velocidad nula.

Para el diseño de estas posiciones iniciales se determinó excluir aquellas en las que el péndulo comenzaba balanceado y sin velocidad dado que esto permitía a las redes que no generaban salidas obtener un buen fitness. Así tomaban desde un principio el dominio de la población y no permitían el progreso de otros individuos que intentaban alguna estrategia.

El fitness de cada individuo se calcula promediando el total de pasos en los 18 intentos en los cuales se mantuvo el péndulo dentro del rango del ángulo de inclinación establecido mas un plus por cada entrada conectada a la red, para premiar de alguna manera a aquellos individuos cuyas redes aprovechan toda la información disponible.

## 9.4. Balance del péndulo doble

El problema anterior si bien constituye un buen banco de pruebas, resulta una tarea sencilla de resolver, aún cuando no se informa a la red las velocidades del carrito y el péndulo. En esta sección describiremos otro problema de control que consiste en una



corridas de 1000 generaciones con poblaciones de tamaño 200. Estos datos pueden verse de manera tabulada en la Tabla 8.

La cantidad de corridas preestablecida se debe a que el XOR no requiere un excesivo tiempo de procesamiento. En cambio debido a la gran cantidad de cálculos necesarios para emular las variables de estado en los problemas de balanceo del péndulo se redujo la cantidad de corridas y se paralelizó la ejecución en un cluster de 16 computadoras independientes comunicadas por interface de red a 100 mbits/s lo cual redujo el tiempo de ejecución de manera considerable. Para más detalles de la modelización de la paralelización de los algoritmos evolutivos aconsejamos retomar el capítulo anterior más precisamente en la página 138.

A estos experimentos agregamos la realización de una prueba para comprobar la efectividad del operador de crossover definido en NeSR. Esta prueba consiste en modificar la selección de los individuos a reproducirse de manera que el primer padre se elige con el método determinado y el segundo se obtiene creando un individuo completamente aleatorio, para luego aplicar el operador de crossover de la manera definida. Si el método funcionando de esa manera obtiene un desempeño inferior al método original, significa que el crossover definido tiene un buen aporte en la búsqueda y por lo tanto es efectivo ya que hace lo que se espera de un operador de crossover. De lo contrario el crossover es similar a una macromutación. Esta prueba se la denomina prueba del *pollo sin cabeza (headless chicken)* [64]. Para esta prueba utilizamos el problema del balance del péndulo doble sin información de velocidades, ejecutando el algoritmo con la misma configuración anteriormente presentada para este problema.

La cantidad de generaciones por corridas responde al problema en sí mismo. La aproximación del operador XOR es un problema clásico, imposible de pasar por alto en la prueba de un nuevo método de neuroevolución pero actualmente resulta de trivial resolución, por lo que es lógico que no requiera muchas más generaciones. En cambio ambos problemas de balanceo resultan una tarea de mayor complejidad para una RNA por lo que decidimos que cada corrida debe tener más tiempo de exploración en la búsqueda de una buena solución. Además como el lector puede advertir entre los problemas de los péndulos resulta mucho más complejo el que debe mantener equilibrados los dos a la vez, por ello también es el que se le asignó mayor cantidad de generaciones.

## 9.6. Configuración de NeSR en los experimentos

En el capítulo correspondiente a la descripción del método (ver Pág. 95) mencionamos la imposición de restricciones a la forma de los Sistemas L (ver Pág. 102) y cuando describimos los operadores genéticos diseñados para NeSR (ver Pág. 103 y 105) utilizamos probabilidades sobre las cuales no inferimos nada en cuanto a los valores de las mismas. En esta sección mostramos dos tablas para presentar los valores y describir el significado de cada uno. En la Tabla 9 mostramos los valores de las restricciones de la forma de los Sistemas L utilizadas mientras que en la Tabla 10 se pueden observar los valores configurados para las probabilidades utilizadas en el método. Los identificadores que aparecen en esta última tabla corresponden a los utilizados en el Algoritmo 6, el Algoritmo 7 y el Algoritmo 8.

	Descripción	XOR	Péndulo Simple	Péndulo doble
NR	Cantidad de reglas que posee un sistema	3	3	3
NC	Cantidad máxima de casos para una regla de un sistema.	2	2	2
NP	Cantidad máxima de parámetros que posee un módulo.	2	3	3
LMin	Cantidad mínima de módulos que puede tener un sucesor.	5	5	5
LMax	Cantidad máxima de módulos que puede tener un sucesor.	15	15	15

Tabla 9: Restricciones de los Sistemas L en los distintos experimentos.

	Descripción	Valor
$P_{cross}$	Probabilidad de realizar un crossover entre dos sistemas.	0.9
$P_{mexp}$	Probabilidad de realizar una mutación sobre una expresión.	0.05
$P_{mest}$	Probabilidad de realizar una mutación estructural sobre un sucesor.	0.15
$P_{ins}$	Probabilidad de insertar módulos aleatorios durante una mutación estructural.	0.30
$P_{remp}$	Probabilidad de reemplazar módulos existentes por aleatorios en una mutación estructural.	0.40
$P_{elim}$	Probabilidad de eliminar módulos existentes durante una mutación estructural.	0.30
InsMods	Cantidad máxima de módulos a insertar en un sucesor en una mutación estructural.	2
ElimMods	Cantidad máxima de módulos a eliminar de un sucesor en una mutación estructural.	2

Tabla 10: Probabilidades y parámetros utilizados por el algoritmo y los operadores genéticos.

Otro dato que debemos destacar y que no se encuentra reflejado en estas tablas es que la reescritura de los sistemas en el momento de construir la RNA se realiza en un máximo de 3 pasos o hasta que la cadena resultante haya superado los 1000 módulos de longitud.

## 9.7. Resultados obtenidos

Una vez presentados todos los datos pertinentes respecto de los experimentos estamos en condiciones de brindar al lector los resultados obtenidos y comparaciones contra otros métodos de neuroevolución también probados para este trabajo.

El método NeSR resolvió el problema del XOR en 45 de los 50 intentos. En promedio fueron necesarias 34 generaciones para obtener una red que resolviera el problema, como máximo se requirieron 195 generaciones y como mínimo 1. En los 5 intentos que no resolvió el problema el mejor individuo obtuvo un fitness que representa el 75% del fitness máximo posible. La Figura 78 y la Figura 79 muestran diferentes gráficos del desempeño del método en la resolución de este problema.

Para que el lector pueda comprender estos gráficos pasaremos a detallar que representan cada uno:

- **Fitness promedio** es el promedio de los fitness de la población en una determinada generación en *todas* las corridas.
- **Mejor fitness en promedio** corresponde al promedio del fitness máximo de cada población en una determinada generación en *todas* las corridas.
- **Corridas con éxito** es similar al ítem anterior con la diferencia que sólo se calcula para las corridas que llegaron a resolver el problema.
- **Corridas fallidas** es el complemento del anterior, es decir se calculan los promedios de los máximos fitness por generación en las corridas que no llegaron a resolver el problema.

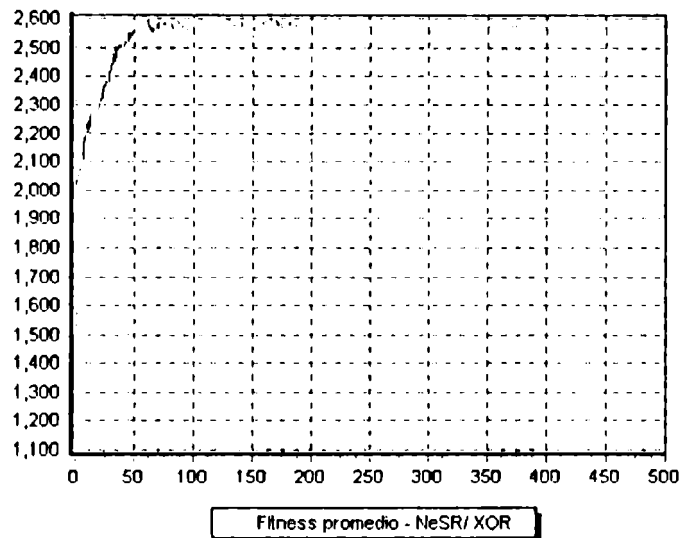


Figura 78: Experimento de NeSR con XOR: Fitness promedio de las poblaciones por generación.

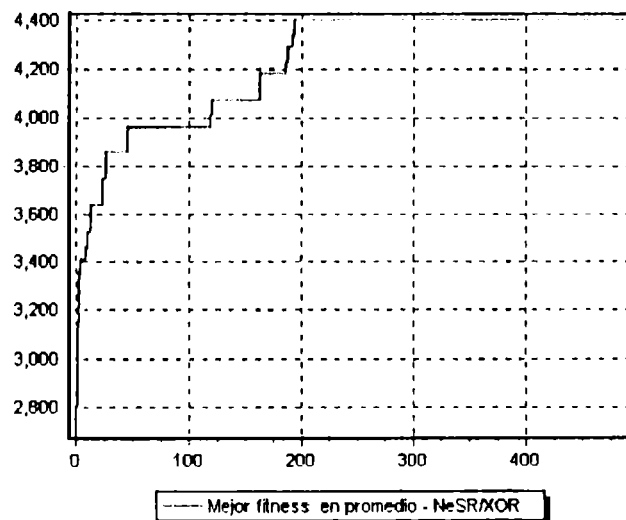


Figura 79: Experimento de NeSR con XOR: Fitness máximo en promedio por generación en todas las corridas.

En el problema del balance del péndulo falló en sólo 1 de los 30 intentos. En promedio fueron necesarias 44 generaciones para encontrar una red que mantuviera el péndulo balanceado en las condiciones impuestas. Como máximo se requirieron 144 generaciones y como mínimo 4. Se pueden ver los resultados de manera gráfica en la Figura 80 y la Figura 81.

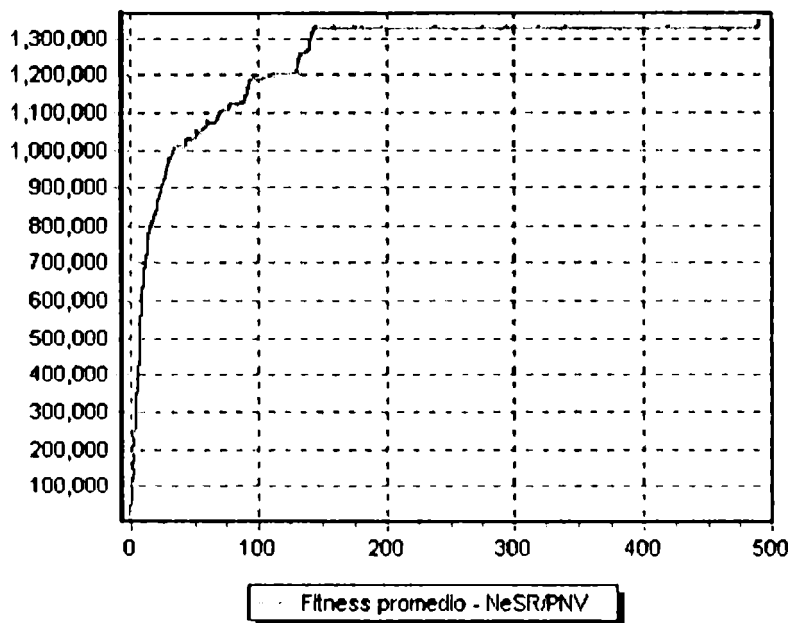


Figura 80: Experimento de NeSR con el balance del péndulo: Fitness promedio de las poblaciones por generación.

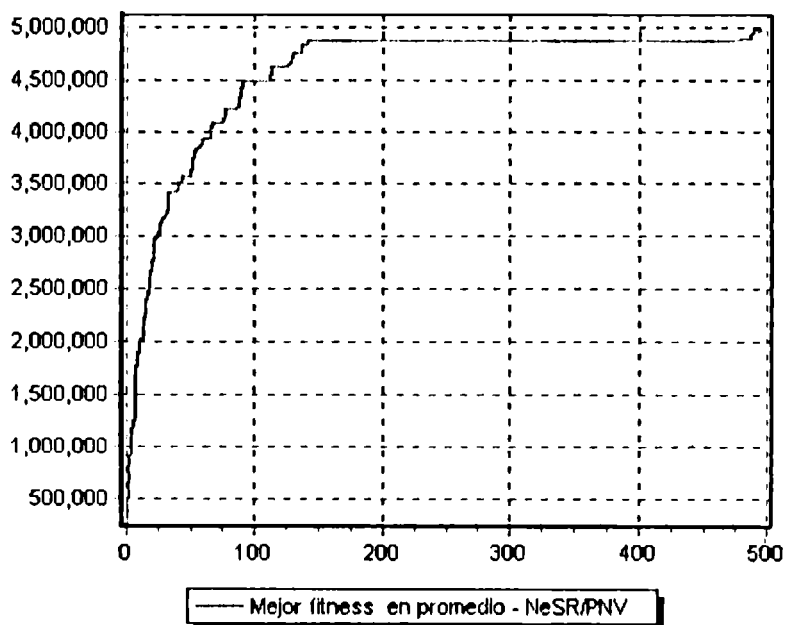


Figura 81: Experimento de NeSR con el balance del péndulo: Fitness máximo en promedio por generación en todas las corridas.

Para el problema del péndulo doble en 9 de los 30 intentos no le fue posible a NeSR resolver el problema. En promedio fueron necesarias 488 generaciones para llegar a una buena solución, como máximo 855 generaciones y como mínimo 30. En la mayoría de los intentos fallidos alcanzó un fitness que representa el 80% del problema resuelto. Como en los casos anteriores, presentamos en la Figura 82 y la Figura 83 una representación gráfica del desempeño del método en el problema.

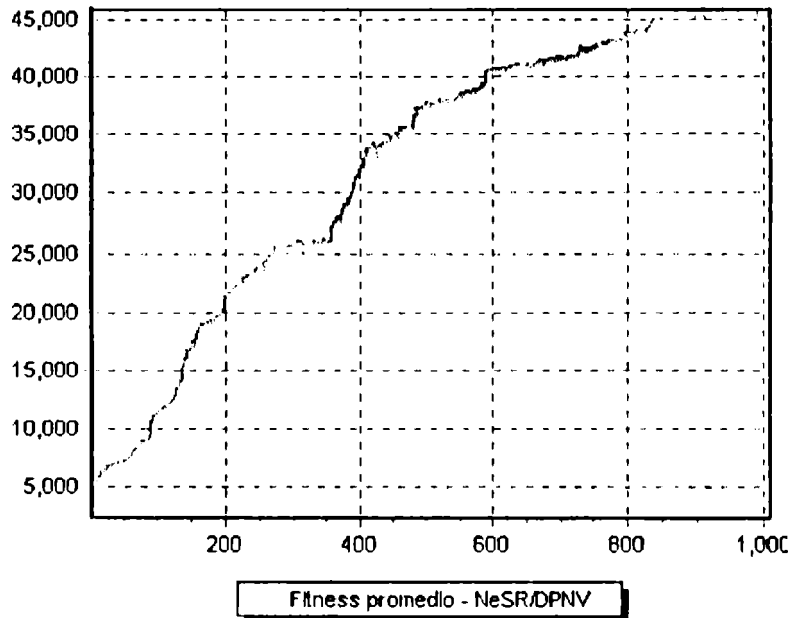


Figura 82: Experimento de NeSR con el balance del péndulo doble: Fitness promedio de las poblaciones por generación.

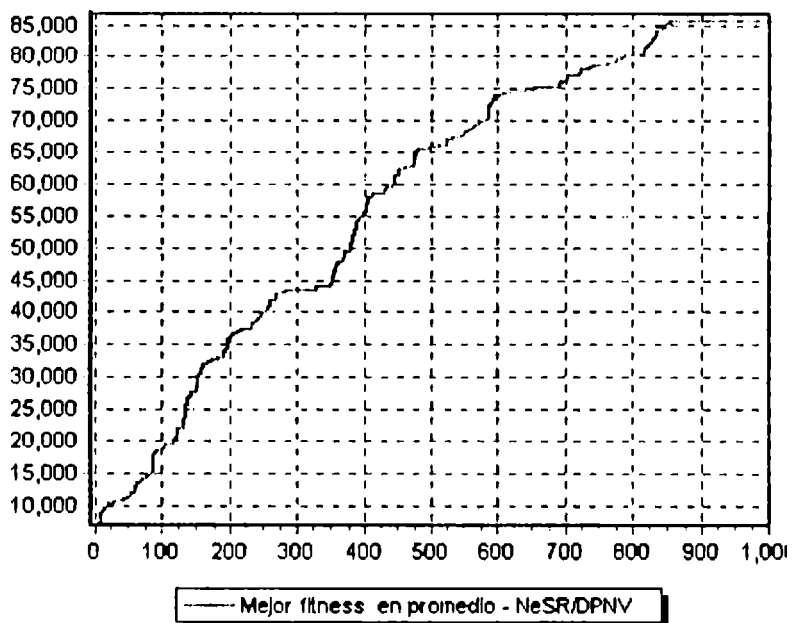


Figura 83: Experimento de NeSR con el balance del péndulo doble: Fitness máximo en promedio por generación en todas las corridas.

Anteriormente hemos hablado de una prueba para la efectividad del crossover. En las 30 corridas de esta prueba, se pudo obtener una buena solución en 10 oportunidades. En promedio necesitó 514 generaciones, siendo la mínima la generación 108 y 907 la máxima. Como se puede ver el desempeño se situó por debajo de la versión original del método corriendo con el mismo problema (balance del péndulo doble). Los gráficos correspondientes se encuentran en la Figura 84 y la Figura 85.

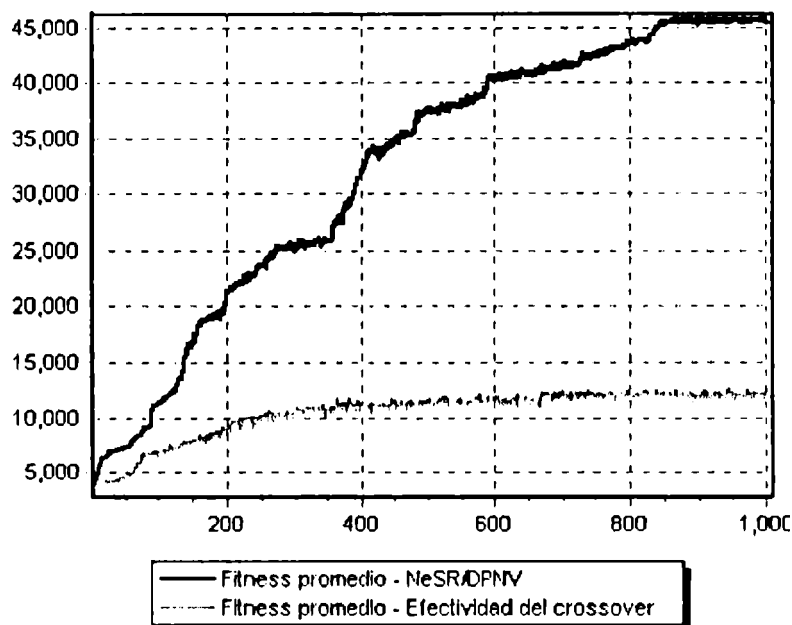


Figura 84: Experimento de efectividad del crossover en NeSR: Comparación del fitness promedio de las poblaciones por generación contra el método ejecutándose de manera normal en el mismo problema.

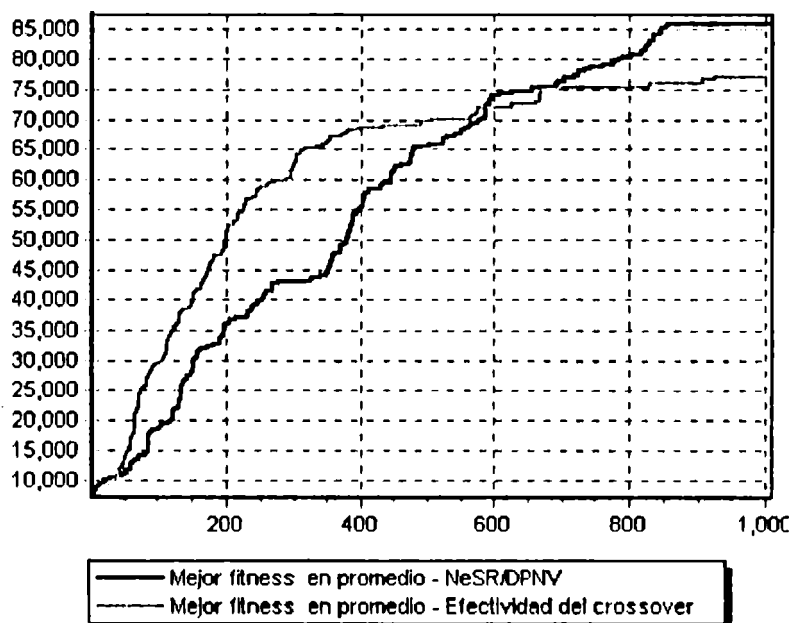


Figura 85: Experimento de efectividad del crossover en NeSR: Comparación del fitness máximo en promedio por generación en todas las corridas contra el método ejecutándose de manera normal en el mismo problema.

En líneas generales podemos afirmar que el método resuelve de manera satisfactoria los problemas en los que fue probado y que además dispone de un operador de crossover que resulta efectivo en la búsqueda genética. Para una visión global del desempeño puede observarse la Tabla 11.

En la misma tabla puede verse que la prueba de la efectividad crossover tuvo un pobre desempeño comparado con el método en su versión original.

Problema	Generación en promedio	Generación máxima	Generación mínima	Intentos exitosos	Intentos fallidos	Efectividad
XOR	34	195	1	45	5	90%
Bal. Péndulo	44	144	4	29	1	96%
Bal. Péndulo Doble	488	855	30	21	9	70%
Efectividad del crossover <sup>a</sup>	514	907	108	10	20	33.33%

Tabla 11: Síntesis de resultados obtenidos por el método NeSR.

Debemos destacar que no fue proporcionada información adicional al método para descubrir una estructura adecuada para la resolución de cada uno de los problemas, sino que el mismo método exploró tanto topologías de RNA como pesos de conexión.

## 9.8. NeSR vs. Neuroevolución basada en cadenas de bits

Para correr el experimento de neuroevolución basado en un algoritmo genético simple fue necesario fijar de antemano una cantidad de neuronas ocultas ya que el método así lo requiere. Esa estructura se mantiene fija en todas las RNA quedando para el proceso evolutivo la tarea de ajustar la conectividad entre las neuronas, es decir, habilitar o deshabilitar conexiones y definición de valores en los pesos de conexión. Hemos configurado para los dos problemas la cantidad de 2 neuronas ocultas. De esta manera queda prácticamente definida la estructura necesaria para resolver cualquiera de los tres problemas. Esto tiene como consecuencia que el espacio de búsqueda se reduce de manera considerable comparado con el de NeSR. Este factor hace que resuelva cada problema en una cantidad menor de tiempo. En la Tabla 12 pueden verse los resultados obtenidos por este método en las tareas de balancear uno y dos péndulos.

Problema	Generación en promedio	Generación máxima	Generación mínima	Intentos exitosos	Intentos fallidos	Efectividad
Bal. Péndulo	29	427	2	30	0	100%
Bal. Péndulo Doble	186	653	16	4	26	86.67%

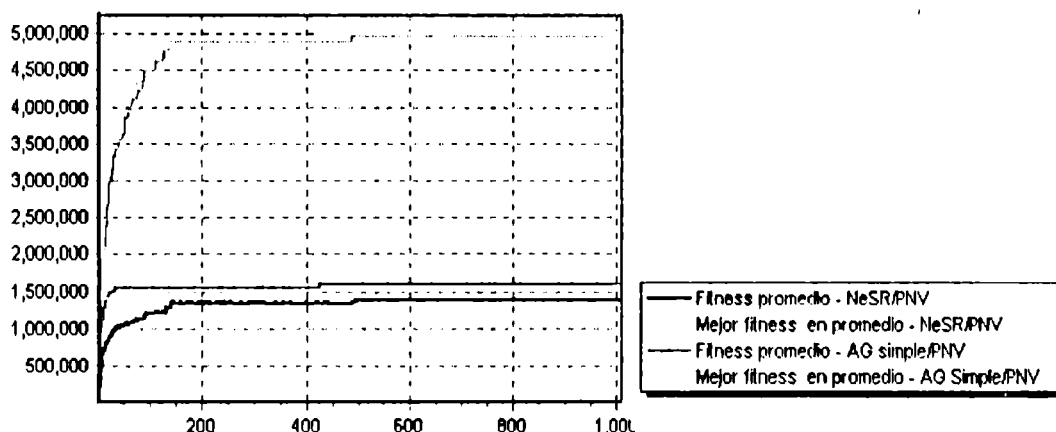
Tabla 12: Desempeño del método de neuroevolución basado en representación de cadenas de bits en los problemas de balance del péndulo.

<sup>a</sup> Prueba realizada con el problema del balance del péndulo doble sin información de velocidades.

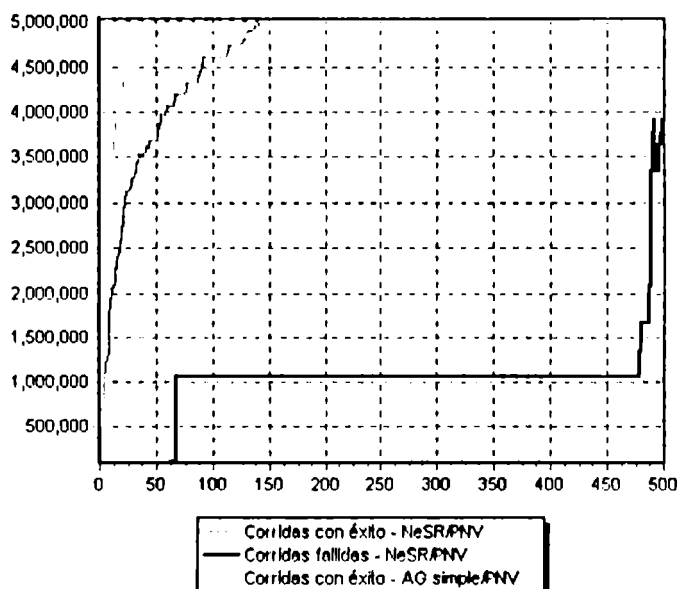
Sin embargo, si no se define la estructura suficiente o bien se utilizan muchas más neuronas que las necesarias, puede resultar imposible encontrar una solución con este método.

La Figura 86 y la Figura 87, en el problema del balance del péndulo, comparan NeSR contra método del algoritmo genético simple. Podemos ver que en general se comportaron de manera similar solo que a NeSR le tomó algunas generaciones más en encontrar una solución (Figura 87).

La Figura 88 y la Figura 89 muestran una comparación en el problema del balance del péndulo doble, en la cual se observa que en las mejores corridas NeSR estuvo por debajo del otro método ya que le tomó más tiempo encontrar una solución, aunque en el promedio general ambos métodos se comportaron de manera similar.



**Figura 86: Comparación entre NeSR y la neuroevolución mediante un algoritmo genético simple del mejor fitness en promedio y del fitness en promedio por generación en cada corrida del experimento del balance del péndulo.**



**Figura 87: Comparación entre NeSR y la neuroevolución mediante un algoritmo genético simple de los promedios de mejores fitness por generación en las corridas que llegan a resolver el problema del balance del péndulo por un lado y de las corridas que no llegan por el otro. Notar que ninguna corrida con el segundo método falló en encontrar una RNA adecuada para el problema.**



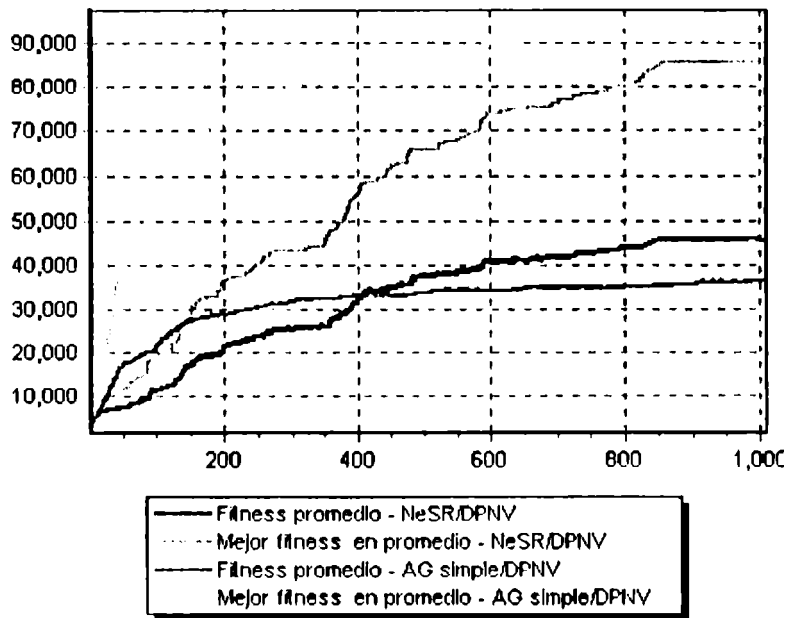


Figura 88: Comparación entre NeSR y la neuroevolución mediante un algoritmo genético simple del mejor fitness en promedio y del fitness en promedio por generación en cada corrida del experimento del balance del péndulo doble.

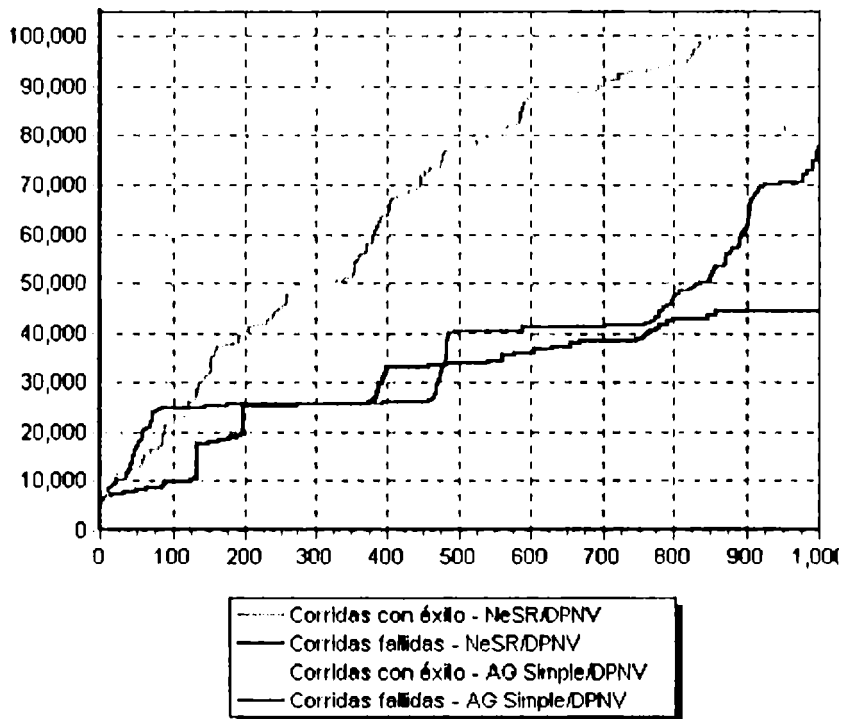


Figura 89: Comparación entre NeSR y la neuroevolución mediante un algoritmo genético simple de los promedios de mejores fitness por generación en las corridas que llegan a resolver el problema del balance del péndulo doble por un lado y de las corridas que no llegan por el otro.

Para experimentar con NEAT se utiliza la configuración que propone el autor del método en [124]. Al igual que en el caso anterior, si bien el método no restringe las topologías de las RNA generadas, tiene una cierta ayuda ya que los cromosomas NEAT se inicializan representando una estructura completamente interconectada entre las entradas y las salidas. Esto hace que resulte mucho más rápido llegar a una solución en los dos problemas de control planteados y con una pequeña modificación llegar a resolver la situación. Ver en la Tabla 13 los resultados obtenidos experimentando con éste método.

Problema	Generación en promedio	Generación máxima	Generación mínima	Intentos exitosos	Intentos fallidos	Efectividad
Bal. Péndulo	153	393	29	27	3	90 %
Bal. Péndulo Doble	217	930	64	30	0	100%

**Tabla 13: Desempeño del método NEAT en los problemas de balance del péndulo.**

Nuevamente debemos recordar que NeSR lo único que conoce acerca de la estructura de la red es que dispone de  $n$  entradas y  $m$  salidas dependiendo  $n$  y  $m$  del problema. A partir de allí debe explorar las conexiones adecuadas para llegar a una buena solución y por ello la diferencia en la cantidad de generaciones que se necesitan para resolver el problema.

Puede verse mejor una comparación entre el desempeño de NEAT y el de NeSR en la Figura 90 y la Figura 91 para el problema del balance del péndulo. En este problema NeSR estuvo por encima de NEAT en los mejores promedios, las corridas exitosas y en las fallidas, pero en promedio general NEAT obtiene individuos con mejor fitness.

En la Figura 92 y la Figura 93 se puede observar una comparación de ambos métodos para el problema del balance del péndulo doble. Como puede verse NEAT se situó por encima de nuestro método tanto en las corridas exitosas como en todos los promedios en general y que además ninguna corrida con NEAT falló en encontrar una RNA adecuada para el problema.

El tipo de codificación utilizada en NeSR tiene como efecto que en una población inicial es poco probable encontrar una red con una estructura aproximada a la necesaria para la resolución del problema. En NEAT ocurre todo lo contrario, para este problema en particular son pocas las modificaciones necesarias a las topologías iniciales para conseguir un buen desempeño en el problema.

Sin embargo encontramos que en muchas oportunidades NeSR resuelve el problema con una red que tiene una estructura de menor cantidad de conexiones comparada con la generada por NEAT.

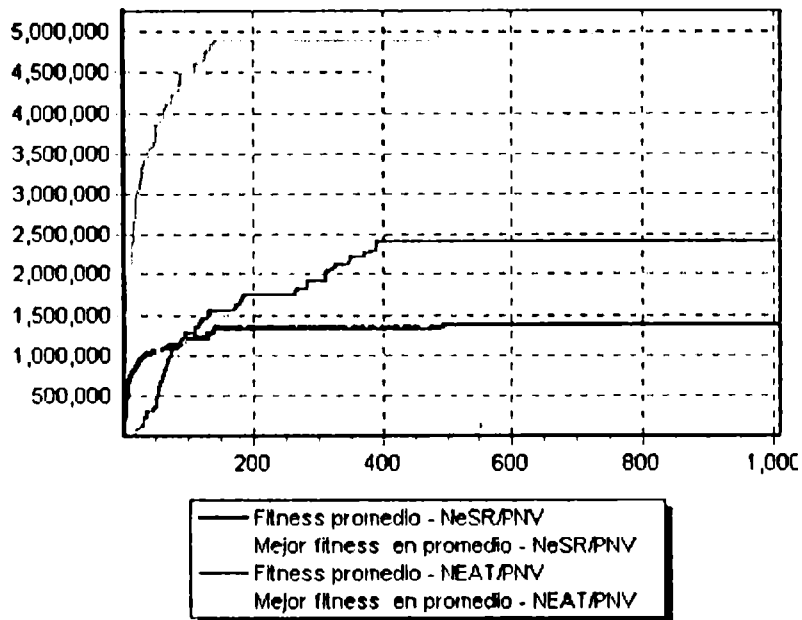


Figura 90: Comparación entre NeSR y NEAT del mejor fitness en promedio y del fitness en promedio por generación en cada corrida del experimento del balance del péndulo.

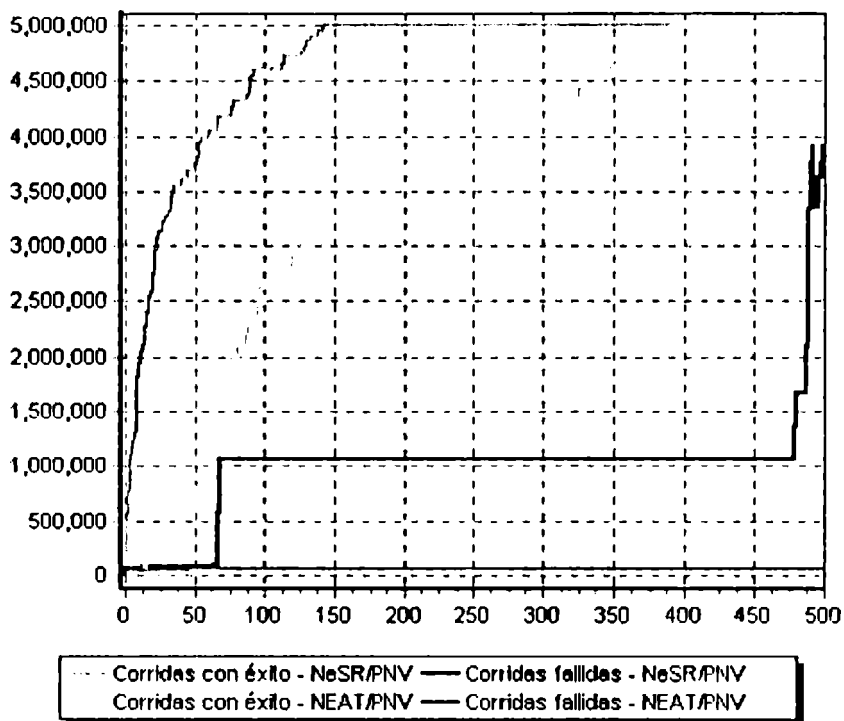


Figura 91: Comparación entre NeSR y NEAT de los promedios de mejores fitness por generación en las corridas que llegan a resolver el problema del balance del péndulo por un lado y de las corridas que no llegan por el otro.

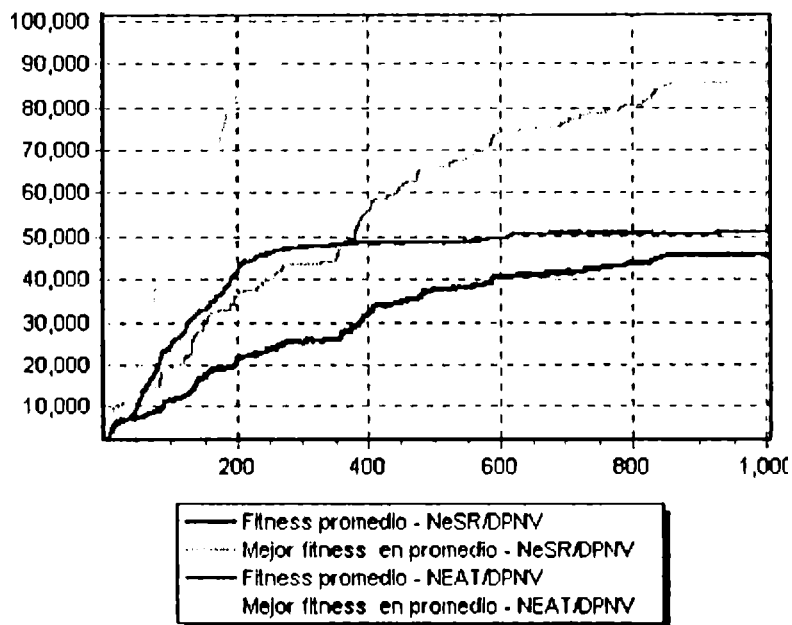


Figura 92: Comparación entre NeSR y NEAT del mejor fitness en promedio y del fitness en promedio por generación en cada corrida del experimento del balance del péndulo doble.

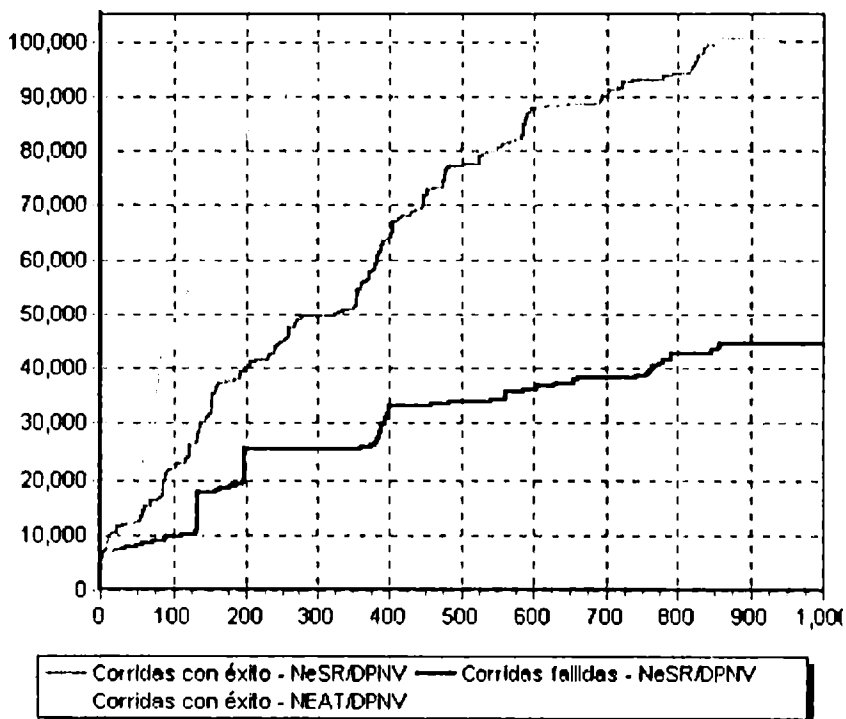


Figura 93: Comparación entre NeSR y NEAT de los promedios de mejores fitness por generación en las corridas que llegan a resolver el problema del balance del péndulo doble por un lado y de las corridas que no llegan por el otro. Notar que ninguna corrida con NEAT falló en encontrar una RNA adecuada para el problema.



## 10. Conclusiones y futuras líneas de trabajo

Los resultados obtenidos por los algoritmos genéticos los han hecho merecedores de reconocimiento como métodos viables para resolver problemas de búsqueda complejos ya que el hecho de trabajar sobre la codificación de las posibles soluciones –genotipo– y no sobre las soluciones mismas –fenotipo– facilita su aplicación prácticamente a cualquier problema, explotando una información fácilmente disponible, y al mismo tiempo permitiendo una gran independencia del problema.

Sin embargo se ha demostrado que un único algoritmo no puede ser la herramienta perfecta de optimización, es decir, adaptable a cualquier problema sin perder su efectividad, atentando esto contra la condición de robustez de los mismos.

Particularmente en el campo de la neuroevolución y especialmente en la evolución de redes neuronales representadas indirectamente, hemos encontrado que no existe un algoritmo genérico óptimo, y siempre será aconsejable su adaptación al problema a resolver. Por esto es que existen una infinidad de esfuerzos realizados en construir y probar diferentes estrategias neuroevolutivas.

No obstante se destaca la resolución eficiente de problemas por parte de las RNAE, ya que la búsqueda de una arquitectura y conjunto de pesos óptimo queda librada al proceso evolutivo. Como hemos visto en este trabajo, con la estrategia adecuada es posible encontrar la solución buscada. Entre otras ventajas de los métodos neuroevolutivos podemos mencionar que manejan mejor el problema de la búsqueda global en superficies del error complejas, multimodales y no diferenciables, disminuyendo la probabilidad de caer en mínimos locales, son más simples de implementar y pueden hibridarse fácilmente con algoritmos de entrenamiento tradicionales.

El paradigma de la neuroevolución ha demostrado fortaleza en campos de aplicación como las simulaciones de conductas observadas en animales inferiores, especialmente en el área de reconocimiento de patrones, clasificación y control automático. Pero también este paradigma presenta sus debilidades ya que su actuación ha sido mucho más débil en áreas en las que las herramientas de la Inteligencia Artificial clásica han simulado con éxito muchas capacidades del pensamiento humano de alto nivel como por ejemplo el análisis gramatical.

El paradigma de la neuroevolución presenta muchas posibles líneas de investigación. El hecho de encontrar evidencia sobre su capacidad para resolver un gran número de problemas, más eficientemente que otras técnicas, junto a discusiones abiertas con finales no resueltos, lo convierten en un objeto de estudio atractivo y muy interesante.

Pero además la neuroevolución es generadora de sus propias discusiones aún sin resolver. Existen investigadores que han presentado pruebas empíricas en donde el entrenamiento evolutivo puede ser significativamente más rápido y confiable que la regla delta generalizada. Sin embargo otros investigadores han presentado resultados donde las variantes más rápidas de Backpropagation son insuperables aún por hibridaciones AG-BP –una técnica que corre un Algoritmo Genético primero y luego se aplica el método de entrenamiento de Backpropagation–. Sin embargo otros artículos reportan excelentes resultados utilizando hibridación de entrenamiento evolutivo y algoritmos de gradiente descendente. Estas discrepancias señalan que no existe todavía un claro ganador en términos del mejor algoritmo de entrenamiento, sino que sus desempeños se ven afectados por la naturaleza del problema.

En cuanto al trabajo de investigación que hemos realizado con motivo de esta tesis podemos decir que:

- Hemos expuesto e implementado una nueva estrategia neuroevolutiva denominada NeSR basada en codificación indirecta que, a diferencia de los métodos convencionales, no requiere poseer conocimiento alguno de la estructura de la RNA que puede ser solución del problema a resolver. Además, NeSR tiene la posibilidad de intercambiar conjuntos de comandos para resolver diferentes problemas lo que le permite aprovechar el potencial de generar estructuras complejas de RNA.
- También hemos analizado y desarrollado un cromosoma, con forma de Sistema L, capaz de representar una red neuronal que evoluciona utilizando un Algoritmo Genético Simple. Para poder aplicarlo ha sido necesario especificar los operadores genéticos adecuados para esta representación. Podemos afirmar, luego de las pruebas realizadas para la efectividad del crossover (ver Pág. 155, *Headless Chicken*), que realmente conseguimos un operador que combina efectivamente los Sistemas L, teniendo éste un bajo nivel destructivo. Sin embargo creemos que pueden realizarse mayores refinamientos.
- Luego de analizar los resultados obtenidos, hemos comprobado que el esquema de codificación indirecta adoptado en NeSR permite explorar amplios espacios de búsqueda. Es importante tener en cuenta que pequeños cambios en el genotipo pueden tener un gran impacto en el fenotipo.
- Además, creamos un framework de evolución, en el cual se puede implementar cualquier estrategia evolutiva reescribiendo solamente el código relacionado a la innovación que la misma propone, sin importar el resto de los mecanismos que no se afectan, logrando así una buena reusabilidad del código. Junto con este framework incluimos una interfaz visual que facilita su uso.
- Hemos realizado exhaustivas pruebas de NeSR contra otros métodos, los cuales también estudiamos. En particular NEAT que se trata de un método reciente y capaz de resolver los problemas propuestos.
- Para reducir el alto tiempo computacional requerido por las pruebas mencionadas en el punto anterior, hemos tenido que recurrir a nuestro conocimiento de la programación paralela que, sumado al hardware adecuado, nos permitió recortar en gran medida los tiempos de procesamiento necesarios para la ejecución.

Igualmente creemos que existen muchos aspectos en NeSR para investigar. Entre otras cosas podemos mencionar:

- **Operadores genéticos:** Creemos que es posible refinar aún más los operadores genéticos, tal como los presentamos, de manera de hacer más efectiva su aplicación. Estos operadores constituyen el tema central del método propuesto en esta tesis y, si bien su desempeño actual es aceptable, sería deseable poder controlar mejor el impacto que tienen las modificaciones del genotipo sobre el fenotipo.

- **Comandos de construcción de RNA:** Durante el desarrollo del presente trabajo de grado hemos probado el método NeSR con una amplia variedad de conjuntos de comandos. El conjunto que seleccionamos resultó ser el más apto para el tipo de problemas en el que evaluamos el método. Sin embargo, queda abierta una futura línea de trabajo para establecer un mecanismo que permita definir conjuntos de comandos de construcción aplicables a un tipo de problema en particular. Especialmente comandos que exploten la potencialidad que posee el método de generar redes de estructuras complejas.

Así concluimos el presente trabajo de grado. Esperamos haber conseguido exponer de forma clara y precisa los conceptos del desarrollo realizado, y haber brindado una completa base para la comprensión del mismo. Además, creemos haber presentado un conjunto interesante de resultados obtenidos por nuestra estrategia ncuroevolutiva, los cuales muestran un buen rendimiento de la misma y esperamos que sirvan de motivación para realizar futuras investigaciones en esta dirección.



BIBLIOTECA  
FAC. DE INFORMÁTICA  
U.N.L.P.

DONACION..... LINT.  
\$.....  
Fecha..... 17-10-03  
Inv. E..... Inv. B..... 002953

TES
03/2

TES 03/2 DIF-02953 SALA	<p>UNIVERSIDAD NACIONAL DE LA PLATA FACULTAD DE INFORMÁTICA Biblioteca 50 y 720 La Plata catálogo:info.unlp.edu.ar biblioteca@info.unlp.edu.ar</p> <p>DIF-02953</p>
----------------------------------	---





# 11. Referencias

- [1] Anderson, C. W. "Learning to control an inverted pendulum using neural networks". IEEE Control Systems Magazine, 9: 31-37, 1989.
- [2] Arabas, J., Michalewicz, Z. y Mulawka, J., "GAVaPS – a Genetic Algorithm with Varying Population Size". Proceedings of the First IEEE International Conference on Evolutionary Computation, Volume 1, pp. 73-78, IEEE Service Center, Piscataway, NJ, 1994.
- [3] Atiya, A. F. & Parlos, A. G. "New Results on Recurrent Network Training: Unifying the Algorithms and Accelerating Convergence". IEEE Transactions on Neural Networks, Vol. 11, N° 3, 2000.
- [4] Baker, J. E. "Reducing bias and inefficiency in the selection algorithm. Genetic algorithms and their applications". Proceedings of the Second International Conference on Genetic Algorithms, pp. 14-21, 1987.
- [5] Baker, R. & Herman, G. T. "Simulation of organisms using a developmental model", parts I and II. Int. J. of Bio-Medical Computing, 3:201--215 and 251--267, 1972.
- [6] Baxter, J. "The evolution of learning algorithms for artificial neural networks". Complex Systems, pp.313-326, IOS Press, Amsterdam, 1992.
- [7] Beasley, D., Bull, D. R. & Martin, R. "An Overview of Genetic Algorithms". Department of Computing Mathematics, University of Cardiff, Department of Electrical and Electronic Engineering, University of Bristol. University Computing, 1993.
- [8] Belew, R. K. , McInerney, J. & Schraudolph N. N. "Evolving networks: using genetics algorithm with connectionist learning". Tech Rep. #CS90-174 (Revised), Computer Science & Engr. Dept. (C-014), Universidad of California at San Diego, La Jolla, CA 92093, USA, 1991.
- [9] Bell, A. "Plant form: An illustrated guide to flowering plants". Oxford University Press, Oxford, 1991.
- [10] Bosarge, W.E. Jr. "Adaptive Processes to Exploit the Nonlinear Structure of Financial Markets". Neural Networks and Pattern Recognition in Forecasting Financial Markets (February 1991), Santa Fe Institute of Complexity Conference, 1991.
- [11] Brindle, A., "Genetic Algorithms for Function Optimization", Doctoral Dissertation, University of Alberta, Edmonton, 1981.
- [12] Chalmers, D. J. "The evolution of learning: an experiment in genetic connectionism". Proceedings of the 1990 Connectionist Models Summer School, pp. 81-90, Morgan Kaufmann, San Mateo, CA, 1990.
- [13] Chen, S., Billings, S., & Grant, P. "Nonlinear system identification using neural networks". Int. J. Control. , 51(6):1191-1214, 1990.

- [14] Collard, J.E. "Commodity Trading with a Three Year Old. Neural Networks in Finance and Investment". R. Trippi and E. Turban, 411-420. Chicago: Probus Publishing Co., 1992.
- [15] Corbalán, L. "Evolución de redes neuronales para comandar criaturas que alcanzan objetivos sorteando obstáculos en un entorno virtual 2D". Tesis de Licenciatura. Facultad de Informática, Universidad Nacional de La Plata, 2002.
- [16] Cottrell, G.W., Munro P. & Zipser, D. "Image Compression by Backpropagation: an Example of extensional programming". Technical Report ICS 8702, Institute for Cognitive Science, University of California, San Diego, California, 1987.
- [17] Davis, L, "Handbook of Genetic Algorithms", Van Nostrand Reinhold, New York, 1991.
- [18] Davis, L., "Job Shop Scheduling with Genetic Algorithms", en Proceedings of the First International Conference on Genetic Algorithms, pp. 136-140, Lawrence Erlbaum Associates, Hillsdale, NJ, 1985.
- [19] De Jong, K. A., "An Analysis of the Behavior of a Class of Genetic Adaptive Systems", Doctoral dissertation, University of Michigan, 1975.
- [20] Dodd, N., Macfarlane, D. & Marland C. "Optimization of artificial neural network structure using genetics techniques implemented on multiple transputers". Proceeding of Transputing'91, pp. 687-700, IOS, Amsterdam, 1991.
- [21] Eiben, A. E., Raue, P. E., & Ruttkay, Z., "Genetic Algorithms with Multi-parent Recombination". Proceedings of the Third International Conference on Parallel Problem Solving from Nature, Lecture Notes in Computer Science, Vol. 866, pp. 78-87, Springer-Verlag, 1994.
- [22] Elman, J.L. "Finding structure in time". Cognitive Science 14, 179-211, 1990.
- [23] Eshelman, L. J., Caruana, R. A. & Schaffer, J. D. "Biases in the Crossover Landscape". Proceedings of the Third International Conference on Genetic Algorithms, pp. 10-19, Morgan Kaufmann Publishers, San Mateo, California, 1989.
- [24] Esquivel, S., Leiva, H. & Gallard, R. "Multiple crossovers between multiple parents to improve search in evolutionary algorithms". Proceedings of the 1999 Congress on Evolutionary Computation (IEEE). Washington DC, pp 1589-1594, 1999.
- [25] Fogel, D. B., "Evolving Artificial Intelligence", PhD Thesis, University of California, San Diego, 1992.
- [26] Fogel, L. J., Owens, A. J., & Walsh, M. J. "Artificial Intelligence through Simulated Evolution", John Wiley, Chichester, UK, 1966.
- [27] Foley, J. D. y Van Dam, A. "Fundamentals of interactive computer graphics", Addison-Wesley, Reading, Massachusetts, 1982.
- [28] Fontanari, J.F. & Meir, R. "Evolving a learning algorithm for the binary perceptrón". Network, vol 2. pp. 353-359, 1991.

- [29] Freeman, James A. & Skapura, David M. "Redes neuronales Algoritmos, aplicaciones y técnicas de programación". Addison-Wesley, 1991. Versión en español de: Rafael García -Bermejo Giner. Addison-Wesley Iberoamericana 1993.
- [30] Frijters, D. "Mechanisms of developmental integration of *Aster novae-angliae* L. and *Hieracium murorum* L". *Annals of Botany*, 42:561--575, 1978.
- [31] Frijters, D. "Principles of simulation of inflorescence development". *Annals of Botany*, 42:549--560, 1978.
- [32] Frijters, D. & Lindenmayer, A. "A model for the growth and flowering of *Aster novaeangliae* on the basis of table (1,0) L-systems". In G. Rozenberg and A. Salomaa, editors, *L Systems, Lecture Notes in Computer Science 15*, pages 24--52. Springer-Verlag, Berlin, 1974.
- [33] Frijters, D. & Lindenmayer, A. "Developmental descriptions of branching patterns with paracladial relationships". In A. Lindenmayer and G. Rozenberg, editors, *Automata, languages, development*, pages 57--73. North-Holland, Amsterdam, 1976.
- [34] Fukushima, K., Miyake S. & Ito T. "Neocognitron: A Neural Network Model for a mechanism of visual pattern recognition". *IEEE Transactions on System, Man and Cybernetics*, 1983.
- [35] Giles, C.L., C.B. Miller, D. Chen, H.H. Chen, G.Z. Sun, & Y.C. Lee, "Grammatical Inference Using Second-Order Recurrent Neural Networks," *Proceedings of the International Joint Conference on Neural Networks*, Seattle, Washington, IEEE Publication, vol. 2, p. 357, 1991.
- [36] Glover, D.E., "Optical Fourier/electronic neurocomputer machine vision inspection system". *Proceedings of the Vision 88 Conference*, Dearborn, MI, 1988.
- [37] Goldberg, D. E. & Lingle, R. "Alleles, Loci, and the TSP", en *Proceedings of the First International Conference on Genetic Algorithms*, pp. 154-159, Lawrence Erlbaum Associates, Hillsdale, NJ, 1985.
- [38] Goldberg, D. E. & Richardson, J. "Genetic algorithms with sharing for multimodal function optimization". In Grefenstette, J. J., editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 148-154, Morgan Kaufmann, San Francisco, California, 1987.
- [39] Goldberg, D. E., "Sizing Populations for Serial and Parallel Genetic Algorithms". *Proceedings of the Third International Conference on Genetic Algorithms*, pp. 70-79, Morgan Kaufmann Publishers, San Mateo, California, 1989.
- [40] Goldberg, D.E., "Genetic Algorithms in Search, Optimization and Machine Learning". Addison- Wesley Publishing Company, 1989.
- [41] Goldberg, D.E., "Optimal Initial Population Size for Binary-coded Genetic Algorithms". TCGA Report No. 85001, Tuscaloosa, University of Alabama, 1985.
- [42] Goldberg, D.E. & Deb, K. & Korb, B. "Do not worry, be messy". *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 24-30. Morgan Kaufmann publishers, San Mateo, California, 1991.

- [43] Gómez Martínez, M. "Redes Neuronales Artificiales" Contenido de los seminarios sobre redes neuronales artificiales dictados en San Nicolás y en Rosario. Universidad Tecnológica Nacional Facultad Regional San Nicolás Secretaría de Ciencia y Tecnología Grupo Ingeniería del Conocimiento, 1997.
- [44] Gomez, F. y Miikkulainen, R. "Incremental Evolution of Complex General Behavior". *Adaptive Behavior*, 5:317-342, 1997.
- [45] Grefenstette, J.J., "Optimization of Control Parameters for Genetic Algorithms". *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 16, No. 1, pp. 122-128, 1986.
- [46] Hamming, R.W. "Digital filters". Prentice-Hall Engelwood Cliffs, NJ, 1983.
- [47] Hanan, J. S. "Parametric L-systems and their application to the modelling and visualization of plants". PhD thesis, University of Regina, 1992.
- [48] Hanan, J. S. "PLANTWORKS: A Software for realistic plant modelling". Master's thesis, University of Regina, 1988.
- [49] Hanan, J. S. "Parametric L-systems and their application to the modeling and visualization of plants". PhD thesis, University of Regina, 1992.
- [50] Hancock, P. J. B., Smith, L. S. & Phillips, W. A. "A biologically supported error-correcting learning rule" *Proceedings of Int'l Conf. on Artificial Neural Networks. ICANN-91*, vol. 1, pp. 531-536, North-Holland, Amsterdam, 1991.
- [51] Harp, S.A., Samad, T & Guha, A. "Designing application-specific neural networks using the genetic algorithm". In *Advances in Neural Information Processing Systems 2*, pp. 447-454, Morgan Kaufmann, San Mateo, CA, 1990.
- [52] Harper, J. L. & Bell, A. D. "The population dynamics of growth forms in organisms with modular construction". In R. M. Anderson, B. D. Turner, and L. R. Taylor, editors, *Population dynamics*, pages 29--52. Blackwell, Oxford, 1979.
- [53] Haykin, S. "Neural Networks. A Comprehensive Foundation". Prentice Hall, 1994.
- [54] Hecht-Nielsen, R. "Neurocomputing". Addison-Wesley Publishing Company. 1991.
- [55] Herman, G. T. & Rozenberg, G. "Developmental systems and languages". North-Holland, Amsterdam, 1975.
- [56] Herman, G. T. & Liu, W. H. "The daughter of CELIA, the French flag, and the firing squad". *Simulation*, 21:33--41, 1973.
- [57] Holland, J. H., "Adaptation in Natural and Artificial Systems",. University of Michigan Press, 1975.
- [58] Honda, H. "Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body". *Journal of Theoretical Biology*, 31.331-338, 1971.
- [59] Hopcroft, J. E. & Ullman, J. D. "Introducción a la teoría de autómatas, lenguajes y computación". Versión en español de Homero Flores Samaniego Addison-Wesley, 1993.

- [60] Hoptroff, R. G., Bramson, M. J., & Hall, T. J. "Forecasting Economic Turning Points with Neural Nets". Proceedings of the IEEE International Joint Conference on Neural Networks (Seattle 1991), vol. I, 347-352, 1991.
- [61] Hung, S. L. & Adeli, H. "Parallel genetic/neural network learning algorithm for MIMD shared memory machines", IEEE Transactions on Neural Networks, vol. 5, no. 6, pp. 900-909, 1994.
- [62] Jain, A. K., Mao, J. y Mohiuddin, K. M. "Artificial neural networks: A tutorial". IEEE Computer, Marzo 1996.
- [63] Janssen, J. M. & Lindenmayer, A. "Models for the control of branch positions and flowering". Sequences of capitula in *Mycelis muralis* (L.) Dumont (Compositac). New Phytologist, 105:191--220, 1987.
- [64] Jones, T. "Evolutionary Algorithms, Fitness Landscapes and Search". PhD thesis, University of New Mexico, 1995.
- [65] Jordan M. "Attractor dynamics and parallelism in a connectionist sequential machine". Proceedings of the Eighth Annual Conference of the Cognitive Science Society. Amherst. MA. 531-546, 1986.
- [66] Kamijo, K. & Tanigawa, T. "Stock Price Pattern Recognition: A Recurrent Neural Network Approach". Proceedings of the IEEE International Joint Conference on Neural Networks (San Diego 1990), vol. I, 215-221. New York: IEEE, 1992.
- [67] Kernighan, B. W. & Ritchie, D. M. "The C programming language". Second edition. Prentice Hall, Englewood Cliffs, 1988.
- [68] Kinnebrock, W., "Accelerating the standard backpropagation method using a genetic approach". Neurocomputing, vol. 6, no. 5-6, pp. 583-588, 1994.
- [69] Kitano, H. "Designing Neural Networks using Genetic Algorithms with graph generation system", Complex Systems, vol 4, no. 4, pp. 461-476, 1990.
- [70] Kitano, H. "Empirical studies on the speed of convergente of neural network training using genetics algorithms". Proceedings of the Eight Nat'l Conf. On AI (AAAI-90), pp. 789-795, MIT Press, Cambridge, MA, 1990.
- [71] Kohonen, T. "Self-Organizing Maps". Springer-Verlag, 1989.
- [72] Koza, J. R., Genetic Programming, MIT Press, Cambridge, MA, 1992.
- [73] Koza, J.R., "Genetic Programming. On the Programming of Computers by Means of Natural Selection", MIT Press, 1992.
- [74] Lapedes, A. & Farber, R. "Nonlinear signal processing using neural networks: prediction and system modelling". Technical Report LA-UR-87-2662, Los Alamos National Laboratory, 1987.
- [75] Larrañaga, P. "Algoritmos Genéticos". Departamento de Ciencias de la Computación e Inteligencia Artificial Universidad del País Vasco.

- [76] Lindenmayer, A. "Adding continuous components to L-systems". In G. Rozenberg and A. Salomaa, editors, *L Systems, Lecture Notes in Computer Science 15*, pages 53--68. Springer-Verlag, Berlin, 1974.
- [77] Lindenmayer, A. "Algorithms for plant morphogenesis". In R. Sattler, editor, *Theoretical plant morphology*, pages 37--81. Leiden University Press, The Hague, 1978.
- [78] Lindenmayer, A. "Developmental algorithms for multicellular organisms: A survey of L-systems". *Journal of Theoretical Biology*, 54:3-22, 1975.
- [79] Lindenmayer, A. "Developmental algorithms: Lineage versus interactive control mechanisms". In S. Subtelny and P. B. Green, editors, *Developmental order: Its origin and regulation*, pages 219--245. Alan R. Liss, New York, 1982.
- [80] Lindenmayer, A. "Developmental systems without cellular interaction, their languages and grammars". *Journal of Theoretical Biology*, 30:455--484, 1971.
- [81] Lindenmayer, A. "Mathematical models for cellular interaction in development", Parts I and II. *Journal of Theoretical Biology*, 18:280--315, 1968.
- [82] Lindenmayer, A. "Models for multicellular development: Characterization, inference and complexity of L-systems". In A. Kelemenov'a and J. Kelemen, editors, *Trends, techniques and problems in theoretical computer science, Lecture Notes in Computer Science 281*, pages 138--168. Springer-Verlag, Berlin, 1987.
- [83] Lindenmayer, A. "Positional and temporal control mechanisms in inflorescence development". In P. W. Barlow and D. J. Carr, editors, *Positional controls in plant development*. University Press, Cambridge, 1984.
- [84] Lindenmayer, A. & Jürgensen, H. "Grammars of development: Discrete-state models for growth, differentiation and gene expression in modular organisms". In G. Rozenberg and A. Salomaa, editors, *Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology*, pages 3--21. Springer-Verlag, Berlin, 1992.
- [85] Lindenmayer, A. & Prusinkiewicz, P. "Developmental models of multicellular organisms: A computer graphics perspective". In C. G. Langton, editor, *Artificial Life*, pages 221--249. Addison-Wesley, Redwood City, 1988.
- [86] Macdonald, N. "Trees and networks in biological models". J. Wiley & Sons, New York, 1983.
- [87] Mahfoud, S. W., "Niching Methods for Genetic Algorithms", Ph.D. thesis, Department of General Engineering, IlliGAL Report 95001, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1995.
- [88] Mandelbrot, B. B. "The fractal geometry of nature". W. H. Freeman, San Francisco, 1982.
- [89] Michalewicz, Z., "Genetic Algorithms + Data Structures = Evolution Programs", Springer-Verlag, 1992.

- [90] Michalewicz, Z. y Janikow, C., "Genetic Algorithm for Numerical Optimization Problems with Linear Constraints", Communications of the ACM, 1992.
- [91] Michie, D. & Chambers, R.A. "BOXES: An experiment in adaptive control". In E. Dale and D. Michie, editors, Machine Intelligence. Oliver and Boyd, Edinburgh, UK, 1968.
- [92] Minsky, M. & Papert, S. "Perceptrons: Perceptrons: An Introduction to Computational Geometry ". MIT Press, Cambridge, MA, 1969.
- [93] Moriarty, D. E. & Miikkulainen, R., "Efficient reinforcement learning through symbiotic evolution", Machine Learning, 22:11-32.
- [94] Moriarty, D. E. "Symbiotic Evolution of Neural Networks in Sequential Decision Tasks". Dissertation, 1997.
- [95] Mühlenbein, H. "Adaptation in open systems: learning and evolution". Workshop Konnektionismus, pp. 122-130, GMD, Postfach 1240, D5205 St., Augustin, Germany, 1988.
- [96] Odom, M.D.& Sharda, R. "A Neural Network Model for Bankruptcy Prediction". Proceedings of the IEEE International Joint Conference on Neural Networks (San Diego 1990), vol. II, 163-168. New York: IEEE, 1990.
- [97] Oliver, I. M., Smith, D. J. & Holland, J. R. C. "A Study of Permutation Crossover Operators on the Traveling Salesman Problem". Proceedings of the Second International Conference on Genetic Algorithms, pp. 224-230, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.
- [98] Omatu, S. & Deris, S. "Stabilization of inverted pendulum by the genetic algorithm". Proceedings of the 1996 IEEE Conference on Emerging Technologies and Factory Automation, ETFA '96. Part 1(of 2), Piscataway, NJ, USA, pp. 282-287, IEEE Press, 1996.
- [99] Omatu, S. & Yoshioka, M. "Self-tuning neuro-PID control and applications". Proceedings of the 1997 IEEE International Conference on Systems, Man, and Cybernetics. Part 3 (of 5) Piscataway, NJ, USA, pp. 1985-1989, IEEE Press, 1997.
- [100] Osella Massa, G. L., García, E. A. & Lanzarini, L. "NeSR - Neuroevolución de Sistemas de Reescritura". IX Congreso Argentino de Ciencias de la Computación, pp. 693-704, 2003.
- [101] Osella Massa, G. L., García, E. A. & Lanzarini, L. "NeSR - Neuroevolución de Sistemas de Reescritura". Conferencia Latinoamericana de Informática, 2003.
- [102] Página de usuario de NEAT: <http://www.cs.utexas.edu/users/kstanley/neat.html>. Sección de implementaciones del método.
- [103] Palmer, A., Montañó, J.J. & Jiménez, R. "Tutorial sobre Redes Neuronales Artificiales: El Perceptrón Multicapa" Área de Metodología de las Ciencias del Comportamiento. Facultad de Psicología. Universitat de les Illes Balears. Publicado en Psicología.com REVISTA ELECTRÓNICA DE PSICOLOGÍA. Vol. 5, No. 2. ISSN 1137-8492, 2001.



- [104] Prusinkiewicz P. & Lindenmayer, A. "The algorithmic beauty of plants". Springer-Verlag, New York. With J. S. Hanan, F. D. Fracchia, D. R. Fowler, M-Verlag, New York, 1990. With J. S. Hanan, F. D. Fracchia, D. R. Fowler, M.J.M. de Boer, and L. Mercer, 1990.
- [105] Prusinkiewicz, P. & Hanan, J. "Lindenmayer systems, fractals, and plants". Volume 79 of Lecture Notes in Biomathematics. Springer-Verlag, Berlin, 1989.
- [106] Prusinkiewicz, P. & Hanan, J. "L-systems: From formalism to programming languages". In G. Rozenberg and A. Salomaa, editors, Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology pages 193--211. Springer-Verlag, Berlin, 1992.
- [107] Prusinkiewicz, P. & Hanan, J. "Visualization of botanical structures and processes using parametric L-systems". In D. Thalmann, editor, Scientific visualization and graphics simulation, pages 183--201. J. Wiley & Sons, Chichester, 1990.
- [108] Radcliffe, N. J. "Genetic Neural Networks on MIMD Computers" (compressed edition). PhD thesis, Dept. of Theoretical Phys. University of Edinburg, Scotland, UK, 1990.
- [109] Raup, D. M. "Geometric analysis of shell coiling: general problems". *Journal of Paleontology*, 40:1178--1190, 1966.
- [110] Raup, D. M. & Michelson, A. "Theoretical morphology of the coiled shell". *Science*, 147:1294--1295, 1965.
- [111] Rechenberg, I. "Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution", Frommann-Holzboog Verlag, Stuttgart, 1973.
- [112] Rozenberg, G. & Salomaa, A. "The mathematical theory of L-systems". Academic Press, New York, 1980.
- [113] Rozenberg, G. & Salomaa, A. "When L was young". In G. Rozenberg and A. Salomaa, editors, *The book of L*, pages 383--392. Springer-Verlag, Berlin, 1986.
- [114] Rueda A., Pedrycz W., Nelly R., & Carelli, R. "Control of a robot manipulator via a stable fuzzy-neuro controller". *Proceedings of the Conference on Modelling and Simulation 1994*. Conference on Modelling and Simulation, 1994.
- [115] A. Salomaa. *Formal languages*. Academic Press, New York, 1973.
- [116] Sarle, W. S. "Frequently Asked Questions". [news://comp.ai.neural-nets](http://news://comp.ai.neural-nets), 1997.
- [117] Schafer, R. W. & Rabiner, L. R. "Digital processing of Speech Signals". Prentice-Hall, Englewood Cliffs N.J., 1978.
- [118] Schaffer, J.D., Caruana, R.A. & Eshelman, L. J. "Using genetic search to exploit the emergent behavior of neural networks". *Physica D*, vol. 42, pp. 244-248, 1990.
- [119] Schwefel, H. P., "Numerical Optimization for Computer Models", John Wiley, Chichester, UK, 1981.

- [120] Sejnowski, T.J. & Rosenberg, C.R. "Parallel Networks that learn to pronounce English text". *Complex Systems*, 1:145-168, 1987.
- [121] Skinner, A. J. & Broughton, J. Q., "Neural networks in computational materials science: Training algorithms". *Modelling and Simulation in Materials Science and Engineering* vol.3, no. 3, pp. 371-390, 1995.
- [122] Smith, A. R. "Plants, fractals, and formal languages". Proceedings of SIGGRAPH '84 (Minneapolis, Minnesota, July 22--27, 1984) in *Computer Graphics*, 18, 3 (July 1984), pages 1--10, ACM SIGGRAPH, New York, 1984.
- [123] Spears, W. M. y De Jong, K. A., "On the Virtues of Parametrized Uniform Crossover". Proceedings of the Fourth International Conference on Genetic Algorithms, pp. 230-236, Morgan Kaufmann Publishers, San Mateo, California, 1991.
- [124] Stanley, K.O. & Miikulainen R. "Efficient reinforcement learning through evolving neural network topologies". Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002). San Francisco, CA, Morgan Kaufmann, 2002.
- [125] Tishby, N. "A dynamical systems approach to speech processing". Proceedings Int. Conf. on Acoustic Speech Signal Processing, pages 365-368, 1990.
- [126] Townshend, B. "Nonlinear prediction of speech". Proceedings Int. Conf. on Acoustic Speech Signal Processing, pages 425-428, 1991.
- [127] Von Koch, H. "Une m'ethode g'eom'etrique 'el'ementaire pour l'etude de certaines questions de la th'eorie des courbes planes". *Acta Mathematica*, 30:145--174, 1905.
- [128] Waller, D. M. & Steingraeber, D. A. "Branching and modular growth: Theoretical models and empirical patterns". In J. B. C. Jackson and L. W. Buss, editors, *Population biology and evolution of clonal organisms*, pages 225--257. Yale University Press, New Haven, 1985.
- [129] White, H., "Economic Prediction Using Neural Networks: The Case of IBM Daily Stock Returns". Proceedings of the IEEE International Conference on Neural Networks. (July 1988), vol. II, 451-458, 1988.
- [130] Whitley, D., "The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best". Proceedings of the Third International Conference on Genetic Algorithms, pp. 116-121. Morgan Kaufmann publishers, San Mateo, California, 1989.
- [131] Wright, A. H., "Genetic Algorithms for Real Parameter Optimization", en Proceedings of the Fourth International Conference on Genetic Algorithms, pp. 205-218. Morgan Kaufmann publishers, San Mateo, California, 1991.
- [132] Yao, X. & Liu, Y "Towards designing artificial neural networks by evolution". *Applied Mathematics and Computation*, vol. 91, no. 1, pp 83-90, 1998.
- [133] Yao, X. "Evolving Artificial Neural networks". School of Computer Science. University of Birmingham Edgbaston, Birmingham B15 2TT. Proceedings of the IEEE, 87(9):1423-1447, 1999.