FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: Controladores Obtenidos por Neuroevolución

AUTORES: A.C. Hernán Luis Vinuesa

DIRECTOR: Prof. Lic Laura C. Lanzarini

CARRERA: Licenciatura en Sistemas

Resumen

El aporte central de esta tesis radica en la definición de estrategias evolutivas que permiten obtener controladores neuronales aplicables directamente al área de la Robótica. A partir del método NEAT (NeuroEvolution of Augmenting Topologies) se ha definido una nueva estrategia con capacidad para combinar módulos neuronales entrenados previamente. El resultado de esta combinación permite obtener una arquitectura adecuada en menor tiempo. Como una segunda alternativa para reducir el tiempo de obtención del controlador se propone combinar las primeras etapas de evolución de NEAT con evolución por torneo binario.
Finalmente, se plantea el uso de una minipoblación de controladores para lograr una adaptación a entornos dinámicos. Los resultados obtenidos fueron aplicados sobre un robot Kephera II con resultados satisfactorios.

Líneas de Investigación

- Análisis de los frameworks existentes para Robótica Evolutiva en lo que hace a definición de escenarios, interacción con robots específicos, plataformas de desarrollo y posibilidades de desarrollos multi-agentes.
- Estudio e investigación de experiencias realizadas en la obtención de controladores en el área de la robótica evolutiva haciendo hincapié en el dinamismo de la representación y en el conocimiento previo necesario para resolver el problema.
- Combinación de distintas arquitecturas para resolver un único problema. Interesa especialmente la reducción del tiempo de adaptación.
- Investigación de nuevas estrategias evolutivas para resolver problemas complejos a través de la combinación de controladores elementales buscando reusar el conocimiento adquirido minimizando el tiempo de adaptación.

Trabajos Realizados

Como resultado de las investigaciones realizadas se han registrado las siguientes publicaciones:
- Improving Controllers based on Neural Networks obtained by Parallel Evolution Strategy. Vinuesa et all. XXVII Jornadas Chilenas de Computación 2008. Chile. Noviembre de 2008.
- Improving Controllers based on Neural Networks obtained by Layered Evolution. Vinuesa et all, XXXIV Conferencia Latinoamericana de Informática. CLEI 2008. Argentina. Septiembre de 2008.
- Continuous Evolution of Neural Modules for Autonomous Robot Controllers. Vinuesa et all. XIII Congreso Argentino de Ciencias de la Computación. CACIC 2007. Argentina. Octubre de 2007. ISBN: 978-950-656-109-3. pp. 1420-1428.
- Control de Robots Autónomos a Partir de la Evolución Continua de Módulos Neuronales. Vinuesa et all. XV Jornadas de Jóvenes Investigadores Asociación de Universidades Grupo Montevideo (AUGM). Paraguay. Octubre de 2007.
- Neural Networks Elitist Evolution. Vinuesa et all. 29th Internacional Conference Information Technology Interfaces. ITI 2007. IEEE: 07EX1589. ISBN: 978-953-7138-09-7. ISSN: 1330-1012. (CD Rom). IEEE Computer Society Press. pp. 457-462.

Conclusiones

Se han presentado distintas estrategias evolutivas que permiten obtener un controlador de un robot autónomo, así como diferentes maneras de descomponer un problema en partes reduciendo la complejidad de las soluciones y el tiempo en el que se obtienen los controladores. Estas estrategias han demostrado ser una buena alternativa a la hora resolver problemas de control de robots, ya que la comparación con otras estrategias evolutivas convencionales, tanto en ambientes de simulación como en ambientes reales, han arrojado resultados satisfactorios.

Trabajos Futuros

A partir de las estrategias descriptas en esta tesis sería deseable establecer un mecanismo adecuado para combinar las funciones de fitness de cada uno de los módulos. También resulta de interés la definición de estrategias de evolución multiobjetivo con capacidad para operar con módulos neuronales. Actualmente se está trabajando en la combinación de la estrategia de Evolución Continua con la estrategia de NEAT con Torneo para que el controlador evolucione a lo largo de su vida útil. Además, se están evaluando técnicas de procesamiento distribuido, así como distintas estrategias para la paralelización de metaheurísticas.

Fecha de la presentación: Diciembre de 2009

Controladores Obtenidos por Neuroevolución

Autor: A. C. Hernán Luis Vinuesa

Directora: Prof. Lic. Laura C. Lanzarini

Tesina de Licenciatura de Sistemas
Facultad de Informática
UNIVERSIDAD NACIONAL DE LA PLATA

1 de noviembre de 2009

Resumen

Esta tesis describe distintas estrategias evolutivas que permiten obtener controladores neuronales aplicables directamente al área de la Robótica.
En particular se utiliza como base el método NEAT (NeuroEvolution of Augmenting Topologies) por su capacidad para generar controladores de tamaño mínimo. Este método, si bien ha sido ampliamente utilizado, se caracteriza por requerir un tiempo de cómputo importante y por generar controladores que carecen de la habilidad de adaptarse al entorno de información. Por tales motivos, su aplicación en el área de Robótica no es una tarea simple.

El aporte central de esta tesis es la definición de nuevas estrategias evolutivas que resuelvan estos problemas facilitando la obtención de los controladores correspondientes.
Con respecto a la reducción del tiempo de cómputo, una de las modificaciones centrales radica en la incorporación del concepto de módulo a la representación del método NEAT. De esta forma, la solución del problema reside en la combinación de elementos básicos previamente entrenados lo que reduce considerablemente el tiempo necesario para la evolución. Una segunda alternativa para reducir el tiempo de obtención del controlador es utilizar las primeras etapas de evolución de NEAT como herramienta para definir una arquitectura mínima y luego pasar a otra estrategia de evolución más rápida como puede ser la evolución por torneo.
Con respecto a la adaptación a un entorno dinámico, se propone una nueva estrategia basada en una minipoblación de controladores obtenidos a partir de NEAT. Los resultados obtenidos fueron aplicados sobre un robot Khepera II con resultados satisfactorios.

Los métodos propuestos y descriptos en esta tesina, as como los resultados de las investigaciones referidas al funcionamiento de los operadores genéticos involucrados, han sido publicados es diferentes congresos informáticos nacionales e internacionales [53] [54] [75] [97] [98] [99] [100].

Prefacio

Esta tesina resume una gran parte de mis investigaciones realizadas en el campo de la Robótica Evolutiva durante mi período como Becario del III-LIDI y posteriormente como Becario de la CIC.
Para facilitar la lectura y comprensión de los métodos que he desarrollado, incluí en los primeros cuatro capítulos de esta tesina los conceptos teóricos básicos relacionados.
A partir del capítulo 5, se detallan las estrategias evolutivas que constituyen el aporte central de esta tesina. El recorrido utilizado para su presentación es el mismo que me llevó a definirlas.
Cuando estudié el método NEAT por primera vez, me encontré con una estrategia que resuelve uno de los principales problemas de la evolución de Redes Neuronales: la aplicación de los operadores genéticos. NEAT define una representación muy útil que permite reconocer el origen de cada parte de una red obtenida por evolución. Esto facilita la combinación de individuos. Además, tiene la particularidad de asegurarse de obtener resultados con tamaño mínimo. Hasta aquí, todo funcionó perfectamente pero cuando tuve que llevarlo a un robot real me encontré con que el tiempo de procesamiento era un problema central. Como forma de resolverlo, junto con Germán Osella Massa, extendimos la representación de NEAT para que soportara el concepto de Módulo Neuronal. La integración automática de módulos es lo que publicamos bajo el título ”Modular Creation of Neuronal Networks for Autonomous Robot Control”. El objetivo de este enfoque fue construir un repositorio de módulos neuronales que puedan combinarse para obtener distintos comportamientos, reduciendo de esta forma el costo de entrenamiento. En esa oportunidad, el énfasis estuvo puesto en el análisis de situaciones donde participan varios agentes con objetivos distintos. En estos casos, la evolución modular permitió combinar comportamientos generales con otros específicos de cada tarea a desarrollar.
Posteriormente y con el objetivo de dar al controlador la capacidad de adaptarse a los cambios dinámicos del entorno, trabajé en la evolución del controlador a lo largo de su vida útil combinando un método basado en evolución de módulos neuronales con un algoritmo evolutivo específico. El primer método fue utilizado para producir el controlador mientras que el segundo lo ajustaba durante su funcionamiento. Como resultado, obtuve un controlador basado en una red neuronal que posee una arquitectura mínima y capacidad de adaptación en la fase de ejecución. Este método fue publicado bajo el título ”Continuous Evolution of Neural Modules of Autonomous Robot Controllers”.
Los operadores genéticos utilizados en los desarrollos anteriores fueron analizados de manera independientes como se describe en el artículo ”Neural Networks Elitist Evolution”. A partir de este análisis se efectuó la selección de operadores

que finalmente fueron considerados en la mayoría de las estrategias propuestas. Todas esta variantes de NEAT son las que se describen en el capítulo 5.
El III-LIDI dispone de cuatro robots Khepera II sobre los cuales busqué aplicar los resultados de los controladores obtenidos. Esto me llevó en una etapa inicial a extender el simulador YAKS para que los incluyera. El uso del simulador en una etapa inicial me ahorró mucho tiempo ya que no es necesario esperar a que el robot ejecute cada orden. Afortunadamente el simulador, si está bien calibrado, se ajusta de una forma muy precisa a los movimientos reales del robot. Problemas sencillos tales como evasión de obstáculos y alcance de objetivos fueron resueltos satisfactoriamente.
Sin embargo, cuando la complejidad del problema aumenta, el tiempo requerido para obtener el controlador se incrementa considerablemente. Por eso, el capítulo 6 describe una manera de resolver esta situación combinando NEAT con Torneo binario. De esta forma, se evoluciona una arquitectura de tamaño mínimo determinada inicialmente por NEAT a través de una estrategia de menor tiempo de cómputo.
Finalmente el capítulo 7 contiene las conclusiones así como algunas líneas de trabajo futuras.

Esta es la lista de publicaciones referidas a mis investigaciones en Robótica Evolutiva

- Trabajos publicados en revistas periódicas
  - *Modular Creation of Neuronal Networks for Autonomous Robot Control*. Osella Massa Germán Leandro, Vinuesa Hernán Luis, Lanzarini Laura. Revista Iberoamericana de Inteligencia Artificial. Vol. 11, No. 35 (2007), pp. 43-53. ISSN: 1137-3601.
- Trabajos publicados en actas de congresos con referato internacional.
  - *Improving Controllers based on Neural Networks obtained by Parallel Evolution Strategy*. Hernán Vinuesa, Laura Lanzarini, Waldo Hasperué, Leonardo Corbalán. XXVII Jornadas Chilenas de Computación. JCC 2008. Punta Arenas. Chile. 10 al 15 de Noviembre de 2008. También publicado en XIV Congreso Argentino de Ciencias de la Computación. CACIC 2008. Chilecito, La Rioja. Argentina. 6 al 10 de Octubre de 2008.
  - *Improving Controllers based on Neural Networks obtained by Layered Evolution*. Hernán Vinuesa, Laura Lanzarini, Germán Osella Massa.

XXXIV Conferencia Latinoamericana de Informática. CLEI 2008. Santa Fe. Argentina. 8 al 12 Septiembre de 2008.

- *Continuous Evolution of Neural Modules for Autonomous Robot Controllers*. Hernán Vinuesa, Germán Osella Massa, Leonardo Corbalán, Laura Lanzarini. XXVI Jornadas Chilenas de Computación. JCC 2007. Iquique. Chile. 5 al 10 de Noviembre de 2007. También publicado en XIII Congreso Argentino de Ciencias de la Computación. CACIC 2007. Corrientes y Resistencia. Argentina. 1 al 5 de Octubre de 2007. ISBN: 978-950-656-109-3. pp. 1420-1428.
- *Control de Robots Autónomos a Partir de la Evolución Continua de Módulos Neuronales*. Hernán Vinuesa, Germán Osella Massa, Leonardo Corbalán, Laura Lanzarini. XV Jornadas de Jóvenes Investigadores Asociación de Universidades Grupo Montevideo (AUGM). Núcleo disciplinario: Ingeniería Mecánica y de la Producción. Campus de la Universidad Nacional de Asunción, Asunción, Paraguay. 24 al 26 de Octubre de 2007.
- *Neural Networks Elitist Evolution*. Vinuesa Hernán, Lanzarini Laura. 29th Internacional Conference Information Technology Interfaces. ITI 2007. Dubrovnik. Crocia. 25 al 28 de Junio de 2007. Catalogo IEEE: 07EX1589. ISBN: 978-953-7138-09-7. ISSN: 1330-1012. (CD Rom). Artículo completo Publicado por IEEE Computer Society Press. pp. 457-462.
- *Modular Creation of Neuronal Networks for Autonomous Robot Control*. Osella Massa Germán Leandro, Vinuesa Hernán Luis, Lanzarini Laura. 3rd IEEE Latin American Robotics Symposium. LARS 2006. Santiago de Chile. Chile. 26 de Octubre de 2006. ISBN: 1-4244-0537-8. pp. 66-73. También publicado en XII Congreso Argentino de Ciencias de la Computación. CACIC 2006. San Luis. Argentina. Octubre 2006. ISBN: 950-609-050-5. pp. 1255-1266.

- Trabajos publicados en actas de congresos con referato nacional.

  - *Sistemas Inteligentes. Aplicaciones en Minería de Datos, Robótica Evolutiva y Redes de Computadoras*. Lanzarini Laura, Hasperue Waldo, Vinuesa Hernán, Corbalán Leonardo,Osella Massa Germán. X Workshop de Investigadores en Ciencias de la Computación WICC 2008. General Pico. Mayo 2008.
  - *Sistemas Inteligentes basados en Neurocomputación*. Lanzarini Laura, Corbalan Leonardo, Osella Massa Germán, Hasperué Waldo, Vinue-

sa Hernán. IX Workshop de Investigadores en Ciencias de la Computación WICC 2007. Trelew. Argentina. Mayo 2007.

Índice general

Capítulo 1

Introducción a las metaheurísticas

La resolución de problemas de optimización es de gran interés en la actualidad y ha motivado el desarrollo de diversos métodos informáticos para tratar de resolverlos. Generalmente en optimización se está frente a la búsqueda de la mejor solución (o lo suficientemente buena) al problema dado, entre muchas alternativas. Los problemas de optimización pueden ser modelados en un conjunto de variables de decisión con sus dominios y limitaciones en relación a las variables de configuración. Naturalmente se dividen en tres categorías:

1. Problemas que tienen exclusivamente variables discretas.
2. Problemas que tienen exclusivamente variables continuas.
3. Problemas que tienen tanto variables discretas tanto como continuas.

Los métodos para resolver problemas de optimización fueron desarrollados originalmente para problemas de clase 1, también conocidos como problemas combinatorios de optimización o problemas CO. Un problema CO puede expresarse como $P = (S, f)$ y es un problema de optimización en el cual se da un conjunto finito de objetos o elementos S y una función objetivo $f : S \longmapsto \mathbb{R}^+$ que asigna un valor de costo positivo a cada objeto $s \in S$. El objetivo es encontrar un elemento con valor de costo mínimo. Los objetos o elementos son típicamente: números enteros, subconjunto de un conjunto de ítems, permutaciones de un conjunto de ítems o estructuras gráficas. Un ejemplo bien conocido es el problema del viajante (Traveling Salesman Problem, TSP [55]).
Debido a la importancia práctica de los problemas CO, se han desarrollado muchos algoritmos para enfrentarlos. Estos algoritmos pueden dividirse en exactos y aproximados.
Los algoritmos exactos garantizan encontrar para cada instancia de un problema CO de tamaño finito una solución óptima en un tiempo limitado [70][76]. Sin embargo, para problemas CO que son $NP - hard$ [28], no existe un algoritmo de

tiempo polinomial, asumiendo $P \neq NP$. Por lo tanto, los métodos exactos necesitan, en el peor caso, un tiempo computacional exponencial. Esto a menudo resulta inviable para propósitos prácticos. Por lo tanto, el uso de métodos aproximados para solucionar problemas CO ha recibido más y más atención en los últimos 30 aros. En los métodos aproximados, se sacrifica la garantía de encontrar soluciones óptimas en pro de tener buenas soluciones con una reducción significativa en la cantidad de tiempo requerido para cumplir con la tarea. Entre los métodos aproximados básicos usualmente se distingue entre *heurísticas constructivas* y *métodos de búsqueda local*.

- Las heurísticas constructivas son típicamente los métodos de aproximación más rápidos. Generan soluciones desde cero mediante el agregado de componentes oportunamente definidos a una solución parcial inicialmente vacía. Esto finaliza antes de que la solución se complete o que se satisfaga otro criterio de detención.

- Los algoritmos de búsqueda local comienzan a partir de una solución inicial e iterativamente tratan de reemplazarla con una mejor obtenida dentro de un vecindario apropiado. Dicho vecindario es definido como una estructura o como una solución mínima a nivel local.

En la década del '70, surgió una nueva clase de algoritmos de aproximación cuyo funcionamiento se basó en la combinación de métodos heurísticos básicos en marcos de nivel superior encaminados hacia una eficiente y eficaz exploración del espacio de búsqueda. Estos métodos, hoy en día, son conocidos como *metaheurísticas*. El término *metaheurística* [31] proviene de la composición de dos palabras griegas: *heurística* que deriva del verbo *heuriskein* que significa "encontrar", mientras que el prefijo *meta* significa "más allá, en un nivel superior". La clase de los algoritmos metaheurísticos incluye (pero no está restringida) Optimización Basadas en Colonias de Hormigas (Ant Colony Optimization, ACO), Computación Evolutiva (Evolutionary Computation, EC) incluyendo a los Algoritmos Genéticos (Genetic Algorithms, GAs), Búsqueda Local Iterativa (Iterated Local Search, ILS), Recocido Simulado (Simulated Annealing, SA), y Búsqueda Tabú (Tabu Search, TS). Las diferentes descripciones de las metaheurísticas encontradas en la literatura permiten extraer algunas propiedades fundamentales que las caracterizan:

- Las metaheurísticas son estrategias que "guían" el proceso de búsqueda.

- El objetivo es explorar eficientemente el espacio de búsqueda con el fin de encontrar (acercarse a) soluciones óptimas.

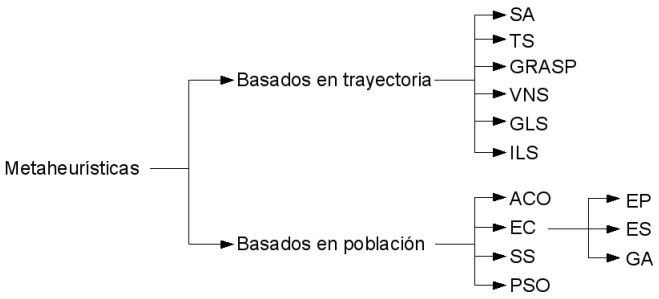

Figura 1.1: Clasificación de las metaheurísticas.

- Las técnicas que constituyen el rango de los algoritmos metaheurísticos van desde procedimientos simples de búsqueda local a complejos procesos de aprendizaje.
- Los algoritmos metaheurísticos son aproximados y usualmente no deterministas.
- Deben incorporar mecanismos para evitar estancarse en áreas confinadas del espacio de búsqueda.
- El concepto básico de las metaheurísticas puede ser descripto en un nivel abstracto (por ejemplo, no atados a un problema específico).
- Las metaheurísticas deben hacer uso de un dominio especifico del conocimiento en la forma en que las heurísticas son controladas por estrategias de niveles superiores.
- Las metaheurísticas no son un problema especifico.
- En estos días, muchas metaheurísticas avanzadas usan la experiencia de búsqueda (embebidos en alguna forma de memoria) para guiar el proceso.

En este capítulo se presenta una introducción al concepto de técnicas metaheurísticas de optimización que se ven en la figura 1.1, llegando a explotar la clasificación que las agrupa en aquellas basadas en la trayectoria y aquellas basadas en la población [5].

1.1. Metaheurísticas basados en trayectoria

El término *métodos basados en trayectoria* se refiere a la manera en que se realiza el proceso de búsqueda. La mayoría de estos métodos son extensiones de

procedimientos interactivos simples mejorados, cuyo desempego es usualmente insatisfactorio. Estos incorporan técnicas que le permiten al algoritmo escapar de mínimos locales. Esto implica la necesidad de un criterio de terminación que no sea simplemente llegar a un mínimo local. Comúnmente se usan criterios de terminación como máximo tiempo de CPU, máximo número de iteraciones, una solución de suficiente calidad o un máximo número de iteraciones de búsqueda sin mejoras [5].

Simulated Annealing (SA)

Es una de las primeras metaheurísticas que ha utilizado una estrategia explícita para escapar de mínimos locales, presentada por primera vez como un algoritmo de búsqueda para problemas CO en [50] y en [15]. Su funcionamiento se basa en en el proceso físico de tratamiento térmico de materiales, en la cual dicho material se calienta hasta alcanzar su límite máximo de energía y luego se enfría gradualmente hasta llegar a un límite inferior preestablecido. Para evitar quedar atrapado en mínimos locales, se permite que durante la etapa de enfriamiento se realicen movimientos hacia soluciones con un valor de función objetivo más desfavorable que el valor actual. Ese movimiento es llamado movimiento hacia arriba. En cada iteración, una solución *s'* que pertenece al vecindario de *s* es elegida al azar. Si *s'* es mejor que *s* entonces *s'* se acepta como la nueva solución actual; de lo contrario, *s'* es aceptada con una probabilidad la cual es el parámetro de una función de temperatura [5] [41].

Tabu Search (TS)

Es una de las metaheurísticas más exitosas para la aplicación de problemas CO introducida en [31], basado en ideas formuladas en [30]. La idea básica de TS es un uso explicito del historial de la búsqueda, tanto para escapar de mínimos locales como para implementar una estrategia exploratoria. Un algoritmo simple de TS está basado en el mejor de una búsqueda local y en el uso de una memoria a corto plazo para escapar de mínimos locales y para evitar ciclos. La memoria a corto plazo está implementada como una *lista tabu* que mantiene pistas de las soluciones recientemente visitadas y las excluye del vecindario de la solución actual. En cada iteración, se elige la mejor solución entre las permitidas como la nueva solución actual. Por otra parte, esta solución se agrega a la lista tabu.

Métodos exploradores de búsqueda local

Greedy Randomized Adaptative Search Procedure (GRASP)

Es una metaheurística simple que combina heurísticas constructivas con búsqueda local [24] [78]. GRASP es un procedimiento iterativo, compuesto por dos fases: construcción de solución y mejora de solución. La mejor solución encontrada es devuelta en el momento de la finalización del proceso de búsqueda. El mecanismo de construcción de la solución es una heurística constructiva aleatoria, generando una solución paso a paso mediante el agregado un nuevo componente de solución tomado de un conjunto finito de componentes de soluciones a la actual solución parcial. La segunda fase del algoritmo es un método de búsqueda local, el cual debe ser un algoritmo básico como, por ejemplo, mejoramiento iterativo o una técnica aun mas avanzada como SA o TS.

Variable Neighborhood Search (VNS)

Es una metaheurística la cual aplica explícitamente estrategias para intercambiar entre diferentes estructuras de vecindarios de un conjunto finito predefinido. El algoritmo es muy general y existen muchos grados de libertad para el diseco de variantes e instanciaciones particulares. En la inicialización del algoritmo, un conjunto de estructuras de vecindario deben ser definidas. Estas estructuras de vecindarios pueden ser elegidas arbitrariamente. Luego, se genera una solución inicial, se inicializa el índice de vecindario, y el algoritmo itera hasta lograr una condición de terminación [40].

Guided Local Search (GLS)

Aplica una estrategia para escapar de mínimos locales muy diferente a la aplicada por TS o VNS. Esta estrategia consiste en cambiar dinámicamente la función objetivo, la cual resulta en un cambio en el alcance de la búsqueda. El objetivo es hacer que el actual mínimo local sea progresivamente menos deseable en el tiempo. El cambio dinámico de la función objetivo está basado en un conjunto de características de las soluciones. Una característica de solución puede ser alguna clase de propiedad o particularidad que pueda ser usada para discriminar entre las soluciones [102].

Iterated Local Search (ILS)

Es una metaheurística basada en un concepto simple pero potente. En cada iteración, la solución actual (la cual es un mínimo local) es perturbada y se aplica un método de búsqueda local a la solución perturbada. Luego, el mínimo local

obtenido por la aplicación del método de búsqueda local puede ser aceptado como la nueva solución o no. Intuitivamente, ILS desempeña una trayectoria a lo largo de mínimos locales sin introducir una estructura de vecindario explícitamente [94] [58] [60].

1.2. Metaheurísticas basados en población

Los métodos basados en población tratan en cada iteración del algoritmo con un conjunto de soluciones (una población) en lugar de con una solución única. De esta manera, los algoritmos basados en población proveen una forma natural e intrínseca de explotación del espacio de búsqueda. Aún así, el desempeño final depende en gran medida en la forma en que la población es manipulada [5]. Los métodos basados en población más estudiados en optimización combinatoria son la Computación Evolutiva (Evolutionary Computation, EC), Búsqueda Dispersa (Scatter Search, SS), Optimización basado en Cúmulos de Partículas (Particle Swarm Optimization, PSO) y la Optimización basado en Colonia de Hormigas (Ant Colony Optimization, ACO). A continuación se presenta una breve descripción de cada uno de ellos.

Evolutionary Computation (EC)

Son algoritmos inspirados en la capacidad de evolucionar de los seres vivos bien adaptados a sus entornos. Los algoritmos EC son modelos computacionales de procesos evolutivos. En cada iteración un numero de operadores se aplican a los individuos de la población actual para generar a los individuos de la población de la siguiente generación (iteración). Usualmente, los algoritmos EC usan operadores llamados *recombinación* o *crossover* (cruce) para producir nuevos individuos a partir del material genético de los existentes. También suelen usar operadores de *mutación* o *modificación* los cuales causan auto-adaptación de los individuos. La fuerza motriz en los algoritmos evolutivos es la *selección* de los individuos basados en su valor de aptitud o *fitness* (el cual depende de la función objetivo, el resultado de un experimento simulado, o en cualquier otra clase de medidas de calidad). Los individuos con fitness más alto tienen mayores probabilidades de ser elegidos como miembros de la población en la siguiente iteración (o como padres para la generación de un nuevo individuo). Esto corresponde al principio de *la supervivencia del más fuerte* en la evolución natural.
Se ha producido una variedad de algoritmos EC ligeramente diferentes a lo largo de los años. Básicamente se dividen en tres categorías las cuales han sido desarrolladas independientemente unas de otras. Estas son la Programación Evolutiva (Evolutionary Programming, EP) [25] [26], Estrategias Evolutivas (Evolutionary

Strategies, ES) [82], y los Algoritmos Genéticos (Genetic Algorithms, GA) [42]. EP surgió del deseo de generar máquinas inteligentes, y si bien originalmente fue propuesta para operar con representaciones discretas de máquinas de estado finito, la mayoría de las variantes actuales son usadas para problemas de optimización continua. A este último también se aplican la mayoría de las variantes actuales de ES, mientras que los GA son aplicados mayormente para solucionar problemas de optimización combinatoria.
Una de las mayores dificultades de los algoritmos EC (especialmente cuando se aplican a búsqueda local) es la prematura convergencia hacia una solución subóptima. El mecanismo más simple para diversificar el proceso de búsqueda es usar un operador de mutación aleatorio. También existen estrategias más complejas en las cuales la aptitud reproductiva de un individuo en una población se reduce proporcionalmente al número de otros individuos que comparten la misma región del espacio de búsqueda [34] [59].
Una característica importante de los algoritmos EC es la forma en que trata a los individuos inviables los cuales pueden ser producidos por operadores genéticos. Hay básicamente tres formas diferentes de manejar esta situación [5]. La acción más simple es *eliminarlos*. No obstante, para algunos problemas es difícil encontrar a los individuos inviables. Por lo tanto, la estrategia de *penalización* en función de la calidad del individuo suele ser la más apropiada. La tercer posibilidad consiste en tratar de *reparar* a la solución inviable.

Scatter Search (SS)

Este método difiere principalmente de la Computación Evolutiva en proveer principios unificados para la recombinación de soluciones basadas en la construcción de caminos generalizados en vecindarios. Estos principios están basados en estrategias originalmente propuestas para la combinación de reglas de decisión y limitaciones en el contexto de la programación entera. Es una estrategia que opera en un conjunto de *soluciones de referencia* que son soluciones factibles para el problema en consideración. El algoritmo comienza generando un conjunto de soluciones, luego se utiliza un método diversificador que iterativamente elige una de ella y genera una nueva con el objetivo de crear una solución lo más diferente posible de las existentes. Las nuevas soluciones se agregan al conjunto de soluciones solo si no están en él. Luego se aplican operaciones de recombinación y mejora. Esto se repite hasta que el criterio de detención se satisfaga [32] [33].

Particle Swarm Optimization (PSO)

Es una metaheurística inspirada en el comportamiento social del vuelo de las bandadas de aves o el desplazamiento de los cardúmenes de peces. La idea

principal del algoritmo es simular la influencia social que posee cada agente por parte del entorno durante la selección de la próxima dirección a seguir para alcanzar una nueva posición dentro del espacio de soluciones. Esta selección no sólo es influenciada por el conocimiento de su entorno, sino también por las experiencias anteriores del agente. Este algoritmo se basa en la idea expuesta en [73] de que los individuos que conviven en una sociedad tienen una opinión que es parte de un conjunto de creencias compartido (el espacio de búsqueda) por todos los posibles individuos. Cada individuo puede modificar su opinión dependiendo de:

- Su conocimiento sobre el entorno (su valor de fitness).
- Su conocimiento histórico o experiencias anteriores (su memoria o conocimiento cognitivo).
- El conocimiento histórico o experiencias anteriores de los individuos situados en su vecindario (su conocimiento social).

En base a esto, cada individuo adapta su conjunto de creencias según las creencias de aquellos con más éxito de su entorno, originando así una cultura en dónde sus individuos tienen un conjunto de creencias estrechamente relacionado.
En base a esto, cada individuo adapta su conjunto de creencias según las creencias de aquellos con más éxito de su entorno, originando así una cultura en dónde sus individuos tienen un conjunto de creencias estrechamente relacionado.

Ant Colony Optimization (ACO)

Es una metaheurística inspirada por el comportamiento de las hormigas reales [21] [22] [23]. Este comportamiento permite a las hormigas encontrar los caminos más cortos entre las fuentes de alimentación y sus nidos. Inicialmente, las hormigas exploran el área alrededor de sus nidos de forma aleatoria. Tan pronto como una hormiga encuentra una fuente de alimento, acarrea algo del alimento encontrado a su nido y mientras camina, la hormiga deposita un rastro de feromona química en el terreno. La calidad de la feromona depositada, la cual puede depender de la cantidad y la calidad de la comida, guía a otras hormigas a la fuente de alimentos. La comunicación indirecta entre las hormigas a través del rastro de feromona les permite encontrar los caminos más cortos entre su nido y las fuentes de alimentos. Esta funcionalidad de colonias de hormigas reales es explotado en colonias de hormigas artificiales para resolver problemas combinatorios de optimización.
En los algoritmos ACO los rastros de feromona química son simulados por un modelo probabilístico parametrizado llamado *modelo de feromona*. El modelo de feromona consiste en un conjunto de parámetros del modelo cuyos valores son llamados *valores de feromonas*. El ingrediente básico de los algoritmos ACO es una

heurística constructiva que es usada para la construcción probabilística de soluciones usando los valores de la feromona. En general, el enfoque de ACO intenta resolver un problema combinatorio de optimización mediante la iteración de dos pasos:

- Las soluciones se construyen usando un modelo de feromona, que es una distribución probabilística parametrizada sobre el espacio de soluciones.
- Las soluciones construidas, y las posibles soluciones que se construirán en las primeras iteraciones, se usan para modificar los valores de feromona de manera que el futuro sesgo de muestreo lleve a soluciones de alta calidad.

En general, diferentes versiones de los algoritmos ACO difieren en la manera en que actualizan los valores de feromona. Esto también se tiene en las actuales variantes ACO de mejor desempeño en la práctica, las cuales son los Sistemas de Colonias de Hormigas (Ants Colony System, ACS), Sistemas de Hormigas *MAX-MIN* (*MAX-MIN* Ant System, *MM*AS), y algoritmos ACO implementados en frameworks de hipercubos. Exitosas aplicaciones de ACO se incluyeron en aplicaciones de ruteo en redes de comunicación, a problemas de ordenamiento secuencial, a scheduling de limitación de recursos de proyectos, y al problema de scheduling del negocio abierto.

Capítulo 2

Redes Neuronales Artificiales

Bajo el nombre de genérico de ”Redes Neuronales” se engloban diferentes estructuras de procesamiento paralelo distribuido, algunas de ellas biológicamente inspiradas, en las que un número de elementos simples de cómputo no lineal se interconectan de forma más o menos densa. Estas estructuras son de utilidad en reconocimiento de patrones, modelado y filtrado no lineal de señales, entre muchas otras aplicaciones. En general, las redes neuronales presentan algún tipo de plasticidad en su estructura que permite sintonizar su operación a fin de optimizar una figura de mérito, como por ejemplo alguna medida de distancia al comportamiento deseado. De esta forma la red puede ”aprender” a realizar una tarea u operación. Esta característica es de gran utilidad allí donde el conocimiento del problema a resolver es impreciso o variante en el tiempo [86].

2.1. Características generales

Las redes neuronales son fundamentalmente estructuras simples las cuales se puede definir de la siguiente manera:

Una red neuronal es una estructura distribuida y paralela de procesamiento de información que consiste en ***elementos de procesamiento*** *interconectados por canales de señales unidireccionales llamados* ***conexiones****. Los elementos de procesamiento pueden poseer una memoria local y pueden llevar a cabo operaciones de procesamiento de información localmente. Cada uno de los elementos de procesamiento tiene una única conexión de salida que se ramifica en tantas conexiones como se desee, donde cada una lleva la misma señal, la señal de salida de los elementos de procesamiento. La información de procesamiento que va dentro de cada elemento de procesamiento puede ser definida arbitrariamente con la res-*

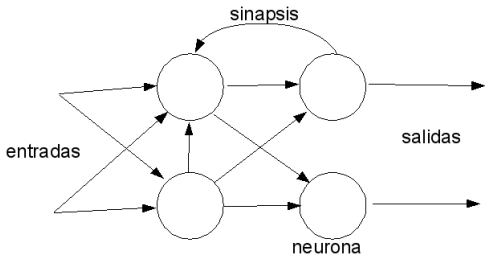

Figura 2.1: Red neuronal genérica.

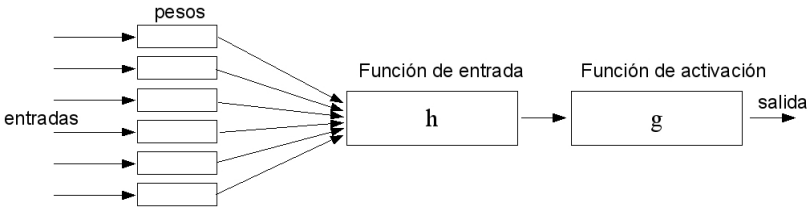

Figura 2.2: Modelo de una red neuronal artificial.

tricción de que esta debe ser completamente local; esto es, esta debe depender solo de los valores actuales de las señales de entrada que llegan al elemento de procesamiento mediante las conexiones indicadas y los valores almacenados en la memoria local de los elementos de procesamiento [41].

El esquema general de una red neuronal típica puede apreciarse en la figura 2.1. De una manera más informal, puede definirse a una red neuronal como un operador no lineal formado por un conjunto de elementos simples de procesamiento o neuronas, que se conectan entre si y con el medio externo a través de conexiones o sinapsis, que poseen un *peso* asociado.
La figura 2.2 ilustra el modelo habitualmente utilizando para una red neuronal artificial. Esencialmente, la operación de la neurona involucra el cálculo de una *función de entrada h*, a partir de las señales que ingresan a la misma y la posterior aplicación de una *función de activación g*, en general no lineal. En algunos casos existe una función umbral (*offset*), que puede ser siempre asociada a una entrada adicional de valor 1 y peso igual al umbral. La salida de cada neurona dependerá, entonces, de sus señales de entrada, del peso asociado a cada entrada y de las características de las funciones de entrada y activación. En la mayoría de los casos, los elementos plásticos de la red son los pesos de las conexiones. El mecanismo utilizado para adaptar esos pesos se conoce como *algoritmo de entrenamiento o aprendizaje*. Las redes neuronales se pueden clasificar de acuerdo a los siguientes

aspectos:

- **Número y disposición de neuronas**. En una red de tipo *feedforward*, las neuronas están organizadas en capas y no hay conexiones recurrentes o retroalimentación de la salida sobre la entrada o entre las internas. Existen entonces, capas directamente conectadas a la salida y capas *ocultas* o intermedias. En una red *recurrente*, por el contrario, existe una retroalimentación entre neuronas, lo que determina la aparición de una dinámica más compleja.

- **No-linealidad presente en cada neurona**. La salida de cada neurona es, en general, una función no lineal de la función de entrada. Si bien pueden utilizarse salidas lineales, su capacidad de modelado es moderada (básicamente la red es un combinador o clasificador lineal). La no-linealidad puede ser *suave* (esto es, con derivada continua con respecto a su entrada) o *abrupta*. Las características de la no-linealidad tienen directa relación con el algoritmo de aprendizaje a ser utilizado en la fase de entrenamiento.

- **Red de interconexión**. Los valores de los pesos de la red de interconexión imponen la influencia que la salida de una neurona tiene sobre las otras. Esta influencia puede ser *excitatoria*, cuando tiende a aumentar el valor de la activación, o *inhibitoria*, cuando tiende a disminuirlo. En otras redes, como las de base radial, los pesos especifican las coordenadas donde la no-linealidad de cada neurona está centrada.

- **Algoritmo de entrenamiento**. En los algoritmos *supervisados*, existe un *tutor* o *salida deseada* que especifica qué valor debería tener la salida de la red para cada posible entrada dentro de un conjunto de datos de entrenamiento. El error de la salida actual de la red y la deseada es utilizada por el algoritmo para modificar adecuadamente el valor de los pesos de la red. En los algoritmos *no supervisados*, por el contrario, no existe un tutor y el objetivo del algoritmo de entrenamiento es detectar alguna regularidad, estructura o característica particular de los datos, de forma que la red se convierte en un *detector de características* (*feature detector*). Los algoritmos de entrenamiento son, generalmente, procesos de optimización, a menudo heurísticos, y como tales sufren de problemas de convergencia y existencia de mínimos locales en la superficie de búsqueda.

- **Comportamiento estático y dinámico**. Una vez entrenadas, las redes que no poseen memoria se comportan como combinadores no lineales. Por el contrario, la existencia de elementos capaces de almacenar valores del estado pasado de la red la transforman en un sistema dinámico no lineal, lo que

amplía significativamente sus capacidades de modelado y agrega complejidad a su comportamiento y tratamiento matemático.

Atendiendo los aspectos arriba citados, es posible caracterizar totalmente una determinada red neuronal. En las secciones que siguen se analizarán algunos de los paradigmas más utilizados [86].

2.2. Modelos de RNA

2.2.1. Perceptron simple

Este modelo se concibió como un sistema capaz de realizar tareas de clasificación de forma automática. La idea era disponer de un sistema que, a partir de un conjunto de ejemplos de clases diferentes, fuera capaz de determinar las ecuaciones de las superficies que hacían de frontera de dichas clases. La información sobre la que se basaba el sistema estaba constituida por los ejemplos existentes de las diferentes clases. A esto se lo llamó patrones o ejemplos de entrenamiento. Son dichos patrones de entrenamiento los que aportan la información necesaria para que el sistema construya las superficies discriminantes, y además actúe como un discriminador para ejemplos nuevos desconocidos. El sistema, al final del proceso, es capaz de determinar, para cualquier ejemplo nuevo, a qué clase pertenece [46].

Descripción del modelo

La arquitectura de la red es muy simple. Se trata de una estructura monocapa, en la que hay un conjunto de entradas, tantas como sea necesario, según los términos del problema; y una célula de salida. Cada una de las entradas tiene conexiones con la célula de salida, y son estas conexiones las que determinan las superficies de discriminación del sistema.
En el ejemplo de la figura 2.3 las entradas son *x1* y *x2*, y la salida, *y*. Los pesos son *w1* y *w2*. Además existe un parámetro adicional llamado umbral denotado por θ. El umbral se utiliza como factor de combinación para producir la salida.
En este esquema la salida de la red se obtiene de las siguiente forma. Primero se calcula la activación de la célula de salida mediante la suma ponderada por los pesos de todas las entradas:

$$y' = \sum_{i=1}^{n} w_i x_i \tag{2.1}$$

La salida definitiva se produce al aplicarle una función de salida al nivel de activación de la célula. En un Perceptron la función de salida es una función escalón

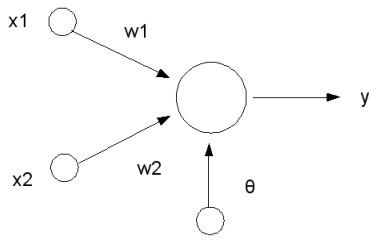

Figura 2.3: Arquitectura de un Perceptron con dos entradas y una salida.

que depende del umbral:

$$y = F(y', \theta) \qquad F(s, \theta) = \begin{cases} 1 & \text{Si } s > \theta \\ -1 & \text{en caso contrario} \end{cases} \tag{2.2}$$

Utilizando las ecuaciones 2.1 y 2.2 la salida se puede describir en una sola ecuación:

$$y = F\left(\sum_{i=1}^{n} w_i x_i + \theta\right) \tag{2.3}$$

donde F ya no depende de ningún parámetro:

$$F(s) = \begin{cases} 1 & \text{Si } s > 0 \\ -1 & \text{en caso contrario} \end{cases} \tag{2.4}$$

Esta ecuación equivale a introducir artificialmente en la salida un nuevo peso θ que está conectado a una entrada ficticia con un valor constante de -1.
La función de salida F es binaria y de gran utilidad en este modelo ya que al tratarse de un discriminante de clases, una salida binaria puede ser fácilmente traducible en una clasificación en dos categorías de la siguiente forma:

- Si la red produce salida 1, la entrada pertenece a la categoría A.
- Si la red produce salida -1, la entrada pertenece a la categoría B.

En el caso de dos dimensiones, la función discriminante será:

$$w_1 x_1 + w_2 x_2 + \theta = 0 \tag{2.5}$$

que es la ecuación de una recta de pendiente $-\frac{w_1}{w_2}$ y que en el origen de ordenadas pasa por $-\frac{\theta}{w_2}$. En este caso los patrones de entrenamiento pueden presentarse como puntos en un espacio bidimensional. Si además dichos puntos pertenecen a una de dos categorías, A o B, la separación en dichas categorías podrá hacerse

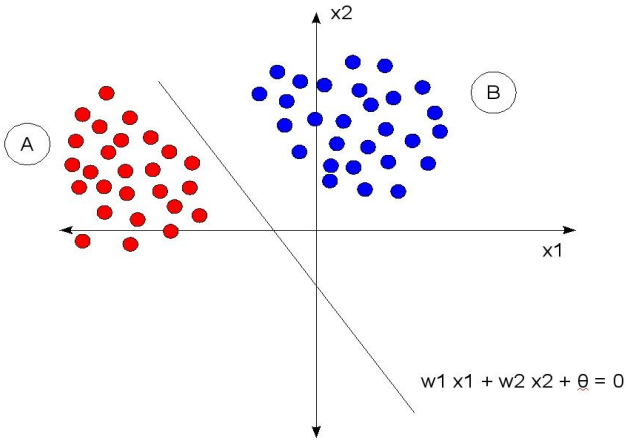

Figura 2.4: Separación en dos clases mediante un Perceptron.

mediante la recta anterior.
Es decir que la red define una recta, que en el caso de ser solución al problema discriminará entre las clases existentes en los datos de entrenamiento. Gráficamente podría representarse como se muestra en la figura 2.4. Si se llama χ a cada uno de los patrones de entrenamiento y $d(\chi)$ a su clase asociada, tomando valores en (1,-1), el proceso de aprendizaje se puede describir de la siguiente forma:

1. Empezar con valores aleatorios para los pesos y el umbral.

2. Seleccionar un vector de entrada χ del conjunto de ejemplos de entrenamiento.

3. Si $y \neq d(\chi)$, la red da una respuesta incorrecta. Modificar w_i de acuerdo con:
$$\Delta w_i = d(\chi)\, x_i \qquad i = 1, .., n \tag{2.6}$$

4. Si no se ha cumplido el criterio de finalización, volver a 2.

En el paso tres se aprecia que si la salida de la red para un patrón es $y(\chi) = 1$, pero su clase es $d(\chi) = -1$, entonces el incremento es negativo, $\delta w_i = d(\chi)\, x_i = -x_i$, mientras que si ocurre lo contrario, es positivo, como se describió anteriormente. Puesto que el umbral es equivalente a un peso adicional, al que se denota por w_0, cuya entrada es siempre 1 ($x_0 = 1$), la ecuación anterior se puede extender para el umbral de la siguiente manera:

$$\Delta w_i = d(\chi)\, x_i \qquad i = 0, .., n \tag{2.7}$$

2.2.2. Adaline

El Perceptron es un sistema simple de aprendizaje basado en ejemplos capaz de realizar tareas de clasificación. Sin embargo, existe un gran número de problemas abordables desde la perspectiva del aprendizaje basado en ejemplos que no se reducen a tareas de clasificación. La característica de clasificador del Perceptron viene dada por la naturaleza de sus salidas. En la capa de salida las neuronas codifican una función escalón (2.4) que transforma la suma ponderada de las entradas que es un valor real, en una salida binaria. De esta forma, sólo pueden codificar un conjunto discreto de estados. Mientras que si fueran números reales, podrían codificar cualquier tipo de salida y se convertirían en sistemas de resolución de problemas generales. En este caso, podrían resolver problemas a partir de ejemplos en los que es necesario aproximar una función cualquiera ($F(x)$), definida por un conjunto de datos de entrada.
Los datos de entrenamiento en este caso son un conjunto de valores enteros, compuestos por un vector de entrada y su salida asociada:

$$P = \{(\vec{x}_1, y_1), ..., (\vec{x}_m, y_m)\} \tag{2.8}$$

En este caso la función sería:

$$F(\vec{x}_i) = y_i, \forall p \in P \tag{2.9}$$

En otros términos, habrá que buscar la función $F(x)$ tal que aplicada a cada una de las entradas x_i del conjunto de aprendizaje P produzca la salida y_i correspondiente a dicho conjunto de aprendizaje.
Sin embargo, la naturaleza del Perceptron no permite producir salidas reales. Esta regla es fundamentalmente una regla de aprendizaje por refuerzo en la que se potencian las salidas correctas y no se tienen en cuenta las incorrectas. No existe ninguna graduación en la regla que indique en que medida resulta errónea la salida producida, y refuerce proporcionalmente a dicha medida de error.

Descripción del modelo

En 1960, Wildrow [106] propuso un sistema de aprendizaje que sí tuviera en cuenta el error producido, y diseñó lo que denominó ADAptative LInear NEuron, o en su acrónimo, ADALINE. Esta es una estructura prácticamente idéntica a la del Perceptron, pero es un mecanismo físico, capaz de realizar aprendizaje. Es un elemento combinador adaptativo, que recibe un conjunto de entradas y las combina para producir una salida. Esta salida puede transformarse en binaria mediante un conmutador bipolar que produce un 1 si la salida es positiva u un -1 si es negativa:

$$\widehat{y} = \sum_{i=1}^{n} w_i \cdot x_i + \theta \tag{2.10}$$

El aprendizaje en este caso incluye la diferencia entre el valor real producido en la capa de salida para un patrón de entrada x^p y el que debería haber producido dicho patrón, es decir, su salida esperada d^p, que está en el conjunto de aprendizaje ($|d^p - y^p|$). A esta regla se la conoce con el nombre de *regla Delta*.
La diferencia con respecto al aprendizaje de Perceptron es la manera de utilizar la salida, una diferencia fundamental entre ambos sistemas. El Perceptron utiliza la salida de la función umbral para el aprendizaje; sin embargo, la regla Delta utiliza directamente la salida de la red, sin pasarla por ninguna función umbral. El objetivo es obtener un valor determinado $y = d^p$ cuando el conjunto de valores X_i^p, $i = 1,...,n$ se introduce en la entrada. Por lo tanto, el problema será obtener los valores de w_i, $i = 1,...,n$ que permitan realizar lo anterior para un número arbitrario de patrones de entrada.
No es posible conseguir una salida exacta, pero sí minimizar la desviación cometida por la red, esto es, minimizar el error cometido por la red para la totalidad del conjunto de patrones de ejemplo. En otros términos, hay que evaluar globalmente el error cometido por la red para todos los patrones, o sea elegir una medida de error global. Habitualmente, la medida de error global utilizada es el error cuadrático medio cuya formula se observa en la ecuación 2.11, aunque otros errores pueden ser usados por este modelo.

$$E = \sum_{p=1}^{m} E^p = \frac{1}{2} \sum_{p=1}^{m} (d^p - y^p)^2 \tag{2.11}$$

Al ser ésta una medida de error global, la regla intentará minimizar este valor para todos los elementos del conjunto de patrones de aprendizaje. La manera de minimizar este error es recurrir a un proceso iterativo en el que se van presentando los patrones uno a uno, y modificando los pesos de las conexiones, mediante la *regla del descenso del gradiente*.
La idea es realizar un cambio en cada peso proporcional a la derivada del error, medida en el patrón actual, respecto del peso:

$$\Delta_p w_j = -\gamma \frac{\partial E^p}{\partial w_j} \tag{2.12}$$

Para el cálculo de la derivada anterior se utiliza la regla de la cadena:

$$\frac{\partial A}{\partial x} = \frac{\partial A}{\partial y} \frac{\partial y}{\partial x} \tag{2.13}$$

Aplicando la regla de la cadena a la expresión anterior queda como sigue:

$$\frac{\partial E^p}{\partial w_i} = \frac{\partial E^p}{\partial y^p} \frac{\partial y^p}{\partial w_i} \tag{2.14}$$

Al ser unidades lineales, sin función de activación en la capa de salida, se cumple:

$$\frac{\partial y^p}{\partial w_i} = x_j \qquad \frac{\partial E^p}{\partial y^p} = -\left(d^p - y^p\right) \tag{2.15}$$

que sustituyendo queda como sigue:

$$\Delta_p w_j = -\gamma \left(d^p - y^p\right) x_j \tag{2.16}$$

Si se compara la expresión de la regla Delta (ecuación 2.16), con la regla de aprendizaje del Perceptron (ecuación 2.7), se aprecia que la diferencia es precisamente la introducción de la diferencia entre la salida deseada y la obtenida en la regla de aprendizaje. Si la salida del Adaline fuese binaria, el conjunto de patrones estaría construido por $P = \left\{\left(\overrightarrow{x_1}, 0\, o\, 1\right), ..., \left(\overrightarrow{x_m}, 0\, o\, 1\right)\right\}$, es decir, $d^p \in 0, 1, \forall p$. Si se incluye a la salida del Adaline el acoplador bipolar comentado con anterioridad para "binarizar"la salida, la regla Delta (ecuación 2.16) quedaría así:

$$\Delta_p w_j = \begin{cases} \gamma x_i & \text{Si } d^p > y^p \\ -\gamma x_i & \text{Si } d^p < y^p \\ 0 & \text{en caso contrario} \end{cases} \tag{2.17}$$

que para un $\gamma = 1$ se convierte en la regla del Perceptron. Así pues, la regla Delta es una extensión de la regla del Perceptron a valores de salida reales.
El procedimiento de aprendizaje definido por la regla Delta será:

1. Inicializar los pesos de forma aleatoria.
2. Introducir un patrón de entrada.
3. Calcular la salida de la red, compararla con la deseada y obtener la diferencia: $(d^p - y^p)$.
4. Para todos los pasos, multiplicar dicha diferencia por la entrada correspondiente, y ponderarla por una tasa de aprendizaje γ.
5. Modificar el peso restando del valor antiguo la cantidad obtenida en 4.
6. Si no se ha cumplido el criterio de convergencia, regresar a 2; si se han acabado todos los patrones, empezar de nuevo a introducir patrones.

En resumen, las diferencias entre ambos modelos de Redes Neuronas Artificiales serán las siguientes:

- En el Perceptron la salida es binaria, mientras que en el Adaline es real.

- En el Perceptron la diferencia entre entrada y salida es 0 si ambas pertenecen a la misma categoría y ±1 si por el contrario pertenecen a categorías diferentes. En el Adaline se calcula la diferencia real entre entrada y salida.
- En el Adaline existe una medida de cuánto se ha equivocado la red; en el Perceptron sólo se determina si se ha equivocado o no.
- En el Adaline hay una razón de aprendizaje (γ) para regular cuánto va a afectar cada equivocación a la modificación de los pesos. Es siempre un valor entre 0 y 1 para ponderar el aprendizaje.

2.2.3. Perceptron multicapa y algoritmo de Retropropagación

El perceptron multicapa o red multicapa con conexiones hacia adelante (red FeedForward) es una generalización del perceptron simple presentado en la sección 2.2.1, y surgió como consecuencia de las limitaciones de dicha arquitectura en lo referente al problema de la separabilidad no lineal. En un Perceptron multicapa no puede usarse la regla de aprendizaje de un Perceptron simple, pues esta regla no puede aplicarse en este escenario. Para adaptar los pesos de la capa de entrada a la capa oculta se desarrolló una manera de retropropagar los errores medidos en la capa de salida de la red hacia las neuronas ocultas, dando lugar a la llamada regla delta generalizada, que no es más que una generalización de la regla delta, descripta en la sección 2.2.2, para funciones de activación no lineales y redes multicapa.
Diferentes autores [19] [45] han demostrado que el Perceptron multicapa es una aproximación universal, en el sentido de que cualquier función continua sobre un compacto de R^n puede aproximarse con un Perceptron multicapa como una nueva clase de funciones (como polinomios, funciones trigonométricas, splines, etc.) para aproximar o interpolar relaciones no lineales entre datos de entrada y salida. La habilidad del Perceptron multicapa para aprender a partir de un conjunto de ejemplos hace que sea un modelo adecuado para abordar problemas reales, sin que esto indique que sean los mejores aproximadores universales.

Arquitectura

La arquitectura del Perceptron multicapa se caracteriza por tener sus neuronas agrupadas en capas de diferentes niveles. Cada una de las capas está formada por un conjunto de neuronas. Se distinguen tres tipos de capas diferentes: la capa de entrada, las capas ocultas y la capa de salida, como se observa en la figura 2.5. Las neuronas de la capa de entrada, no actúan como neuronas propiamente dichas, sino que se encargan únicamente de recibir las señales o patrones que proceden del exterior y propagar dichas señales a todas las neuronas de la siguiente capa. La

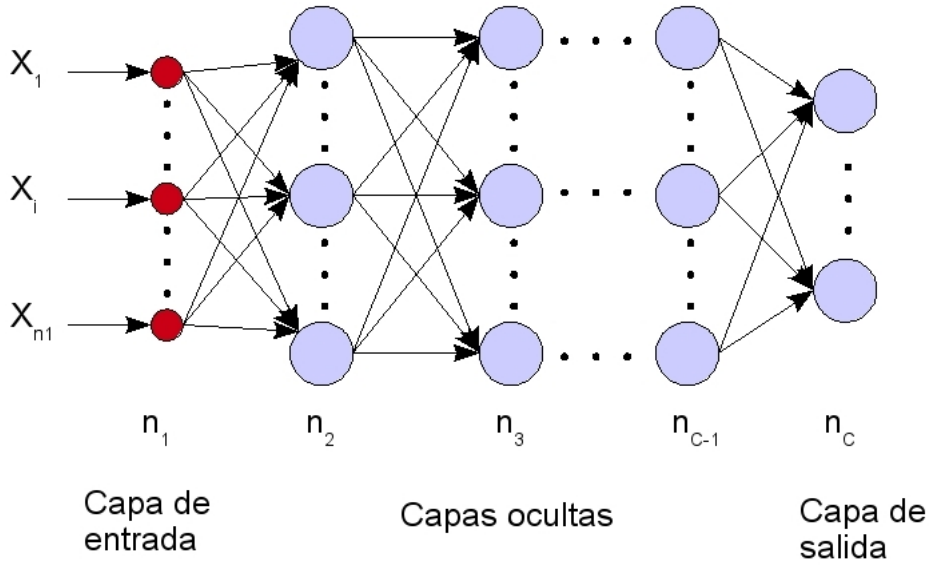

Figura 2.5: Arquitectura del Perceptron multicapa.

última capa actúa como salida de la red, proporcionando al exterior la respuesta de la red para cada uno de los patrones de entrada. Las neuronas de las capas ocultas realizan un procesamiento no lineal de los patrones recibidos.
Como se observa en la figura 2.5, las conexiones del Perceptron multicapa siempre están dirigidas hacia adelante, es decir, las neuronas de una capa se conectan con las neuronas de la siguiente capa, de ahí que reciban también el nombre de redes alimentadas hacia adelante o redes ”feedforward”. Las conexiones entre las neuronas llevan asociado un número real, llamado peso de la conexión. Todas las neuronas de la red llevan también asociado un umbral, que en el caso del perceptron multicapa suele tratarse de como una conexión más a la neurona, cuya entrada es constante e igual a 1.
Aunque en la mayor parte de los casos la arquitectura del Perceptron multicapa está asociada al esquema de la figura 2.5, es posible también englobar dentro de este tipo de redes a arquitecturas con las siguientes características:

- Redes con conexiones de todas o ciertas neuronas de una determinada capa a neuronas de capas posteriores, aunque no inmediatamente posteriores.
- Redes en las que ciertas neuronas de ciertas capas no están conectadas a neuronas de la siguiente capa, es decir, el peso de la conexión es constante e igual a cero.

Cuando se busca resolver un problema utilizando este tipo de estructuras uno de los primeros pasos a realizar es el diseño de la arquitectura de la red. Este diseño

implica la determinación de la función de activación a emplear, el número de neuronas y el número de capas de la red.
La elección de la función de activación se suele hacer basándose en el recorrido deseado, y el hecho de elegir una u otra, generalmente, no influye en la capacidad de la red para resolver el problema.
En general, el número de neuronas de la capa de entrada, como el número de neuronas de la capa de salida, vienen dadas por las variables que definen el problema. Sin embargo, existen problemas para los cuales el número de variables de entrada relevantes no se conoce con exactitud. En estos casos, se dispone de un gran número de variables, algunas de las cuales podrían no aportar información relevante a la red, y su utilización podría dificultar el aprendizaje, pues implicaría arquitecturas de gran tamaño y con alta conectividad.
Por otro lado, el número de capas ocultas y el número de neuronas en estas capas deben ser elegidos por el diseñador. No existe ningún método o regla que determine el número óptimo de neuronas ocultas para resolver un problema dado. En la mayor parte de las aplicaciones prácticas, estos parámetros se determinan por prueba y error. Partiendo de una arquitectura ya entrenada, se realizan cambios aumentando y disminuyendo el número de neuronas ocultas y el número de capas hasta conseguir una arquitectura adecuada para el problema a resolver, que si bien puede no ser la óptima, proporciona una solución.

Propagación de los patrones de entrada

El Perceptron multicapa define una relación entre las variables de entrada y las variables de salida de la red. Esta relación se obtiene propagando hacia adelante los valores de las variables de entrada. Para ello, cada neurona de la red procesa la información recibida por sus entradas y produce una respuesta o activación que se propaga, a través de las conexiones correspondientes, hacia las neuronas de la siguiente capa.

Sea un Perceptron multicapa con C capas ($C-2$ capas ocultas) y n_c neuronas en la capa c, para $c = 1, 2, ..., C$. Sea $W^c = \left(w_{ij}^c\right)$ la matriz de pesos asociada a las conexiones de la capa c a la capa $c+1$ para $c = 1, 2, ..., C-1$, donde w_{ij}^c representa el peso de la conexión de la neurona i de la capa c a la neurona j de la capa $c+1$; y sea $U^c = \left(u_i^c\right)$ el vector de umbrales de las neuronas de la capa c para $c = 2, ..., C$. Se denota a_i^c a la activación de la neurona i de la capa c; estas activaciones se calculan del siguiente modo:

- **Activación de las neuronas de la capa de entrada** $\left(a_i^1\right)$. Las neuronas de la capa de entrada se encargan de transmitir hacia la red las señales recibidas

del exterior. Por tanto:

$$a_i^1 = x_i \quad para \quad i = 1, 2, ..., n_1 \tag{2.18}$$

donde $X = (x_1, x_2, ..., x_{n_1})$ representa el vector o patrón de entrada a la red.

- **Activación de las neuronas de la capa oculta** c $\left(a_i^c\right)$. Las neuronas ocultas de la red procesan la información recibida aplicando la función de activación f a la suma de los productos de las activaciones que recibe por sus correspondientes pesos, es decir:

$$a_i^c = f\left(\sum_{j=1}^{n_{c-1}} w_{ij}^{c-1} a_j^{c-1} + u_i^c\right) \; para \; i = 1, 2, ..., n_c \; y \; c = 2, 3, ..., C-1 \tag{2.19}$$

donde a_j^{c-1} son las activaciones de las neuronas de la capa $c-1$.

- **Activación de las neuronas de la capa de salida** $\left(a_i^C\right)$**.** Al igual que en el caso anterior, la activación de estas neuronas viene dada por la función de activación f aplicada a la suma de los productos de las entradas que recibe por sus correspondientes pesos:

$$a_i^C = f\left(\sum_{j=1}^{n_{C-1}} w_{ij}^{C-1} a_j^{C-1} + u_i^C\right) \; para \; i = 1, 2, ..., n_C \tag{2.20}$$

donde $Y = (y_1, y_2, ..., y_{n_C})$ es el vector de salida de la red.

La función f es la llamada *función de activación* y las más utilizadas son la función sigmoidal y la función tangente hiperbólica.

- Función sigmoidal:

$$f_1(x) = \frac{1}{1 + e^{-x}} \tag{2.21}$$

- Función tangente hiperbólica:

$$f_2(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \tag{2.22}$$

Ambas son funciones crecientes con dos niveles de saturación: el máximo, que proporciona salida 1, y el mínimo, salida 0 para la función de activación sigmoidal y salida -1 para la tangente hiperbólica, como se observa en la figura 2.6. Generalmente, la función de activación en el Perceptron multicapa es común a

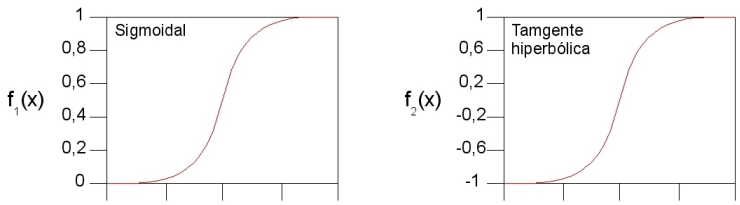

Figura 2.6: Funciones de activación del Perceptron multicapa.

todas las neuronas de la red y es elegida por el diseñador, elección que se realiza únicamente basándose en los valores de activación que se desee que alcancen las neuronas. Ambas funciones están relacionadas mediante la expresión $f_2(x) = 2f_1(x) - 1$, por lo que la utilización de una u otra manera se elige únicamente en función del recorrido de interés.
En ocasiones, y dependiendo de la naturaleza del problema, las neuronas de salida se distinguen del resto de las neuronas de la red, utilizando otro tipo de función de activación. En este caso, las más usadas son la función identidad y la función escalón.

Algoritmo de Retro Propagación

El algoritmo de entrenamiento es el mecanismo por el cual se van adaptando y modificando todos los parámetros de la red. En el caso del Perceptron multicapa se trata de un algoritmo supervisado, es decir, la modificación de los parámetros se realiza para que la salida sea lo más próxima posible a lo esperado. Por lo tanto, para cada patrón de entrada es necesario disponer de un patrón de salida deseada. Para esto se formula el aprendizaje de la red como un problema de minimización del siguiente modo:

$$Min_W E \tag{2.23}$$

siendo W el conjunto de parámetros de la red (pesos y umbrales) y E una función error que evalúa la diferencia entre las salidas de la red y las salidas deseadas. La función error se define como:

$$E = \frac{1}{N}\sum_{n=1}^{N} e(n) \tag{2.24}$$

donde N es el número de patrones o muestras y $e(n)$ es el error cometido por la red para el patrón n, dado por:

$$e(n) = \frac{1}{2}\sum_{i=1}^{n_c} (s_i(n) - y_i(n))^2 \tag{2.25}$$

siendo $Y(n) = (y_1(n), ..., y_{n_c}(n))$ y $S(n) = (s_1(n), ..., s_{n_c}(n))$ los vectores de salida de la red y las salidas deseadas para el patrón n, respectivamente.
De este modo, si W^* es un mínimo de la función error E, en dicho punto el error es próximo a cero, lo cual implica que la salida de la red es próxima a la salida deseada, alcanzando así la meta de la regla de aprendizaje.
Por lo tanto, el aprendizaje del Perceptron multicapa es equivalente a encontrar un mínimo en la función error. En el contexto de redes neuronales, la dirección de búsqueda más comúnmente utilizada es la dirección negativa del gradiente de la función E (método de descenso del gradiente).
Aunque, estrictamente hablando, el aprendizaje de la red debe realizarse para minimizar el error total (ecuación 2.24), el procedimiento más utilizado está basado en métodos del gradiente estocástico, los cuales consisten en una sucesiva minimización de los errores para cada patrón, $e(n)$, en lugar de minimizar el error total E. Por lo tanto, aplicando el método de descenso estocástico, cada parámetro w de la red se modifica para cada patrón de entrada n de acuerdo con la siguiente ley de aprendizaje:

$$w(n) = w(n-1) - \alpha \frac{\partial e(n)}{\partial w} \tag{2.26}$$

donde $e(n)$ es el error para el patrón n dado por la ecuación 2.25 y α es la razón o tasa de aprendizaje, parámetro que influye en la magnitud del desplazamiento en la superficie del error.
Debido a que las neuronas de la red están agrupadas en capas de distintos niveles, es posible aplicar el método del gradiente de forma eficiente, resultando el conocido algoritmo de RetroPropagación [84] o *regla delta generalizada*. El término de retropropagación se utiliza debido a la forma de implementar el método del gradiente en el Perceptron multicapa, pues el error cometido en la salida de la red es propagado hacia atrás, transformándolo en un error para cada una de las neuronas ocultas de la red.

2.3. RNA Recurrentes

Las redes neuronales estudiadas hasta el momento, suelen tener una fuerte restricción, que consiste en no permitir conexiones entre neuronas creando ciclos o bucles. En cambio, las RNA recurrentes no están sometidas a esta restricción en la conectividad.
Las redes neuronales recurrentes se caracterizan porque crean bucles en las neuronas de la red mediante el uso de las llamadas conexiones recurrentes, pudiendo aparecer en la red conexiones de una neurona a ella misma, conexiones entre neuronas de la misma capa o conexiones de las neuronas de una capa a la capa

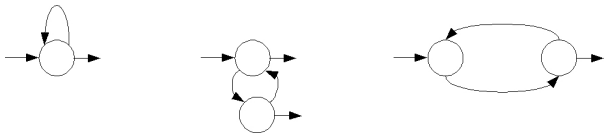

Figura 2.7: Ejemplos de neuronas con conexiones recurrentes.

anterior, como se muestran en la figura 2.7. Al introducir conexiones recurrentes creando bucles, la activación de una neurona ya no depende sólo de las activaciones de las neuronas de la capa anterior, sino que depende también del estado o activación de cualquier otra neurona conectada a ella, o incluso de su propia activación. Por lo tanto, es necesario incluir la variable tiempo en la activación o estado de una neurona.
La presencia de la variable tiempo en las activaciones de las neuronas recurrentes, hace que estas redes posean un comportamiento dinámico o temporal. Dicho comportamiento temporal puede entenderse de dos formas diferentes, lo cual implica dos maneras distintas de entender el modo de actuación y aprendizaje dentro del grupo de las redes recurrentes. Éstas son:

- **Evolución de las activaciones de la red hasta alcanzar un punto estable.** Para estas redes el modo de activación consiste en evolucionar la red, es decir las activaciones de sus neuronas, desde un estado inicial hasta conseguir que las activaciones de todas las neuronas de la red no se modifiquen, momento en el que se considera que la red ha alcanzado un punto estable. Generalmente, el estado inicial viene dado por el patrón de entrada y el estado estable representa el patrón de salida de la red.

- **Evolución de las activaciones de la red en modo continuo.** En este caso, para cada instante de tiempo se dispone de la salida de la red, la cual depende de la entrada en el instante inmediatamente anterior. En términos generales, el aprendizaje de este tipo de redes se puede llevar a cabo de dos modos diferentes:

  - Aprendizaje por épocas: dado un intervalo de tiempo o época, la red se evoluciona durante dicho intervalo, y cuando se alcanza el instante final se adaptan o modifican los pesos de la red. Una vez transcurrida la época, la red se reinicia y se entra en una nueva época.
  - Aprendizaje en tiempo real o continuo: la ley de aprendizaje para modificar los pesos de la red se aplica en cada instante de tiempo, siempre y cuando exista la salida deseada para la red en dicho instante.

Dentro del grupo de las neuronas recurrentes que poseen este modo de activación se pueden englobar las redes conocidas como redes parcialmente recurrentes y redes totalmente recurrentes. Las primeras se caracterizan porque sólo unas pocas conexiones recurrentes son introducidas en la red, mientras que las segundas no tienen restricciones en la consideración de conexiones recurrentes. Ambos tipos de redes utilizan algoritmos de aprendizaje supervisados para la modificación de sus parámetros.

El comportamiento dinámico de las redes recurrentes facilita el tratamiento de información temporal o patrones dinámicos, es decir, patrones que dependen del tiempo en el sentido de que el valor del patrón en un determinado instante depende de sus valores en instantes anteriores de tiempo. En principio, las redes recurrentes son las más indicadas para abordar este tipo de problemas. Sin embargo, éstas no son las únicas redes de neuronas, como el Perceptron multicapa; basta considerar como entrada a la red una secuencia finita temporal del patrón en el pasado.
Aunque la aplicación fundamental de las redes recurrentes es el procesamiento de patrones dinámicos, se pueden aplicar también a patrones estáticos, es decir, patrones en los que no interviene la variable tiempo y cuyo procesamiento no depende del orden de presentación a la red.
A pesar de la complejidad que poseen las redes de neuronas recurrentes y la dificultad, en ovaciones, de realizar el aprendizaje, éstas has sido utilizadas para abordar diferentes tipos de problemas. Por ejemplo, problemas de modelización neurológica [6], tareas lingüísticas [29], reconocimiento de palabras y fonemas [83], control de procesos dinámicos [77] [80], entre otros.

Capítulo 3

Algoritmos Genéticos y Estrategias Evolutivas

Durante varias décadas, los algoritmos evolutivos se han utilizado exitosamente para resolver problemas de optimización en campos diversos como la ingeniería, arte, diseño, biología, economía y física.
Los algoritmos evolutivos mantienen una población de soluciones candidatas al problema de optimización, conocidas como individuos. La población tiende a evolucionar hacia mejores soluciones usando un proceso iterativo.
Dentro de la familia de los algoritmos evolutivos, se incluyen los Algoritmos Genéticos, las Estrategias de Evolución, la Programación Evolutiva y la Programación Genética.
Este capítulo tiene como objetivo presentar los aspectos más destacados de los Algoritmos Genéticos y las Estrategias Evolutivas.

3.1. Algoritmos Genéticos

En los años 1960, gracias a John Holland, surgió una de las líneas más prometedoras de la Inteligencia Artificial: los Algoritmos Genéticos, llamados así porque se inspiran en la evolución biológica y su base genético-molecular [43] [44]. Los algoritmos genéticos son algoritmos de búsqueda basados en los mecanismos de selección genética naturales. Estos a partir de un grupo de soluciones combinan la supervivencia de la estructura más adecuada con un intercambio aleatorio de información estructurada para formar un algoritmo de búsqueda con alguna de las innovaciones propias de la búsqueda humana.
Los algoritmos genéticos operan con una o varias poblaciones de individuos, con cadenas codificadas para representar al conjunto de parámetros subyacente. Distintos operadores genéticos se aplican a los individuos de la o las poblaciones para

crear nuevos individuos. Estos operadores simples, no son mas complejos que la generación de un número aleatorio, copia de cadenas y extracciones parciales de cadenas y debido a esta simplicidad, el desempeño de la búsqueda resultante es muy bueno y diverso [42].
Existen cuatro diferencias que separan a los algoritmos genéticos de las técnicas de optimización convencionales:

1. Manipulación directa de la codificación.

2. Para la búsqueda, se utiliza una población de soluciones, no un solo punto.

3. La búsqueda se realiza mediante la prueba y error, es una búsqueda a ciegas.

4. Se usan operadores estocásticos para la búsqueda, regidos bajo reglas deterministas.

3.1.1. Características

Los algoritmos genéticos poseen como principal característica que trabajan con un conjunto de potenciales soluciones, generalmente denominado población. A cada una de las soluciones se las suele llamar individuo. Cada individuo de la población está compuesto por un conjunto de parámetros de longitud finita en los cuales se codifican las variables del problema de optimización. Por ejemplo, considérese el problema de la caja negra de interruptores ilustrada en la figura 3.1. Este problema consiste en un dispositivo de caja negra con cinco interruptores. Por cada configuración de los 5 interruptores hay una señal de salida f, matemáticamente $f = f(s)$, donde s es una configuración particular de los cinco interruptores. El objetivo del problema es configurar los interruptores de manera de obtener el máximo valor posible para f. Con otros métodos de optimización se debe trabajar directamente con la configuración de los parámetros (la configuración de los interruptores) y cambiar la posición de los interruptores usando reglas de transición de un método particular. Con los algoritmos genéticos, primero se deben codificar los 5 interruptores en una cadena de longitud fija. Una simple codificación puede ser una cadena de cinco unos y ceros donde cada interruptor está representado por un 1 si el interruptor está encendido y 0 si está apagado. Con esta codificación, la cadena 11110 codifica la configuración donde los cuatro primeros interruptores están encendidos y el quito interruptor está apagado [34].
En muchos métodos de optimización, se va moviendo gradualmente de un punto en el espacio de soluciones al siguiente usando alguna regla de transición para determinar el siguiente punto. Este método de punto-a-punto es peligroso porque se puede caer en máximos locales en espacios de búsqueda multimodales (varios

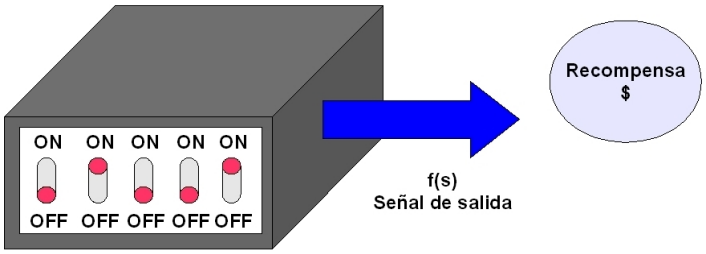

Figura 3.1: Problema de optimización de la caja negra con cinco interruptores ilustra la idea de la codificación y de la medida de la recompensa. Los algoritmos genéticos solo requieren de estas dos cosas, no necesitan saber el funcionamiento de la caja negra.

máximos y mínimos). En cambio, los algoritmos genéticos trabajan con una amplia información de los puntos simultáneamente (una población de cadenas), escalando varios máximos paralelamente; así, la probabilidad de encontrar un máximo local es reducida comparada con los métodos que avanzan punto-a-punto. Como ejemplo, considérese nuevamente el problema de optimización de la caja negra de la figura 3.1. Con otras técnicas, para resolver este problema se debería empezar con una configuración de los interruptores, aplicar algunas reglas de transición, y generar una nueva configuración de los interruptores. Un algoritmo genético comienza con una población de cadenas y para luego generar sucesivas poblaciones de cadenas. Por ejemplo, un comienzo aleatorio de la posición de los interruptores lanzando una moneda (cara = 1, seca = 0) y creando una población de $n = 4$ individuos (pequeña para los algoritmos genéticos estándar):

```
01101
11000
01000
10011
```

Luego de este comienzo, sucesivas poblaciones serán generadas usando el algoritmo genético con el objetivo de maximizar la señal de salida de la caja [34].

Función Objetivo

La mayoría de las aplicaciones actuales de algoritmos genéticos son problemas de optimización. La función objetivo representa la función que se quiere optimizar. El algoritmo genético usa la función objetivo para asignar a cada individuo de la población un valor de aptitud que será utilizado posteriormente por el mecanismo de selección para identificar las soluciones más prometedoras.

La función objetivo depende de cada problema y debe ser proporcionada por el usuario. Algunas veces, el problema que se quiere optimizar puede ser expresado como una función matemática que puede ser comprendida por los humanos, pero algunas veces no existe una formulación simbólica del objetivo que se desea optimizar. Por ejemplo, en un problema de diseño se puede desear obtener la forma que minimice la fricción del aire sobre una estructura, pero no existe una fórmula simbólica, sino que la evaluación de la aptitud se efectúa en un simulador. Matemáticamente, la función objetivo puede ser representada como $f : Dom \longrightarrow R$ (donde *Dom* es el dominio de la función y el resultado es un número real), pero los algoritmos genéticos no requieren que la función se pueda expresar de forma simbólica [86].
Los algoritmos genéticos funcionan bien en situaciones en que la función objetivo es estocástica, esto es, que puede devolver resultados diferentes utilizando las mismas entradas. De hecho se han creado algoritmos en los que la función objetivo es la opinión subjetiva de humanos, por ejemplo, para crear música o arte gráfico [47] [12].

Representación

La forma en la que las soluciones se representan en los individuos determina en gran manera el éxito de los algoritmos genéticos. En la práctica, el usuario debe decidir la representación más adecuada al problema que se desea resolver.
En algunos casos, la decisión de la representación es sencilla. Por ejemplo, en problemas en los que se debe elegir un subconjunto de objetos a partir de un conjunto finito conviene utilizar una representación binaria. En particular, en problemas del tipo de ”El problema de la mochila” en el que se supone un conjunto dado de objetos, cada uno con cierto peso y utilidad, se debe elegir un subconjunto de objetos que maximicen la utilidad total sin exceder un peso determinado. La selección se puede representar con un 0 o un 1 [49]. Así un subconjunto de los objetos está representado por una cadena binaria. Problemas similares ocurren en selección de atributos para aprendizaje automático, en los que el objetivo es identificar los atributos que resulten en la mejor desempeño de algún algoritmo clasificador [85].
En otros problemas, pueden existir varias representaciones adecuadas. Por ejemplo se podría utilizar una codificación binaria tradicional para representar números enteros para un problema de optimización. Además de esta codificación, podríamos haber utilizado una codificación binaria con códigos de Gray o utilizar números enteros directamente, pero estas elecciones requieren de operadores de variación más complejos.
Los códigos de Gray para codificaciones binarias tienen ciertas ventajas sobre la codificación binaria tradicional [63]. En los códigos de Gray los enteros consecu-

tivos están representados por cadenas binarias en las que un solo bit cambia, lo que no ocurre en la codificación tradicional.
Si las variaciones de un problema son siempre números enteros, entonces convendrá utilizar una representación con números de este tipo. Por ejemplo, en el diseño de una red hidráulica para repartir agua en una zona urbana puede utilizarse un algoritmo genético para escoger los diámetros de las tuberías que deben utilizarse. Las tuberías existen en ciertos diámetros estándar, así que es posible utilizar una codificación de números enteros que representen los diferentes diámetros [87].
Otras codificaciones con números enteros se utilizan frecuentemente en problemas en los que el objetivo es encontrar una permutación de objetos que optimicen la función objetivo. Por ejemplo, en problemas del vendedor viajero se debe encontrar el orden en el que un viajero debe recorrer un conjunto de ciudades para minimizar la distancia que recorre [55] [57] [62]. Todas las ciudades deben ser visitadas y cada ciudad se puede visitar una sola vez. Los problemas que involucran permutaciones requieren de operadores de variación que preserven las permutaciones (todos los objetos deben aparecer exactamente una vez en la representación).
Por supuesto, es posible que en un problema existan diversas clases de variables: enteras, reales, binarias; y es perfectamente aceptable combinarlas en la representación.

3.1.2. Reproducción: Selección y Reemplazo

El proceso de selección es crucial para el éxito de un algoritmo genético. Después de todo, el proceso de selección se encarga de dirigir al algoritmo hacia regiones del espacio de búsqueda que parecen prometedoras. Esta dirección ocurre cuando la selección identifica miembros de la población a los que se les asignó valores de aptitud que parecen ventajosos.
Hay diversos mecanismos de selección, pero todos tienden a escoger a los mejores individuos (con mejores valores de aptitud) y a descartar a los peores [35]. La mayoría de los mecanismos de selección son estocásticos, pero también existen versiones deterministas. Por ejemplo, la selección por truncamiento selecciona a todos los individuos que exceden cierto valor de aptitud (generalmente determinado por el promedio, la media, o algún otro percentil) o que están ranqueados entre los primeros m elementos de la población. Estos métodos aseguran que los mejores individuos siempre son seleccionados. La selección determinista es utilizada frecuentemente por los algoritmos genéticos conocidos como estrategias evolutivas.

Selección por Torneo

Este método elige al azar a s individuos de la población y selecciona al que tenga el mejor valor de aptitud. El parámetro s es dado por el usuario y determina la intensidad de la selección. Cuanto mayor sea el parámetro s, la competencia por sobrevivir es más aguda y es menos probable que individuos con aptitudes bajas sobrevivan.
Los participantes del torneo se pueden elegir muestreando la población con o sin reemplazo. En muestreo sin reemplazo, una vez que se elige un individuo en un torneo, el individuo no puede participar en otros torneos. Esto asegura que todos los individuos participen en exactamente un torneo. Cuando la población se muestrea con reemplazo, los participantes de cada torneo regresan a la población y pueden ser considerados para otros torneos. Es importante destacar que en muestreos con reemplazo es posible que algunos individuos participen en múltiples torneos y otros no participen en ninguno [65] [35].

Selección Proporcional (Ruleta)

Quizá el método se selección que de relaciona más frecuentemente a los algoritmos genéticos es la selección proporcional. Este método consiste en asignar a cada individuo i una probabilidad de ser seleccionado p_i de acuerdo a la razón de su valor de aptitud con respecto a la suma de todos los valores de aptitud: $p_i = f_i / \sum_j f_j$. Después se muestrea la población utilizando estas probabilidades. Los individuos con mayores aptitudes tendrán una probabilidad mayor de ser seleccionados que los menos aptos. Este método también se conoce como el método de la ruleta porque las probabilidades se pueden visualizar como las rebanadas de una ruleta.
El método proporcional tiene varias desventajas. Una de ellas es que el algoritmo se comporta de manera diferente si la función objetivo es transpuesta, por ejemplo sumándole una constante a la función $f'(x) = f(x) + c$. Al optimizar f' el denominador de las probabilidades p_i es distinto a lo que sería optimizando f. Otra desventaja es que conforme la ejecución del algoritmo avanza, la población tiende a estar compuesta de individuos con aptitudes similares (y si todo ha salido bien, con aptitudes altas). Cuando las aptitudes son similares, las probabilidades de selección tienden a ser uniformes. Así las probabilidades de seleccionar a los mejores individuos son solo un poco mayores que las probabilidades de seleccionar a los peores. Esto ocasiona que el algoritmo genético no progrese rápidamente [20].
Para disminuir este problema de estancamiento se han propuesto varias alternativas. Una alternativa es escalar (p.e., linealmente) los valores de aptitud de cada individuo y utilizar los valores escalados para determinar las probabilidades de

selección.
Otra alternativa es ordenar a los individuos de acuerdo a su aptitud y basar la selección en el rango (ranking). Así se evita que los individuos muy aptos dominen la población rápidamente, reduciendo la diversidad de la población que es necesaria para explorar más soluciones. Utilizando el rango para seleccionar también evita que la búsqueda se estanque en las etapas avanzadas [34].

Elitismo

Una variación que se puede incorporar a muchos métodos de selección es el elitismo. En esta variación se identifica a cierto número de los mejores individuos en cierta generación y estos individuos se insertan en la población después de aplicar un método de selección estocástico. Esto asegura que las mejores soluciones encontradas no se pierdan porque tengan la mala suerte de no ser seleccionadas. Generalmente se elige un número pequeño de los mejores individuos (1 o 2). También es posible insertar los individuos después de realizar las operaciones de variación para asegurar que los mejores individuos no se pierdan porque fueron destruidos por esos operadores.
Además de actuar como un seguro para evitar perder a las mejores soluciones, generalmente el elitismo acelera la evolución. Un método de selección conocido como truncamiento puede ser visto como un caso extremo de elitismo. Este método simplemente consiste en seleccionar determinísticamente un cierto porcentaje (generalmente alrededor de la mitad) de los mejores individuos de cada generación [86].

Reemplazo

Los métodos de selección también pueden ser utilizados para determinar qué individuos recién creados por los operadores de variación entrarán a la población. En los algoritmos genéticos simples se reemplaza a toda la población con los individuos recién creados, pero es posible pensar en ejecutar torneos entre los nuevos individuos y aquellos presentes en la población o reemplazar probabilísticamente a los peores individuos usando un método de ruleta [34].
Los mecanismos de selección consideran únicamente los valores de aptitud, ignorando la representación utilizada en las soluciones y los operadores que se aplicados para generar nuevos individuos [86].

3.1.3. Operadores de variación

Los operadores de variación son los componentes de los algoritmos genéticos que generan nuevas soluciones a partir de las soluciones existentes. La forma es-

pecífica de estos operadores depende en gran medida de la representación utilizada. De la misma manera en que existen representaciones típicas (usando números binarios o reales, o árboles o autómatas) también existen operadores de variación típicos.

Mutación

Quizás el método más sencillo para producir nuevos algoritmos es la mutación. En representaciones binarias este operador identifica aleatoriamente los bits que se va a mutar y cambia sus valores de 0 a 1 o viceversa. En los algoritmos genéticos generalmente se asocia una probabilidad pequeña para aplicar este operador a cada posición de cada individuo. Históricamente, esta probabilidad se situaba entre 0,01 y 0,001 independientemente del problema, pero más recientemente se ha vuelto común utilizar una probabilidad de $p_m = 1/l$, donde l es la longitud (número de bits) de los individuos. Una forma de implantar este método de mutación es generar un número aleatorio entre 0 y 1 para cada posición de cada individuo. En aquellas posiciones donde el número aleatorio sea menor que p_m se cambia el valor.
En representaciones discretas con alfabetos con más de dos símbolos, la mutación identifica aleatoriamente cuáles posiciones se van a mutar (posiblemente utilizando la implementación descripta anteriormente). Luego, para determinar el nuevo valor de dicha posición, se elige aleatoriamente un símbolo del alfabeto. Una opción es omitir el valor original entre los valores candidatos. Así, en un problema que utiliza los símbolos 1,2,3, si una posición que se eligió para mutar tiene el valor 1, solo los símbolos 2 y 3 son considerados como nuevos valores.
Pero también es frecuente que se consideren todos los símbolos como opciones para nuevos valores. Cuando se reportan resultados es importante notar si en la elección del nuevo valor se consideró al valor existente como una opción.
El operador de mutación para representaciones de números reales también puede tomar diferentes formas. La mutación más simple para este caso, es elegir un valor aleatorio dentro del rango de valores posibles para la variable que se desea mutar. Otra opción utilizada comúnmente en algoritmos genéticos es sumar al valor actual un número aleatorio que se obtiene de una distribución simétrica. Generalmente se utilizan distribuciones normales pero es posible utilizar otras (Cauchy o uniforme en cierto rango). En los algoritmos genéticos se continúa utilizando la mutación con una probabilidad muy baja, pero en los algoritmos evolutivos la mutación es el operador de variación principal y se utiliza con mayor frecuencia [11].

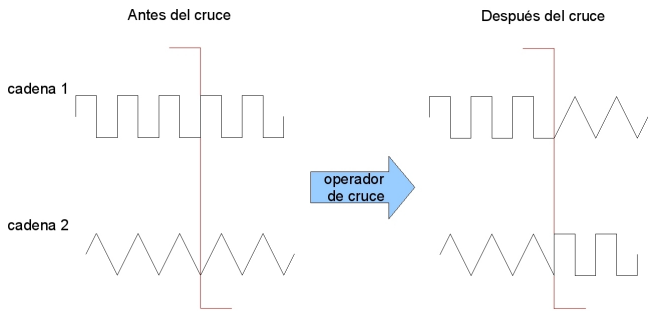

Figura 3.2: Un esquema de cruce simple en el que se observa la alineación de las dos cadenas y el intercambio parcial de información, usando un punto de cruce aleatorio.

Cruzamiento

El operador de cruce probablemente más sencillo es el que realiza cruce en un punto. Consiste en elegir al azar un punto en el interior de los cromosomas de dos individuos donadores y formar dos nuevos individuos copiando el segmento izquierdo del cromosoma de un padre con el segmento derecho el otro, como se observa en la figura 3.2 [34].

El cruce de un punto se puede generalizar a cruce en n puntos, donde los donadores se parten en $n + 1$ segmentos al azar y los descendientes se obtienen tomando segmentos alternos de cada donador. Por ejemplo, en cruce con dos puntos se obtienen tres segmentos de un donador X que podemos designar $X_1X_2X_3$ y tres segmentos del donador Y, $Y_1Y_2Y_3$. Los dos individuos que resultan del cruce en dos puntos son $X_1Y_2X_3$ y $Y_1X_2Y_3$.

Otro operador de cruce utilizado frecuentemente en la práctica es el cruce uniforme. Este operador trata cada posición de los descendientes independientemente y decide aleatoriamente el donador par cada posición de los hijos. Esto es, en lugar de dividir los cromosomas en segmentos que se intercambian, para cada posición se elige un donador al azar. Generalmente se utiliza una probabilidad igual para cada padre (50 %) y se utiliza con una probabilidad para cada posición, aunque es posible pensar en mecanismos para adaptar las probabilidades [88].

El cruce en n puntos tiende a mantener juntos los genes que están cercanos entre sí en la representación. En cambio, el cruce uniforme ignora completamente la posición de las variables en la representación.

Al igual que el operador de mutación, el operador de cruce tiene asociado una probabilidad de cruce. Históricamente, en los algoritmos genéticos, esta probabilidad es relativamente alta (alrededor de 0,6 a 1) en comparación con las probabilidades de mutación.

Los operadores de cruce en varios puntos y el cruce uniforme intercambian información de los padres y son adecuados para representaciones discretas, como la binaria u otros alfabetos de mayor cardinalidad. Aunque también es posible utilizar estos operadores de recombinación discreta cuando se utilizan números reales, en este caso es natural utilizar operadores que combinan los valores de cada parámetro.
Existen muchas maneras de hacer esto. Los operadores de recombinación aritmética (o recombinación intermedia) son una clase importante de operadores de cruce para representaciones reales. En general, estos operadores crean un nuevo valor para la posición i usando $z_i = \alpha x_i + (1 - \alpha)\, y_i$ para alguna α entre [0,1].
Existen diversas variantes de operadores de recombinación intermedia. El más común es conocido como recombinación aritmética completa y simplemente promedia todas las variables de dos donadores. Para crear dos descendientes se usa

$$Hijo_1 = \alpha\bar{x} + (1 - \alpha)\,\bar{y}$$
$$Hijo_2 = \alpha\bar{y} + (1 - \alpha)\,\bar{y}$$

donde $\bar{x}$ y $\bar{y}$ denotan los vectores de todas las variables del primer y segundo padres respectivamente. Otras variaciones de recombinación intermedia no cambian todas las variables. Por ejemplo un operador conocido como recombinación simple elige un punto k y copia las primeras k variables de cada padre a los hijos [86]. El resto de las variables se recambian usando el mismo método que la recombinación aritmética completa. Otra variación es la recombinación aritmética sencilla donde únicamente se combinan las variables de un punto k elegido al azar. El resto de las variables se copian de los padres sin ningún cambio.
Los operadores de recombinación son sencillos pero tienen la desventaja de disminuir los rangos de las variables. Después de todo, el promedio de dos números siempre será un número intermedio. Cuando se usa recombinación intermedia, la creación de valores fuera del rango de los valores de los padres queda a cargo de la mutación [34]. Existen otros operadores de recombinación que no limitan el rango de los hijos al rango de los padres. Por ejemplo se puede extender el rango de α en la recombinación a valores mayores a 1. Pero otros operadores obtienen el nuevo valor de forma aleatoria utilizando una distribución que está parametrizada con propiedades de los padres. Por ejemplo, se puede obtener el valor nuevo de una distribución normal centrada en el promedio aritmético de los padres y utilizando una desviación estándar que permita que el nuevo valor sea ligeramente mayor o menor a los padres.

3.1.4. Algoritmos Genéticos Paralelos

Los algoritmos genéticos paralelos se han vuelto muy populares y existe un gran número de implementaciones y algoritmos [10]. Las principales razones de

su éxito son, en primer lugar, el hecho de que los algoritmos genéticos son naturalmente paralelos y en segundo lugar, que la mayoría de los operadores de variación pueden ser fácilmente paralelizados. Sin embargo, una observación interesante es el uso de una población estructurada, una distribución espacial de los individuos, en forma de un conjunto de islas [95], o en una grilla de difusión [89], es la responsable de los principales beneficios.
Hay una larga tradición en el uso de poblaciones estructuradas en los algoritmos genéticos, especialmente asociados a distribuciones paralelas como por ejemplo: [1] [8] [10] [7] [3] [38]. La descentralización de una única población puede realizarse mediante particionamiento en varias subpoblaciones (islas) la cual se puede realizar de acuerdo a diferentes modelos de paralelismo.

Modelo de Ejecuciones Independientes

Este modelo consiste principalmente en ejecutar en paralelo el mismo algoritmo secuencial, sin comunicación alguna entre las ejecuciones independientes. Este método extremadamente simple de hacer en un trabajo simultáneo resulta muy útil, ya que puede ser usado para la ejecución de varias versiones del mismo problema con diferentes condiciones iniciales, permitiendo así generar datos estadísticos sobre el problema en cuestión. Como los algoritmos genéticos son naturalmente estocásticos, la posibilidad de contar con esta clase de estadísticas es muy importante [5].

Modelo de Maestro y Esclavo

El modelo de maestro-esclavo es fácil de visualizar. Consiste en la distribución de las evaluaciones de la función objetivo en varios procesadores mientras que el bucle principal del algoritmo genético se ejecuta en un procesador maestro. Este paradigma paralelo es bastante simple de implementar y su exploración del espacio de búsqueda es conceptualmente igual al de un algoritmo genético ejecutado en un procesador simple. En otras palabras, el número de procesadores usados es independiente de las soluciones evaluadas, excepto por el tiempo. Este paradigma se ilustra en la figura 3.3 donde el procesador maestro envía los parámetros necesarios para la evaluación de la función objetivo a los esclavos y el valor de la función es retornado una vez que el valor es calculado [9].

Modelo Distribuido

En este modelo, la población está estructurada en subpoblaciones más pequeñas relativamente aisladas unas de otras, lo cual es conveniente para la implementación.

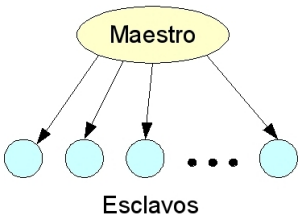

Figura 3.3: Modelo Maestro-Esclavo.

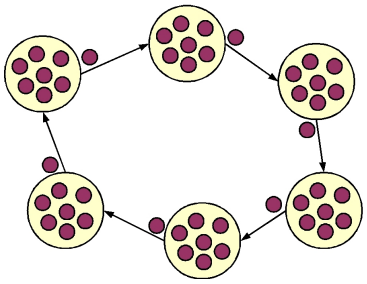

Figura 3.4: Modelo Distribuido.

Los algoritmos genéticos paralelos basados en este paradigma a veces se los llama algoritmos genéticos multipoblacionales. A parte de su nombre, la característica clave de esta clase de algoritmos es que los individuos dentro de una subpoblación particular (o isla) ocasionalmente puede migrar a otra. Este paradigma se ilustra en la figura 3.4. Hay que notar que los canales de comunicación que se observan son a modo de ejemplo. Las asignaciones específicas se hacen como parte de la estrategia de migración de los algoritmos genéticos y se mapean sobre una red física [4].

Modelo Celular

El paradigma de algoritmos genéticos celulares paralelos normalmente se encuentra asociado al concepto de una única población, donde cada procesador mantiene solo unos pocos individuos (normalmente uno o dos). Esto es porque este modelo a veces se lo conoce como paralelismo de *grano fino*. Su característica principal es la estructuración de la población en vecindarios, y los individuos solo pueden interactuar con sus vecinos. Así, las buenas soluciones (posiblemente) surgen en diferentes áreas en toda la topología y se dispersan lentamente a través de toda la estructura (población). Este modelo se ilustra en la figura 3.5 y las implementaciones de algoritmos genéticos celulares encajan naturalmente en él.

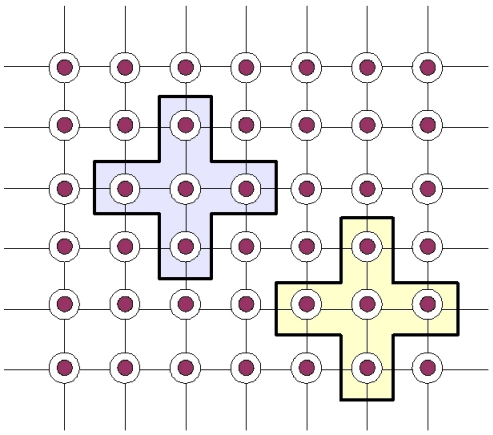

Figura 3.5: Modelo Celular.

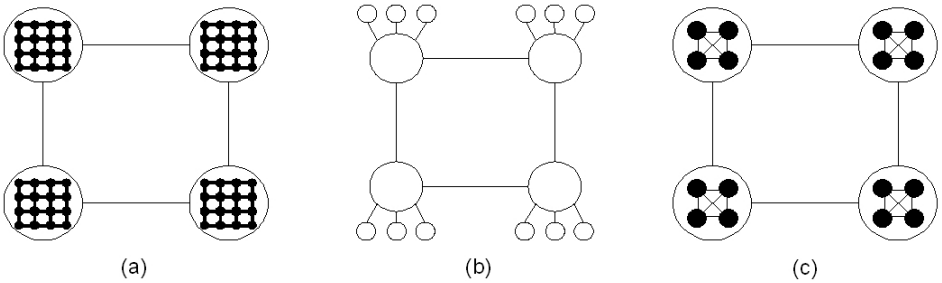

Figura 3.6: Modelos Híbridos.

Los algoritmos genéticos celulares se diseñaron inicialmente para el trabajo de un alto grado de paralelismo, aunque este mismo modelo ha sido adaptado para sistemas distribuidos [61] y maquinas monoprocesador [2] [38].

Otros Modelos

Es posible encontrar muchas implementaciones difíciles de clasificar en la literatura. En general, se los llama *algoritmos híbridos* a partir de las características de los diferentes modelos. Por ejemplo, la figura 3.6 muestra tres arquitecturas híbridas en las cuales se observa un enfoque de dos niveles de paralelización. En los tres casos el nivel más alto de paralelización es un algoritmo genético distribuido. En la figura 3.6 (a) se observa un modelo de islas básicas de un algoritmo genético celular que trata de combinar las ventajas de dos modelos [2]. En la figura 3.6 (b) se observan varios niveles de paralelización conectados en un modelo distribuido, obteniendo así una explotación del paralelismo para hacer evoluciones

rápidas y para obtener la evolución de poblaciones separadas al mismo tiempo [9]. Finalmente, la figura 3.6 (c) presenta varias islas de algoritmos distribuidos con un nivel más alto de distribución, permitiendo la migración entre islas conectadas [5].

3.2. Estrategias Evolutivas

Las estrategias evolutivas han sido utilizadas por varias décadas para resolver problemas complejos en muchas áreas. Estas estrategias se basan en principios de evolución biológica y genética. Existen muchas variantes dentro de estas estrategias, en esta sección solo se mencionarán las características más generales y, tomando como base los algoritmos genéticos vistos en la sección 3.1, se enumerarán las diferencias entre estos algoritmos genéticos y las estrategias evolutivas.

3.2.1. Reproducción

Los mecanismos de selección y reemplazo de las estrategias evolutivas son un caso extremo de lo que ocurre en los algoritmos genéticos . En estos algoritmos, se considera que la población entera es seleccionada para reproducirse, pero solo algunos individuos recién creados sobrevivirán para formar la siguiente población. En estas estrategias evolutivas se crean λ descendientes a partir de μ padres y λ es varias veces (generalmente alrededor de siete veces) mayor que μ. En una variación conocida como $\mu + \lambda$ los mejores μ individuos se eligen de la unión de los padres y los descendientes. Esta es una forma extrema de elitismo. En otra variación conocida como (μ, λ) los mejores μ se eligen sólo entre los descendientes y no hay elitismo [107].

3.2.2. Mutación

De acuerdo con lo visto en la sección 3.1.3, en los algoritmos genéticos se cuenta con una probabilidad de mutación relativamente baja, es decir que p_m toma valores entre 0,01 y 0,001. En las estrategias evolutivas todos los individuos son mutados (esto es $p_m = 1,0$). En la variación más sencilla de estrategias evolutivas cada componente i de una solución x muta de la siguiente manera:

$$x'_i = x_i + N(0, \sigma)$$

donde $n(\mu, \sigma)$ denota la distribución normal con media μ (0 en este caso) y σ es la desviación estándar. Para cada componente i se obtiene un nuevo valor aleatorio. La desviación estándar es un parámetro importante porque determina magnitud

de la mutación. En la variación más simple de estrategias evolutivas se utiliza un único valor de σ para obtener las mutaciones de todos los componentes de la solución (y de hecho para todas las soluciones candidatas). Este valor se puede ajustar durante la ejecución del algoritmo. Una forma sencilla para realizar esto consiste en actualizar el valor de σ después de cierto número de iteraciones usando la frecuencia de mutaciones que resultaron en una mejor solución.
Existen métodos más sofisticados de mutación en los que se incluyen parámetros que determinan la correlación entre diferentes componentes. Esto permite realizar mutaciones en direcciones arbitrarias, pero en la práctica son utilizadas poco frecuentemente porque requieren un número cuadrático $\left(O\left(l^2\right)\right)$ de operaciones y de espacio para mutar cada solución en la población [101].

3.2.3. Cruce y Recombinación

En la sección 3.1.3, se mencionaron de operadores de cruce que combinan de alguna manera la información de exactamente dos padres. En estrategias evolutivas es común utilizar la población completa para producir dos descendientes: para cada posición de los hijos se eligen al azar dos padres y se aplica recombinación directa o intermedia. La elección de los padres se realiza nuevamente en cada posición. Recuérdese que en estrategias evolutivas el cromosoma de los individuos contiene las variables del problema que se desea resolver y también los parámetros que controlan la mutación. Se acostumbra utilizar recombinación discreta para la parte del cromosoma con las variables del problema y recombinación intermedia para los parámetros de mutación [86].
La recombinación produce nuevos individuos combinando la información de dos o más padres. Si bien existen varios métodos de recombinación para individuos con variables discretas, los métodos de recombinación de la sección 3.2.3 son sólo aplicables a variables reales.

Recombinación Intermedia

En la recombinación intermedia [68] los valores de las variables de los descendientes son elegidos alrededor y entre los valores de las variables de los padres. Los descendientes se producen de acuerdo a esta regla:

$$Var_i^0 = Var_i^{P_1} \times a_i + Var_i^{P_2} \times (1 - a_i) \tag{3.1}$$

donde:

- $i \in (1, 2, ..., Nvar)$.
- $a_i \in [-d, 1 + d]$ aleatorio uniforme.

Figura 3.7: Área de los valores de las variables de los descendientes comparados con los valores de las variables de los padres en la recombinación intermedia.

- $d = 0,25$.
- a_i para cada nuevo i, siendo a un factor de escala uniformemente aleatorio en el intervalo $[-d, 1 + d]$ para cada nueva variable a.

El valor del parámetro d define el tamaño del área para los posibles descendientes. El valor $d = 0$ define el área para los descendientes del mismo tamaño que el área entre los padres. Este método se lo llama recombinación intermedia (estándar) porque la mayoría de las variables de los descendientes no son generadas en el borde, reduciendo el área para las variables a lo largo de las generaciones.Para asegurar que el área variable de los descendientes sea la misma que el área variable definida por las variables de los padres se utiliza un valor mayor en d. El valor $d = 0,25$ asegura estadísticamente este resultado. En la figura 3.7 se observa el área del rango de las variables de los descendientes definidas por las variables de los padres. La recombinación intermedia es capaz de producir cualquier punto dentro de un hipercubo un poco más grande que el definido por los padres. En la figura 3.8 muestra la posible área de descendientes después de la recombinación intermedia.

Recombinación Lineal

La recombinación lineal [68] es similar a la recombinación intermedia, excepto en que sólo se utiliza un único valor de a para todas las variables:

$$Var_i^0 = Var_i^{P_1} \times a + Var_i^{P_2} \times (1 - a) \tag{3.2}$$

donde:

- $i \in (1, 2, ..., Nvar)$.
- $a \in [-d, 1 + d]$ aleatorio uniforme.
- $d = 0,25$.

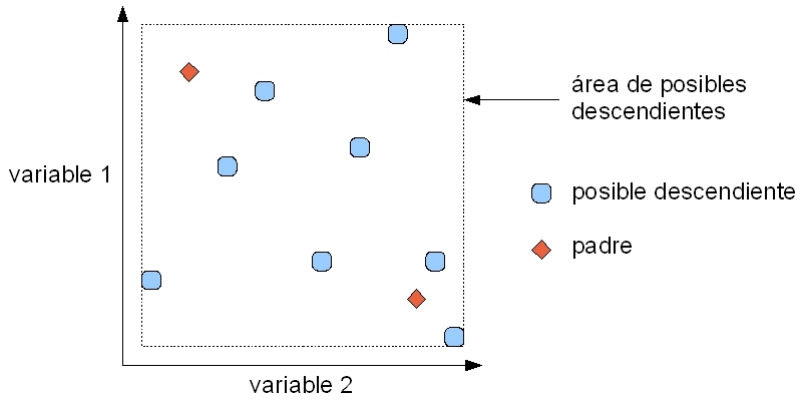

Figura 3.8: Posible área de los descendientes después de la recombinación intermedia.

- a para todos los i idéntico.

Para los valores de d, las aclaraciones dadas para la recombinación intermedia también valen en la recombinación lineal.
La recombinación lineal puede generar cualquier punto en la linea definida por los padres. En la figura 3.9 se observan las posibles posiciones de los descendientes después de la recombinación lineal.

Recombinación Lineal Extendida

La recombinación lineal extendida [69] genera descendientes en una línea definida por los valores de las variables de los padres. De todas maneras, la recombinación lineal extendida no está restringida a la línea entre los padres y una pequeña área fuera. Los padres definen la línea donde los posibles descendientes pueden ser creados; el tamaño del área se define por el dominio de las variables. Dentro de esta área, los posibles descendientes no están uniformemente distribuidos, ya que la probabilidad de la creación de descendientes cercanos a los padres es alta.
Los descendientes son producidos de acuerdo a la formula:

$$Var_i^0 = Var_i^{P_1} + s_i \times r_i \times a \times \frac{Var_i^{P_2} - Var_i^{P_1}}{\| Var_i^{P_1} - Var_i^{P_2}} \tag{3.3}$$

donde:

- $a = 2^{-ku}$, con a idéntico para todos los i, donde:
  - k es la precisión de mutación.

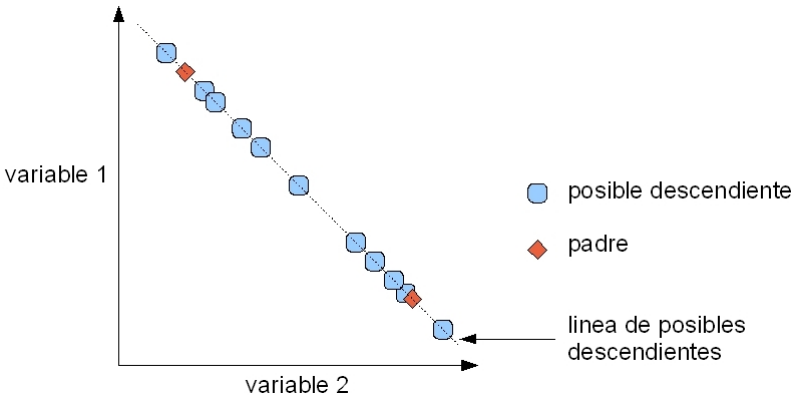

Figura 3.9: Posibles posiciones de los descendientes después de la recombinación lineal.

- $u \in [0, 1]$ aleatorio uniforme.

- $r \times dominioR$, donde $dominioR$ es el rango de pasos de recombinación.

- $s_i \in \{-1, +1\}$ aleatorio uniforme.

La creación de los descendientes usa características similares al operador de mutación para variables de valores reales. El parámetro a define el tamaño relativo del paso, el parámetro r el máximo tamaño de paso y el parámetro s la dirección del paso de la recombinación. La figura 3.10 muestra este efecto de la recombinación lineal extendida. El parámetro k determina la precisión usada para la creación de los pasos de recombinación. Con un k grande se producen pasos más pequeños. Para todos los valores de k el máximo valor de a es $a = 1$ ($u = 0$). Los valores típicos para el parámetro de precisión k están entre 4 y 20.
Un robusto valor para el parámetro r (rango de pasos de recombinación) es del 10 % del dominio de la variable. De todos modos, según el dominio definido para las variables o para casos especiales, se puede reajustar este parámetro. Mediante la selección de un valor pequeño para r la creación de los descendientes se realiza en una área más pequeña alrededor de los padres.
Si el parámetro s (dirección de búsqueda) se fija en -1 o $+1$ con igual probabilidad se realizará una recombinación sin dirección. Si la probabilidad de $s = +1$ es superior a $0,5$, se realizará una recombinación dirigida donde los descendientes son creados desde el peor padre hasta el mejor, donde el mejor padre siempre debe ser el primero.

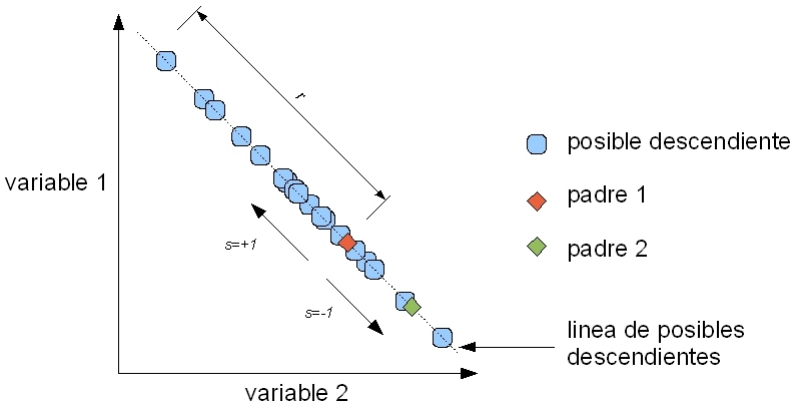

Figura 3.10: Posibles posiciones de los descendientes después de la recombinación lineal extendida de acuerdo a la posición de los padres y del área de definición de las variables.

3.2.4. Neuroevolución

La neuroevolución es una estrategia evolutiva que combina a los algoritmos evolutivos con redes neuronales artificiales (RNA). La evolución de redes neuronales artificiales se refiere a una clase especial de redes neuronales cuya evolución es otro proceso fundamental de adaptación en el aprendizaje [109]. Los algoritmos evolutivos se usan para desempeñar varias tareas como el entrenamiento de los pesos de las conexiones, el diseño de la arquitectura, adaptación de reglas de aprendizaje, selección de características de entrada, inicialización de los pesos de las conexiones, etc. Una característica distintiva de la neuroevolución es su adaptabilidad a entornos dinámicos. En otras palabras, la neuroevolución puede realizar adaptaciones a medida que los cambios se van sucediendo en el entorno. Las dos formas de realizar la adaptación en la neuroevolución, evolución y aprendizaje, hacen su adaptación mucho más efectiva y eficiente. En un sentido más amplio, la neuroevolución puede ser vista como un framework general para sistemas adaptativos, por ejemplo, sistemas que pueden cambiar su arquitectura y aprender reglas apropiadamente sin intervención humana.
La evolución ha sido introducida a las redes neuronales artificiales en tres niveles diferentes: los pesos de las conexiones, las arquitecturas y las reglas de aprendizaje.

- La evolución de los pesos de las conexiones introduce un enfoque de entrenamiento adaptativo y global, especialmente en el refuerzo del aprendizaje y el paradigma de aprendizaje de redes recurrentes, donde los algoritmos de entrenamiento basados en gradiente suelen sufrir grandes dificultades.

- La evolución de la arquitectura le permite a las redes neuronales artificiales adaptar su topología a diferentes tareas sin intervención humana y así proveer un enfoque hacia el diseño automático de redes neuronales donde ambos, los pesos de las conexiones y las estructuras, pueden ser evolucionados.

- La evolución de las reglas de aprendizaje pueden ser considerada como un proceso de ”aprendiendo a aprender” en redes neuronales donde la adaptación de las reglas de aprendizaje registrado a través de evolución. También pueden ser considerada como un proceso adaptativo de una búsqueda automática de nuevas reglas de aprendizaje.

De acuerdo con la simplicidad y generalidad de la evolución y el hecho de que los algoritmos de entrenamiento basados en gradiente con frecuencia deben ser ejecutados varias veces ya que suelen quedar atrapados en óptimos locales, el enfoque de la evolución resulta ser bastante competitivo.
La evolución puede ser usada para encontrar automáticamente una arquitectura de una red neuronal artificial cercana al óptimo. Esto tiene varias ventajas por sobre los métodos heurísticos de diseño de arquitecturas [110].

Capítulo 4

NeuroEvolution of Augmenting Topologies

NeuroEvolución (NE), la adaptación de las redes neuronales a través de estrategias evolutivas, ha mostrado ser una buena herramienta en el aprendizaje de tareas complejas [37] [39] [67] [79] [104]. La neuroevolución busca, a través del espacio de soluciones, determinar redes que puedan desarrollar correctamente una tarea dada. Este enfoque de resolver problemas complejos de adaptación representa una alternativa a las técnicas estadísticas que tratan de estimar la utilidad de acciones particulares en estados particulares del mundo. La NE es un enfoque prometedor para la resolución de problemas, ya que busca un comportamiento en lugar de una función de valor y es efectivo en problemas con espacios de estados continuos y de grandes dimensiones. Además, la memoria es fácil de representar a través de conexiones recurrentes en las redes neuronales.
En los enfoques tradicionales de la NE, se elige una topología para evolucionar las redes antes de comenzar con los experimentos. Normalmente, la topología de la red es una simple capa de neuronas ocultas, con cada neurona oculta conectada a cada entrada y a cada salida de la red. La evolución busca en el espacio de los pesos de las conexiones en esta topología completamente conectada permitiendo redes de alto rendimiento para reproducirse. El espacio de los pesos se explora a través del cruce de los vectores de los pesos de la red y a través de mutaciones simples en los pesos de las redes. Así, el objetivo de la NE de topología fija optimiza los pesos de las conexiones que determinan el funcionamiento de una red.
De todos modos, los pesos de las conexiones no son el único aspecto de las redes neuronales que contribuyen en su comportamiento. La topología, o *estructura*, de las redes neuronales también afecta a su funcionamiento.
Una importante cuestión en Neuroevolución es como ganar ventaja a partir de la topología de las redes neuronales evolutivas junto con los pesos.
En esta sección se presenta un método, NeuroEvolution of Augmenting Topolo-

gies (NEAT), el cual supera al mejor método de topología fija en un exigente punto de referencia como el aprendizaje de tareas [91]. Se trata de aumentar la eficiencia de acuerdo a:

1. El empleo de un método principal de cruce de diferentes topologías.

2. La protección de la innovación estructural usando especiación.

3. El crecimiento incremental desde una estructura mínima.

NEAT hace una importante contribución a los algoritmos genéticos porque muestra que por evolución se puede optimizar y, simultáneamente, hacer más complejas las soluciones, dando la posibilidad de evolucionar aumentando la complejidad de las soluciones a cada generación, fortaleciendo la analogía con la evolución biológica.

4.1. Codificación Genética

En el proceso de evolución de NEAT, se realizan diversos procesos de mutación que pueden cambiar tanto los pesos de las conexiones como la estructura de la red. Los pesos de las conexiones mutan como en cualquier sistema de NE, con cada conexión, ya sea perturbado o no en cada generación [109]. Las mutaciones de la estructura ocurren de dos maneras. Ambos tipos de mutación se ilustran en la figura 4.1 con los genes de conexión de una red junto con sus fenotipos. El número superior de cada genoma es el *número de innovación* del gen. Los números de innovación son marcas históricas que identifican el antecesor original histórico de cada gen. A los nuevos genes se les asignan nuevos números cada vez más altos. Cuando se agrega una conexión, un gen simple de una nueva conexión se agrega al final del genoma y dando el siguiente número de innovación disponible. Cuando se agrega un nuevo nodo, el gen de conexión a dividirse se deshabilita (DES) y dos nuevos genes de conexión se agregan al final del genoma. El nuevo nodo se sitúa entre las dos nuevas conexiones. Un nuevo gen de nodo (no graficado en 4.1) se agrega al genoma representando a este nuevo nodo.

Cada mutación expande el tamaño del genoma agregando uno o varios genes. En la mutación *agregar conexión*, se agrega un gen de una nueva conexión simple con un peso aleatorio que conecte dos nodos previamente desconectados. En la mutación *agregar un nodo*, se divide una conexión existente y se coloca un nuevo nodo donde la conexión estaba, como se observa en la figura 4.1. La vieja conexión se deshabilita y 2 nuevas conexiones se agregan al genoma. La nueva conexión que llega al nuevo nodo se le coloca 1 como valor del peso y la nueva conexión que sale recibe el peso que la vieja conexión. Esta forma de agregar

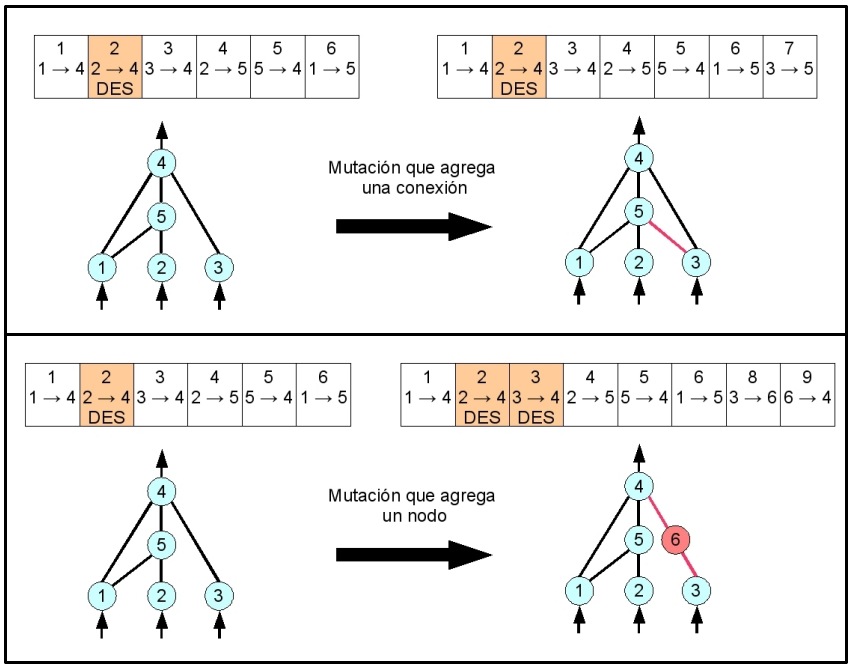

Figura 4.1: Los dos tipos de mutación de NEAT.

nodos permite minimizar el efecto inicial de la mutación. La nueva no linealidad en la conexión cambia ligeramente la función, pero los nuevos nodos pueden ser integrados inmediatamente a la red, en contraposición a agregar estructuras extrañas que tengan que ser evolucionadas luego dentro de la red. De esta manera, por la especiación, la red tendrá tiempo de optimizarse y hacer uso de su nueva estructura.
A través de la mutación, los genomas en NEAT aumentarán su tamaño gradualmente, dando como resultado a genomas de variados tamaños, a veces con diferentes conexiones en las mismas posiciones. El problema con topologías de diferentes tamaños y diferentes combinaciones de pesos es un resultado inevitable de permitir que los genomas crezcan sin límites [91].

4.2. Seguimiento de Genes

Hay información sin explotar en la evolución que puede decir exactamente cuales son los genes que pueden combinarse entre individuos de una población topológicamente diversa. Esta información es el origen histórico de cada gen. Dos genes con el mismo origen histórico deben representar la misma estructura (aunque posiblemente difieran en los pesos), desde que son derivados del mis-

mo gen ancestro en algún punto en el pasado. Así, todo el sistema necesita hacer saber cuales son los genes que se pueden combinar para mantener el seguimiento del origen histórico de cada gen en el sistema.
El seguimiento de los orígenes históricos es muy fácil de computar. Cuando un nuevo gen aparece (a través de la mutación estructural) se incrementa un *número de innovación global* y se lo asigna al gen. Así, los números de innovación representan una cronología de apariciones de cada gen en el sistema. Por ejemplo, suponiendo que las dos mutaciones de la figura 4.1 ocurren una después de la otra en el sistema, al nuevo gen de conexión creado en la primera mutación se le asigna el número 7 y a los nuevos genes de conexión agregados durante la mutación de un nuevo nodo se les asignan los números 8 y 9. En el futuro, cuando estos genomas se reproduzcan, los descendientes tendrán los mismos números de innovación en cada gen; los números de innovación nunca cambian. Así, el origen histórico de cada gen en el sistema es conocido en toda la evolución.
Cuando se cruzan, se alinean los genes en ambos genomas con el mismo número de innovación. Estos genes se conocen como genes *coincidentes*. Los genes que no coinciden se los llaman *disjuntos* o *exceso*, dependiendo de si se producen dentro o fuera del rango de los números de innovación del otro padre. Estos representan la estructura que no está presente en el otro genoma. En la composición del descendiente, los genes se eligen aleatoriamente de cada padre de entre los genes coincidentes, considerando que todos los genes exceso o disjuntos se incluyen siempre del padre más adecuado. De esta manera, las marcas históricas le permiten a NEAT llevar a cabo un cruce usando genomas lineales sin la necesidad de un costoso análisis de las topologías.
Las marcas históricas le dan a NEAT una nueva y poderosa capacidad. El sistema ahora sabe exactamente que genes combinan con cuales como se ve en la Figura 4.2. Aunque el padre 1 y el padre 2 se vean diferentes, sus números de innovación (es el número en la parte superior de cada gen) dicen cual gen encaja con cual. Los genes que coinciden se heredan aleatoriamente, mientras que los genes disjuntos (que no coinciden en el medio) y genes de exceso (que no coinciden al final) se heredan desde el padre más apropiado. En este caso, se asume igual fitness, por lo tanto los genes disjuntos como los genes de exceso también se heredan aleatoriamente. Los genes deshabilitados pueden habilitarse nuevamente en las futuras generaciones: hay una chance predeterminada que un gen heredado se deshabilite si está deshabilitado en alguno de los padres.
Si se desea tener un mayor detalle tanto de la codificación como del segmento de genes de NEAT, consultar [91] [92].
Mediante el agregado de genes a la población y la sensatez del apareamiento de genomas que representan diferentes estructuras, el sistema puede formar una población de diversas topologías. Sin embargo, resulta que dicha población en si misma no puede mantener las innovaciones topológicas, porque las estructuras

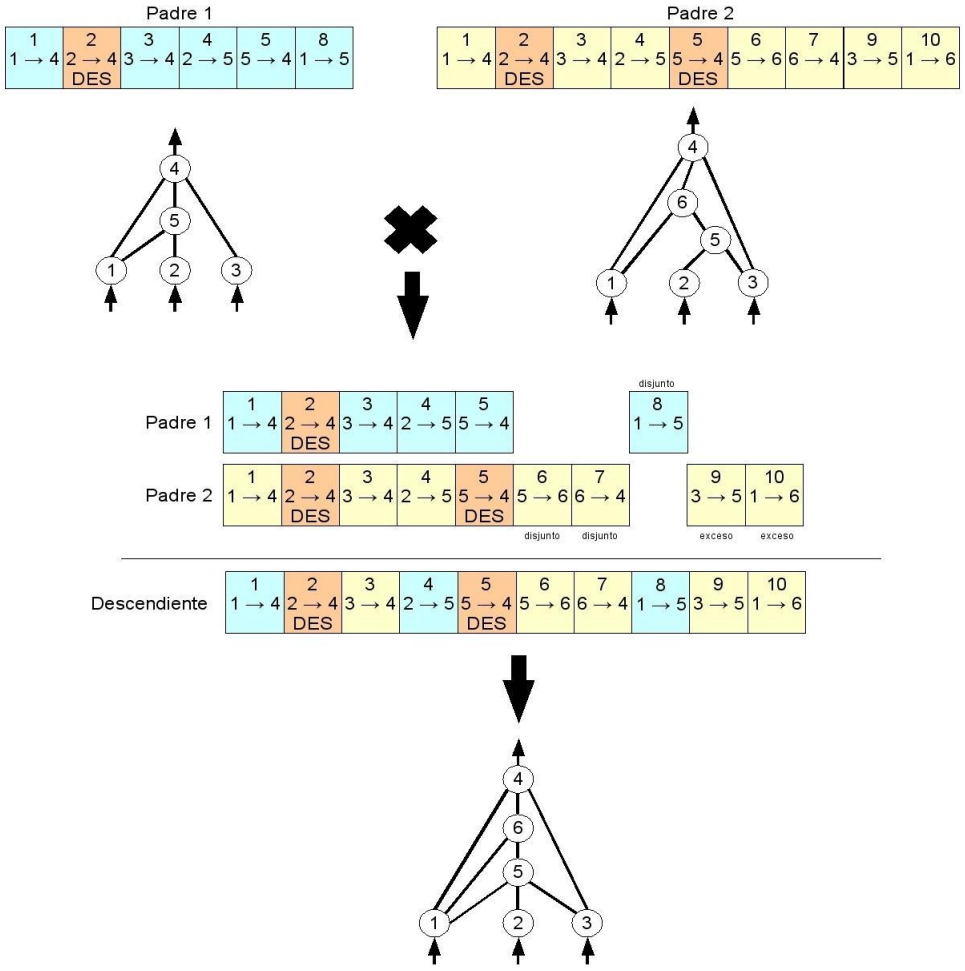

Figura 4.2: Combinación de genomas de redes de diferentes topologías.

más pequeñas se optimizan más rápido que las estructuras grandes y el agregado de nodos y conexiones, normalmente, decrece inicialmente el fitness de la red [91]. Las estructuras recientemente aumentadas tienen una pequeña esperanza de sobrevivir más de una generación aunque las innovaciones que representan pueden ser cruciales para la resolución de la tarea en una larga ejecución. La solución es proteger la innovación mediante la especiación de la población, como se explica a continuación.

4.3. Especiación

La especiación de la población le permite a los organismos competir principalmente dentro de sus propios nichos en lugar de hacerlo con la población en general. De esta manera, las innovaciones topológicas están protegidas en un nuevo nicho donde tienen tiempo de optimizar sus estructuras a través de la competencia en el nicho. La idea es dividir la población en especies de tal manera que cada una contenga topologías similares. Esta tarea parece ser un problema de coincidencia de topología. Sin embargo, resulta que de nuevo las marcas históricas ofrecen una solución eficiente.
El número de genes disjuntos y excesos entre un par de genomas es una medida natural de su distancia de compatibilidad. Cuanto más disjuntos sean dos genomas, menos historia evolutiva comparten y por lo tanto son menos compatibles. Entonces, se puede medir la distancia de compatibilidad δ de diferentes estructuras en NEAT como una simple combinación lineal del número de genes excesos E y el número de genes disjuntos D, así como el peso medio de las diferencias de los genes de coincidencia $\overline{W}$, incluyendo los genes deshabilitados:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{4.1}$$

Los coeficientes c_1, c_2 y c_3 permiten ajustar la importancia de los tres factores y el factor N, la cantidad de genes del genoma más largo, normaliza el tamaño del genoma (N puede tomar el valor 1 si ambos genomas son pequeños, por ejemplo si consiste en alrededor de 20 genes).
La medida de distancia δ permite la especiación usando un umbral de compatibilidad δ_t. Se mantiene una lista ordenada de las especies. En cada generación los genomas se sitúan secuencialmente en las especies. Cada especie existente se representa mediante un genoma aleatorio perteneciente a la especie de la *generación anterior*. Dado un genoma g en la generación actual se ubica en la primer especie en la cual g es compatible con el genoma representativo de dicha especie. De esta manera, las especies no se superponen. Si g no es compatible con ninguna especie existente, se crea una nueva especie con g como su representante.

Como mecanismo de reproducción de NEAT, se usa *fitness explícito compartido* [34], donde los organismos dentro de la misma especie deben compartir el fitness de su nicho. Así, una especie no puede darse el lujo de ser demasiado grande aún si varios de sus organismos tienen un buen desempeño. Por lo tanto, es poco probable que una especie abarque toda la población, lo cual es crucial para que la evolución por especiación funcione. El fitness ajustado f_i' para el organismo i se calcula de acuerdo a su distancia δ de cualquier otro organismo j en la población:

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} sh\left(\delta\left(i, j\right)\right)} \tag{4.2}$$

La función compartir sh vale 0 cuando la distancia $\delta\left(i, j\right)$ está por encima del umbral δ_t; caso contrario, $sh\left(\delta\left(i, j\right)\right)$ vale 1. Así, $\sum_{j=1}^{n} sh\left(\delta\left(i, j\right)\right)$ se reduce al número de organismos en la misma especie que el organismo i. Esta reducción es natural desde que las especies están agrupadas por compatibilidad usando el umbral δ_t. A cada especie se le asigna un número potencial de descendientes en proporción a la suma de fitness ajustados f_i' de sus organismos miembros. Las especies se reproducen a continuación, en primer lugar la eliminación de los miembros de la población de más bajo rendimiento. Luego, la población entera se reemplaza por los descendientes de los organismos restantes en cada especie.
El efecto deseado de la especiación de la población es proteger las innovaciones topológicas. El objetivo final del sistema, entonces, es llevar a cabo lo más eficientemente posible la búsqueda de soluciones. Este objetivo se logra a través de reducir al mínimo la dimensionalidad del espacio de búsqueda [91].

4.4. Crecimiento Incremental

NEAT sesga la búsqueda hacia espacios de dimensiones mínimas comenzando con una población uniforme de redes sin nodos ocultos (ej: todas las entradas se conectan directamente a la salida). Nuevas estructuras son introducidas incrementalmente a medida que ocurren las mutaciones estructurales y solo sobreviven aquellas estructuras que son encontradas útiles mediante la evaluación de su fitness. En otras palabras, la elaboración estructural que ocurre en NEAT siempre está justificada. Dado que la población comienza en una forma reducida, se minimiza la dimensionalidad del espacio de búsqueda y NEAT siempre busca en dimensiones reducidas. El minimizar la dimensionalidad le da a NEAT una ventaja en cuanto al desempeño comparada con otras aproximaciones.

4.5. Conclusiones

El método NEAT es una excelente herramienta para obtener RN de arquitectura mínima. Su desempeño se debe a las marcas históricas, a la especiación y al crecimiento incremental a partir de una estructura mínima. También se introduce una nueva técnica de visualización de especies para obtener una mejor comprensión de la dinámica del sistema.
Lo aquí expuesto es un resumen del método propuesto por Kenneth O. Stanley y Risto Miikkulainen. Para más detalles se recomienda consultar [91].

Capítulo 5

Robótica Evolutiva

La Robótica Evolutiva (RE) es el área de la robótica autónoma en la que se desarrollan controladores para robots utilizando estrategias evolutivas generalmente aplicadas a redes neuronales [36]. Este tipo de estructura, además de satisfacer la necesidad que la RE tiene por comprender el funcionamiento de su par biológico, posee la característica de ser fácilmente representable dentro de un algoritmo genético.
Dentro de las soluciones existentes, donde cada individuo de la población representa una opción dentro del espacio de soluciones y en función de la complejidad del problema a resolver, existen distintas propuestas que van desde adaptaciones de una única estructura mínima [72] hasta la división de la tarea en partes más simples que pueden ser aprendidas independientemente para luego combinarse [75]. También se han presentado alternativas donde la respuesta al problema no se encuentra dada por un único individuo sino por la población en su conjunto [66] o por combinación de subpoblaciones [16] [18]. En todos los casos el proceso de adaptación suele ser una tarea computacionalmente costosa.
En general, el método NEAT visto en el capítulo 4 es una buena alternativa a la hora de resolver problemas mediante la neuroevolución. Los resultados publicados por los autores del método muestran buen un desempeño para la resolución de distintas tareas. Sin embargo, la evolución de las redes neuronales mediante el método NEAT suele ser costoso en cuanto a tiempo de computo. Esto se debe, en parte, a que el método no solo evoluciona los pesos de las conexiones sino que evoluciona toda la arquitectura.
Por otro lado, las tareas de control complejas pueden ser resueltas dividiéndolas en módulos más específicos y fáciles de manejar. Varios autores han desarrollado distintas soluciones que combinan técnicas de Evolución por Módulos con Redes Neuronales Evolutivas dando lugar a controladores formados por varias redes. En este tipo de soluciones la elección del módulo a utilizar en cada caso no es un problema de fácil solución.

El presente capítulo describe uno de los principales aportes de esta tesina que es la incorporación de módulos al método de NEAT [75], es decir que incorpora una etapa previa de generación de una biblioteca de módulos neuronales básicos, donde cada uno de ellos es capaz de resolver una tarea simple, que al combinarse en un único controlador pueden resolver una tarea más compleja, reduciendo considerablemente el tiempo de entrenamiento. Por lo tanto, se propone que para la resolución de una tarea compleja, se divida la tarea en subtareas más simples de dos formas distintas: vertical y horizontalmente.
La forma de dividir al problema en subtareas depende de la forma en que las mismas deben ser resueltas. Si las subtareas deben resolverse en forma conjunta en un mismo instante de tiempo, se dice que la división será horizontal. Por el contrario, si existe un orden de resolución entre las tareas de acuerdo a dependencias jerárquicas, donde una tarea no puede ser resuelta hasta que no se resuelvan las tareas de las cuales depende, se dice que la división es vertical.
Por lo tanto, se plantean las dos formas de dividir y resolver las tareas como extensiones al método NEAT; en la sección 5.1 se refiere a la división horizontal y en el capítulo 6 trata sobre la división vertical.

5.1. NEAT con Módulos

Como forma de reducir el tiempo de evolución, se le realizaron modificaciones al método NEAT para incorporar la combinación de módulos pre-entrenados. En esta sección se describen las modificaciones realizadas al método NEAT que permiten combinar los módulos que resuelven las distintas partes de un problema dando lugar a una única red neuronal recurrente.
De esta forma, se trabaja con módulos simples que son entrenados independientemente del problema a resolver. La comunicación entre ellos se establece por evolución dando lugar a una única red neuronal que representa a la solución deseada.
Las modificaciones propuestas en esta sección ha sido utilizado para resolver el problema de evasión de obstáculos y alcance de objetivos utilizando un robot Khepera II. Las pruebas realizadas tanto en el ambiente simulado como sobre el robot real han arrojado resultados satisfactorios.

5.1.1. Incorporación de módulos al método NEAT

El enfoque de descomponer al problema original en módulos independientes que, luego de ser entrenados individualmente, combinan su funcionamiento integrando una única estructura permite reducir el tiempo de entrenamiento necesario para obtener una estructura que resuelva el problema completo ya que cada parte

implica una adaptación más simple. Como consecuencia de esta descomposición se obtendrán módulos generales reusables basados en redes neuronales.
Como se describió anteriormente, determinado tipo de tareas complejas pueden verse como la unión de varias tareas más simples. Si cada una de estas tareas simples es resuelta con éxito en forma independiente, debería ser posible combinar dichas soluciones para resolver la tarea compleja. Sobre esta premisa es que se introduce una nueva extensión al método NEAT, incorporando el concepto de composición de módulos.
Esta propuesta asume que se dispone de un conjunto de redes neuronales donde cada una de ellas, denominada módulo, es capaz de resolver una de las tareas simples. El objetivo de este trabajo será conseguir una red neuronal unificada, constituida por la combinación de todos estos módulos, que sea capaz de resolver la tarea compleja. La incorporación del concepto de módulos al método de NEAT involucra la realización de varias modificaciones. La primera de ellas tiene que ver con la topología de las redes neuronales que componen la población inicial. En la propuesta original, se asume que no se cuenta con información suficiente del problema como para especificar dicha topología. Además, comenzar con redes mínimas conduce al método a explorar primero las soluciones más simples. En el caso de la extensión propuesta, se conocen redes que resuelven distintas partes del problema, con lo que es posible comenzar el proceso evolutivo formando la población inicial con variaciones de una red neuronal unificada.
Esta red neuronal unificada se construye a partir de fusionar dentro de una misma estructura a cada uno de los módulos disponibles. Dado que las tareas resueltas por cada módulo son parte de una única tarea compleja, es de esperar que más de un módulo utilice las mismas entradas o produzca las mismas salidas. La red neuronal unificada poseerá como entradas la unión de las entradas de cada uno de los módulos, los cuales son conectados a ellas sin sufrir modificaciones. Las salidas de la red unificada dependerán de la tarea a resolver y por ello, la red contará con tantas neuronas de salida como necesite el problema en cuestión.
Más de un módulo puede generar la misma salida de la red. También es posible que distintos módulos produzcan estímulos opuestos para entradas similares, ya que las tareas que resuelven cada uno de ellos pueden ser contradictorias. Para que el proceso evolutivo ajuste el aporte de cada módulo a las salidas de la red unificada, potenciando respuestas deseadas y compatibilizando estímulos opuestos, las neuronas de salida de cada módulo se convierten en neuronas ocultas. A cada una de estas neuronas convertidas se le agrega una nueva conexión hacia la neurona de la red unificada encargada de producir la salida que daba originalmente la neurona en cuestión. La conexión se establece con peso 1.0, de manera que el estímulo original llegue a la neurona de salida sin que sea afectado. Esta nueva conexión no se considera como parte de algún módulo sino que pertenece a la red neuronal unificada. La Figura 5.1 ejemplifica el proceso de combinación de dos

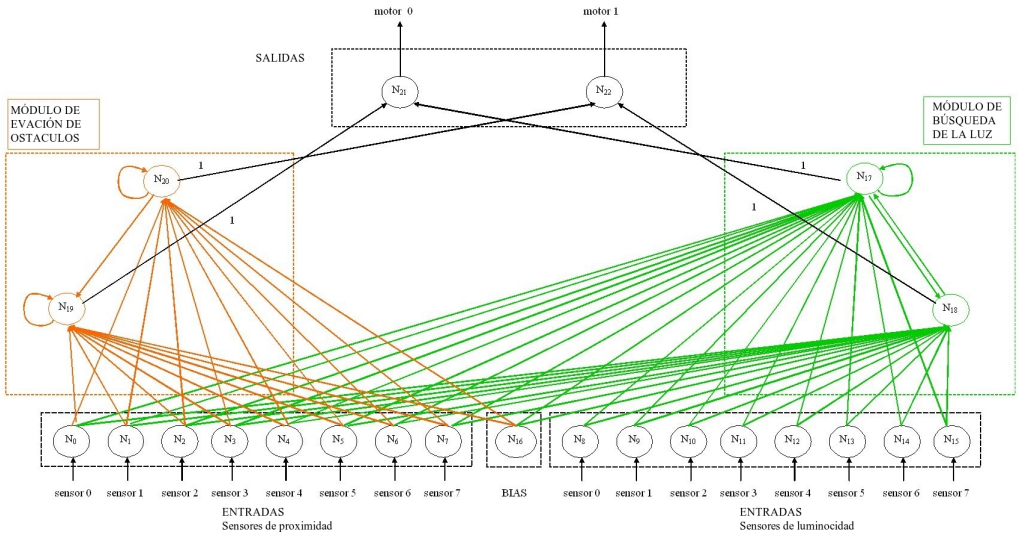

Figura 5.1: Combinación de dos módulos en una única red neuronal.

módulos para producir una red neuronal unificada.

Durante el proceso de construcción de la red unificada, tanto las conexiones como las neuronas que se incorporan a la red son marcadas con un identificador asociado al módulo al que pertenecían. Esto se hace para facilitar el seguimiento de los módulos que forman las redes una vez comenzada la evolución.

Otra de las modificaciones a NEAT que se proponen es la forma en que los operadores genéticos se aplican para producir nuevos genomas. Originalmente, el operador de mutación era el responsable de generar innovaciones, alterando pesos, agregando conexiones nuevas entre neuronas existentes o insertando una nueva neurona luego de dividir en dos una conexión existente, como se muestra en la Figura 5.2.

Dentro de las modificaciones realizadas al método NEAT, se ha restringido la aplicación del operador de mutación. Sólo es posible cambiar el peso de una conexión si no pertenecía originalmente a ninguno de los módulos que forman a la red sometida a mutación. De la misma manera, no se permite establecer nuevas conexiones entre neuronas de un mismo módulo, siendo únicamente válido hacerlo entre neuronas de módulos diferentes. Por último, sólo es posible agregar una neurona si previamente se divide una conexión existente, la cual, nuevamente, no debe ser una conexión aportada por alguno de los módulos. Estas restricciones conducen a que el método evolutivo genere la estructura necesaria para que los módulos originales interactúen y logren alcanzar la solución a la tarea compleja planteada.

El resto del método evolutivo no difiere de NEAT entandar; se mantiene una

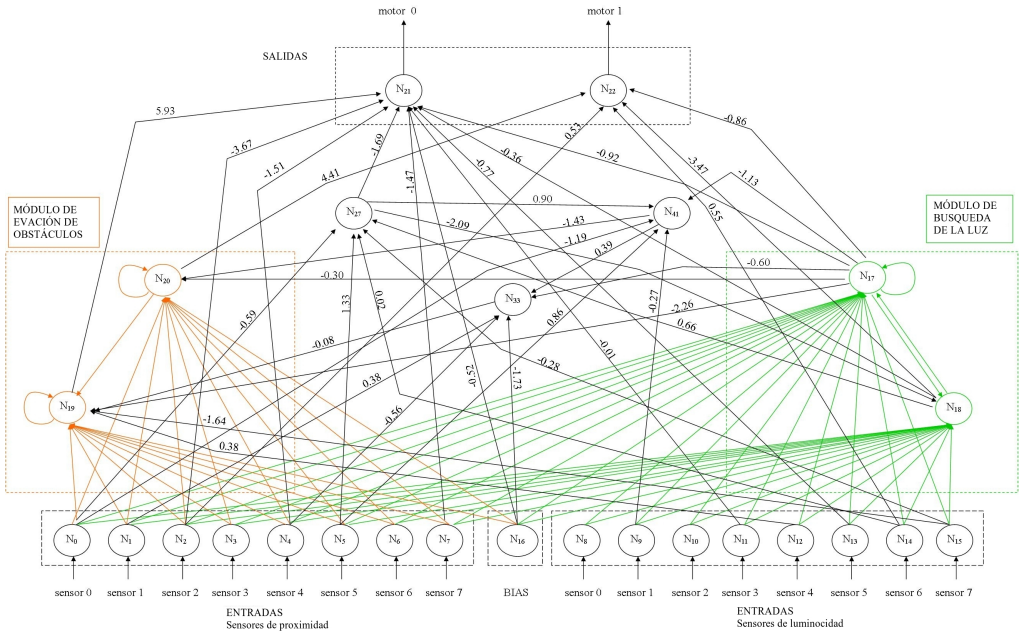

Figura 5.2: Red neuronal unificada y evolucionada.

población de genomas que emplean marcado histórico, se utiliza el operador de crossover original y se divide a la población en especies de acuerdo a un criterio de compatibilidad, dividiendo el fitness de cada miembro en proporción a la cantidad de genomas que pertenecen al mismo nicho.
El motivo por el cual no se permite que la topología y los pesos de conexión de cada módulo cambie se debe a que al estar éstos fijos, el proceso evolutivo debe buscar en un espacio más reducido que si tuviera que hacerlo sobre una red completamente mutable. Esto debería favorecer una convergencia más rápida hacia mejores soluciones. Cabe mencionar que se corre el riego de generar una red neuronal con una estructura innecesariamente mayor a la que se podría conseguir si se partiera de una topología mínima. Sin embargo, cuando la dificultad de la tarea aumenta, la complejidad de la red neuronal crece en proporción y generar una estructura que haga de interfaz entre los módulos resulta más simple que resolver el problema con NEAT estándar [75].

5.1.2. Prueba de las modificaciones

Para probar las modificaciones introducidas al método de NEAT, se ha desarrollado un controlador basado en redes neuronales evolutivas que permita dotar a un robot Khepera II de la habilidad de esquivar obstáculos y alcanzar objetivos.

Dado que se han utilizado únicamente los sensores de luz y proximidad del robot, los objetivos buscados estarán representados por luces ubicadas en cualquier posición dentro de su entorno de navegación. Se espera lograr que el robot se desplace libremente, sin colisionar con los obstáculos hasta acercarse a una zona iluminada donde deberá intentar permanecer.
Para tal fin, utilizando la extensión al método de NEAT anteriormente descripta, se combinarán dos módulos obtenidos de manera independiente: uno que permita esquivar obstáculos y otro que permita alcanzar la luz más próxima. Cada uno de estos módulos se encuentran representados por una red neuronal recurrente obtenida a través del método de NEAT convencional [91].
La estrategia propuesta en esta sección, si bien ha sido aplicada para obtener un controlador a partir de dos comportamientos adquiridos previamente, puede ser extendida fácilmente incorporando módulos adicionales con sus correspondientes funciones de fitness. De esta forma, el disponer de módulos independientes previamente entrenados facilita la obtención de controladores más complejos.

5.1.3. Módulos básicos

Tanto el módulo de evasión de obstáculos como el que permite al robot alcanzar la luz han sido evolucionados utilizando NEAT convencional a lo largo de 500 generaciones. En ambos casos, el fitness de cada individuo se obtiene como el promedio de su calificación en cuatro intentos independientes. La diferencia en cada caso radica en la posición y orientación inicial del robot. Dado que las pruebas han sido realizadas en un laberinto rectangular, cada intento comienza con el robot en un extremo diferente. La figura 5.3 contiene el pseudocódigo del algoritmo utilizado.
De esta forma, el valor de fitness de un individuo está dado por el promedio de los resultados de las cuatro pruebas a las que fue sometido. A continuación se describen las características específicas de cada uno de los módulos.

Módulo para evasión de Obstáculos

Se trata de una red neuronal recurrente que utiliza 8 entradas, una para cada uno de los sensores de proximidad del robot y dos neuronas de salida, una para cada uno de los motores.
Las entradas a la red son linealmente escaladas al rango [0..1] partir de los valores recogidos por los sensores.
Todas las neuronas de la red, salvo las de entrada y la de bias, aplican la función *logsig* como función de transferencia. Esto implica que la salida de la red deberá ser escalada entre [-1..1] a fin de poder controlar adecuadamente los motores que actúan sobre las ruedas del robot.

```
Generar una población inicial de 150 individuos.
repetir
  para cada individuo de la población
    para i = 1 : 4
      Ubicar al individuo en la posición i
      Calcular la aptitud del individuo en esta etapa
      representado por Eval. Si durante esta la evaluación
      el robot colisiona, se interrumpe su recorrido y se
      retorna el valor de Eval con lo acumulado hasta este punto.
    fin para
    Calcular el fitness del individuo como el promedio de las 4
    evaluaciones anteriores.
  fin para
  Seleccionar
  Recombinar y mutar utilizando NEAT estándar.
hasta (alcanzar las 500 generaciones)
```

Figura 5.3: Pseudocódigo del algoritmo evolutivo.

La estructura inicial está formada por 8 neuronas lineales de entrada, dos no lineales de salida y una neurona adicional de bias con capacidad de conectarse con cualquier otra neurona de la red que no sea de entrada.
Para medir la aptitud de un individuo durante la simulación se utiliza la siguiente función de fitness:

$$Eval_{Obs} = zonas \times \sum_{t=1}^{500} \Big(M_{izq} + M_{der}\Big) \times \Big(1 - |M_{izq} - M_{der}|\Big) \times (1 - S_{ir}) \qquad (5.1)$$

donde:

- M_{izq} y M_{der} son valores en el intervalo [-1..1] correspondientes a la velocidad de los motores izquierdo y derecho respectivamente.
- S_{ir} es el máximo valor de los sensores de proximidad en el intervalo [0..1].
- *zonas* es un valor proporcional a la cantidad de sectores recorridos por el agente durante el entrenamiento.

La optimización del término $\Big(M_{izq} + M_{der}\Big)$ obliga al agente a maximizar su desplazamiento ya que el mayor valor lo obtiene cuando el robot va hacia adelante a máxima velocidad.
El termino $\Big(1 - |M_{izq} - M_{der}|\Big)$ hace referencia a la rotación del robot. Por ejemplo, si el robot se encuentra girando sobre si mismo la velocidad en ambos motores es

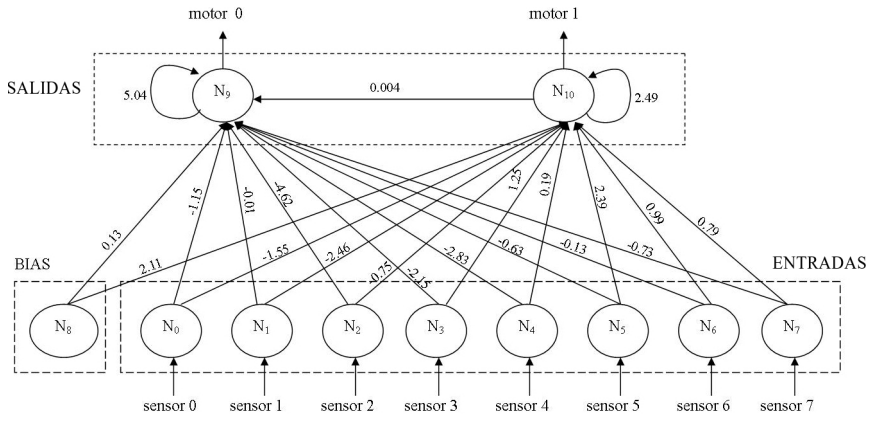

Figura 5.4: Red para la evasión de obstáculos.

la misma pero con signo contrario, alcanzando su mayor valor cuando la velocidad sea máxima. Por lo tanto, el agente deberá minimizar este efecto para incrementar su fitness.
Por su parte, el término $(1 - S_{ir})$ obliga al agente a alejarse de los obstáculos.
Con el objetivo de obtener individuos capaces de cubrir grandes distancias, se dividió el entorno en un reticulado de 100x100 sectores iguales y se utilizó el coeficiente *zonas* para medir la cantidad de sectores por los que el agente pasó a lo largo de una prueba.

$$zonas = \frac{\sum_{x=1}^{100}\sum_{y=1}^{100} zona_{xy}}{100 \times 100} \tag{5.2}$$

$$zona_{xy} = \begin{cases} 1 & \text{Si el agente paso por el sector(xy)} \\ -1 & \text{en caso contrario} \end{cases} \tag{5.3}$$

En resumen, en la formula 5.1 se pondera el recorrido del robot durante 500 pasos (sumatoria) en forma proporcional a la cantidad de sectores recorridos.
La figura 5.4 muestra la arquitectura obtenida.

Módulo para el alcance de la luz

Se trata de una red neuronal recurrente que utiliza 16 entradas, 8 para cada uno de los sensores de proximidad del robot, 8 para cada uno de los sensores de luz y dos neuronas de salida, una para cada uno de los motores.
Al igual que en el módulo anterior, las entradas a la red son linealmente escaladas

al rango [0..1] a partir de los valores recogidos por los sensores y todas las neuronas de la red, salvo las de entrada y la de bias, aplican la función *logsig* como función de transferencia.
La estructura inicial está formada por 16 neuronas lineales de entrada, dos no lineales de salida y una neurona adicional de bias con capacidad de conectarse con cualquier otra neurona de la red que no sea de entrada.
Para medir la aptitud de un individuo durante la simulación se utiliza la siguiente función de fitness:

$$Eval_{lig} = \sum_{t=1}^{500} \alpha \times \left(M_{izq} + M_{der}\right) \times \left(1 - abs\left(M_{izq} - M_{der}\right)\right) + \beta \times (1 - S_{li}) \quad (5.4)$$

donde:

- M_{izq} y M_{der} son valores en el intervalo [-1..1] correspondientes a la velocidad de los motores izquierdo y derecho respectivamente.
- S_{li} es el máximo valor de los sensores de luz en el intervalo [0..0,9]. El valor cero corresponde a la ausencia de luz y 0,9 a la detección de luz plena.
- α y β son constantes de escala.

En la formula 5.4 se pondera el movimiento del robot de la misma forma que el módulo anterior, es decir que el agente intentará maximizar su desplazamiento minimizando las rotaciones. Sin embargo, el hecho de sumar una cantidad proporcional al mayor valor de luz sensado permite que el desempeño del agente siga siendo considerado como bueno aun en el caso en que no se desplace. Esto último permite que el robot permanezca cerca de la luz. La figura 5.5 muestra arquitectura obtenida.

5.1.4. Resultados

Los módulos representados en las figuras 5.4 y 5.5 son integrados en una única estructura siguiendo el método descrito en la sección 5.1.1. La figura 5.1 corresponde a la arquitectura inicial. En ella puede observarse que los módulos comparten la información suministrada por los sensores de proximidad.
La red neuronal unificada combina la salida de cada uno de los módulos a través de dos neuronas nuevas que serán las encargadas de comandar a los motores del robot. Así, la red queda constituida con 16 entradas, 8 para cada uno de los sensores de proximidad del robot y 8 para cada uno de los sensores de luz, 4 neuronas ocultas y 2 neuronas de salida, una para cada uno de los motores. La población inicial está formada por redes cuyos genomas se obtienen al aplicar el operador de

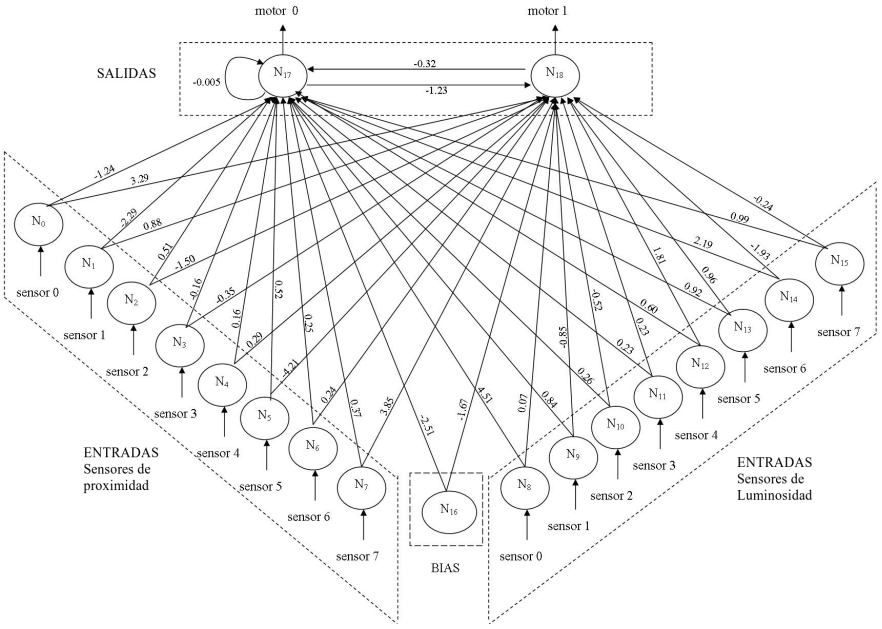

Figura 5.5: Red para el alcance de la fuente de luz.

mutación definido en la sección 5.1.1 a esta estructura inicial. Como se explicó en esa sección, esta población se evoluciona según el método NEAT con las restricciones impuestas a los operadores genéticos.
El fitness de un individuo estará dado por su desempeño al combinar los comportamientos de cada uno de los módulos originales. Las funciones de fitness empleadas para cada uno de los módulos se integran en una única expresión:

$$fitness = coef_{obs} \times Fit_{obs} + coef_{lig} \times Fit_{lig} \tag{5.5}$$

donde:

- Fit_{Obs} y Fit_{lig} corresponden a los fitness del individuo respecto de la evasión de obstáculos y el alcance de luz respectivamente.
- $coef_{obs}$ y $coef_{lig}$ son constantes que permiten ajustar el peso relativo de la tarea que originalmente desempeñaba cada módulo, a fin de que intervengan adecuadamente en la ponderación del fitness de la red unificada. En esta solución específica se utilizaron los valores 1 y 1.3 respectivamente con el fin de poner un mayor énfasis en el alcance de la luz.

La figura 5.6 resume la evolución anteriormente descripta.
La extensión al método de NEAT propuesta en este artículo fue comparada con

```
Generar una población inicial de 150 individuos a partir de
mutaciones de la red unificada.
repetir
   para cada individuo de la población
      para i = 1 : 4
          Ubicar al individuo en la posición i
          Calcular EvalObs utilizando (eq.1)
          Calcular EvalLig utilizando (eq.2)
      fin para
      Calcular FitObs como el promedio de las 4 evaluaciones
      anteriores.
      Calcular FitLig como el promedio de las 4 evaluaciones
      anteriores.
      Fitness = coef_obs * FitObs + coef_lig * FitLig
   fin para
   Seleccionar
   Recombinar y mutar utilizando la nueva extensión de NEAT
   con módulos
hasta (alcanzar las 500 generaciones)
```

Figura 5.6: Pseudocódigo del cálculo del fitness.

la solución de NEAT convencional y con una versión intermedia que utiliza la misma topología inicial que la versión con módulos (como el ejemplo de la figura 5.1) pero permite que toda la arquitectura evolucione, es decir, que se utilizan los operadores genéticos de NEAT estándar.
Para comparar el desempeño de las tres opciones presentadas, se realizaron 30 ejecuciones independientes de cada una de ellas, cada una con una duración de 500 generaciones. En la figura 5.7 se muestra el promedio de los mejores valores de fitness por generación de cada una de las variantes. En la figura 5.8 se muestran por separado cada uno de estos resultados incluyendo el valor mínimo y máximo de fitness, con el objetivo de visualizar la variación de estos valores por generación. Cabe destacar que el método propuesto supera el desempeño de NEAT estándar tanto en la cantidad generaciones necesarias para alcanzar un valor de fitness dado así como para el máximo valor obtenido. También, debe notarse que comenzar el proceso evolutivo con redes unificadas que son completamente mutables no mejora el desempeño de NEAT estándar. Eso puede justificarse si se tiene en cuenta que al iniciar la evolución con una estructura más compleja, el espacio de búsqueda es mayor. Por lo tanto, encontrar la combinación adecuada para la topología y los pesos resulta ser un problema más complicado.

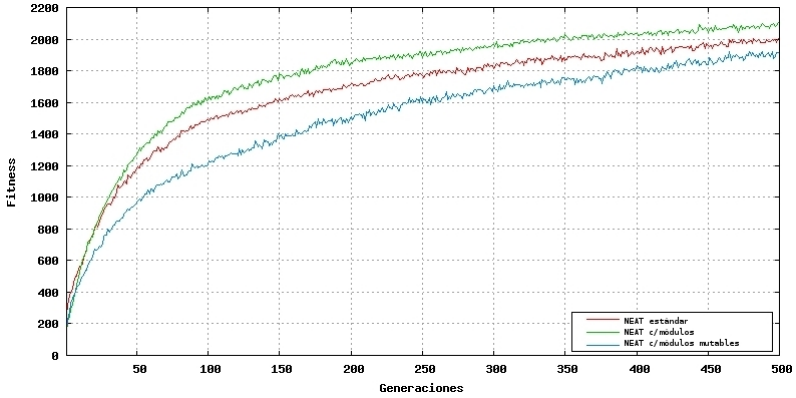

Figura 5.7: Promedios de mejores valores de fitness por generación.

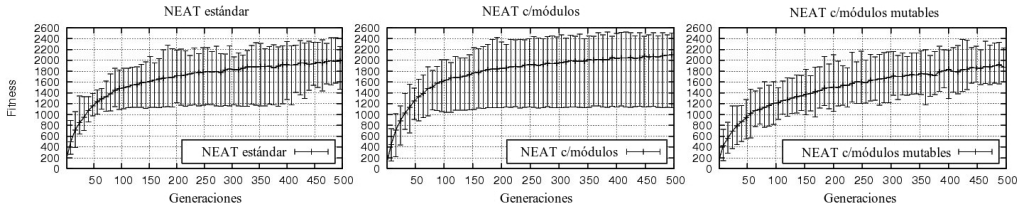

Figura 5.8: Promedios discriminados de mejores valores de fitness por generación.

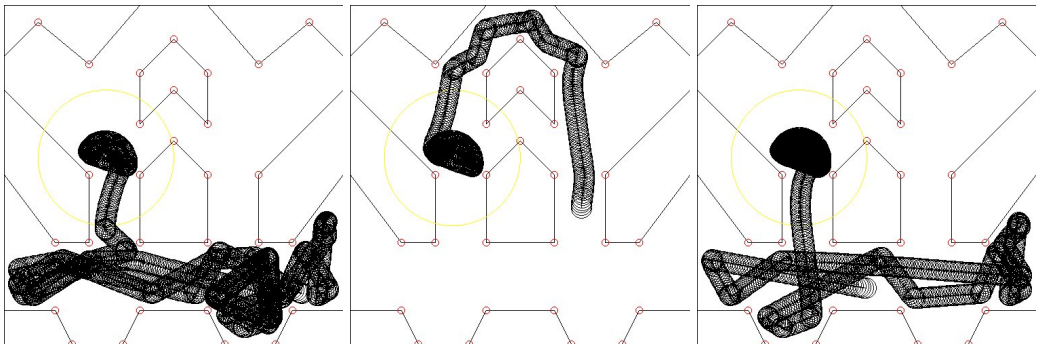

Figura 5.9: Comportamiento del mejor individuo generado utilizando NEAT a partir de una topología inicial basada en los módulos.

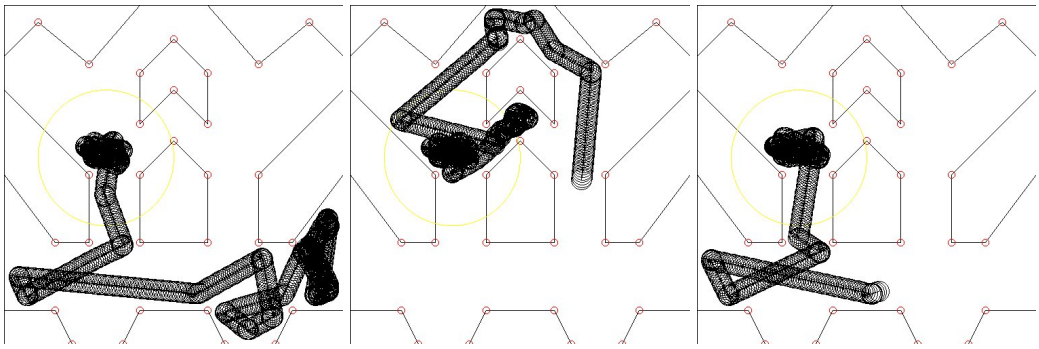

Figura 5.10: Comportamiento del mejor individuo generado utilizando NEAT con módulos .

Las figuras 5.9 y 5.10 muestran el funcionamiento del mejor individuo encontrando en las 30 ejecuciones para el caso de NEAT estándar, figura 5.9, y de NEAT con módulos, figura 5.10. En las tres simulaciones, el robot comenzó desde posiciones diferentes a las utilizadas durante la evaluación de su fitness.

5.2. Evolución Continua

A lo largo de los años, se realizaron diversas experiencias en la investigación en técnicas basadas en algoritmos evolutivos para el desarrollo de controladores aplicables a robots autónomos. Si bien los algoritmos evolutivos son una de las herramientas más utilizadas en este tipo de tareas debido a su capacidad de adaptación al entorno, en su gran mayoría su aplicación se concentra en la fase de generación del controlador no permitiendo realizar adaptaciones con posterioridad. Esto perjudica la aplicación del controlador en ambientes dinámicos.
En esta sección se propone extender la evolución del controlador a lo largo de su vida útil combinando el método basado en evolución de módulos neuronales visto en la sección 5.1 con un algoritmo evolutivo específico. El primer método es utilizado para producir los controladores mientras que el segundo ajusta al controlador durante su funcionamiento. Como resultado, se obtiene un controlador basado en una red neuronal que posee una arquitectura mínima y capacidad de adaptación en la fase de ejecución.
También se realizó un estudio comparativo entre distintos operadores genéticos con el objetivo de poder establecer en que situaciones y bajo que circunstancias es conveniente utilizar uno u otro operador. Este estudio permite elegir el mejor operador genético para realizar la segunda etapa de la evolución [97].
Los controladores producidos por la primera etapa consisten de una única estruc-

tura obtenida de la combinación de módulos neuronales simples e independientes, que han sido evolucionados previamente. Esto busca reducir el tiempo de entrenamiento necesario para obtener un buen desempeño.
Si el ambiente en el que se desempeñará el robot no sufriera modificaciones, el mejor de este conjunto de controladores sería suficiente para resolver el problema ya que su interacción con el entorno estará contemplada en la información suministrada durante el entrenamiento. Sin embargo, en el mundo real, se producen modificaciones, por ejemplo los obstáculos cambian de lugar o el objetivo a alcanzar modifica sus características externas; esto lleva a realizar ajustes sobre el controlador en la etapa de uso. Ese es el objetivo de la segunda etapa, ajustar la interacción del robot con el medio. Para ello se utilizará una pequeña población formada por los tres mejores controladores obtenidos en la primera etapa y se aplicará sobre ellos un segundo algoritmo evolutivo diseñado para ejecutarse sobre el robot.

5.2.1. Estrategia evolutiva utilizada

La estrategia evolutiva utilizada en esta sección se divide en dos etapas. Para la primera etapa se utilizó estrategia descripta en la sección 5.1 y para la segunda etapa el algoritmo que se describe a continuación.
La segunda fase de la evolución se desarrolla sobre el robot físico y puede tomarse como una etapa infinita o de larga duración ya que, en teoría, nunca debería terminar. Dentro del robot se mantiene una pequeña población de tres individuos que provienen de la primera etapa. La elección de los individuos que componen esta población inicial puede realizarse de diferentes maneras. Una estrategia de selección puede ser tomar los tres individuos con mejor desempeño de la población como candidatos. Otra alternativa sería elegir el mejor individuo de cada una de las tres mejores especies. En este trabajo se optó por la primera opción. Los individuos seleccionados contendrán el material genético que permitirá adaptar el comportamiento del robot ante los cambios del entorno. Dicho proceso de adaptación consiste en realizar pequeñas modificaciones sobre los controladores, causando cambios en su comportamiento. Para ello, en cada generación se evalúan los tres controladores. A partir de los dos individuos con el mejor desempeño se obtendrá un nuevo controlador que reemplazará al peor individuo de la población original. De esta manera se asegura que nunca se pierda al mejor individuo. Para determinar el operador genético más conveniente para esta tarea, quién será el encargado de realizar pequeñas modificaciones sobre la población de 3 individuos, se realizó el estudio descripto a continuación en la sección 5.2.2.

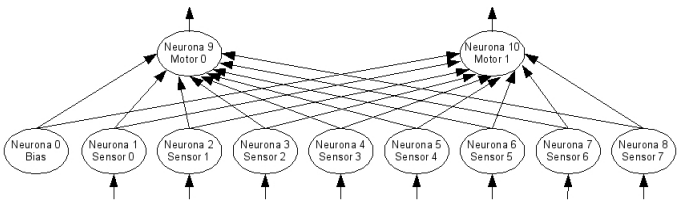

Figura 5.11: Arquitectura del controlador.

5.2.2. Evaluación de Operadores

Realizar la adaptación del controlador a través de un proceso evolutivo implica indefectiblemente la modificación o adecuación de los operadores genéticos a fin de favorecer la mejora de la aptitud de los individuos a lo largo de las generaciones [56]. En general, se ha comprobado que el operador de crossover resulta perjudicial a la hora de combinar dos redes neuronales [110].
Para reducir el efecto destructivo del cruce (crossover) se propone utilizar una estrategia de creación de varios hijos por cada par de padres seleccionando adecuadamente la manera de efectuar los reemplazos. Los resultados obtenidos muestran que aunque el número de hijos es elevado, la cantidad de evaluaciones de fitness realizadas es muy inferior a la de un algoritmo evolutivo convencional. De esta forma, se plantea una alternativa que reduce el costo computacional del proceso adaptativo logrando una respuesta adecuada para la resolución de distintos problemas.

Arquitectura del controlador

Se busca obtener por evolución la arquitectura de la red neuronal feedforward de la Figura 5.11, capaz de comandar un robot Khepera II [64]. Está formada sólo por dos capas: una de entrada y otra de salida. La primera contiene nueve neuronas de las cuales ocho reciben la información de los sensores de proximidad del robot y la restante se utiliza como bias. La capa de salida está formada por dos neuronas; cada una de ellas controla uno de los motores de las ruedas del robot.

Proceso evolutivo

Para realizar esta evaluación se usó un algoritmo evolutivo que utiliza una única población de redes neuronales de dimensión fija. En la población, cada uno de sus individuos constituye una solución completa del problema planteado y la

```
Crear la población inicial P de N individuos.
Calcular el fitness de todos los individuos de P
Mientras (no se haya alcanzado el criterio de terminación)
  Ordenar la población P por su valor de fitness.
  Sea P1 = el 25% mejor de P.
  Sea P2 = el 75% peor de P.
  Sea K = cantidad de individuos de P2.
  R = 0; ITERA = 0.
  Mientras (R < K) y (ITERA < MaxITERA)
    ITERA = ITERA + 1;
    Sean F1 y F2 dos padres seleccionados de P1 por el método de
    la ruleta.
    Generar 10 hijos según 5 operadores genéticos
    Calcular el fitness de los hijos.
    Sean H1 y H2 los 2 mejores hijos.
    Sean S1 y S2 los peores indiv. de P2.
    Si (fitness (H1) > fitness (F1)) o
    (fitness (H1) > fitness(F2))
      Reemplazar S1 por H1; R = R + 1.
      Si (fitness (H2) > fitness (F2))
        Reemplazar S2 por H2; R = R + 1.
  Fin Mientras
  P = P1 + P2
Fin Mientras
```

Figura 5.12: Pseudocódigo de la estrategia propuesta.

adaptación será realizada únicamente en el espacio de los pesos de la red. Con el fin de reducir el costo computacional del proceso evolutivo se realizará una fuerte presión selectiva tanto para elegir los individuos que se van a cruzar como para efectuar los reemplazos que darán lugar a la nueva población y para evitar los efectos adversos del crossover, aplicado a redes neuronales, se generarán 10 hijos a partir de cada par de padres utilizando distintos criterios. En cada generación, cada par de padres a utilizar en el proceso de reproducción es seleccionado por el método de la ruleta aplicada únicamente sobre el 25 % de los mejores individuos. A partir de ellos se generan 10 descendientes y se seleccionan los 2 mejores. Estos comparan su valor aptitud con el de sus padres y si son mejores se insertan en la población reemplazando a los 2 peores individuos. Esto se realizará hasta que toda la población (excepto el 25 % que corresponde a los padres) haya sido reemplazada o hasta que se haya realizado una determinada cantidad de iteraciones. La Figura 5.12 muestra el pseudocódigo correspondiente a este algoritmo.

Operadores genéticos estudiados

Para la generación de los 10 hijos se utilizaron cinco operadores genéticos diferentes, siendo P1 el mejor de los dos padres:

- **Mutación**

$$H_j = P_j + m_j \times V_j \tag{5.6}$$

donde:

- $j = 1, 2$.
- V_j es un vector binario con tantos elementos como pesos (arcos) haya en la red neuronal. Su valor es 0 en todas sus posiciones excepto una, seleccionada al azar, que vale 1.
- m_j es un valor pequeño que representa la magnitud de cambio.

De esta forma, cada hijo es igual al padre que lo generó salvo por una pequeña variación en uno de sus pesos. Esto lleva a que en muchos casos se conserve gran parte del comportamiento original. Considerando que los padres pertenecen al 25 % de los mejores individuos, esto es un aspecto deseable. Sin embargo, tratándose de redes neuronales, esto puede llevar a obtener resultados muy diversos.

- **Cruce (Crossover)** Los hijos H_3 y H_4 se obtienen intercambiando las neuronas de salida de los padres. Es decir que, para la arquitectura propuesta, H_3 tendrá el control del motor 1 del padre 1 y el del motor 2 del padre 2; mientras que H4 controlará su motor 1 según el padre 2 y el motor 2 según el padre 1.
  Este operador busca mantener la diversidad genética ya que, salvo que los dos padres sean muy similares entre sí, los hijos mostrarán un comportamiento nuevo.

- **Recombinación Lineal**

$$H_5 = \alpha P_1 + (1 - \alpha) P_2 \tag{5.7}$$

$$H_6 = P_1 (1 - \alpha) + \alpha P_2 \tag{5.8}$$

donde: $\alpha \in (0.,1]$
Para valores de α cercanos a 0 o a 1 los hijos tendrán una fuerte influencia de los padres; mientras que para valores cercanos a 0.5 su comportamiento será totalmente nuevo ya que se están combinando los valores de los pesos dos redes. Esto evita la reducción del espacio de búsqueda del algoritmo genético [69].

- **Recombinación Intermedia**

$$H_{7i} = a_i P_{1i} + (1 - a_i) P_{2i} \tag{5.9}$$

$$H_{7i} = a_i P_{1i} + (1 - a_i) P_{2i} \tag{5.10}$$

donde $a_i \in [-0{,}25, 1 + 0{,}25]$ y es diferente para cada uno de los pesos. El subíndice i representa cada uno de los pesos de la red. Este operador tiene como objetivo mantener la diversidad genética ampliando el espacio de búsqueda del algoritmo [69].

- **Recombinación Lineal Extendida**

$$H_{9i} = P_{1i} + \frac{s_i \times r_i \times a \times (P_{2i} - P_{1i})}{||P_1 - P_2||} \tag{5.11}$$

$$H_{10i} = P_{2i} + \frac{s_i \times r_i \times a \times (P_{1i} - P_{2i})}{||P_1 - P_2||} \tag{5.12}$$

donde:

  - $i \in$ (1,2,...,NroLinks).
  - $a_i = 2^{-ku}, k = 20$ (percepción de mutación).
  - $u \in [0,1]$ aleatorio uniforme.
  - a_i igual para todos los i.
  - $r_i = r\times$ dominio de r. Dominio de r es el rango de recombinación.
  - $s_i \in \{-1, +1\}$, aleatorio uniforme.

Cada hijo estará basado claramente en uno de los padres y contendrá una pequeña influencia del otro progenitor. Para más detalles respecto de este operador consultar [69].

Descripción de la evaluación

La evaluación de los operadores se realizó mediante la resolución del problema de evasión de obstáculos, considerando más exitosos aquellos controladores que logren recorrer la mayor parte de su entorno. Se trabajará con una estrategia de población fija de 100 individuos donde cada uno constituye una solución completa del problema planteado. En cada caso, la aptitud es medida en un mismo ambiente realizando tres intentos o épocas. En cada época la posición inicial del robot es aleatoria. El valor de aptitud del individuo se calcula como la suma de los fitness de cada uno de los 3 intentos. El cálculo del fitness cuantifica la aptitud de la red neuronal para evadir obstáculos cubriendo la mayor distancia posible dentro del entorno. La expresión utilizada para ello es la descripta en la sección 5.1.3.

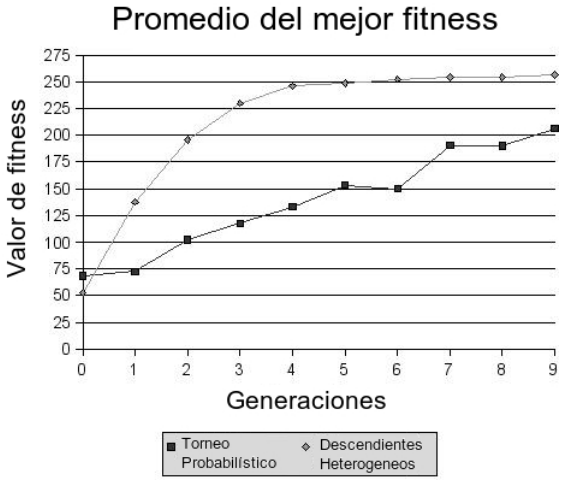

Figura 5.13: Comparación entre ambos métodos.

Resultados de la evaluación

Para poder realizar las mediciones del algoritmo descripto se realizaron algunas adaptaciones al simulador YAKS [108]. Se trata de una herramienta sumamente útil para trabajar con los robots Khepera II, está escrito en C++ y es de acceso libre. Además, no solo cuenta con un entorno simulado sino que también puede utilizarse como controlador de un robot real conectado a una PC vía RS-232. Para más información vea el apéndice B.
El algoritmo descripto en la sección 5.2.2, fue implementado en YAKS y comparado con un algoritmo genético convencional que utiliza selección por torneo probabilístico binario y mutación. La aplicación de ambas estrategias a la resolución del problema de evasión de obstáculos muestra una sensible diferencia en la evolución del fitness promedio.
La figura 5.13 ilustra el crecimiento del mejor fitness a lo largo de las 10 primeras generaciones de cada estrategia. En ambos casos, el valor indicado para cada generación representa el promedio de 10 evaluaciones realizadas de manera independiente. Puede observarse que el proceso evolutivo descripto en la sección 5.2.2 produce mejores individuos en un número de generaciones menor.

Dado el costo que representa la generación de múltiples hijos en cada operador, se hicieron distintas mediciones con el objeto de reducir el número de hijos en cada paso. Se efectuaron 10 corridas con distintas cantidades de descendientes: 4, 6 y 10 hijos. Para las corridas de los 10 descendientes se utilizaron todos los operadores de la sección anterior, para las de 6 descendientes se utilizaron los o-

Cuadro 5.1: Distintas cantidades de descendientes.

Hijos	Promedio del mejor fitness	Fitness promedio	Promedio de individuos evaluados
4	129,75	116,68	500.296
6	270,24	171,85	47.912
10	299,04	214,51	27.417

peradores de recombinación lineal, recombinación lineal extendida y mutación; y para las de 4 descendientes se utilizaron los operadores de recombinación lineal y recombinación lineal extendida. En todas las corridas, por cada uno de los operadores se obtuvieron 2 descendientes.
La tabla 5.1 muestra el promedio de 10 ejecuciones para cada una de las tres cantidades de descendientes antes mencionadas. En cada caso se incluye el promedio de los fitness de los mejores individuos, el fitness promedio de la población y la cantidad promedio de individuos evaluados. Este último dato representa el esfuerzo realizado por el método ya que la evaluación del fitness de un individuo implica la realización de tres pruebas de 500 pasos cada una.
Como puede observarse, existe una marcada diferencia entre las evaluaciones de 10 descendientes y el resto. Si bien, el valor alcanzado en el promedio de los mejores fitness entre las evaluaciones de 6 y 10 descendientes es similar, se necesita evaluar al doble de individuos duplicando el esfuerzo computacional. Por lo tanto, la opción de utilizar 10 descendientes resulta la más adecuada.

Finalmente se analizó la intervención de cada uno de los operadores genéticos dentro de la estrategia propuesta. La figura 5.14 muestra que en las primeras generaciones tanto la mutación como la recombinación lineal extendida basadas en el mejor padre, se destacan por sobre el resto; y a medida que el fitness de la población va mejorando, esta diferencia de éxito disminuye evidenciando una participación uniforme por parte de cada uno de los operadores. Este aspecto final es el que justifica los resultados de la tabla 5.1.

5.2.3. Resultados

La estrategia evolutiva empleada en la sección 5.2 ha sido utilizada para desarrollar un controlador basado en redes neuronales evolutivas que permite dotar a un robot Khepera II de la habilidad de esquivar obstáculos y alcanzar objetivos. Dicho controlador debe tener la capacidad de adaptarse a modificaciones del entorno una vez que se encuentre en su etapa de uso.

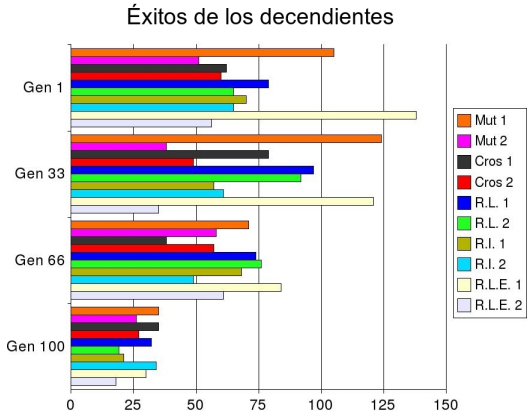

Figura 5.14: Éxitos de los descendientes.

Dado que se han utilizado únicamente los sensores de luz y proximidad del robot, los objetivos buscados estarán representados por luces ubicadas en cualquier posición dentro de su entorno de navegación. Se espera lograr que el robot se desplace libremente, sin colisionar con los obstáculos hasta acercarse a una zona iluminada donde deberá intentar permanecer.
La población inicial de controladores fue generada con la extensión del método de NEAT descripta en la sección 5.1. Se combinaron dos módulos obtenidos de manera independiente: uno que permite esquivar obstáculos y otro que permite alcanzar la luz más próxima. Cada uno de estos módulos se encuentra representado por una red neuronal recurrente obtenida a través del método de NEAT convencional descripto en la sección 4. La figura 5.1 corresponde a la arquitectura inicial. En ella puede observarse que los módulos comparten la información suministrada por los sensores de proximidad.
La red neuronal unificada combina la salida de cada uno de los módulos a través de dos neuronas nuevas que serán las encargadas de comandar realmente los motores del robot. Por lo tanto, queda formada una red con 16 entradas, 8 para cada uno de los sensores de proximidad del robot, 8 para cada uno de los sensores de luz y dos neuronas de salida, una para cada uno de los motores. La población inicial está formada por redes cuyos genomas se obtienen de realizar pequeñas modificaciones a esta estructura inicial. Como se explicó en la sección 5.1 esta población se evoluciona según el método NEAT con las restricciones impuestas a los operadores genéticos.
La figura 5.2 representa la red unificada con mejor desempeño obtenida como resultado.
Los tres mejores individuos obtenidos como resultado de esta etapa de entrenamiento inicial fueron seleccionados para constituir la primera población de

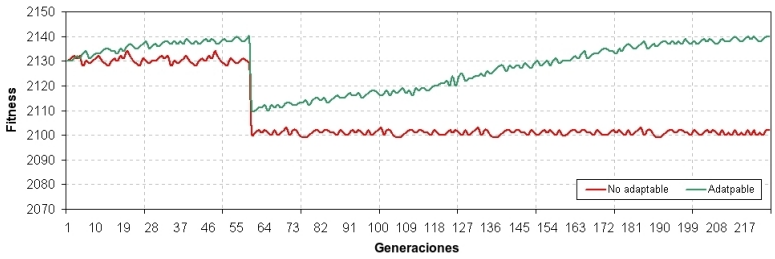

Figura 5.15: Valor de fitness promedio por generación de individuos adaptables y no adaptables.

controladores encargada de comandar al robot Khepera II y sobre quienes se aplicará la estrategia evolutiva en forma continua.
El tamaño extremadamente reducido de esta población busca minimizar el tiempo de cómputo necesario para evaluar la aptitud de cada uno de ellos al desempeñar la tarea.
Cada uno de los tres individuos es evaluado sobre el robot real en forma independiente de los otros dos. Una vez concluida la evaluación de la población, se ordena a los individuos de acuerdo a su valor de fitness y se seleccionan los 2 mejores para realizar la operación de la recombinación lineal extendida. Este operador ha sido seleccionado en base a las mediciones realizadas en la sección 5.2.2. Como resultado de esta operación se obtiene un nuevo controlador basado en el mejor padre y en dirección del segundo mejor padre. Este nuevo individuo será el reemplazante del peor individuo de la población.
La frecuencia en que este proceso deberá realizarse dependerá de la taza de cambio que tenga el entorno en el que se mueve el robot, por lo tanto, puede no ser necesario en todas las generaciones. Igualmente es recomendable aplicarlo en las primeras generaciones de la etapa del uso del controlador para permitir a los individuos adaptar el comportamiento aprendido en el simulador al mundo real.
También se realizaron distintas pruebas para analizar la capacidad de adaptación del robot ante cambios del ambiente. La figura 5.15 muestra una comparación generación a generación entre los fitness promedio de dos robots diferentes: A uno se le aplica el algoritmo evolutivo de la segunda epata (línea sólida) y al otro no (línea punteada). En la generación 60 se introduce un cambio en el entorno que dura hasta la generación 225.
En las primeras 60 generaciones, se puede observar una mejora en el fitness del robot que realiza una adaptación al entorno comparado con el que no la hace. Las pequeñas variaciones en el fitness se deben a diferencias en las condiciones con las que se evaluó al controlador. Esto justifica la variación en el fitness del robot que no hace realiza ajustes.

En la generación 60, las condiciones en el entorno cambian y el fitness de ambos controladores cae inmediatamente. Esto se debe a que ninguno de los controladores fue producido para las condiciones del nuevo entorno. Sin embargo, a medida que el algoritmo evolutivo de la segunda etapa se ejecuta sobre el robot adaptable, puede observarse una mejora en el fitness del robot. Después de 200 generaciones, el fitness de ese robot termina siendo tan bueno como era antes de los cambios del entorno. Por otro lado, el robot no adaptable no mejoró después del cambio, como era lógico.

La figura 5.16 muestra el comportamiento de 2 individuos antes de la modificación del entorno, la figura 5.17 muestra el comportamiento del mismo par de individuos después de modificar el entorno y la figura 5.18 el comportamiento de estos 2 individuos una vez que se realizó la adaptación.

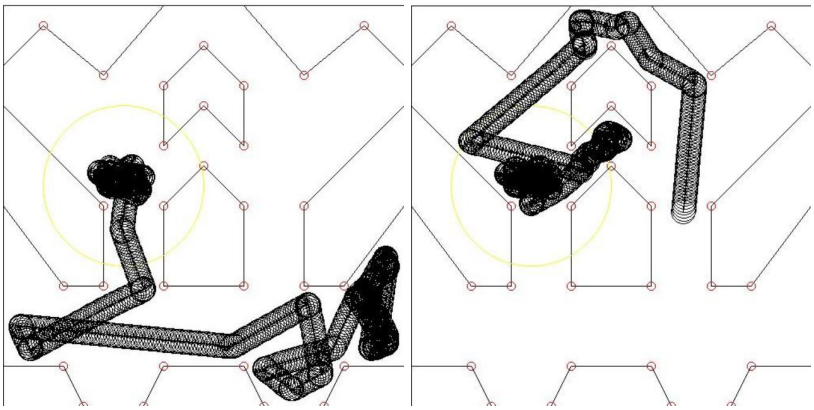

Figura 5.16: Antes de modificar el entorno.

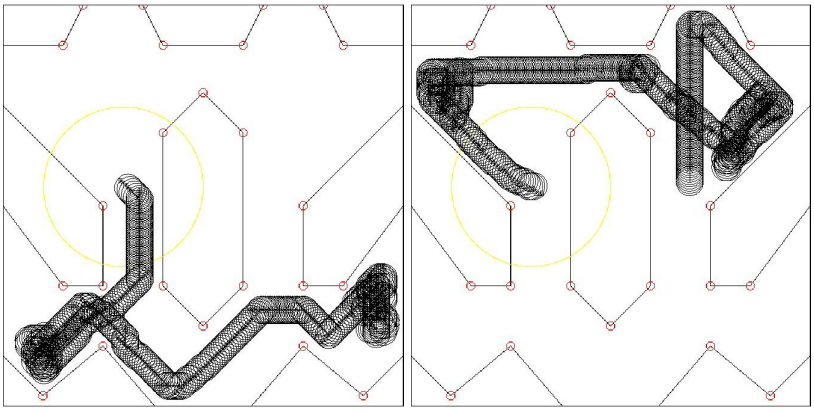

Figura 5.17: Después de modificar el entorno y sin adaptarse.

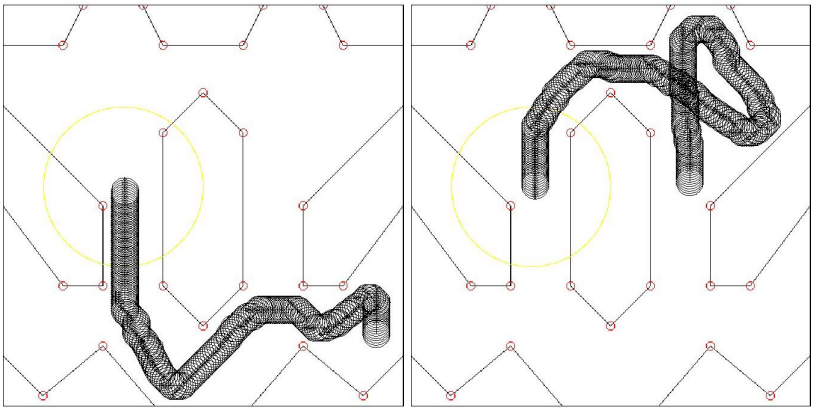

Figura 5.18: Una vez que se adaptó al nuevo entorno.

Capítulo 6

NEAT con Torneo

Como se mencionó en capítulos anteriores, NEAT es una excelente estrategia evolutiva para la obtención de controladores de robots autónomos basados en redes neuronales con una arquitectura mínima. Pero uno de los problemas centrales de este método es el tiempo de computo requerido para lograr este objetivo, por lo tanto el aporte de este capítulo es una mejora al método de NEAT en esta dirección, utilizándolo solo en las primeras etapas de la evolución con el objetivo de aprovechar su capacidad de generar controladores de tamaño mínimo para luego utilizar otra estrategia, como por ejemplo, el torneo binario que posee un menor tiempo de procesamiento.
Por otra parte, se describe una variante de la división vertical de un problema planteada en el capítulo 5 y se focaliza en la presentación de una nueva estrategia de evolución por capas, la cual permite la mejora de controladores basados en redes neuronales obtenidos a través de estrategias de evolución paralela. Su funcionamiento se basa en la combinación de NEAT con un mecanismo de selección por torneo y mutación uniforme. Durante el proceso, con la intención de reducir el tiempo de adaptación, la aptitud de los individuos es evaluada en paralelo.
El método propuesto ha sido aplicado para la generación de un controlador que le permite a un robot encontrar una pelota, posicionarse adecuadamente y golpearla en una dirección específica. Las pruebas realizadas tanto en el ambiente simulado como sobre el robot real han arrojado resultados satisfactorios.

6.1. Objetivo

Esta investigación está basada en trabajos llevados a cabo previamente en los campos de la evolución por capas [93] [105] utilizando algoritmos neuroevolutivos y proponiendo una alternativa la cual permite obtener mejoras en las soluciones propuestas.

El objetivo del capítulo 6 es presentar una nueva estrategia basada en evolución a través de la cual pueden obtenerse eficientemente los subcontroladores encargados de resolver cada una de las partes del problema. El proceso de adaptación no sólo permite alcanzar el comportamiento esperado sino que establece automáticamente la estructura mínima necesaria para cada controlador. Con el propósito de reducir el tiempo de adaptación, la evaluación de la aptitud de los individuos durante el proceso se realiza en forma paralela.

6.2. Descripción de la propuesta

La estrategia de adaptación propuesta en el capítulo 6 permite obtener un controlador formado por tantas redes neuronales recurrentes como subtareas se hayan definido. Cada red se obtiene a partir de una evolución por capas en función de la dependencia establecida entre las subtareas. El método empleado para llevar a cabo este proceso adaptativo no sólo permite alcanzar el comportamiento esperado sino que establece automáticamente la estructura mínima necesaria para cada caso.
Estudios realizados previamente [75] han demostrado que NEAT posee la capacidad suficiente para resolver este tipo de situaciones. Sin embargo, el tiempo de cálculo empleado para obtener la red neuronal adecuada para resolver cada subtarea puede resultar excesivo.
Por lo tanto, se propone realizar la evolución en dos partes o etapas; la primera es comandada por el método NEAT y la segunda por un Torneo Binario aplicado a todos los individuos de la población. El pseudocódigo 6.1 detalla este proceso.

6.3. Descripción del problema

El método propuesto en la sección 6.2 ha sido aplicado para la generación de un controlador que le permite a un robot Khepera II localizar una pelota dentro un área de juego y colocarla en la zona de anotación. El juego se desarrolla en un entorno rectangular del cual ni la pelota ni el robot pueden salir y finaliza cuando el robot consigue anotar.
La figura 6.2 muestra el entorno donde se desarrolla el juego. En ella se ejemplifican dos recorridos independientes efectuados por el robot hasta ubicarse en la posición que le permite golpear a la pelota en dirección al arco o zona de interés.

```
Sea C=[c1,c2,..,cn] la lista de subcontroladores a obtener ordenados
según sus dependencias.
Sea Oi el objetivo del subcontrolador ci con i=1:n
Para cada subcontrolador ci, con i de 1 a n.
  Generar una población inicial aleatoria.
  Evolucionar utilizando NEAT una cantidad mínima de generaciones
  evaluando en forma paralela el fitness de cada individuo.
  Mientras (no se alcance un nro máximo de generaciones) y
    (el objetivo Oi no este realizado)
    Realizar torneos entre pares de individuos seleccionados
    aleatoriamente de toda la población.
    La cantidad de pares corresponde al 45% del tamaño de la
    población.
    La nueva población quedará formada por
      El 10% de los individuos con mejor fitness de la población
      anterior (elitismo).
      Los ganadores de los torneos binarios.
      Nuevos individuos obtenidos al aplicar mutación uniforme
      a los arcos de c/u de los ganadores de los torneos.
    Evaluar en forma paralela el fitness de todos los individuos
    de la población.
  Fin Mientras
Fin Para
```

Figura 6.1: Pseudocódigo del proceso evolutivo.

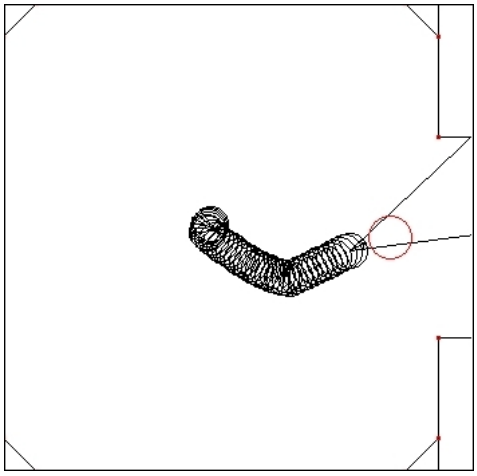

Figura 6.2: Entorno simulado.

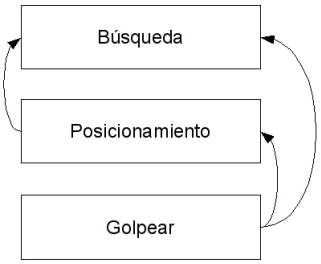

Figura 6.3: Orden de dependencia de capas.

6.3.1. Descomposición del problema en subtareas simples

Este juego puede descomponerse en tres subtareas. Cada una de ellas es llevada a cabo por una red neuronal diferente obtenida por evolución:

- Búsqueda: El objetivo de esta red neuronal es brindar al robot la capacidad de explorar el entono hasta localizar la posición de la pelota, para luego aproximarse.
- Posicionamiento: Esta red neuronal es la encargada de posicionar adecuadamente al robot. Dado que el Khepera II utilizado no cuenta con ningún soporte adicional para ”enganchar” la pelota, es de fundamental importancia que quede alineado correctamente con la pelota y la zona de anotación.
- Golpear: El objetivo de esta red neuronal es golpear a la pelota lo más fuerte posible, de manera tal de introducirla dentro del área de anotación.

6.4. Aprendizaje por capas

Una vez que se lleva a cabo la división de tareas, se establece un orden de dependencias entre ellas, las cuales indican la secuencia de entrenamiento. La figura 6.3 muestra estas dependencias para el problema propuesto.
Cada rectángulo representa una subtarea y las flechas indican las dependencias entre ellas. Una subtarea podrá realizar su aprendizaje una vez que el resto de las subtareas de las cuales dependa hayan finalizado su aprendizaje.

Esto se conoce como aprendizaje por capas, basado en las dependencias existentes en el orden de aprendizaje de las diferentes subtareas. Desde otro punto de vista, podría ser considerada como una estructura inicial que tengan una capa formada por las subtareas que no necesitan de otras para realizar el aprendizaje. Luego, en la capa siguiente, aquellas tareas que pueden realizar su aprendizaje a partir de las anteriores se lleva a cabo, y así sucesivamente.

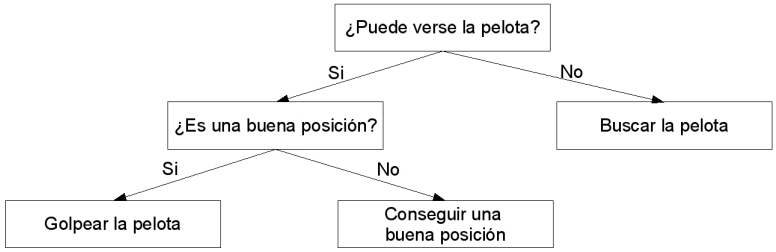

Figura 6.4: Árbol de decisión.

Observe que este aprendizaje no muestra la forma de resolver el problema completo, pero si la forma de realizar el aprendizaje para llevar a cabo cada una de las subtareas.

6.5. Resolución del problema

Una vez que se obtuvieron las redes, un árbol de decisión es el encargado de seleccionar la red neuronal que puede ser usada en cada instante. De esta forma, se obtiene un controlador simple basado en controladores específicos para cada subtarea. La Figura 6.4 muestra el árbol de decisión usado para la resolución de este problema.

6.6. Aspectos de implementación

El robot Khepera utilizado en todas las pruebas sólo posee una cámara de visión K213 con capacidad para distinguir una línea de 64 píxeles correspondientes a los tonos de grises ubicados dentro de su visión angular. Por tal motivo, las paredes del entorno rectangular cerrado utilizado fueron pintadas de negro, la pelota de color blanco y la zona de anotación de color gris. Las colisiones son detectadas a través de los sensores de proximidad.
El controlador de cada subtarea es comandado por una red neuronal compuesta por 72 neuronas lineales de entrada, dos neuronas no lineales de salida, y una neurona adicional de bias la cual se puede conectar a cualquier otra neurona, excepto a las neuronas de salida. Las entradas de la red están escaladas linealmente en el rango [0..1] de los valores tomados por los sensores donde los primeros 8 valores corresponden a los sensores de proximidad y los 64 restantes corresponden a la cámara k213. Las salidas de la red están escaladas entre [-1..1] para controlar la velocidad de los motores de cada una de las ruedas del robot para coincidir con los requerimientos del simulador. La arquitectura inicialmente se gráfica en la figura

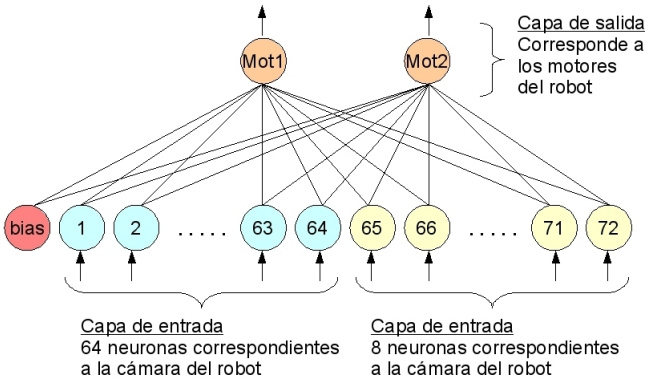

Figura 6.5: Red inicial, previa al proceso de evolución.

6.5 y la arquitectura final a utilizar queda determinada luego de haber aplicado NEAT.

Para determinar el fitness de un individuo se mide el desempeño de su controlador comenzando desde 4 posiciones distintas como lo gráfica la figura 6.6. Dado que las pruebas han sido realizadas en un área rectangular, cada intento comienza con el robot en un extremo diferente. En cada caso, la posición de la pelota también varía.

Finalmente, el valor de fitness de un individuo está dado por el promedio de los resultados de las cuatro pruebas a las que fue sometido. El pseudocódigo 6.7 muestra el algoritmo utilizado.
A continuación se detalla la forma en que fueron calculados los fitness correspondientes.

6.6.1. Módulo de búsqueda

Para medir el comportamiento en cada prueba se utiliza la siguiente función de evaluación:

$$Eval_{bus} = \sum_{t=1}^{500} \left[\left(M_{izq} + M_{der} \right) \times (1 - S_{ir}) \times \sum_{i=1}^{64} (camara_i \times vector_i) \right] \qquad (6.1)$$

donde:

- $camara_i$ es la posición i en el arreglo de valores correspondientes a la cámara k213. Se trata de un valor del intervalo [0,1] correspondiente a la escala de grises donde 0 representa negro y 1 blanco.

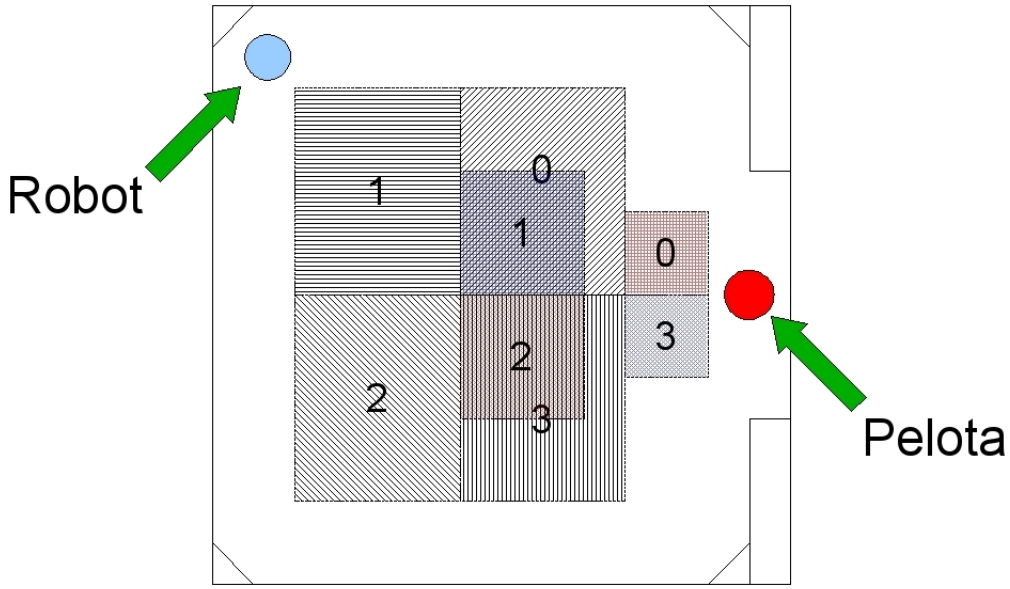

Figura 6.6: Las cuatro posiciones iniciales de evolución.

```
para cada individuo de la población
  para i = 1 : 4
    Ubicar al individuo en la posición i
    Realizar 500 iteraciones con el controlador actual.
    Calcular la aptitud del individuo en esta etapa
    representado por Eval como la suma de los fitness
    de c/generación. Si el robot colisiona, se
    interrumpe su recorrido y se retorna el valor de
    Eval con lo acumulado hasta este punto.
  fin para
  Calcular el fitness del individuo como el promedio de
  las 4 evaluaciones anteriores.
fin para
```

Figura 6.7: Pseudocódigo del proceso de evaluación del individuo.

- $vector_i$ es la posición i en el arreglo de valores escalares con distribución normal. Este vector busca escalar la información de la cámara dándole mayor importancia a los píxeles de la zona del centro.
- M_{izq} y M_{der} son valores en el intervalo [-1..1] correspondiente a la velocidad de los motores izquierdo y derecho del robot, respectivamente. Estas son las salidas de la red neuronal.
- S_{ir} es el máximo valor de los sensores de proximidad en el intervalo [0..1].

El término $(camara_i \times vector_i)$ toma su máximo valor cuando el robot esté lo más cerca posible de la pelota y más al centro del ángulo de visión de la cámara se encuentre la misma. El término $\left(M_{izq} + M_{der}\right)$ lleva al controlador a maximizar su movimiento al obtener el valor más alto cuando el robot se dirige hacia adelante a máxima velocidad. Finalmente, el termino $(1 - S_{ir})$ fuerza al robot a que se aleje de los obstáculos para aumentar su puntuación.

6.6.2. Módulo de posicionamiento

Para poder medir el comportamiento del controlador en cada prueba, se utilizó la siguiente función de evaluación:

$$Eval_{pos} = zonas \times \sum_{t=1}^{500} \left(\left[\left(M_{izq} + M_{der} \right) \times \left(1 - |M_{izq} - M_{der}| \right) \times (1 - S_{ir}) \right] + D_{pel} + D_{ar} \right) \tag{6.2}$$

donde:

- M_{izq} y M_{der} son valores en el intervalo [-1..1] correspondientes a la velocidad de los motores izquierdo y derecho respectivamente.
- S_{ir} es el máximo valor de los sensores de proximidad en el intervalo [0..1].
- $zonas$ es un valor proporcional a la cantidad de sectores recorridos por el agente durante el entrenamiento.
- D_{pel} es un valor en el intervalo [0..1] que indica la distancia a la pelota.
- D_{ar} es un valor en el intervalo [0..1] que indica la distancia al área de anotación.

El termino $\left(1 - |M_{izq} - M_{der}|\right)$ hace referencia a la rotación del robot. Por ejemplo, si el robot se encuentra girando sobre si mismo la velocidad en ambos motores es la misma pero con signo contrario, alcanzando su mayor valor cuando la velocidad sea máxima. Por lo tanto, el agente deberá minimizar este efecto para incrementar

su fitness.
Con el objetivo de obtener individuos capaces de cubrir grandes distancias, se dividió el entorno en un reticulado de 100x100 sectores iguales y se utilizó el coeficiente *zonas* para medir la cantidad de sectores por los que el agente pasó a lo largo de una prueba.

$$zonas = \frac{\sum_{x=1}^{100}\sum_{y=1}^{100} zona_{xy}}{100 \times 100} \tag{6.3}$$

$$zona_{xy} = \begin{cases} 1 & \text{Si el agente paso por el sector(xy)} \\ -1 & \text{en caso contrario} \end{cases} \tag{6.4}$$

En resumen, en la formula 6.2 se pondera el recorrido del robot durante 500 pasos (sumatoria) en forma proporcional a la cantidad de sectores recorridos.

6.6.3. Módulo para golpear la pelota

$$Eval_{golp} = \sum_{t=1}^{500}\left[\left(M_{izq} + M_{der}\right) \times \left(1 - |M_{izq} - M_{der}|\right) \times (1 - S_{ir}) \times \sum_{i=1}^{64} (cam_i \times vec_i)\right] \tag{6.5}$$

- cam_i es la posición i en el arreglo de valores correspondientes a la cámara k213.
- vec_i es la posición i en el arreglo de valores escalares con distribución normal.
- M_{izq} y M_{der} son valores en el intervalo [-1.. 1] correspondiente a la velocidad de los motores izquierdo y derecho del robot, respectivamente. Estas son las salidas de la red neuronal.
- S_{ir} es el máximo valor de los sensores de proximidad en el intervalo [0..1].

6.7. Resultados

Para determinar la eficiencia y eficacia del método propuesto, se han considerado las siguientes alternativas:

1. **Subcontroladores basados en redes neuronales feedforward**: En cada caso, la estructura de la red neuronal utilizada fue la arquitectura feedforward más eficiente que pudo definirse manualmente. El entrenamiento se llevó a cabo mediante un torneo binario.

2. **Subcontroladores obtenidos únicamente utilizando NEAT**: Esta forma de establecer los subcontroladores no requiere de ningún conocimiento previo de la arquitectura de la red neuronal ya que posee la capacidad de establecerlo durante la adaptación.

3. **Subcontroladores obtenidos combinando NEAT y selección por torneo**: Esta alternativa es la propuesta en la sección 6 y corresponde a lo expuesto anteriormente.

Para cada método se efectuaron 30 ejecuciones independientes. Para adaptar cada una de las tres redes neuronales que componen un mismo controlador se utilizaron poblaciones de 100 individuos y se las evolucionó durante 100 generaciones. Por lo tanto, estas 30 corridas dieron como resultado 100 individuos por cada método evolutivo. De cada una de estas poblaciones, se seleccionaron los 3 mejores en cuanto a su fitness, uno de cada método evolutivo. Con estos tres controladores se realizaron nuevamente 40 pruebas contabilizando la cantidad de veces que el robot logró anotar. La figura 6.9 muestra el porcentaje de éxitos promediando las 40 corridas antes mencionadas.
Una vez finalizada la tercer etapa, se toman los 100 controladores de la última generación de cada método evolutivo. A cada controlador se le dieron 4 intentos de convertir el gol, por lo tanto, por cada método evolutivo se efectuaron 400 intentos de convertir un gol. La figura 6.8 muestra la cantidad de goles convertidos por la población en la ultima generación.
En cada caso, se realizaron 30 corridas independientes del proceso necesario para obtener el controlador completo. Esto implica que en cada corrida se generaron los subcontroladores correspondientes utilizando 100 generaciones para cada uno. Con el objetivo de medir la capacidad de cada método para generar el controlador buscado, se seleccionó entre las 30 pruebas, para cada uno de ellos, el individuo que mejor resuelve la subtarea de golpear la pelota. La elección de estos 3 individuos permite definir el controlador completo ya que los subcontroladores restantes son los utilizados durante los entrenamientos.
Con estos tres controladores se realizaron nuevamente 40 pruebas contabilizando la cantidad de veces que el robot logró anotar. La figura 6.9 nuestra el porcentaje de éxitos promediando las 40 corridas antes mencionadas.
Si se analiza el desempeño de la población de cada controlador en lo que se refiere a la cantidad de goles realizados, puede verse que el desempeño de NEAT + Torneo es muy superior. La figura 6.8 muestra la cantidad de éxitos de toda la

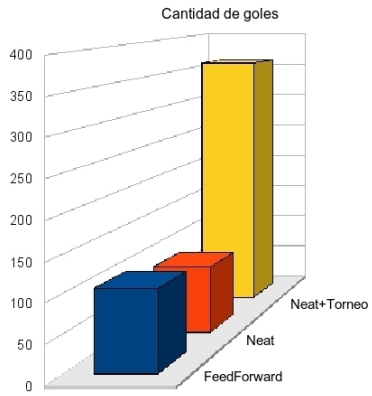

Figura 6.8: Número de goles en la última generación.

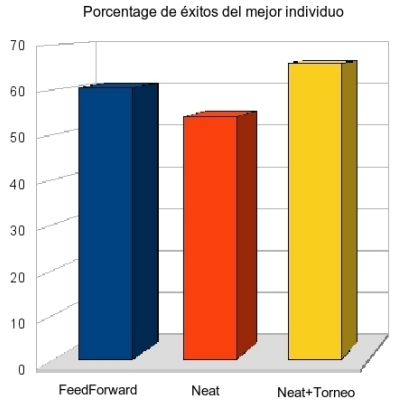

Figura 6.9: Promedio de éxitos del mejor individuo.

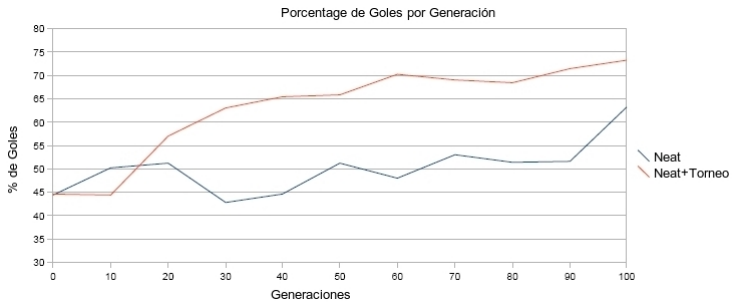

Figura 6.10: Promedio de goles por generación.

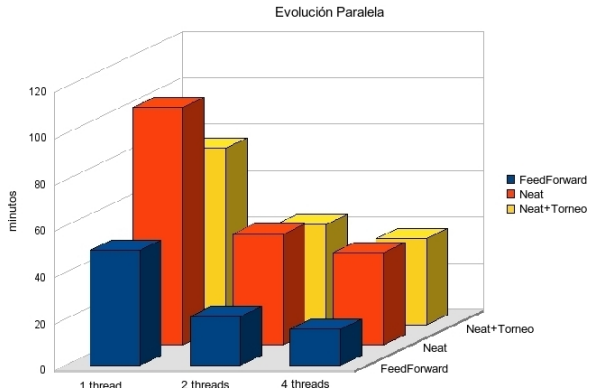

Figura 6.11: Promedio de tiempo de evolución.

población en la última generación.
Con el fin de medir las mejoras introducidas por el método propuesto en este trabajo, el comportamiento de los mejores controladores obtenidos con NEAT y NEAT + torneo, se analizo cada 10 generaciones del proceso de evolución, su utilización para golpear la bola 100 veces consecutivas, tratando de introducir en el área de anotación.

La figura 6.10 muestra los valores medios correspondientes a los goles convertidos por los controladores en las generaciones indicadas anteriormente durante 30 corridas independientes.

En cuanto a tiempo de cálculo necesario para aplicar cada uno de los métodos puede decirse que el valor más significativo se encuentra en la evaluación del fitness de cada controlador a lo largo de las sucesivas generaciones ya que cada ejecución implica estimar el desempeño de 30000 redes neuronales.
La figura 6.11 muestra el tiempo de evolución promedio de los distintos métodos con diferentes grados de paralelismo. En ella, puede observarse que la paraleliza-

ción del algoritmo reduce el tiempo de ejecución en promedio a la mitad. Además, se observa el bajo costo computacional de las redes feedforward; esto se debe a que, a diferencia de las estrategias basadas en NEAT, la arquitectura es definida a priori y no sufre ninguna modificación durante el proceso. Con respecto a NEAT y la estrategia propuesta en la sección 6, puede verse que en las versiones paralelas emparejan sus tiempos de evolución, aunque el método propuesto en este artículo siempre mantiene una ventaja sobre NEAT estándar. Los tiempos indicados en la figura corresponden a la ejecución del algoritmo en máquinas de 3 GHz Pentium 4 HT.

Capítulo 7

Conclusión y Trabajos Futuros

7.1. Conclusión

Las estrategias evolutivas si bien han demostrado ser capaces de ofrecer excelentes resultados en diversas áreas, poseen un par de características desfavorables. En primer lugar, el proceso de adaptación suele ser lento y costoso en tiempo de ejecución y en segundo lugar, resulta difícil reutilizar el conocimiento adquirido.
En el sección 5.1 se vio una estrategia que busca resolver ambos problemas proponiendo la combinación de módulos simples, basados en redes neuronales recurrentes, generados independientemente del problema a resolver. Finalmente, la comunicación entre ellos se establece por evolución dando lugar a una única red neuronal que representa a la solución deseada.
Los resultados obtenidos en la resolución del problema de evasión de obstáculos y alcance de luz utilizando un robot Khepera II han demostrado que la estrategia propuesta es más eficiente.
El motivo de esta mejora puede atribuirse al hecho de incorporar soluciones parciales en lugar de no contar con algún conocimiento previo.

Por otro lado, la obtención de controladores utilizando herramientas adaptables al entorno resulta, en general, una tarea computacional costosa. Por lo tanto, suele separarse abruptamente la etapa de generación de las soluciones de la etapa de su uso. En esta dirección, en la sección 5.2, se utilizó una estrategia que reduce el tiempo de entrenamiento inicial, aprovechando el conocimiento almacenado en módulos con comportamientos simples previamente generados y permite al robot seguir adaptando su respuesta a futuros cambios del entorno. En otras palabras, se evoluciona un controlador a lo largo de su vida útil combinando un método basado en evolución de módulos neuronales con un algoritmo evolutivo específico.

Los resultados obtenidos en la resolución del problema de evasión de obstáculos y alcance de objetivos utilizando un robot Khepera II han demostrado que esta estrategia permite adaptar el comportamiento del robot a los cambios del entorno.

En el capítulo 6 se ha presentado una nueva estrategia que permite mejorar el desempeño de controladores obtenidos utilizando evolución por capas logrando reducir considerablemente el tiempo de cálculo sin afectar sensiblemente la calidad del controlador final. Su funcionamiento se basa la combinación de un método capaz de generar una red neuronal de estructura mínima con un algoritmo genético que utiliza selección por torneo y mutación uniforme.
Su aplicación en la resolución de un problema concreto ha sido testeada tanto en el ambiente simulado como sobre el robot real con resultados muy satisfactorios. Distintas experiencias realizadas con NEAT han permitido establecer que con el 20 % de las generaciones máximas es suficiente para tener una población con un desempeño básico sobre la cual es factible aplicar torneo, optimizando de esta forma el tiempo de ejecución.

A lo largo de este trabajo se han presentado distintas estrategias evolutivas que permiten obtener un controlador de un robot autónomo, así como diferentes maneras de descomponer un problema de partes reduciendo la complejidad de las soluciones y el tiempo en el que se obtienen los controladores. Estas estrategias, basadas en redes neuronales y algoritmos evolutivos, han demostrado ser una buena alternativa a la hora resolver problemas de control de robots, ya que la comparación con otras estrategias evolutivas convencionales, tanto en ambientes de simulación como en ambientes reales, han arrojado resultados satisfactorios.

7.2. Trabajos futuros

Aun queda pendiente establecer un mecanismo adecuado para la combinación de las funciones de fitness de cada uno de los módulos. En la sección 5.1 se tuvo que recurrir a un escalamiento de los valores de una de las funciones para que se encontraran en un intervalo comparable, evitando que los valores de una de las funciones opaquen a los de la otra. Además, la elección de los coeficientes tuvo que hacerse manualmente mediante prueba y error. En futuros trabajos, se medirá el desempeño del método en problemas de mayor complejidad que puedan descomponerse en un número mayor de módulos. Para simplificar la definición de la función de fitness para la red unificada, se estudiará la posibilidad de incorporar estrategias de evolución multiobjetivo. Esto permitiría obtener un conjunto de redes neuronales que se aproximen a la frontera de Pareto, de modo que todas representen variaciones del comportamiento pedido, permitiendo optar por la más

apropiada.
También se está trabajando en la posibilidad de combinar la estrategia propuesta en el capítulo 6 con lo visto en la sección 5.2, o sea, tomar los tres mejores controladores obtenidos mediante la estrategia evolutiva de NEAT con Torneo e instalar en el robot una minipoblación con estos tres controladores y que la misma evolucione a lo largo de su vida útil [98].
En estos momentos se está trabajando en una adaptación de la estrategia evolutiva presentada en el capítulo 6 para realizar el proceso evolutivo con un grado mayor de paralelismo, a fin de reducir aún más el tiempo de evolución de los controladores. Para ello se están evaluando distintas técnicas de procesamiento distribuido, como clustering o el modelo de computación de ciclos redundantes, así como distintas estrategias para la paralelización de metaheurísticas.

Appendices

Apéndice A

Robots Khepera II

El robot Khepera II fue originalmente diseñado como una herramienta de investigación y enseñanza en el framework del Programa Suizo de Investigación Prioritaria (Swiss Research Priority Program). El cual fue desarrollado por primera vez en 1992, por un equipo de investigación del Microprocessor and Interface Laboratory (LAMI) en el Swiss Federal Institute of Technology Lausanne (EPFL). Permite la confrontación del mundo real con los algoritmos desarrollados en simulaciones para la ejecución de trayectorias, evasión de obstáculos, preprocesamiento de la información de los sensores, hipótesis sobre procesamiento de comportamiento, etc. El robot Khepera es usado actualmente alrededor del mundo como una plataforma de variados experimentos y aplicaciones en robótica. El robot Khepera II a extendido sus capacidades, es completamente compatible con el original y puede utilizar cualquiera de los accesorios del Khepera. Es fácil de usar, robusto y con una plataforma estándar para varias aplicaciones robóticas [48].

A.1. Programación

Para facilitar la programación del robot, LabVIEW a propuesto un entorno de desarrollo, basado en programación gráfica, básicamente dedicado a la instrumentación, lo que permite un rápido desarrollo de interfaces de entrada-salida.
Algunos otros productos conocidos que soportan la comunicación con el robot Khepera son:

- Matlab de *MathWorks*
- Sysquake de *Calerga*
- Webots de *Cyberbotics*
- YAKS freeware [108]

Figura A.1: Robot Khepera II junto a una moneda de 2 euros.

El código también puede cargarse en la memoria del Khepera para una ejecución independiente. Los programas, ya sean escritos en el lenguaje C o en el lenguaje ensamblador para M68000, puede ser compilado en varios entornos usando un compilador cruzado y luego cargarlo en la RAM o en la memoria FLASH no volátil. Está disponible una completa API para programas que interactúan con el harward del robot, tanto para C o el lenguaje ensamblador [48].

A.2. Características

El cuerpo del robot tiene un diámetro de 70 mm, una altura de 30 mm y un peso aproximado de 80 g. La figura A.1 compara al robot junto a una moneda.
Para la ejecución de los programas, el robot cuenta con un hardware que se encarga de controlar al robot, cuenta con un procesador Motorola 68331 25MHz, 512 KB de memoria RAM, 512 KB de memoria FLASH para almacenar los programas (memoria no volátil) y una interfaz serie RS232 para conectar al Khepera con una computadora. Este puerto serie puede utilizarse para controlar al robot desde la computadora o para cargarle un programa compilado. La figura A.2 muestra el resultado de una inspección externa del robot [48].

A.2.1. Sensores

El robot Khepera II consta de 8 sensores que están ubicados como lo indica la figura A.3 y cada uno de ellos puede ser utilizado de 2 maneras distintas: como sensor de proximidad de objetos y como sensor de luminosidad ambiental. En el primer caso, se utiliza a los sensores como emisores y receptores de pulsos infrarrojos y en el segundo, solo la función de recepción. En ambos casos los valores retornados se encuentran entre 0 y 1023 [48].

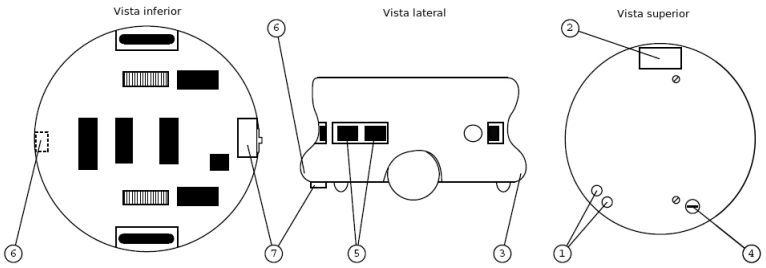

Figura A.2: Revisión del robot Khepera.

1. LEDs.
2. Conector de linea serie.
3. Botón de Reset.
4. Selector del modo de ejecución.
5. Sensores de proximidad infrarrojos.
6. Conector del cargador de la batería.
7. Interruptor de encendido-apagado.

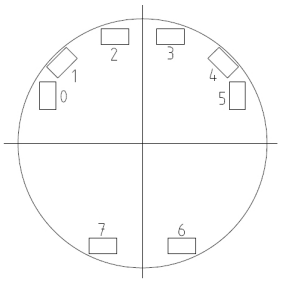

Figura A.3: Posición de los 8 sensores.

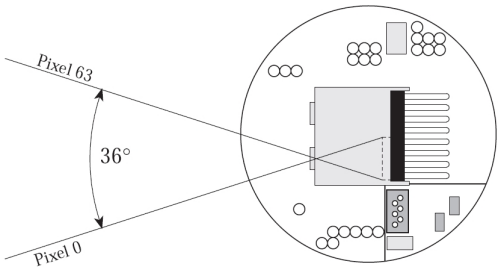

Figura A.4: Ángulo de visión de la cámara.

Sensores de proximidad

Los sensores de proximidad entregan un valor más alto cuanto más cerca se encuentre de un objeto. El rango de detección está entre 0 y 10 cm., dependiendo de la capacidad de reflexión de rayos infrarrojos que tenga el objeto en cuestión [48].

Sensores de luminosidad

Los sensores de luz retornarán un valor más bajo cuanto mayor sea la intensidad lumínica. Es decir que 1023 representa la oscuridad total. El rango de detección depende de la intensidad de la fuente. Por ejemplo, una fuente de 50W a 40 cm produce valores significativos en los sensores [48].

Cámara K213

Además de los sensores de proximidad y de luz, el robot cuenta con un sensor llamado Torreta de visión K213, que es una cámara muy básica que consta de una resolución de 64x1 píxeles, en escala de 256 tonos de grises, con un ángulo de 36 grados y con un rango de distancia entre 5 y 50 cm como se ilustra en la figura A.4 [48].

A.2.2. Motores

El Khepera cuenta con 2 motores de corriente continua, donde cada motor controla una de las 2 ruedas con una caja de cambios con una reducción de 25:1. Un codificador incremental situado en eje del motor, otorga 24 pulsos por cada revolución del motor. Esto permite una resolución de 600 pulsos por cada revolución de la rueda, que corresponde a 12 pulsos por milimetro de la ruta del robot.
El procesador principal tiene un control directo sobre la alimentación del motor y puede leer los pulsos del codificador incremental. Una rutina detecta cada pulso

del codificador incremental y actualiza un contador de la posición de la rueda. Los motores tienen un rango de valores enteros entre -128 y 127, donde los valores negativos hacen que el robot retroceda, los positivos que avance y el 0 que el motor del robot se detenga. Cuanto más alto sea el módulo del valor aplicado a un motor, más alta será la velocidad. Cada unidad de los motores representa una velocidad de 8 milímetros por segundo[48].

Apéndice B

YAKS (Yet Another Khepera Simulator)

YAKS (Yet Another Khepera Simulator) [108] es un sistema de simulación de un ambiente donde se mueve un robot Khepera II, realizado por la Universidad de Skövde, Suecia [96]. Está realizado en C++, es de código abierto, que no solo cuenta con las herramientas necesarias para simular un entorno real, si no que también posee los algoritmos necesarios para desarrollar controladores basados en redes neuronales y algoritmos genéticos.
En el ambiente simulado, se pueden agregar distinta clase de objetos para que el robot interactúe. Los objetos pueden ser paredes, obstáculos circulares y fuentes de luz. Para poder simular el proceso de reconocimiento de los objetos o de luz, así como el movimiento de los motores, el simulador cuenta con tablas de muestras, donde cada una sirve para reconocer un objeto en particular. Estas tablas se generaron a partir de mediciones hechas sobre un robot real, interpolando los valores faltantes, lo que le da al simulador mayor realismo. Incluso, permite agregar ruido a los sensores.

B.1. Características principales

La idea de YAKS es controlar el comportamiento de los robots a través de redes neuronales, pero dichas redes no se entrenan sino que son evolucionadas mediante algoritmos genéticos. Para realizarlo, el simulador provee un conjunto de clases para construir y manipular redes neuronales, así como para trabajar con algoritmos genéticos.
Es muy útil contar con un simulador durante el desarrollo de los controladores de los robots, para probar el aprendizaje en un ambiente controlado y seguro antes de

```
Leer los parámetros del archivo de configuración
Crea el ambiente especificado en los parámetros
Generar un población inicial.
Mientras no satisfaga la condición de fin.
  Para cada robot 1..#robot
    Activar las redes con las entradas deseadas.
    Calcular el fitness de cada red (individuo).
  Seleccionar las redes que formaran las siguiente generación
  en base al fitness.
  Generar la siguiente generación.
```

Figura B.1: Pseudocódigo del algoritmo evolutivo del simulador.

llevar al robot a un ambiente real.
El robot del simulador, al igual que el robot real, tiene 8 sensores de proximidad cuyos valores están normalizados al rango [0..1]. También posee 8 sensores de luz con valores normalizados entre [0..1], donde la oscuridad es 0. Por seguridad, no se permite que los motores se desplacen a una velocidad mayor que 10. Los valores para los motores se encuentran normalizados al rango [0..1], donde 0 significa que retrocede a velocidad 10, 1 significa que avanza a velocidad 10 y 0.5 hace que el motor se pare.
YAKS permite crear y manipular redes neuronales de varias maneras. Una opción es construir una red con alguna de las arquitecturas predefinidas, otra posibilidad es armar la red manualmente, especificando cada neurona que la compone, y la tercera opción es leer la configuración de la red neuronal desde un archivo.
También posee una implementación para algoritmos genéticos que esta adaptada al control de robots mediante redes neuronales. Esto implica, entre otras cosas, que de los tres operadores básicos (reproducción, cruza y mutación) solo están presentes en YAKS la reproducción y la mutación. Estos operadores trabajan solamente sobre los pesos de las redes, es decir, no es posible evolucionar la arquitectura de las redes. Sin embargo, en algunos casos es posible que la población puede estar formada por redes con arquitecturas diferentes, lo que permite al menos, elegir la mejor de las arquitecturas presentes.
Evidentemente, otra limitación es que los individuos a evolucionar siempre son redes, no se puede evolucionar individuos ”genéricos”.
El algoritmo evolutivo básico para evolucionar redes neuronales se describe el el pseudocódigo B.1.

B.2. Configuración

Existe un archivo de configuración *.opt* en el cual se especifican ubicado en el directorio raíz de la aplicación en el cual se especifican los parámetros necesarios para la ejecución del simulador. Una ventaja importante del archivo de configuración es que no necesita recompilación. Entre los parámetros que se pueden configuran se destacan la cantidad de robots intervinientes en la simulación, la cantidad y tipo de sensores para cada robot, el archivo en el que describe el ambiente en donde se mueve el robot y la ubicación de archivos externos necesarios. También pueden especificarse otros parámetros relacionados con el proceso de evolución como el número de iteraciones que tendrá el algoritmo genético, la cantidad de épocas por cada generación, la cantidad de veces que el robot se mueve en cada época, la cantidad de individuos padres que permanecerán en la próxima generación, la cantidad de descendientes que se obtendrán por cada padre, el método por el cual se obtiene la nueva generación, la cantidad de individuos del algoritmo genético, etc.
En el archivo que describe el ambiente simulado se definen las coordenadas en las que se ubican las paredes, las coordenadas y el radio de los obstáculos y las zonas, y las posiciones de las fuentes de luz.

B.3. Clases principales

- **SIM.CPP**: Esta es la clase que contiene el main. Permite testear el código ya sea en el robot real o en el simulador.

- **GA.CPP**: Define los métodos referentes al algoritmo genético, entre los cuales se encuentran los distintos métodos de selección:

    - Selección por torneo con mutación: Crea una nueva generación usando selección por torneo. Los nuevos individuos son mutados. Note que cada individuo puede ser comparado con si mismo. Este método es seguro de usar con un algoritmo genético que usa diferentes arquitecturas de redes neuronales.
    - Selección por torneo con mutación y elitismo: Crea una nueva generación usando selección por torneo y elitismo. Los nuevos individuos son mutados. Note que cada individuo puede ser comparado con si mismo. Este método es seguro de usar con un algoritmo genético que usa diferentes arquitecturas de redes neuronales.
    - Nueva generación con mutación en los pesos: Crea una nueva generación con mutación en los pesos. Los individuos deben tener la misma

arquitectura en la red neuronal. La generación nueva es padres x descendientes.

- Nueva generación con mutación: Crea una nueva generación con mutación en los pesos, guardando el mejor individuo. Los individuos deben tener la misma arquitectura en la red neuronal. La generación nueva es padres x descendientes.

- **ANN.CPP**: Define los métodos referentes a las redes neuronales. Dentro de los métodos más importantes están guardar y cargar todos los pesos de la red en un archivo, imprimir la arquitectura de la red con sus pesos y valores de activación, mutar todos los pesos de los enlaces de todas las neuronas de la red, duplicar la red neuronal y comparar 2 redes, agregar y borrar una neurona o un enlace, ingresar y obtener los valores de entrada y de salida de la red, conectar 2 capas de acuerdo al tipo de red (los tipos de redes se describen en BUILDN.CPP). También se encuentran los métodos referentes al entrenamiento usando Back Propagation.

- **BUILDN.CPP**: Contiene las estructuras e implementación de las distintas redes neuronales. Esta es la clase que se debe modificar para agregar una nueva arquitectura de redes neuronales. Las redes provistas en el simulador son redes Feedforward, Elman, EcoState y Modular, entre otras.

B.4. Desarrollos específicos

Para poder implementar y evaluar distintas técnicas evolutivas es que se decidió introducir modificaciones al simulador de acuerdo a las siguientes necesidades:

- La primera necesidad surge desde el primer intento de tratar de entender el funcionamiento interno de YAKS, de quien no se posee ninguna documentación. Si bien el código es bastante claro, con comentarios, está bien modularizado y con un correcto uso del paradigma de programación orientado a objetos, la función principal carece de modularización y excede considerablemente la extensión en lineas de código recomendables (más de 600 líneas).

- La segunda necesidad surge al momento de tratar de ampliar la potenciabilidad del simulador, en particular la capacidad de desarrollar controladores basados en redes neuronales donde la arquitectura de las mismas no sea establecida desde antemano, si no que esta evolucione a lo largo de las distintas generaciones. Además se desea añadir al simulador el operador básico

faltante (operador de cruzamiento), donde este operador intenta generar individuos a partir de una combinación de dos individuos, llamados padres. Existen diferentes estrategias para combinar 2 individuos de manera que las características más deseables permanezcan en los individuos resultantes, llamados hijos.

- La tercera razón para realizar un proceso de reingeniería parte de la necesidad de reducir el tiempo de procesamiento de los individuos, ya que es considerable.

B.4.1. Metodología

Análisis del sistema

En una primera etapa se realizó un análisis integral del simulador, así como una prueba del código fuente, dando como resultado un diagrama de clases del simulador y una lista de las modificaciones a realizarse. La lista de modificaciones incluye limpieza de código no necesario, modularización de la función main, detección y corrección de errores, y preparación del simulador para la incorporación del nuevo código.
Para añadir las funcionalidades necesarias se resolvió utilizar una estrategia de evolución de redes neuronales y algoritmos evolutivos desarrollada por Kenneth O. Stanley [90] y Risto Miikkulainen en la Universidad Central de Florida, Facultad de Ingeniería Electrónica y Ciencias de la Computación, en Estados Unidos. Dicha estrategia se conoce como NEAT (NeuroEvolution of Augmenting Topologies) [91] y se describe en el capítulo 4. Se cuenta con un código fuente escrito por los mismos autores de esta estrategia que no solo está disponible en C++, sino que también puede encontrarse en Java, C#, Matlab, y en varios lenguajes más [71].
También se realizó un análisis sobre el código de la nueva estrategia, donde se resolvió el modo en que se hará su integración al sistema (resolución de espacios de nombres, definición de variables y nuevas funciones, etc).
Para reducir el tiempo de procesamiento de los individuos, se planteó una estrategia para paralelizar el algoritmo de evaluación de los mismos, ya que la evaluación de cada uno de los individuos de la población son totalmente independientes entre si. Se realizó un análisis de viabilidad sobre la paralelización y se concluyo que es posible realizar dicha modificación.

Análisis Funcional

Se realizó un análisis sobre la funcionalidad del sistema de simulación, y a cada una de la funciones se le asignó un coeficiente de complejidad de desarrollo estimativo de 1 a 5, donde 5 es el más complejo y 1 el más simple. Las funciones

identificadas son las siguientes:

1. Definición de redes neuronales de arquitectura fija. (2)

2. Creación de una población de redes neuronales de arquitectura fija. (3)

3. Creación de un entorno simulado. (5)

4. Creación de un robot simulado. (4)

5. Creación de una pelota simulada. (3)

6. Representación visual del entorno simulado mediante la interfaz gráfica. (5)

7. Interacción de todos los objetos en el entorno simulado. (4)

8. Obtención de datos de un entorno simulado. (2)

9. Obtención de datos de un robot simulado. (2)

10. Procesamiento de datos a través de una red neuronal de arquitectura fija. (2)

11. Modificación del robot simulado en base a la salida de una red neuronal de arquitectura fija. (2)

12. Modificación del entorno simulado en base al movimiento de un robot. (3)

13. Evaluación de una red neuronal de arquitectura fija en base a su comportamiento. (3)

14. Mutación de los pesos de los arcos de una red de arquitectura fija. (2)

15. Creación de la siguiente generación de la población de redes neuronales de arquitectura fija. (4)

16. Archivar las mejores redes neuronales de arquitectura fija. (1)

Se realizó análisis funcional similar al realizado sobre el sistema de simulación sobre el método de NEAT, con los siguientes coeficientes de complejidad estimativo:

17. Definición de una red neuronal de arquitectura mínima variable. (3)

18. Creación de una población de redes neuronales de arquitectura mínima variable. (4)

19. Clasificación de los individuos de la población en especies. (3)

20. Procesamiento de datos a través de una red neuronal de arquitectura mínima variable. (3)

21. Mutación de los pesos de los arcos de una red neuronal de arquitectura mínima variable. (3)

22. Mutación de la arquitectura de una red neuronal de arquitectura mínima variable. (4)

23. Cruce de dos redes neuronales de arquitectura mínima variable pertenecientes a la misma especie. (5)

24. Creación de la siguiente generación de la población de redes neuronales de arquitectura mínima variable. (5)

25. Archivar las mejores redes neuronales de arquitectura mínima variable. (1)

A continuación se enumeran las funciones por implementar con sus respectivos coeficientes de complejidad de desarrollo estimativo:

26. Modificación del robot simulado en base a la salida de una red neuronal de arquitectura mínima variable. (3)

27. Evaluación de una red neuronal de arquitectura mínima variable en base a su comportamiento. (3)

28. Paralelización de la evaluación de una red neuronal de arquitectura fija. (2)

29. Paralelización de la evaluación de una red neuronal de arquitectura mínima variable. (2)

También se analizaron las distintas funciones a implementar, para resolver los distintos problemas propuestos:

30. Inicialización del entorno simulado, el robot y la pelota para la evolución de redes neuronales de arquitectura fija para que el robot busque la pelota. (3)

31. Inicialización del entorno simulado, el robot y la pelota para la evolución de redes neuronales de arquitectura mínima variable para que el robot busque la pelota. (3)

32. Inicialización del entorno simulado, el robot y la pelota para la evolución de redes neuronales de arquitectura fija para que el robot se posicione detrás de la pelota. (3)

33. Inicialización del entorno simulado, el robot y la pelota para la evolución de redes neuronales de arquitectura mínima variable para que el robot se posicione detrás de la pelota. (3)

34. Inicialización del entorno simulado, el robot y la pelota para la evolución de redes neuronales de arquitectura fija para que el robot golpee la pelota. (3)

35. Inicialización del entorno simulado, el robot y la pelota para la evolución de redes neuronales de arquitectura mínima variable para que el robot golpee la pelota. (3)

36. Función de aptitud de una red neuronal de arquitectura fija para que el robot busque la pelota. (4)

37. Función de aptitud de una red neuronal de arquitectura mínima variable para que el robot busque la pelota. (4)

38. Función de aptitud de una red neuronal de arquitectura fija para que el robot se posicione detrás de la pelota. (4)

39. Función de aptitud de una red neuronal de arquitectura mínima variable para que el robot se posicione detrás de la pelota. (4)

40. Función de aptitud de una red neuronal de arquitectura fija para que el robot golpee la pelota. (4)

41. Función de aptitud de una red neuronal de arquitectura mínima variable para que el robot golpee la pelota. (4)

42. Implantación de un árbol de decisión, quien comandará las redes neuronales de arquitectura fija de acuerdo donde se encuentra el robot y la pelota respecto al entorno simulado. (5)

43. Implantación de un árbol de decisión, quien comandará las redes neuronales de arquitectura mínima variable de acuerdo donde se encuentra el robot y la pelota respecto al entorno simulado. (5)

Análisis de costos

Se realizó un análisis de costo en cuanto al tiempo de desarrollo del directo del simulador en comparación con la aplicación de un método de reingeniería del software. Se realizó una estimación de tiempos de desarrollo en base a la complejidad de las funciones, asignando a cada una de las funciones un valor en horas de desarrollo. La estimación de las horas se realizó tomando como parámetro el

Cuadro B.1: Tabla de estimación de horas.

Complejidad	1	2	3	4	5
Horas estimadas	Menos de 15	20 – 25	30 – 35	40 – 45	Más de 50

tiempo de desarrollo de las funciones implementadas. Esta estimación de horas puede verse en la tabla B.1.

En comparación, el costo en cuanto a horas del desarrollo directo (aproximadamente 1435 horas) es muy superior al costo en horas del proceso de reingeniería (aproximadamente 578 horas). El costo de aplicar reingeniería al sistema es un 60 % menor que el costo del desarrollo directo.

Modificaciones

En la segunda etapa se realizan las modificaciones planteadas en la primer etapa. Comenzando con una limpieza del código innecesario, como la eliminación del código necesario para realizar el entrenamiento con el robot físico conectado vía RS-232 ya que solo nos interesa el desarrollo del controlador en un ambiente de simulación, debido a la velocidad de entrenamiento. También se eliminó el código necesario para que el sistema pueda ser compilado bajo Windows, esto se realizó para la simplificación del código.
Se modularizaron segmentos de código de la función main en módulos independientes tales como funciones de inicialización del simulador, de la interfaz y del proceso de simulación, calculo del valor de aptitud del individuo (fitness) y el mapeo entre el robot y los sensores. También se implementaron módulos para la inicialización de la población y para la ejecución de un individuo o de una red neuronal, así como módulos auxiliares para la implementación de las distintas estrategias evolutivas.
Se prepararon ambos códigos fuente, tanto el del simulador como el de la estrategia evolutiva a incorporar, para realizar la integración definiendo espacios de nombres diferentes para evitar conflictos tanto de nombres de variables como de nombres de clases.
Para implementar la paralelización del algoritmo de evolución se decidió utilizar la estrategia de ”bolsa de tareas” ya que permite un mejor aprovechamiento de los ciclos ociosos de la CPU, así como parametrizar la cantidad de tareas que se procesarán concurrentemente. De esta manera se define a la evaluación de un individuo como una tarea, siendo la población una bolsa de n tareas. También se definen m threads o hilos que serán los encargados de tomar una tarea de la bolsa,

o sea de evaluar un individuo de la población.

Índice de figuras

Índice de cuadros

Bibliografía

[1] Alba, E. Troya, J. M. *A Survey of Parallel Distributed Genetic Algorithms*. Complexity, 4(4). Pags 31-52. 1999.

[2] Alba, E. Troya, J. M. *Cellular Evolutionary Algorithms: Evaluating the Influence of Ratio*. Parallel Problem Solving from Nature (PPSN VI). Vol 1917 de Lecture Notes in Computer Science. Pags 29-38. Spring-Verlag. Heidelberg. 2000.

[3] Alba, E. Troya, J. M. *Improving Flexibility and Efficiency by Adding Parallelism to Genetic Algorithms*. Statistics and Computing, 12(2). Pags 91-114. 2002.

[4] Alba, E. Luna, F. Nebro, A. J. *Advances in Parallel Heterogeneous Genetic Algorithms for Continuous Optimization*. International Hournal of Applied Mathematics and Computer Science, 14(3). Pags 317-333. 2004.

[5] Alba, E. *Parallel Metaheuristics. A New Class of Algorithms*. Wiley, 2005.

[6] Anastasio, T. *A recurrent neural network model of velocity storage in the vestibulo-ocular reflex*. Advances in Neural Information Processing Systems. Pags. 32-38. Morgan Kaufmann. 1991.

[7] Baluja, S. *Structure and Performance of the Fine-Grain Parallelism in Genetic Search*. Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA). Pags 155-162. Morgan Kaufmann. 1993.

[8] Belding, T. C. *The Distributed Genetic Algorithm Revised*. Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA). Pags 114-121. Morgan Kaufmann. 1995.

[9] Cantú-Paz, E. *Designing Efficient Master-Slave Parallel Genetic Algorithms*. IlliGAL Report No. 97004. May 1997

[10] Cantú-Paz, E. *Efficientand Accurate Parallel Genetic Algorithms*. Kluwer Academic Press. 2000.

[11] Bäck, T. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press. New York. 1996.

[12] Banzhaf, W. *Selection: Interactive Evolution*. Handbook of Evolutionary COmputation. Pags C2.9:1-C2.9:6. Institute of Phisics Publishing and Oxford University Press. Bristol New York. 1997.

[13] Bruce, J. Miikkulainnen, R. *Evolving Populations Of Expert Neural Networks*. Department of Computer Sciences, The University of Texas at Austin. Proceedings of the Genetic and Evolutionary Computation Conference. (GECCO-2001, San Francisco, CA), pp. 251-257. 2001.

[14] Bryant, B. Miikkulainnen, R. *Neuroevolution for Adaptive Teams*. Congress on Evolutionary Computation (CEC-2003). 2003.

[15] Cerny, V. *A Themodynamical Approach to the Travelling Salesman Problem: An Efficient simulation algorithm*. Journal of Optimization Theory and Applications, 45. Pags 41-51. 1985.

[16] Corbalán, L. Lanzarini, L. *Evolving Neural Arrays. A new mechanism for learning complex action sequences*. CLEI Electronic Journal. Special Issue of Best Papers presented at CLEI'2002. Volumen 6. Number 1, December 2003. Uruguay. http://www.clei.cl/cleiej/volume.php.

[17] Corbalán, L. Osella Massa, G. Lanzarini, L. De Giusti, A. *ANELAR. Arreglos Neuronales Evolutivos de Longitud Adaptable Reducida*. X Congreso Argentino de Ciencias de la Computación. CACIC 2004. Universidad Nacional de La Matanza. Bs.As. Argentina. ISBN 987-9495-58-6. October 2004.

[18] Corbalán, L. Lanzarini, L. De Giusti, A. *ALENA. Adaptive-Legth Evolving Neural Arrays*. Journal of Computer Science and Technology. Vol 5, nro. 4. (http://journal.info.unlp.edu.ar). ISSN: 1666-6038. Pags. 59-65. 2005.

[19] Cybenko, G. *Approximation by superposition of a sigmoid function*. Mathematics of Control, Signals and Systems, 2. Pags. 303-314. 1989.

[20] De La Maza, M. Tidor, B. *An Analisys of Selection Procedures with Particular Attention Paid to Proporcional and Boltzmann Selection*. Proceedings of the Fifth International Conference on Genetic Algorithms. Pags 124-131. San Mateo, CA. Morgan Kaufmann. 1993.

[21] Dorigo, M. Maniezzo, V. Colorni, A. *Ant Systems: Optimization by e Colony of Cooperating Agents*. IEEE Transactions on Systems, Man and Cybernetics - Part B, 26(1). Pags 29-41. 1996.

[22] Dorigo, M. Di Caro, G. Gambardella, L. M. *Ant Algorithms for Discrete Optimization*. Artificial Life, 5(2). Pags 137-172. 1999.

[23] Dorigo, M. Stützle, T. *Ant Colony Optimization*. MIT Press. Cambridge. MA. 2004.

[24] Feo, T. A. Resende, M. G. C. *Greedy Randomized Adaptive Search Procedures*. Journal of Global Optimizations, 6. Pags 109-133. 1995.

[25] Fogel, L. J. *Toward Inductive Inference Automata*. Proceedings of the International Federation for Information Processing Congress. Pags 395-399. Munich. 1962.

[26] Fogel, L. J. Owens, A. J. Walsh, M. J. *Artificial intelligence through Simulated Evolution*. Wiley. 1966.

[27] Freeman, James A. Skapura, David M. *Redes Neuronales. Algoritmos, aplicaciones y técnicas de programación*. Addison-Wesley, 1993.

[28] Garey, M. R. Johnson, D. S. *Computers and intractability; a Guide to the theory of NP-completeness*. W. H. Freeman, 1979.

[29] Giles, C. Chen, D. Miller, C. Chen, H. Sun, G. Lee, Y. *Second-order recurrent neural network for grammatical interference*. International Joint Conference on Neural Networks. Volumen 2. Pags 273-281. 1991.

[30] Glover, F. *Heuristics for Integer Programming Using Surrogate Constraints*. Decision Sciences, 8. Pags 156-166. 1977.

[31] Glover, F. *Future Paths for Integer Programming and Links to Artificial Intelligence*. Computers & Operations Research, 13. Pags 533-549. 1986.

[32] Glover, F. *Scatter Search and Path Relinking*. En Corne, D. Dorigo, M. Glover, F., editores, New Ideas in Optimization. Advanced Topics in Computer Science Series. McGraw-Hill. 1999.

[33] Glover, F. Laguna, M. Martí, R. *Fundamentals of Scatter Search and Path Relinking*. Control and Cybernetics, 29(3). Pags 658-684. 2000.

[34] Goldberg, D.E. Richardson, J. *Genetic Algorithms with Sharing for Multimodal Function Optimization.* En Grefenstette, J. J., editor, Genetic Algorithms and their Applications. Pags. 41-49. Lawrence Erlbaum Associates. Hillsdale. NJ. 1987.

[35] Goldberg, D.E. Deb, K. *A Comparative Analisys of Selection Schemes Used in Genetic Algorithms*. Foundations of Genetic Algotithms, 1. Pags 69-93. 1991.

[36] Gomi, T. Griffith, A. *Evolutionary robotics-an overview*. Evolutionary Computation, 1996. Proceedings of IEEE International Conference on 20-22 May 1996 pp:40 - 49.

[37] Gomez, Faustino J. Miikkulainen, Risto. *Solving Non-Markovian Control Tasks with Neuro-Evolution*. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI99, Stockholm, Sweden). Denver:Morgan Kaufmann, 1999.

[38] Gordon, V. S. Whitley. *Serial and Parallel Genetic Algorithms as Function Optimizers*. Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA). Pags 177-183. Morgan Kaufmann. 1993.

[39] Gruau, F. Whitley, D. Pyeatt, L. *A Comparison between Cellular Encoding and Direct Encoding for Genetic Neural Networks*. NeuroCOLT Technical Report Series, NC-TR-96-048. July 1996.

[40] Hansen, P. Mladenovic. *Variable Neighborhood Search: Principles and Aplications*. European Journal of Operational Research, 130. Pags 449-467. 2001.

[41] Hecht-Nielsen, Robert. *Neurocomputing*. Addison-Wesley, 1990.

[42] Holland, J. H. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press. Ann Harbor. MI. 1975.

[43] Holland, J. H. *Concerning Efficient Adaptive Systems*. Spartan Books, 1962.

[44] Holland, J. H. *Outline for a logical theory of adaptive systems*. Journal of the Association for Computing Machinery, 9. Pag 297-314. ISSN:0004-5411. 1962.

[45] Hornik, K. Stinchcombe, M. White, H. *Multilayer feedforward networks are universal approximators*. Neural Networks, v.2 n.5, p.359-366, 1989.

[46] Isasi Viñuela, Pedro. Galván León, Inés. *Redes Neuronales Artificiales. Un Enfoque Practico*. Pearson Prenitice Hall, 2004.

[47] Johanson, B. Poli, R. *GP-music: An interactive genetic programming system for music generation with automated fitness raters*. Genetic Programing 98. San Francisco. Pags 181-186. Morgan Kaufmann Publishers. 1998.

[48] *Khepera II User Manual. Version 1.1*. K-Team. 2002.

[49] Khuri, S. Bäck, T. Heitkötter, J. *The Zero/One Multiple Knapsack Problem and Genetic Algorithms*. Proceedings of the 1994 ACM Symposium on Applied Computing. Pags 188-193. 1994.

[50] Kirkpatrick, S. Gelatt, C. D. Vecchi, M. P. *Optimization by Simulated Annealing*. Science, 200 (4598). Pags 671-680. 1983.

[51] Lanzarini, L. *Reconocimiento de Patrones en Imágenes Médicas utilizando Redes Neuronales*. Journal of Computer Science and Technology. Vol.4 . Dic 2000.

[52] Lanzarini, L. Yanivello, D. *Reconocimiento de Comandos Gestuales utilizando GesRN*. X Congreso Argentino de Ciencias de la Computación. CACIC 2004. Bs.As. Argentina. Oct/04. ISBN 987-9495-58-6

[53] Lanzarini, L. Corbalan, L. Osella Massa, G. Hasperué, W. Vinuesa, H. *Sistemas Inteligentes basados en Neurocomputación*. IX Workshop de Investigadores en Ciencias de la Computación WICC 2007. Trelew. Argentina. Mayo 2007.

[54] Lanzarini, L. Hasperue, W. Vinuesa, H. Corbalán, L. Osella Massa, G. *Sistemas Inteligentes. Aplicaciones en Minería de Datos, Robótica Evolutiva y Redes de Computadoras*. X Workshop de Investigadores en Ciencias de la Computación WICC 2008. General Pico. Mayo 2008.

[55] Lawer, E. Lenstra, J. K. Rinnooy Kan, A. H. G. Shmoys, D. B. *The Traveling Salesman Problem*. John Wiley and Son, New York, NY, 1985.

[56] Leung, Frank H. F. Lam, H. K. Ling, S. H. Tam, Peter K. S.. *Tuning of the Structure and Parameters of a Neural Network Using an Improved Genetic Algorithm*. IEEE Transactions On Neural Networks, Vol. 14, No. 1, Jan/2003.

[57] Liepins, G. E. Hilliard, M. R. Richardson, J. Palmer, M. *Genetic Algorithms applications to Set Covering and Traveling Salesman Problems*. Operations Research and Artificial Intelligence: The Integration of The Problem Solving Strategies. Pags 29-57. Kluwer Academic Publishers. Boston. 1990.

[58] Lourenço, H. R. Martin, O. Stützle. *Iterated Local Search* . En Glover, F. kochenberger, G. editores. Handbook of Metaheuristics, volume 57 of International Series in Operations Research & Management Science. Page 321-353. Kluwer Academic Publishers, Norvel, MA. 2002.

[59] Mahfoud, S. W. *Niching Methods for Genetic Algorithms*. PhD thesis. University of Illinois at Urbana-Champaign. Urbana. IL. 1995.

[60] Martin, O. Otto, S. W. Felten, E, W. *Large-step Markov Chains for the Traveling Salesman Problem* . Complex Systems, 5(3). Pags. 299-326. 1991.

[61] Maruyama, T. Hirose, T. Konagaya, A. *A Fine-Grained Parallel Genetic Algorithm for Distributes Parallel System*. Proceedings of the Fifth International Conference on Genetic Algorithms (ICGA). Pages 184-190. Morgan Kaufmann. 1993.

[62] Mathias, K. Whitley, D. *Genetic Operators, the Fitness Landscape and the Traveling Salesman Problem*. Parallel Problem Solving from Nature, 2. Pags 219-228. 1992.

[63] Mathias, E. Whitley, L. D. *Transforming the Search Space with Gray Coding*. Proceedings of the First IEEE COnference on Evolutionary Computation. Vol 1. Pags 513-518. IEEE Science Center. 1994.

[64] Miglino, O. Lund, H. H. Nolfi, S. *Evolving Mobile Robots in Simulated and Real Environments*. Massachusetts Institute of Technology. Artificial Life 2: 417-434 (1995)

[65] Miller, B. L. Goldberg, D. E. *Genetic Algorithms, Tournament Selection, and the Effects of Noise*. Complex Systems, vol 9. Pags 193-212. 1995.

[66] Moriarty, D. E. Miikkulainen, R. *Efficient Reinforcement Learning through Symbiotic Evolution*. Department of Computer Sciences, The University of Texas at Austin. Austin, TX 78712. Machine Learning, Vol. 22, (1996), pp.11-33

[67] Moriarty, David E. Miikkulainen, Risto. *Forming Neural Networks Through Efficient and Adaptive Coevolution*. Journal of Evolutionary Computation, vol. 5, pp 373–399. 1997.

[68] Mühlenbein, H. Schlierkamp-Voosen, D. *Predictive Models for the Breeder Genetic Algorithm: I. Continuous Parameter Optimization*. Evolutionary Computation, 1 (1), pp. 25-49, 1993.

[69] Mühlenbein, H. *The Breeder Genetic Algorithm. A provable optimal search algorithm and its applications*. GMD. D-53754 Sankt Augustin 1, Germany. 1994.

[70] Nemhauser, G. L. Wolsey, A. L. *Integer and Combinatorial Optimization*. John Wiley and Son, New York, 1988.

[71] *Pagina de Usuarios NEAT*. http://www.cs.ucf.edu/ kstanley/neat.html

[72] Nolfi, S. Parisi, D. *Learning to adapt to changing environments in evolving neural networks*. Department of Neural Systems and Artificial Life. Technical Report 95-15. 15, Viale Marx. 00137. Rome. Italy. Nov 1995 (revised Sep 1996).

[73] Kennedy, J. Eberhart, R. Shi, Y. *Swarm Intelligence*. Morgan Kaufmann Publishers, 2001.

[74] Olivera, J. Lanzarini, L. *Cyclic Evolution. A new strategy for improving controllers obtained by layered evolution*. VI Workshop de Agentes y Sistemas Inteligentes 2005. Concordia, Entre Ríos. Argentina. ISBN: 950-698-166-3. October 2005.

[75] Osella Massa, Germán. Vinuesa, Hernán. Lanzarini, Laura. *Modular Creation of Neuronal Networks for Autonomous Robot Control*. 3rd IEEE Latin American Robotics Symposium. LARS 2006. Chile. Oct/06. También publicado en Journal Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial. Vol.11. Nro.35. pp.43-53 2007.

[76] Papadimitriou, C. H. Steiglitz, K. *Combinatorial Optimization - Algorithms and Complexity*. Dover Publications, Inc., New York, 1982.

[77] Parlos, A. Chong, F. Atiya, A. *Application of recurrent multilayer perceptron in modeling complex process dynamics*. IEEE Transaction on Neural Networks, 5(2). 1994.

[78] Pitsoulis, L. S. Resende M. G. C. *Greedy Randomized Adaptive Search Procedure*. En Pardalos, P. M. Resende, M. G. C., editores, Handbook of Applied Optimizations. Pags 168-183. Oxford University Press. 2002.

[79] Potter, Mitchell A. De Jong, Kenneth A. Grefenstette, John J. *A coevolutionary approach to learning sequential decision rules*. Proceedings of the Sixth International Conference on Genetic Algorithms, pp 366–372. Morgan Kaufmann, 1995.

[80] Puskoris, G. Feldkamp, L. *Neurocontrol of nonlinear dynamical systems with filther trained recurrent networks*. IEEE Transaction on Neural Networks, 5(2). 1994.

[81] Radcliffe, N.J. *Genetic set recombination and its application to neural network topology optimization.* Neural computing and applications 1. 1993.

[82] Rechenberg, I. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Frommann-Holzboog. 1973.

[83] Robinson, A. Fallside, F. *A recurrent error propagation speech recognition system*. Computer Speech and Language, 5. Pags 259-274. 1991.

[84] Rumelhart,D., Hinton, G. and Williams, R. *Parallel Distributed Processing*. Chapter Learning representations by backpropagating errors. MIT Press. 1986.

[85] Siedleck, W. Sklansky, J. *On Automatic Feature Selection*. International Journal of Pattern Recognition and Artificial Intelligence, 2(2). Pags 197-220. 1988.

[86] Sierra Araujo, Basilio. *Aprendizaje Automático: Conceptos Básicos y Avanzados*. Pearson Prentice Hall, 2006.

[87] Simpson, A. R. Dandy, G. C. Murphy, L. J. *Genetic Algorithms Compared to other Techniques for Pipe Optimization*. Journal of Water Resources Planning and Management, 120(4). 423-443. 1994.

[88] Spears, w. M. De Jong, K. A. *On the Virtues of Parametrized uniform crossover*. Proceedings of the Fourth International Conference on Genetic Algorithms. Pags 230-236. San Mateo Ca. Morgan Kaufmann. 1991.

[89] Spiessens, P. Manderick, B. *A Massively Parallel Genetic Algorithm*. Proceedings of the Fourth International Conference on Genetic Algorithms (ICGA). Pags 279-286. Morgan Kaufmann. 1991.

[90] *Pagina de Kenneth O. Stanley*. http://www.cs.ucf.edu/ kstanley/

[91] Stanley, K.O. Miikkulainen, R. *Evolving neural networks through augmenting topologies.* Evolutionary Computation 10. 2002. Pags. 99-127.

[92] Stanley, K.O. Miikkulainen, R. *Competitive coevolution through evolutionary complexification*. Journal of Artificial Intelligence Research 21. 2004.

[93] Stone, P. *Layered Learning in Multiagent Systems*. PhD Thesis. CMU-CS-98-187. School of Computer Science. Carnegie Melon University. 1998.

[94] Stützle, T. *Local Search Algorithms for Combinatorial Problems - Analisis, Algorithms and New Applications*. DISKI - Dissertationen zur Künstliken Intelligenz. Infix, Sankt Augustin, Germany. 1999.

[95] Tanese, R. *Distributed Genetic Algorithms*. Proceedings of the Third International Conference on Genetic Algorithms (ICGA). Pags 434-439. Morgan Kaufmann. 1989.

[96] *Universidad de Skövde, Suecia*. http://www.his.se

[97] Vinuesa, Hernán. Lanzarini, Laura. *Neural Networks Elitist Evolution*. Proceedings del 29th International Conference on Information Technology Interfaces. ITI 2007. Dubrovnik, Croacia. 25 a 28 de junio de 2007. ISBN: 978-953-7138-09-7 / ISSN: 1330-1012 (CD Rom). Artículo completo – Publicado por IEEE Computer Society Press – Pág. 457-462.

[98] Vinuesa, Hernán. Osella Massa, Germán. Corbalán, Leonardo. Lanzarini, Laura . *Continuous Evolution of Neural Modules of Autonomous Robot Controllers*. Jornadas Chilenas de Computación 2007. Iquique, Chile. Noviembre de 2007.

[99] Vinuesa, Hernán. Lanzarini, Laura. Osella Massa, Germán. *Improving Controllers based on Neural Networks obtained by Layered Evolution*. Conferencia Latinoamericana de Informática. CLEI 2008. Santa Fe. Argentina. 8 al 12 Septiembre de 2008.

[100] Vinuesa, Hernán. Lanzarini, Laura. Hasperué, Waldo. Corbalán, Leonardo. *Improving Controllers based on Neural Networks obtained by Parallel Evolution Strategy*. XXVII Jornadas Chilenas de Computación. JCC 2008. Punta Arenas. Chile. 10 al 15 de Noviembre de 2008. También publicado en XIV Congreso Argentino de Ciencias de la Computación. CACIC 2008. Chilecito, La Rioja. Argentina. 6 al 10 de Octubre de 2008.

[101] Voigt, H. M. Anheyer, T. *Modal Mutations in Evolutionary Algorithms*. Proceedings of the First IEEE International Conference on Evolutionary Computation. Vol 1. Pags 88-89. Orlando. 1994.

[102] Voudouris, C. Tsang, E. *Guided Local Search*. European Journal of Operational Research, 113(2). Pags 469-499. 1999

[103] Walker, J. Garrett, S. Wilson, M. *The Balance Between Initial Training and Lifelong Adaptation in Evolving Robot Controllers*. IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—PART B: CYBERNETICS, VOL. 36, NO. 2. APRIL 2006.

[104] Whitley, Darrell. Dominic, Stephen. Das, Rajarshi. Anderson, Charles W. *Genetic Reinforcement Learning for Neurocontrol Problems*. Machine Learning, 13, 259-284. Kluwer Academic Publishers, Boston. 1993.

[105] Whiteson, S. Stone, P. *Concurrent Layered Learning*. Second International Conference on Autonomous Agents and Multiagent Systems - AAMAS'03 pp 14-18.Julio 2003.

[106] Widrow, B. *An adaptative 'adaline' neuron using chemical 'memistors'*. Technical Report 1553-2, Standford Electronics Laboratory. 1960.

[107] Wieczorek, W. Czech, Z. J. *Selection Schemes in Evolutionary Algorithms*. Intelligent Information Systems. Advances in Soft Computing. 2002.

[108] *YAKS - Yet Another Khepera Simulator*. URL : http:/www.his.se/iki/yaks

[109] Yao, X. Liu, Y. *Ensemble Structure of Evolutionary Artificial Neural networks*. Computational intelligence Group, School of Computer Sciencie University College. Australian Defence Force Academy, Canberra, ACT, Australia 2600. 1996.

[110] Yao, Xing. *Evolving Artificial Neural networks*. School of Computer Sciencie The University of Birmingham Edgbaston, Birmingham B15 2TT. Proceedings of the IEEE. Vol.87, No.9, (September 1999), pp.1423-1447