Evolución de controladores definidos por Redes Neuronales

Autor: Javier Hugo Olivera
Número de alumno: 2473/3
Director: Laura Lanzarini

Facultad de Informática
Universidad Nacional de La Plata

El método denominado “Evolución Cíclica” propuesto por J. H. Olivera y L. Lanzarini y descripto en esta Tesis de Grado fue publicado en el XI Congreso Argentino de Ciencias de la Computación CACIC 2005 realizado en Concordia - Entre Ríos del 17 al 21 de octubre de 2005. Una posterior actualización de este artículo fue seleccionada para ser publicada en el S*pecial Issue on Selected Papers from CACIC 2005* correspondiente al Journal of Computer Science and Technology (JCS&T) *Vol. 5 - No. 4 - December 2005 - ISSN 1666-6038*

Tabla de contenidos

INTRODUCCIÓN

Las Redes Neuronales han demostrado ser una excelente herramienta para la resolución de tareas complejas, entendiendo como tales aquellas cuya solución no es directa sino que involucra el aprendizaje de una estrategia para lograr el objetivo esperado. Tal es el caso de un robot que debe trasladar distintos tipos de elementos dentro de un escenario con obstáculos.

También es importante considerar que existen situaciones que no pueden ser resueltas por un único agente. Tal es el caso del fútbol robótico donde varios jugadores combinan sus acciones para lograr un único objetivo. Este tipo de problemas pertenece a lo que se conoce como sistemas multi agente. Los sistemas multi agentes (MAS – Multi Agent System) pertenecen a una rama de la inteligencia artificial (AI – Artificial Intelligence) que apunta a proveer principios para la construcción de sistemas complejos involucrando múltiples agentes y mecanismos para coordinar comportamientos independientes de los mismos. Es importante notar que, más allá de las diferencias entre los agentes, es el grupo el que debe llevar a cabo la estrategia.

Diversas investigaciones han demostrado que este tipo de situaciones pueden ser resueltas dividiendo el problema original en partes más simples, llamadas subtareas, permitiendo de esta forma un aprendizaje gradual de la respuesta buscada.

Cuando la situación a resolver es compleja, es difícil establecer a priori el controlador a utilizar y es aquí donde la descomposición del problema toma importancia. Este proceso, que comienza por aprender las tareas más simples y a partir de ellas va incorporando habilidades más complejas, se conoce como aprendizaje por capas.

Por otro lado, a menos que se disponga de la información inicial necesaria para resolver cada subtarea, resulta ideal disponer de algún mecanismo que permita realizar la adaptación de la manera más automática posible. En esta dirección se han desarrollado distintas soluciones que combinan técnicas de Evolución Incremental con Redes Neuronales Evolutivas con el objeto de proveer un mecanismo adaptivo que minimice el conocimiento previo necesario para obtener un buen desempeño dando lugar a controladores formados por varias redes. Otro aspecto a tener en cuenta es la forma de determinar cual es la red neuronal que debe desenvolverse en cada instante de tiempo; en esta línea existen diferentes alternativas que van desde el uso de un árbol de decisión diseñado ad-hoc hasta mecanismos que organizan la estructura en forma automática.

1.1 Organización del trabajo de grado

El capítulo 2 presenta una introducción a la compunción evolutiva y un análisis más detallado sobre los algoritmos genéticos. De estos últimos se describe la forma en que operan, las posibles representaciones de las estructuras sobre las cuales trabajan, las operaciones que utilizan y los fundamentos teóricos sobre los cuales basan su funcionamiento. Finalmente se presentan algunos ejemplos y algunos campos de aplicación.

En el capítulo 3, se analiza un tema fundamental para el desarrollo de esta tesis denominado neuroevolución. Las redes neuronales son capaces de resolver problemas complejos mediante técnicas de aprendizaje, éstas les permiten adaptarse a un cierto entorno y generar la respuesta buscada. La neuroevolución utiliza dichas redes para la resolución de problemas, pero la adaptación de las mismas, no se realiza mediante los métodos tradicionales de aprendizaje, sino que se logra por medio de procesos evolutivos. El capítulo presenta una breve introducción a las redes neuronales, luego se estudia el tema de redes neuronales evolutivas con sus distintas variantes y finalmente se exponen algunos métodos neuroevolutivos mejorados.

El capítulo 4 presenta una introducción a los sistemas multiagente. Luego se detallan algunos mecanismos para el aprendizaje de tareas complejas los cuales serán utilizados para resolver el juego keepaway. Primero se analizan algunos métodos tradicionales y finamente se presenta un método nuevo propuesto para esta tesis.

En el capítulo 5 se describe el juego elegido para el estudio de cada uno de los métodos de aprendizaje presentados en el capítulo 4. En este juego, denominado keepaway, hay tres jugadores que deben hacerse la mayor cantidad de pases posibles con una pelota, mientras otro debe intentar robarla. En este capítulo, se realiza un análisis detallado de la aplicación de cada método de aprendizaje al juego keepaway así como también de los resultados obtenidos.

El capítulo 6 presenta un análisis global de todos los métodos de aprendizaje y las conclusiones finales.

2 COMPUTACIÓN EVOLUTIVA Y ALGORITMOS EVOLUTIVOS

El término computación evolucionaria o evolutiva, engloba una serie de técnicas inspiradas en los principios de la teoría Neo-Darwiniana de la evolución natural.
El Neo-Darwinismo establece que la historia de la vasta mayoría de la vida en nuestro planeta puede ser explicada a través de un puñado de procesos estadísticos que actúan sobre y dentro de las poblaciones y especies: la reproducción, la mutación, la competencia y la selección.

La reproducción es una propiedad obvia de todas las formas de vida de nuestro planeta, pues de no contar con un mecanismo de este tipo, la vida misma no tendría forma de producirse. En cualquier sistema que continuamente se reproduce a sí mismo y que está en constante equilibrio, pueden existir cambios repentinos o mutaciones. El contar con una cantidad finita de recursos y de espacio para albergar la vida en la Tierra garantiza la existencia de la competencia. La selección se vuelve la consecuencia natural del exceso de organismos con respecto a los recursos disponibles. Es por ello, que los individuos más aptos tendrán más posibilidades de continuar con vida. La evolución es, por lo tanto, el resultado de estos procesos estocásticos (probabilísticos) que interactúan entre sí en las poblaciones, generación tras generación.

La computación evolucionaria usa modelos computacionales de procesos evolutivos naturales como elementos claves en el diseño e implementación de sistemas capaces de resolver problemas utilizando una computadora. Hay una variedad de modelos computacionales evolutivos que han sido propuestos y estudiados, los cuales comúnmente son llamados algoritmos evolucionarios. Ellos comparten una base conceptual común simulando la evolución de estructuras por medio de procesos de selección y reproducción. Estos procesos dependen del funcionamiento o aptitud (fitness) de las estructuras dentro de un ambiente.

```
Procedimiento EA; {
t = 0;
Inicializar_la_ poblacion P(t);
Evaluar P(t);
Hasta (fin) {
        t = t + 1;
        Selección_de_padres P(t);
        Modificaciones P(t);
        Evaluar P(t);
        Sobrevivir P(t);
        }
}
```

Figura 2.1. Un algoritmo evolucionario típico.

Más precisamente, los algoritmos evolucionarios mantienen una población de individuos que evolucionan acorde a reglas de selección y otros operadores, tales como recombinación y mutación. Cada individuo en la población recibe una medida de aptitud en un ambiente. La selección focaliza la atención en individuos con fitness alto. La recombinación y mutación perturba esos individuos, permitiendo la exploración. Aunque desde un punto de vista biológico simplista, estos algoritmos son suficientemente complejos para proveer robustos y poderosos mecanismos de búsqueda adaptivos.

La Figura 2.1 esquematiza un algoritmo evolucionario (EA – evolutionary algorithm) típico. Una población de estructuras (individuos) es inicializada y luego evolucionada de generación en generación mediante aplicaciones repetidas de evaluaciones, selecciones, y algunos operadores que modifican la estructura de los individuos. El tamaño de la población *N,* generalmente se mantiene constante en el algoritmo evolucionario, aunque no hay una razón a priori para imponer esta restricción.

Un algoritmo evolucionario típicamente inicializa su población en forma aleatoria, aunque un conocimiento específico del dominio puede mejorar la respuesta del mismo. Las evaluaciones cuantifican la aptitud de cada individuo, de acuerdo a su desempeño en algún ambiente. La evaluación puede ser tan simple como computar una función de fitness o tan compleja como ejecutar una extensa simulación. La selección se realiza comúnmente en dos pasos, selección de padres y supervivencia. El primer paso, decide quienes son los individuos que se reproducirán y cuantos hijos se obtendrán a partir de ellos. Los hijos son creados mediante recombinación, y mutación. La recombinación, también conocida como crossover, intercambia material genético de los padres generando descendencia con características comunes a ambos. La mutación perturba algunos individuos de forma aleatoria, modificando ciertos aspectos de los mismos. Luego, la nueva población es evaluada y finalmente, el paso de supervivencia decide que individuos sobrevivirán.

Los algoritmos evolucionarios engloban técnicas que permite encontrar soluciones aproximadas a problemas de optimización. Un problema de optimización, es un problema donde se intenta encontrar el óptimo global (o los óptimos globales). En matemáticas, existe un área que se ocupa de desarrollar los formalismos que permitan garantizar la convergencia de un método hacia el óptimo global de un problema. Existen muchos tipos de problemas de optimización, pero los más comunes son los de optimización numérica, en los cuales se intenta maximizar o minimizar una función determinada.

El origen de los algoritmos evolucionarios se remite a la década de 1950 (Fraser, 1957; Box 1957). En las últimas décadas emergieron tres metodologías enmarcadas dentro de los algoritmos evolucionarios conocidas como: “Programación evolucionaria” " (Fogel et al., 1966) , “Estrategias evolucionarias” " (Rechenberg, 1973), y “Algoritmos genéticos.” (Holland, 1975).

Cada uno de estos métodos se originó de manera independiente y con motivaciones diferentes. Aunque ellos son similares desde una perspectiva de alto nivel, implementan los algoritmos evolucionarios de distinta manera. Las diferencias están en casi todos los aspectos de los algoritmos, incluyendo la representación de las estructuras de los

individuos, los mecanismos utilizados para la selección, las formas de los operadores genéticos y la medida del rendimiento.

Estas aproximaciones también han inspirado el desarrollo de algoritmos evolucionarios adicionales tales como “sistemas clasificadores” (Holland, 1986), “Los sistemas LS” (Smith, 1983), “Sistemas de operadores adaptivos” (Davis, 1989), GENITOR (Whitley, 1989), SAMUEL (Grefenstette, 1989), "Programación genética" (de Garis, 1990; Koza, 1991), "GAs difusos" (Goldberg, 1991), y la “aproximación CHC” (Eshelman, 1991).

2.1 Programación evolucionaria

La programación evolucionaria (EP – Evolucionary programming), desarrollada por Fogel et al. (1966), fue utilizada en un principio para hacer evolucionar autómatas de estados finitos, los cuales eran expuestos a una serie de símbolos de entrada (el ambiente), y se esperaba que, eventualmente, fueran capaces de predecir las secuencias futuras de símbolos que recibirían.

Esta técnica ha usado tradicionalmente representaciones que fueron pensadas para un problema en particular. Por ejemplo en problemas de optimización con valores reales, los individuos son vectores de valores reales. Del mismo modo, se han utilizado listas ordenadas en la resolución del “problema del viajante” (traveling salesman problem), y grafos para aplicaciones con máquinas de estados finitos. La EP se usa a menudo como una técnica de optimización, aunque ésta se originó con el deseo de generar máquinas inteligentes.

La Figura 2.2 muestra el esquema de un programa evolucionario. Después de la inicialización, los *N* individuos son seleccionados como padres y luego son mutados,

```
Procedimiento EP; {
t = 0;
Inicializar_la_ poblacion P(t);
Evaluar P(t);
Hasta (fin) {
        t = t + 1;
        Selección_de_padres P(t);
        Mutar P(t);
        Evaluar P(t);
        Sobrevivir P(t);
        }
}
```

Figura 2.2. Esquema de un algoritmo de programación evolucionaria.

produciendo N hijos. Los hijos son evaluados y *N* sobrevivientes son elegidos de los 2*N* individuos (padres + hijos), usando una función de probabilidad basada en el fitness. En

otras palabras, individuos con mayor fitness tienen mayor chance de sobrevivir. La forma de la mutación está basada en la representación usada, y es a menudo adaptiva. La recombinación generalmente no es aplicada dado que la EP pretende modelar el proceso evolutivo a nivel de especies y no a nivel de individuos. Además, dado que las formas de mutación usadas son muy flexibles, pueden producir perturbaciones similares a la recombinación.

2.2 Estrategias evolucionarias

Las estrategias Evolutivas (ESs - Evolucionary Strategies) fueron desarrolladas por Rechenberg (1963). Esta técnica, en su versión original, utilizó operadores de selección y mutación con poblaciones de tamaño uno. Luego Schwefel (1981) introdujo la recombinación y poblaciones de más de un individuo. También proporcionó una buena comparación de las ESs con técnicas de optimización tradicionales. Debido al interés inicial en problemas de optimización hidrodinámicos de alta complejidad, las estrategias evolutivas utilizan, generalmente, representaciones con vectores de valores reales.

```
Procedimiento ES; {
t = 0;
Inicializar_la_ poblacion P(t);
Evaluar P(t);
Hasta (fin) {
        t = t + 1;
        Selección_de_padres P(t);
        Recombinar P(t);
        Mutar P(t);
        Evaluar P(t);
        Sobrevivir P(t);
        }
 }
```

Figura 2.3. Esquema de un algoritmo de estrategia evolucionaria.

La Figura 2.3 muestra el esquema de una estrategia evolucionaria típica. Después de la inicialización y evaluación, los individuos son seleccionados de forma aleatoria como futuros progenitores. En las ESs estándar, pares de padres producen hijos por medio de recombinación, luego los hijos son perturbados por medio de una operación de mutación. El número de hijos creados es mayor que *N*. La supervivencia es determinística y es implementada en una de dos formas:

- La primera permite a los N mejores hijos sobrevivir, y reemplaza a los padres con esos hijos.
- La segunda permite a los N mejores hijos y padres sobrevivir.

A diferencia de la programación evolucionaria, la recombinación juega un papel importante en las estrategias evolucionarias.

2.2.1 Estrategias Evolucionarias vs. Programación Evolucionaria

La Programación Evolutiva usa normalmente selección estocástica, mientras que las estrategias evolutivas usan selección determinística.
Ambas técnicas no requieren que las variables del problema sean codificadas.
La programación evolutiva es una abstracción de la evolución al nivel de las especies, por lo que no se requiere el uso de un operador de recombinación (diferentes especies no se pueden cruzar entre sí). En contraste, las estrategias evolutivas representan una abstracción de la evolución al nivel de un individuo, por lo que la recombinación es posible.

2.3 Algoritmos genéticos

Existe una amplia clase de problemas interesantes para los cuales no han sido desarrollados algoritmos que sean razonablemente rápidos. Muchos de estos problemas son de optimización. Dado un problema de optimización complejo, en ocasiones, es posible encontrar un algoritmo eficiente cuya solución se aproxime a la óptima.

En general cualquier tarea que se quiera realizar puede ser resuelta de varias formas, en particular una forma interesante de encarar la resolución consiste en pensar un proceso de búsqueda a través de un espacio de potenciales soluciones. Puesto que estamos interesados en la mejor solución, podemos ver esta tarea como un proceso de optimización. Para espacios pequeños, generalmente son suficientes métodos exhaustivos de búsqueda; para grandes espacios, deben ser empleadas técnicas especiales de inteligencia artificial. Los algoritmos genéticos (GAs – Genetic Algorithms) pertenecen a este último tipo de técnicas; ellos son algoritmos estocásticos cuyo método de búsqueda modela algún fenómeno natural como puede ser herencia genética y lucha por la supervivencia Darwiniana.

La idea tras los algoritmos genéticos es hacer lo que la naturaleza hace. Tómese un conejo como ejemplo: en un momento dado hay una población de conejos. Alguno de ellos son más rápidos y listos que otros. Esos conejos rápidos y listos están menos propensos a ser comidos por zorros y, por consiguiente, la mayoría de ellos sobrevivirá para hacer lo que mejor saben: producir más conejos. Por supuesto, algunos de los conejos más lentos y menos listos también sobrevivirán sólo porque tuvieron suerte. La población de conejos comienza a reproducirse. Esta reproducción resulta en una buena mezcla del material genético de los conejos: algunos conejos lentos se aparean con conejos rápidos, algunos rápidos con otros rápidos, algunos más listos con otros menos listos, etc. Características nuevas pueden aparecer en los conejos debido a mutaciones en su material genético. Los conejos bebés de la nueva población serán (en promedio) más rápidos y listos que los pertenecientes a la población original, porque esta población tiene en su mayoría conejos rápidos y listos que son los que sobreviven a los zorros.

Un algoritmo genético sigue paso a paso un proceso que se asemeja a la historia de los conejos.

Los algoritmos genéticos usan un vocabulario tomado de la genética natural. Se habla de individuos (o genotipos) en una población; a menudo estos individuos son llamados cromosomas o cadenas. Cada célula de todo organismo de una especie dada lleva un cierto número de cromosomas, por ejemplo el hombre posee 46 de ellos. Sin embargo, el interés será puesto en individuos de un solo cromosoma. Los cromosomas están formados por elementos llamados genes (también conocidos como caracteres o decodificadores) organizados en sucesión lineal. Cada gen controla la herencia de uno o varios caracteres. Los genes de ciertos caracteres están localizados en posiciones del cromosoma llamadas locus. Los caracteres de los individuos (como el color de ojos) pueden manifestarse de maneras diferentes; los genes tienen distintas formas llamadas alelos. La constitución interna de un cromosoma se conoce como genotipo y las características externas observables se conocen como fenotipos.

Cada genotipo debería representar una potencial solución a un problema, el significado de cada cromosoma, su fenotipo, es definido externamente; un proceso evolutivo que corre en una población de cromosomas corresponde a una búsqueda a través de un espacio de potenciales soluciones. Esta búsqueda requiere balancear objetivos aparentemente conflictivos, la explotación de la mejor solución y la exploración del espacio de búsqueda.
Una técnica conocida como Hillclimbing es un ejemplo de una estrategia que explota la mejor solución para posibles mejoras; por otro lado ésta descuida la exploración del espacio de búsqueda. La búsqueda aleatoria es un ejemplo típico de una estrategia la cual explora el espacio de búsqueda ignorando la explotación de regiones prometedoras del mismo.
Los algoritmos genéticos son una clase de métodos de búsqueda de propósito general (independiente del dominio) los cuales tienen un remarcable balance entre exploración y explotación del espacio de búsqueda.

Los algoritmos genéticos han sido aplicados exitosamente a problemas de optimización como el ruteo de cables, organización de eventos, control adaptivo, juegos, optimización de consultas en bases de datos, problemas de viajantes, etc. Sin embargo los problemas de optimización representan el mayor campo de aplicabilidad para los GAs.

Durante la última década, el significado de optimización ha crecido aún más debido a que muchos problemas de optimización combinatoria a gran escala y problemas de ingeniería muy complicados pueden ser resueltos solo aproximadamente con los equipos de cómputos actuales.

Los algoritmos genéticos apuntan a este tipo de problemas complejos y son muy diferentes a los algoritmos de búsqueda aleatorios. Los GAs combinan elementos de búsqueda directa y estocástica. Como consecuencia de ello, estos algoritmos son también más robustos que otros métodos de búsqueda directa. Otra propiedad importante de los métodos de búsqueda basados en conceptos genéticos, es que mantienen una población de potenciales soluciones. Todos los otros métodos procesan solo un único punto del espacio de búsqueda.

Los métodos denominados Hillclimbing usan una técnica de mejora iterativa; esta técnica es aplicada a un único punto por vez en el espacio de búsqueda. Durante una única iteración, se selecciona un nuevo punto de una vecindad del actual (por esto también se

conoce a esta técnica como búsqueda vecina o local). Si el nuevo punto provee un mejor valor de la función objetivo, éste se vuelve el actual. De otra forma, algún otro punto vecino es seleccionado y testeado contra el actual. El método termina cuando la función objetivo no puede ser mejorada por ningún punto de la vecindad.

Es claro que el método hillclimbing provee solo valores óptimos locales y esos valores dependen fuertemente de la selección del punto de comienzo. Además, no hay información disponible en el error relativo de la solución encontrada con respecto al óptimo global.
Para incrementar las chances de éxito, los métodos de tipo hillclimbing generalmente son ejecutados sobre un gran número de puntos iniciales. Estos no necesitan ser seleccionados de forma aleatoria, una selección de un punto inicial para una ejecución puede depender del resultado de la ejecución previa.

La técnica de templado simulado (simulated annealing) elimina la mayoría de las desventajas de los métodos hillclimbing: las soluciones no dependen del punto de comienzo y los resultados están generalmente cercanos al punto óptimo global. Esto se logra introduciendo una probabilidad *p* de aceptación (por ejemplo, reemplazo del punto actual por uno nuevo): p = 1, si el nuevo punto provee un mejor valor de la función objetivo;

Como se mencionó anteriormente, un GA realiza una búsqueda multidireccional manteniendo una población de potenciales soluciones y alienta la formación de información e intercambio entre esas direcciones. La población sufre una evolución simulada. En cada generación las soluciones relativamente buenas se reproducen, mientras soluciones relativamente malas mueren. Para distinguir entre diferentes soluciones se utiliza una función de evaluación que representa el ambiente donde deben desenvolverse.

La estructura de un algoritmo genético simple es la misma que la de un programa evolutivo. Durante la iteración *t*, un algoritmo genético mantiene una población de potenciales soluciones (cromosomas), $P(t) = \{x_1^t,..., x_n^t\}$. Cada solución x_i^t es evaluada para dar alguna medida de su aptitud Luego, en la iteración t+1, se forma una nueva población seleccionando los individuos más calificados. Algunos miembros de esta nueva población sufrirán alteraciones por medio del crossover y la mutación, para formar nuevas soluciones. El crossover combina las características de dos cromosomas progenitores para formar dos descendientes similares intercambiando distintos segmentos de los padres. La mutación altera, arbitrariamente, uno o más genes de un cromosoma seleccionado. Esta modificación es guiada por una probabilidad denominada velocidad de mutación.

Un algoritmo genético debe tener los siguientes cinco componentes:

- Una representación genética para potenciales soluciones del problema
- Una forma de crear una población inicial de soluciones potenciales.
- Una función de evaluación que juegue el papel del ambiente, calificando soluciones en término de sus fitness.
- Operadores genéticos que alteren la estructura de los cromosomas.
- Valores para varios parámetros que el algoritmo genético usa (tamaño de población, probabilidades de aplicación de operadores genéticos, etc.)

Un algoritmo genético representa una técnica de búsqueda usada en ciencias de la computación para encontrar soluciones aproximadas a problemas de búsqueda y optimización. La Figura 2.4 muestra un esquema de un algoritmo genético típico.

```
Procedimiento GA; {
t = 0;
Inicializar_la_ poblacion P(t);
Hasta (fin) {
        t = t + 1;
        Evaluar P(t);
        Selecciónar_padres P(t);
        Recombinar P(t);
        Mutar P(t);
        }
}
```

Figura 2.4. Esquema de un algoritmo genético.

Los algoritmos genéticos están comúnmente implementados, como una simulación en la cual una población, que representa soluciones candidatas llamadas cromosomas o individuos, evoluciona hacia mejores soluciones. Tradicionalmente, las soluciones son representadas en formato binario como cadenas de 0s y 1s, pero existen diferentes codificaciones. La evolución comienza desde una población de individuos completamente aleatorios y sucede en generaciones. En cada generación, se evalúa el fitness de toda la población, luego se seleccionan múltiples individuos y finalmente, mediante operaciones de recombinación y mutación, se modifican para formar una nueva población que será utilizada en la siguiente iteración del algoritmo.

2.3.1 Operación de un algoritmo genético

Se requieren dos elementos para que cualquier problema pueda ser encarado con un algoritmo genético que permita buscar una solución. Primero, debe haber un método para representar una solución de una manera que pueda ser manipulada por el algoritmo. Tradicionalmente, una solución puede ser representada por una cadena de bits, números o caracteres. Segundo, debe haber algún método para medir la calidad de cualquier solución propuesta, usando una función de fitness.

Por ejemplo, para un problema que involucra acomodar tantos objetos pesados como sea posible dentro de una mochila sin romperla; una representación de la solución podría ser una cadena de bits, donde cada bit representa un peso diferente, y el valor del bit (0 o 1) indica si el peso es agregado a la mochila o no. El fitness de la solución sería determinado midiendo el peso total de la solución propuesta. Cuanto más alto es el peso, mayor es el fitness, suponiendo que la solución es posible.

2.3.2 Inicialización

El proceso comienza con la generación en forma aleatoria de muchos individuos para formar una población inicial. El tamaño de la población depende de la naturaleza del problema, pero típicamente contiene cientos o miles de posibles soluciones. Tradicionalmente, la población es generada en forma aleatoria, cubriendo todo el rango de posibles soluciones (el espacio de búsqueda). Ocasionalmente, las soluciones pueden ser generadas (sembradas) en áreas donde es más probable encontrar soluciones óptimas.

2.3.3 Evaluación

Durante la evaluación, se decodifica un cromosoma, convirtiéndose en una serie de parámetros que conforman una posible solución de un problema. Luego se le da una puntuación a esa solución en función de lo cerca que esté de la mejor. A esta puntuación se la llama fitness. El fitness determina siempre los cromosomas que se van a reproducir, y aquellos que se van a eliminar.
Una vez evaluado el fitness, se debe crear una nueva población teniendo en cuenta que los buenos rasgos de los mejores cromosomas se transmitan a ésta. Para ello, hay que seleccionar a una serie de individuos encargados de esta tarea.

2.3.4 Selección

El operador de selección juega un papel importante en el proceso evolutivo de un algoritmo genético, ya que tiene la responsabilidad de guiar la búsqueda hacia regiones prometedoras del espacio de soluciones.

Durante cada sucesiva generación, se selecciona una porción de la población existente para reproducirse y formar una nueva generación. Soluciones particulares son seleccionadas a través de un proceso basado en el fitness, las más aptas (según la función de fitness) tienen generalmente más probabilidades de ser elegidas.

Actuando conjuntamente, los operadores de mutación y crossover exploran el espacio de búsqueda mientras la selección explota la información representada dentro de la población. El balance entre explotación y exploración, o en otras palabras, la creación de diversidad y su reducción debido a la predilección por individuos con fitness elevados, es crítico para lograr un comportamiento razonable de un algoritmo genético en el caso de problemas de optimización complicados. El operador de selección provee un mecanismo para afectar dicho balance hacia la explotación, incrementando el énfasis sobre los mejores individuos, o hacia la exploración, permitiendo que toda la población tenga chances similares de sobrevivir aún para los peores individuos. Informalmente, el término presión selectiva es ampliamente usado para caracterizar el mayor (presión selectiva fuerte) o menor (presión selectiva débil) énfasis del operador de selección sobre los mejores individuos.

Thomas Bäck realizó una investigación sobre el impacto de distintos métodos de selección sobre la presión selectiva [Bäck, T.]. Otros autores como Goldberg y Bed hicieron análisis comparativos de los distintos esquemas de selección [Goldberg, D. E. And Deb, K.]

La mayoría de los operadores de selección son probabilísticos y diseñados de modo que una pequeña proporción de las soluciones menos aptas sea seleccionada. Esto ayuda a mantener la diversidad de la población, previniendo la convergencia prematura a soluciones pobres. Por el contrario, en las estrategias evolutivas, la selección es extintiva. Es decir, que las soluciones menos aptas no tienen probabilidad de sobrevivir.
Métodos populares y bien estudiados de selección incluyen, selección proporcional y selección por torneo.

2.3.4.1 Selección proporcional

La selección proporcional describe a un grupo de esquemas de selección originalmente propuestos por Holland en [Holland, J. H. 1975] en los cuales se eligen individuos en función de su contribución de aptitud con respecto al total de la población.

La forma de selección proporcional más comúnmente usada desde los orígenes de los algoritmos genéticos, se conoce como selección por rueda de ruleta. En este método, una función de fitness les asigna a las posibles soluciones o cromosomas un valor de aptitud específico. Este nivel de aptitud es usado para asociar una probabilidad de selección a cada cromosoma particular. Mientras las soluciones candidatas con fitness elevado tendrán menos probabilidades de ser eliminadas, también existe la posibilidad de que soluciones débiles puedan sobrevivir al proceso de selección. Este factor es positivo, porque a pesar de que la solución sea débil, puede incluir algún componente que puede resultar útil en el proceso de recombinación.

La analogía con la rueda de ruleta puede ser pensada gráficamente como una rueda en la cual cada solución candidata representa una cavidad dentro de ella; el tamaño de cada cavidad es proporcional a la probabilidad de selección de la solución. La selección de N cromosomas de la población es equivalente a jugar N partidos en la ruleta y el individuo indicado por la bolilla es el seleccionado para la recombinación. (Véase Figura 2.5)

La rueda de ruleta se construye como sigue (se asume que los valores de fitness son positivos, de otra manera, se puede usar algún mecanismo de escalado):

- Calcular el valor de fitness $fit(vi)$ para cada cromosoma v_i (i = 1,..., N = tamaño de la población).
- Encontrar el fitness total de la población $F = \Sigma_{i=1..N} fit(vi)$.
- Calcular la probabilidad de una selección p_i para cada cromosoma v_i (i = 1,...,N): $p_i = fit(vi) / F$.
- Calcular una probabilidad acumulativa q_i para cada cromosoma v_i (i = 1,...,N): $qi = \Sigma_{j=1..i} p_j$.

El proceso de selección se basa en hacer girar la rueda de ruleta N veces; cada vez se selecciona un único cromosoma para la nueva población de la siguiente forma:

- Se genera un numero real *r* en el rango [0..1].
- Si $r < q_1$ entonces se selecciona el primer cromosoma (v_i); si no se selecciona el i-ésimo cromosoma v_i ($2 \leq i \leq N$) tal que $q_{i-1} < r < q_i$.

Ejemplo:

Individuos	Fitness	Fitness Total
(1)	25	F = (1)+(2)+(3)+(4) = 286
(2)	81	
(3)	36	
(4)	144	

Prob. de selección	Prob. acumulada
25 / 286 = 0,087	0,087
81 / 286 = 0,28	0,087 + 0,28 = 0,367
36 / 286 = 0,12	0,367 + 0,12 = 0,487
144 / 286 = 0,504	0,487 + 0,504 = 1

Luego se generan cuatro números aleatorios en el rango [0..1], por ejemplo {0,1; 0,6; 0,46; 0,91}. Para estos números los individuos seleccionados son:

Valor	Individuo sleccionado
0,1	(2) ya que 0,087 < 0,1 < 0,367
0,6	(4) ya que 0,487 < 0,6 < 1
0,46	(3) ya que 0,367 < 0,46 < 0,487
0,91	(4) ya que 0,487 < 0,91 < 1

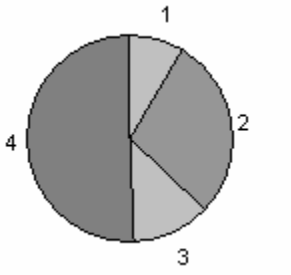

Figura 2.5 Rueda de ruleta del ejemplo, cada cavidad es proporcional probabilidad de selección de la solución.

La rueda de ruleta es uno de los peores métodos de selección, por las siguientes causas:

- La presión selectiva depende exclusivamente del fitness, sobre lo cual no se tiene control.
- Es ineficiente, lo cual puede consumir mucha CPU en algoritmos aplicados sobre grandes poblaciones y en problemas que usan una función de fitness rápida.
- Los individuos menos aptos pueden ser seleccionados más de una vez.

2.3.4.2 Selección por torneo

La selección por torneo es uno de los muchos métodos de selección en algoritmos genéticos que realiza una "competencia" entre unos pocos individuos elegidos al azar de una población y selecciona un ganador.

Hay dos versiones de la selección por torneo: la probabilística y la determinística.
El algoritmo de la versión determinística es el siguiente:

- Mezclar los individuos de la población.
- Escoger un número *k* de individuos (típicamente 2).
- Compararlos con respecto a su aptitud.
- El ganador del "torneo" es el individuo más apto.
- Debe mezclarse la población un total de *k* veces para seleccionar *N* padres (donde *N* es el tamaño de la población).

Nótese que si $k = 1$, el proceso selecciona a todos los individuos de forma aleatoria para la posterior reproducción.

Ejemplo con k=2:

Individuos	Fitness	Mezcla 1	Ganadores	Mezcal 2	Ganadores
(1)	254	(2)		(4)	
(2)	47	(6)	(6)	(1)	(1)
(3)	457	(1)		(6)	
(4)	194	(3)	(3)	(5)	(6)
(5)	85	(5)		(2)	
(6)	310	(4)	(4)	(3)	(3)

Las parejas de padres resultantes son: (6) y (1), (3) y (6), (4) y (3).

La versión probabilística es idéntica a la anterior, excepto por el paso en que se escoge al ganador. En vez de seleccionar siempre al individuo con aptitud más alta, se aplica una función booleana *f(p)* que devuelve verdadero con probabilidad *p* y falso con probabilidad 1-*p*. Si el resultado es verdadero, se selecciona al más apto. De lo contrario, se selecciona al menos apto. El valor de *p* permanece fijo a lo largo de todo el proceso evolutivo y se escoge dentro del rango [0,5 ; 1].

Nótese que si p es igual a uno, la técnica se reduce a la versión determinística.

Ejemplo con p = 0,7 ; k=2 y los mismo individuos que el ejemplo anterior:

Mezcla 1:

(2)
(6) $f(0,7)$ = verdadero, por lo tanto, gana (6)

(1)
(3) $f(0,7)$ = falso, por lo tanto, gana (1)

(5)
(4) $f(0,7)$ = verdadero, por lo tanto, gana (4)

Esta variante reduce un poco la presión de selección, permitiendo que el individuo menos apto gane el torneo.

La selección por torneo tiene varios beneficios:

- La versión determinística garantiza que el mejor individuo será seleccionado k veces.
- Es eficiente, fácil de implementar y trabaja en arquitecturas paralelas
- Puede introducir una presión de selección muy alta (en la versión determinística) porque a los individuos menos aptos no se les da oportunidad de sobrevivir.
- La presión selectiva puede ser fácilmente ajustada cambiando el tamaño del torneo. Si el tamaño de la competencia es alto, los individuos débiles tienen pocas oportunidades de ser seleccionados.

2.3.4.3 Selección por Estado Uniforme

En este tipo de selección, la descendencia de los individuos seleccionados en cada generación vuelve al acervo genético preexistente, reemplazando a algunos de los miembros menos aptos de la siguiente generación. Se conservan algunos individuos entre generaciones.

2.3.4.4 Selección Elitista

En este tipo de selección, se garantiza la selección de los miembros más aptos de cada generación. La mayoría de los GAs no utiliza elitismo puro, sino que usan una forma modificada mediante la cual el mejor individuo, o algunos de los mejores, son copiados hacia la siguiente generación en caso de que no surja nada mejor.

2.3.4.5 Selección Generacional

En la selección generacional, la descendencia de los individuos seleccionados en cada generación se convierte en la siguiente generación. No se conservan individuos entre las generaciones.

2.3.4.6 Selección Jerárquica

En la selección jerárquica, los individuos atraviesan múltiples rondas de selección en cada generación. Las evaluaciones de los primeros niveles son más rápidas y menos discriminatorias, mientras que los que sobreviven hasta niveles más altos son evaluados más rigurosamente. La ventaja de este método, es que reduce el tiempo total de cálculo porque elimina a la mayoría de los individuos que se muestran poco o nada prometedores en etapas tempranas. Luego somete a una evaluación de aptitud más rigurosa y más costosa sólo a los que sobreviven a esta prueba inicial.

2.3.5 Reproducción

El siguiente paso de un algoritmo genético es generar una segunda población a través de los operadores genéticos de crossover (o recombinación) y mutación. Dicha población conformará una nueva generación de posibles soluciones.

Las nuevas soluciones se crean a partir de un par de progenitores mediante la aplicación de operadores de crossover y mutación. Estas soluciones, comúnmente comparte muchas de las características de sus padres pero en ocasiones pueden ser diferentes. El proceso de reproducción continúa hasta obtener una nueva población de soluciones de un tamaño apropiado. La selección de padres se realiza por medio de alguno de los métodos descriptos anteriormente.

El proceso finalmente resulta en una nueva población de cromosomas diferente a la población de la generación inicial. Comúnmente, el fitness promedio de la población se incrementará, dado que los mejores organismos de cada generación tienen más probabilidades de ser seleccionados para reproducirse.

2.3.5.1 Crossover

Como en la vida real, los individuos de un algoritmo genético pueden formar parejas y producir descendencia. Esto es logrado usando el operador de crossover. Este operador toma dos cromosomas y combina su información genética para producir descendientes. El objetivo de este proceso es combinar de forma eficiente dos individuos para producir otros aún mejores. Esto corresponde a compartir el conocimiento de puntos diferentes en el espacio de búsqueda. El crossover no necesariamente se aplica a todos los individuos. El usuario tiene que definir la probabilidad de aplicar el operador antes de comenzar el algoritmo genético.

El crossover es visto con frecuencia como la fuerza impulsora detrás de la evolución, mientras la mutación es solo una operación secundaria. Algunas comunidades en computación evolutiva, evitan usar la mutación y se concentran solamente en el crossover.

El crossover es un operador utilizado para variar la estructura de uno o más cromosomas de una generación a la siguiente. Existen muchas técnicas de crossover dependiendo de la estructura que usan las representaciones de los cromosomas. Entre los más comunes se encuentran el crossover de un punto, el de dos puntos y el uniforme.

2.3.5.2 Crossover de un punto

El método consiste en seleccionar un único punto de cruce al azar. Este, se aplica a ambos padres dividiendo a cada uno en dos partes. Cada hijo hereda la primer parte de un padre y la segunda parte, a partir del punto de cruce, se completa con los datos del otro. (Véase Figura 2.6)

Por ejemplo, para los siguientes progenitores:
v_1 = (1000|**10111011**) y v_2 = (0110|**00101000**), se obtiene la siguiente descendencia:
v_1' = (1000|**00101000**) y $v_{2'}$ = (0110|**10111011**).

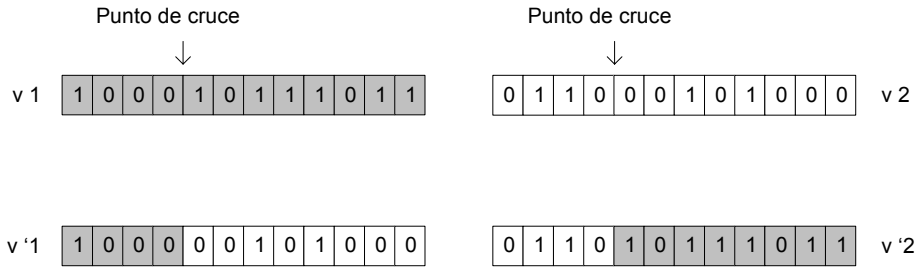

Figura 2.6: Crossover de un punto.

2.3.5.3 Crossover de dos puntos

En este método se seleccionan dos puntos de cruce en los individuos progenitores, quedando divididos en tres partes. Cada hijo se obtiene copiando, posición a posición, la primer y ultima parte de un padre, y la parte central del otro.
Por ejemplo, para v_1 = (1000|**1011**|1011) y v_2 = (0110|**0010**|1000), se obtiene la siguiente descendencia: v_1' = (1000|**0010**|1011) y $v_{2'}$ = (0110|**1011**|1000). (Véase Figura 2.7)

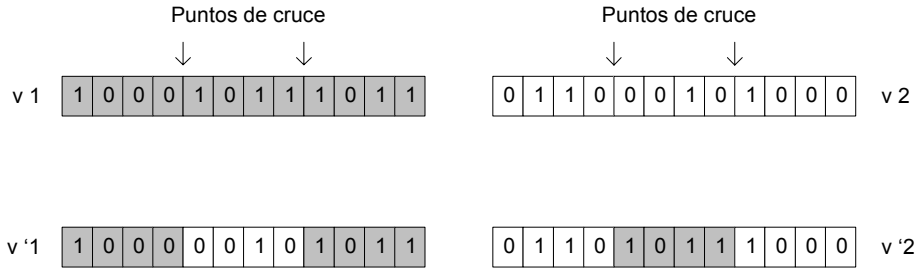

Figura 2.7: Crossover de dos puntos.

2.3.5.4 Crossover multipunto

El crossover de dos puntos puede ser generalizado fácilmente a n puntos de cruce. Estos n puntos generan n+1 segmentos en los cromosomas padres. Los segmentos en las posiciones pares son intercambiados para obtener así los dos descendientes. Por ejemplo para:
v_1 = (100|**010**|111|**011**) y v_2 = (011|**000**|101|**000**), se obtiene la siguiente descendencia:
v_1' = (100|**000**|111|**000**) y $v_{2'}$ = (011|**010**|101|**011**). (Véase Figura 2.8)

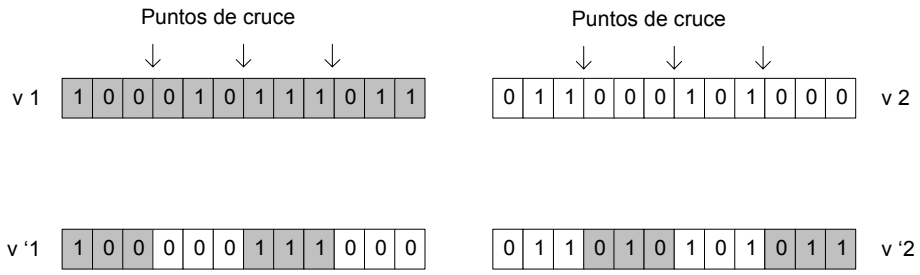

Figura 2.8: Crossover multipunto.

2.3.5.5 Crossover uniforme

El crossover uniforme es una generalización del crossover de un punto, de dos puntos y multipunto. Por cada bit en los cromosomas padres se genera un número aleatorio entre cero y uno, si éste supera cierto umbral *p* los bits son intercambiados. (Véase Figura 2.9)
Por ejemplo para *p* = 0,5, las cadenas v_1 = (100010111011) y v_2 = (011000101000), y los siguientes números aleatorios (0,1; 0,2; 0,6; 0,1; 0,8; 0,7; 0,8; 0,4; 0,3; 0,6; 0,1; 0,9)
se obtiene la siguiente descendencia:

v_1' = (1_1 0_1 **1_2** 0_1 **0_2** 0_2 1_2 1_1 1_1 0_2 1_1 **0_2**) y v_2' = (0_2 1_2 **0_1** 0_2 **1_1** 0_1 1_1 0_2 1_2 0_1 0_2 **1_1**).

Los subíndices 1 y 2 indican los padres (v_1 y v_2) que aportaron el bit . Las posiciones de los caracteres en negrita indican donde hubo un cambio de valor con respecto al padre.

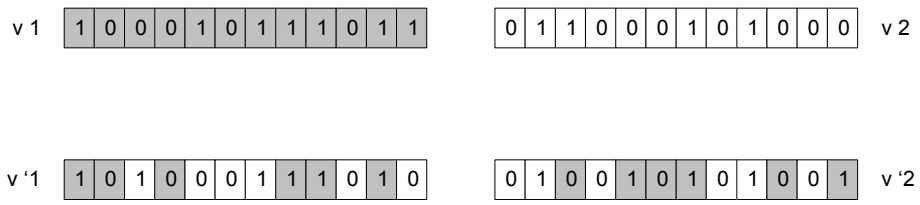

Figura 2.9: Crossover uniforme.

Como el crossover uniforme intercambia bits en lugar de segmentos, éste puede combinar aspectos independientemente de su posición relativa. Para algunos problemas esta habilidad puede pesar más que la desventaja de destruir los bloques constructores. Sin embargo, para otros problemas, el crossover uniforme tiene menor desempeño que el crossover de dos puntos.

2.3.5.6 Mutación

La mutación en GAs simula los cambios en el ADN (ácido desoxirribonucleico) aparentemente accidentales de un individuo. Algunas veces, tales cambios son beneficiosos y resultan en incrementos del fitness de los individuos. En la mayoría de los casos, sin embargo, tienden a reducirlo. El operador de selección previene que este tipo de individuos pase a la siguiente generación.

La mutación es una parte importante de la evolución, dado que ésta ayuda a introducir cambios a la población y permite mantener diversidad en la misma. Como resultado, se pueden evitar óptimos locales y nuevas partes del espacio pueden ser exploradas. Sin mutación, la población podría volverse más y más similar, lo que provocaría una desaceleración o una detención en el proceso evolutivo. Este razonamiento también explica porqué la mayoría de los algoritmos genéticos evita solo tomar los individuos más calificados de una población para generar la siguiente en favor de una selección aleatoria (o casi aleatoria) con inclinación hacia esos que son más calificados.

El tipo de mutación usada, depende en gran parte de la codificación de los cromosomas. La mutación de cadenas binarias se realiza generalmente de modo que cada bit sea invertido con una cierta probabilidad p_m. Esta probabilidad es muy pequeña en comparación a la probabilidad del crossover. Generalmente, está en el rango [0.005; 0,1]. La mutación intenta introducir pequeños cambios en un individuo. Como resultado, las zonas cercanas a buenos puntos en el espacio de búsqueda son analizadas para encontrar mejores soluciones.

Desgraciadamente, la cantidad de cambios introducidos en las características de un individuo no depende directamente de la probabilidad de la mutación. La razón de ello, se debe a la fuerte dependencia del peso de un gen con su posición en el cromosoma. Por ejemplo, si en un cromosoma formado por cuatro dígitos binarios cuyo fenotipo es un determinado número decimal se invierte el primer bit, se genera un gran salto en el nuevo número decimal representado. Esto puede ser visto si se toma el número binario 0000, el cual corresponde al número decimal 0. Invirtiendo el primer bit lo transforma al número binario 1000, el cual corresponde al número decimal 8. Para evitar que este fenómeno ocurra, existen mecanismos alternativos de codificación más apropiados.

2.3.6 Reemplazo

En un algoritmo genético simple, la descendencia producida por los operadores genéticos forma una nueva población que reemplaza completamente la población vieja. Esto significa, que no hay solapamiento entre las generaciones. Los individuos de la vieja población y de la nueva no compiten entre ellos. Los algoritmos genéticos que utilizan este tipo de reemplazo son también conocidos como algoritmos genéticos generacionales.

En contraste, los algoritmos genéticos estables utilizan poblaciones solapadas. En cada generación, una porción de la población es reemplazada por la nueva descendencia generada. Una estrategia de reemplazo define que miembros de la población actual son

forzados a morir para hacer lugar a la nueva descendencia. Esta estrategia de reemplazo podría estar basada en diferentes criterios. Una forma de realizar el reemplazo es remover individuos que tienen valores de fitness bajos. Otras estrategias de reemplazo intentan preservar la diversidad de la población, removiendo individuos con baja contribución a la diversidad.

En la mayoría de los algoritmos genéticos estables uno o dos miembros nuevos son insertados en la población. Sin embargo, cualquier número menor que el tamaño de la población podría ser usado para determinar la cantidad de solapamiento entre dos generaciones consecutivas. En un caso extremo, donde se reemplaza la población entera, el algoritmo estable se vuelve un algoritmo genético simple.

2.3.7 Terminación

Los pasos explicados anteriormente son repetidos hasta que una condición de terminación es alcanzada. Condiciones de terminación comunes son:

- Se encuentra una solución que satisface un criterio mínimo.
- Se alcanza un número fijo de generaciones.
- Utilización máxima de recursos alcanzado.
- La solución con el fitness más alto ha sido encontrada, o se ha alcanzado una meseta tal que sucesivas iteraciones no logran producir mejores resultados.
- Inspección manual.
- Combinaciones de las condiciones anteriores.

2.3.8 Optimización de una función simple

En esta sección se discutirán los aspectos básicos de un algoritmo genético para la optimización de una función simple de una variable y será útil a modo de ejemplo para entender su funcionamiento. La función sobre la cual se trabajará, representada en la Figura 2.10, es la siguiente:

$$f(x) = x \,.\, \text{sen}(\, 10\pi.x\,) + 1{,}0$$

El problema es encontrar un valor de x en el rango [-1..2] que maximice la función f, en otras palabras encontrar un valor x_0 tal que

$$f(x_0) \geq f(x), \text{ para todo } x \in [-1..2]$$

Es un tanto complicado analizar la función f para determinar sus máximos y mínimos en el rango de interés. Primero hay que determinar cuando la derivada f'es igual a 0:

$$f'(x) = \text{sen}(10\pi.x) + 10\pi x \,.\, \cos(10\pi.x) = 0;$$

la fórmula es equivalente a:

$$\tan(10\pi.x) = -10\pi.x$$

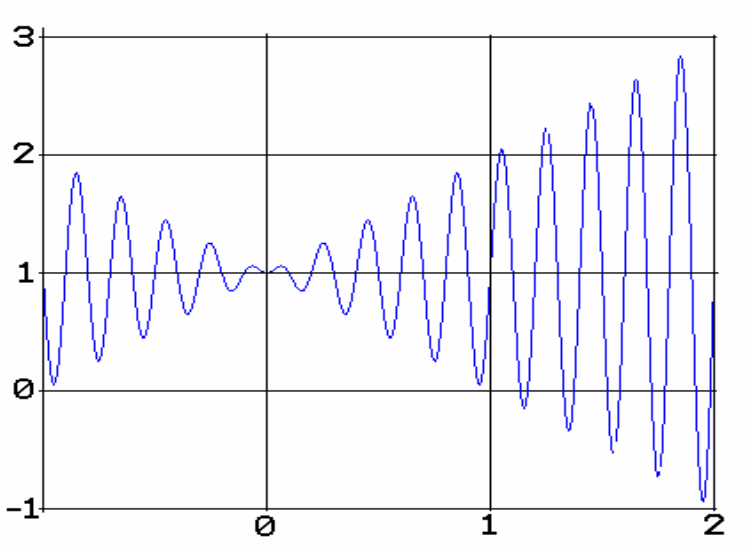

Figura 2.10. Gráfico de la función $f(x) = x \,.\, \text{sen}(\, 10\pi \,.\, x) + 1{,}0$

Es claro que la ecuación de arriba tiene infinitas soluciones. Dichas soluciones pueden ser obtenidas con las siguientes fórmulas:

$x_i = (2i - 1) / 20 + e_i$, para $i >= 1$
$x_i = 0$, para $i = 0$;
$x_i = (2i + 1) / 20 - e_i$, para $i <= -1$

Donde los términos e_i son valores muy pequeños que hacen que las x_i se correspondan exactamente con el centro de cada pico. Nótese también que ***f*** alcanza sus máximos locales para $\boldsymbol{x_i}$ si **i** es un entero impar, y sus mínimos locales para $\boldsymbol{x_i}$ si **i** es un entero par (véase Figura 2.10).

Dado que el dominio del problema es $x \in [-1..2]$, la función alcanza su máximo para $x_{19} = 37/20 + e_{19} = 1{,}85 + e_{19}$, donde $f(x_{19})$ es apenas mayor que $f\,(1{,}85) = 2{,}85$.

Supóngase que se quiere construir un algoritmo genético que permita resolver el problema anterior, por ejemplo maximizar la función f en el rango [-1..2]. En la sección siguiente se discutirán las partes principales de este algoritmo.

2.3.8.1 Representación del cromosoma

Los cromosomas estarán representados por cadenas binarias. Estas cadenas codificarán los números reales pertenecientes al dominio de la función *f* en el rango [-1..2]. La longitud de la cadena depende de la precisión deseada para los números. En este ejemplo se usarán seis dígitos después de la coma decimal, lo cual permite obtener entre un número entero y el siguiente 1000000 de números. El dominio de la variable x tiene longitud 3; la precisión requerida implica que el rango [-1..2] debería ser dividido en 3 * 1000000 lugares iguales. Por lo tanto se necesita una cadena binaria que permita representar 3000000 de números. Esto se puede lograr perfectamente con 22 bits.

$$2097152 = 2^{21} < 3000000 < 2^{22} = 4194304.$$

El mapeo de un cadena binaria $<b_{21}\ b_{20}...b_0>$ a un número real es trivial y se hace en dos etapas:

- Se convierte la cadena binaria $<b_{21}\ b_{20}...b_0>$ de base 2 a base 10:
  $(<b_{21}\ b_{20}...b_0>)_2 = (\Sigma_{i=0...21}\ bi * 2^i)_{10} = x'$

- Se obtiene el número real x correspondiente a x'
  $x = -1{,}0 + (x'/ (2^{22}-1)) * 3,$
  donde –1,0 es el límite izquierdo del dominio y 3 es la longitud.

Por ejemplo, un cromosoma (1000101110110101000111) representa el número 0,637197, dado que

$$x' = (1000101110110101000111)_2 = 2288967_{10}$$

y

$$x = -1.0 + (2288967 / 4194303) * 3 = 0{,}637197.$$

Queda claro que los cromosomas:

(00000000000000000000)

y

(11111111111111111111)

representan los límites del dominio, -1,0 y 2,0 respectivamente.

2.3.8.2 Población inicial

El proceso de inicialización es muy simple, se crea una población de cromosomas en la cual cada cromosoma es una cadena binaria de 22 bits, cada bit es inicializado de forma aleatoria.

2.3.8.3 Función de evaluación

La función de evaluación "eval" de un cromosoma **c** es equivalente a la función f:

$$\text{eval}(\mathbf{c}) = f(x),$$

donde el cromosoma **c** representa el valor real x. Como se mencionó anteriormente, la función de evaluación juega el rol del ambiente, clasificando potenciales soluciones en términos de su fitness. Por ejemplo tres cromosomas:

$$v_1 = (10001011101101010001 11),$$

$$v_2 = (00000011100000000010000),$$

$$v_3 = (11100000000111111000101),$$

corresponden a valores x_1 = 0,637197, x_2 = -0958973, y x_3 = 1627888, respectivamente. Consecuentemente, la función de evaluación evalúa cada uno de ellos de la siguiente manera:

$$eval(v_1) = f(x_1) = 1{,}586345,$$

$$eval(v_2) = f(x_2) = 0{,}078878,$$

$$eval(v_3) = f(x_3) = 2{,}250650.$$

Claramente, el cromosoma v_3 es el mejor de los tres cromosomas, puesto que la función *eval* devuelve el valor más alto.

2.3.8.4 Operadores genéticos

Durante la fase de alteración del algoritmo genético, se usarán dos operadores clásicos, ellos son crossover y la mutación.

Como se mencionó anteriormente, la mutación altera uno o más genes (posiciones en un cromosoma) con una probabilidad igual a la tasa de mutación predefinida. Asúmase que el quinto gen del cromosoma v_3 fue seleccionado para ser mutado. Puesto que ese gen en ese cromosoma es 0, éste tomaría el valor 1 después de sufrir la mutación. De este modo el cromosoma v_3 quedaría así:

$$v_3' = (11101\mathbf{1}00000111111000101).$$

Este cromosoma representa el valor x_3'=1,721638 y $f(x_3') = -0{,}082257$. Lo cual indica que esta mutación en particular resulta en un decremento significativo del valor del cromosoma v_3. Por otro lado, si el décimo gen fuera seleccionado para ser mutado en el cromosoma v_3, entonces:

v_3'' = (11100000**1**111111000101).

El valor correspondiente es x_3''= 1,630818 y $f(x_3$'') = 2,343555, lo cual representa una mejora sobre el valor original $f(x_3)$ = 2,250650.

Ahora se mostrará como funciona el operador de crossover sobre los cromosomas v_2 y v_3. Asúmase que el punto de cruce (aleatoriamente seleccionado) resultó estar después del quinto gen:

v_2 = (**00000 | 01110000000010000**),

v_3 = (11100 | 00000111111000101).

Los dos descendientes resultantes son:

v_2' = (**00000** | 00000111111000101),

v_3' = (11100 | **01110000000010000**).

Las evaluaciones para los descendientes son:

$f(v_2$') = f(-0,998113) = 0,940865,

$f(v_3$') = f(1,666028) = 2,459245.

Nótese que el segundo descendiente tiene mejor evaluación que cualquiera de sus padres.

2.3.8.5 Parámetros

Para este problema en particular se han usado los siguientes parámetros: tamaño de población *tam_pob* = 50, probabilidad de crossover p_c = 0,25 , probabilidad de mutación p_m = 0,01. La siguiente sección presenta algunos resultados experimentales para este algoritmo genético.

2.3.8.6 Resultados experimentales

La Figura 2.11 muestra en una columna los números de generaciones en las cuales hubo una mejora dada por la función de evaluación y en la otra columna el valor de la función.

Número de generación	Funcion de evaluación
1	1,441942
6	2,250003
8	2,250283
9	2,250284
10	2,250363
12	2,328077
39	2,344251
40	2,345087
51	2,738930
99	2,849246
137	2,850217
145	2,850227

Figura 2.11. Resultados de 150 generaciones.

El mejor cromosoma después de 150 generaciones fue:

$$\mathbf{v}_{max} = (111100110100010000101),$$

el cual corresponde al valor $x_{max} = 1{,}850773$.
Como se esperaba $x_{max} = 1{,}85 + e$, y $f(x_{max})$ es apenas mayor que 2,85.

2.3.9 ¿Cómo trabajan los algoritmos genéticos?

En esta sección se presenta el funcionamiento de un algoritmo genético para un problema simple de optimización paramétrico. La maximización de una función es un problema de este tipo.

Se pueden asumir, sin pérdida de generalidad, solo problemas de maximización. Si el problema de optimización fuera minimizar una función *f*, esto sería equivalente a maximizar una función *g*, donde $g = -f$. Por ejemplo,

$$\min f(x) = \max g(x) = \max\{-f(x)\}.$$

También se puede asumir que la función objetivo *f* siempre tiene codominio positivo. En el caso de que una función *g* tenga codominio negativo, se le puede adicionar una constante *C* de modo de desplazarla hacia valores positivos, por ejemplo,

$$\max g(x) = \max\{ g(x) + C \}$$

Supóngase que se quiere maximizar una función de *k* variables, $f(x_1,...,x_k) : R^k \rightarrow R$. Supóngase, además, que cada variable x_i puede tomar valores en el dominio
$D_i = [a_i,b_i] \subseteq R$ y $f(x_1,...,x_k) > 0$ para todo $x_i \in D_i$.

Además, se quiere maximizar la función *f* con alguna precisión requerida, supóngase que se desean seis lugares decimales.

Es claro que para lograr esta precisión cada dominio D_i debería estar dividido en $(b_i - a_i) * 10^6$ lugares iguales. Se define la variable m_i como el valor entero más chico tal que $(b_i - a_i) * 10^6 \leq 2^{mi} - 1$. Entonces una representación teniendo cada variable codificada como una cadena binaria de longitud m_i claramente satisface la precisión requerida. La siguiente fórmula permite interpretar cada una de las cadenas binarias:

$$x_i = a_i + decimal(n_2) * (b_i - a_i) / (2^{mi} - 1),$$

donde *decimal* (n_2) representa el valor decimal de una cadena binaria.

Ahora, cada cromosoma (solución potencial) es representado por una cadena binaria de longitud $m = \Sigma_{i=1..k} m_i$, donde los primeros m_1 bits se mapean a un valor en el rango $[a_1,b_1]$, el siguiente grupo de m_2 bits se mapean a un valor en el rango $[a_2,b_2]$, y así siguiendo hasta el último grupo de m_k bits que se mapean a un valor en el rango $[a_k,b_k]$.
Para crear una población, se puede establecer algún número de cromosomas (*tam_pob*) e inicializar cada uno de forma aleatoria. Sin embargo, si se tiene algún conocimiento acerca de soluciones potenciales, se puede usar esa información para formar la población inicial.

El resto del algoritmo es sencillo, en cada generación se evalúa cada cromosoma (usando la función *f* en la secuencia de variables decodificadas). Luego se selecciona una nueva población de cromosomas con respecto a la distribución de probabilidades basada en su fitness, y se alteran los mismos en la nueva población con operaciones de mutación y crossover. Después de algún número de generaciones, cuando no se observan mejoras, el mejor cromosoma representa una solución óptima, posiblemente la solución global. Generalmente el algoritmo termina después de un número fijo de iteraciones dependiendo de ciertos criterios de velocidad y recursos.

Para el proceso de selección (selección de una nueva población con respecto a la distribución de probabilidades basada en los valores de fitness), una rueda de ruleta puede ser utilizada, dicha rueda está dividida en sectores proporcionales al fitness de cada cromosoma.

Como se explicó en la sección 2.3.4.1, la rueda de ruleta se construye como sigue (se asume que los valores de fitness son positivos, de otra manera, se puede usar algún mecanismo de escalado):

- Calcular el valor de fitness eval(v_i) para cada cromosoma v_i (i = 1,...,*tam_pob*).
- Encontrar el fitness total de la población
  $F = \Sigma_{i=1..tam\_pob}$ eval(v_i).
- Calcular la probabilidad de una selección p_i para cada cromosoma v_i
  (i = 1,...,*tam_pob*): p_i = eval(v_i) / *F*.
- Calcular una probabilidad acumulativa q_i para cada cromosoma v_i
  (i = 1,...,*tam_pob*): $qi = \Sigma_{j=1..i} p_j$.

El proceso de selección se basa en hacer girar la rueda de ruleta *tam_pob* veces; cada vez se selecciona un único cromosoma para la nueva población de la siguiente forma:

- Se genera un numero real *r* en el rango [0..1].
- Si $r < q_1$ entonces se selecciona el primer cromosoma (v_i); si no se selecciona el i-ésimo cromosoma v_i ($2 \leq i \leq tam\_pob$) tal que $q_{i-1} < r < q_i$.

Algunos cromosomas pueden ser seleccionados más de una vez. Esto está demostrado en el teorema de los esquemas (Véase sección 2.3.10): los mejores cromosomas generan más copias, los promedio se mantienen, y los peores mueren.

Ahora se puede aplicar el operador de recombinación, crossover, a los individuos en la nueva población. Como se mencionó antes, uno de los parámetros de un sistema genético es la probabilidad de un crossover p_c. Esta probabilidad representa el número esperado $p_c * tam\_pob$ de cromosomas que sufrirán una recombinación. Se procede de la siguiente manera:

Por cada cromosoma en la nueva población:

- Generar un número aleatorio real *r* en el rango [0..1];

- Si $r < p_c$, seleccionar dicho cromosoma para crossover.

Ahora se forman parejas de cromosomas seleccionados al azar: por cada par de cromosomas se genera un número entero aleatorio *pos* en el rango [1..*m*-1] (m es la longitud total –número de bits– en el cromosoma). El número *pos* indica la posición del punto de cruce. Dos cromosomas

$$(b_1 b_2 ... b_{pos} b_{pos+1} ... b_m)$$

y

$$(c_1 c_2 ... c_{pos} c_{pos+1} ... c_m)$$

son reemplazados por un par de su descendencia :

$$(b_1 b_2 ... b_{pos} c_{pos+1} ... c_m)$$

y

$$(c_1 c_2 ... c_{pos} b_{pos+1} ... b_m)$$

El siguiente operador que se aplica, es la mutación. Esta operación, se realiza bit por bit utilizando otro de los parámetros de un sistema genético, la probabilidad de mutación p_m. Este parámetro representa la probabilidad del número esperado de bits mutados $p_m * m * tam\_pob$. Cada bit (de todos los cromosomas de toda la población) tiene la misma chance de sufrir una mutación, por ejemplo, cambiar de 0 a 1 o viceversa. Ahora se procede de la siguiente manera.

Por cada cromosoma en la población actual (por ejemplo después de un crossover) y por cada bit dentro del cromosoma:

- Se genera un número real aleatorio r en el rango [0..1];
- Si $r < p_m$, mutar el bit.

Después de realizadas las operaciones de selección, crossover y mutación, la nueva población está lista para su siguiente evaluación. Esta evaluación es usada para construir la distribución de probabilidades (para el siguiente proceso de selección), por ejemplo, para la construcción de una rueda de ruleta cuyos sectores sean de tamaño relativo a los fitness obtenidos. El resto de la evolución es una repetición cíclica de los pasos anteriores.

2.3.10 ¿Por que funcionan los algoritmos genéticos?

Aunque los algoritmos genéticos son simples de describir y programar, su comportamiento puede ser muy complicado, y todavía existen muchas preguntas abiertas acerca de como funcionan y sobre el tipo de problemas en los que son más adecuados.

La teoría tradicional de los algoritmos genéticos (ver [Holland, J. H. 1975],) se basa en el concepto de "bloques constructores". Un bloque constructor es un grupo pequeño y compacto de genes que han evolucionado de tal forma que su introducción en cualquier cromosoma tiene una alta probabilidad de incrementar la aptitud de dicho cromosoma. Los fundamentos teóricos de los GAs, formulan que estos trabajan descubriendo, enfatizando y recombinando buenos bloques constructores de una manera altamente paralelizada.

Para las demostraciones teóricas, Holland utilizó una representación de soluciones mediante cadenas binarias, e introdujo el concepto de esquemas para formalizar la noción de bloques constructores.
Puede decirse que un esquema representa una plantilla que permite la exploración de similitudes entre cromosomas. Este se construye utilizando el símbolo asterisco ($*$) en el alfabeto de los genes. Un esquema representa todas las cadenas (un hiperplano, o un subconjunto del espacio de búsqueda) que coinciden exactamente con éste, en todas las posiciones donde no haya un ' $*$ '.

Por ejemplo, considérense las cadenas y esquemas de longitud 10.
El esquema ($*$111100100) coincide con dos cadenas:

{(0111100100), (1111100100) },

y el esquema ($*1*$1100100) coincide con cuatro cadenas:

{(0101100100), (0111100100) , (1101100100), (1111100100)}.

El esquema (1001110001) representa una cadena sola: (1001110001), y el esquema (∗∗∗∗∗∗∗∗∗) representa todas las cadenas de longitud 10. Es claro que un esquema coincide exactamente con 2^r cadenas, donde r es el número de símbolos asteriscos ' ∗ ' en él. Por otro lado, cada cadena de longitud m está representada por 2^m esquemas. Por ejemplo, considérese la cadena (1001110001). Esta cadena está representada por los siguientes 2^{10} esquemas:

(1001110001) (∗001110001)...(100111000∗)
(∗∗01110001) (∗0∗1110001)...(10011100∗∗)
(∗∗∗1110001).....(∗∗∗∗∗∗∗∗∗∗)

Considerando cadenas de longitud m, hay en total 3^m posibles esquemas. En una población de tamaño n pueden ser representados entre 2^m y $n * 2^m$ esquemas diferentes.

Los esquemas tienen diferentes características entre sí. Ya se aclaró que el número de asteriscos en un esquema determina la cantidad de cadenas que coinciden con éste. Los esquemas tienen dos propiedades importantes, orden y longitud de definición; el Teorema de los Esquemas será formulado sobre la base de estas propiedades.

El *orden* del esquema S (denotado por $o(S)$) es el número de posiciones con 0s y 1s presentes en este. En otras palabras, es la longitud de la plantilla menos el número de símbolos asteriscos. El orden define la especialización de un esquema. Por ejemplo los siguientes tres esquemas, cada uno de longitud 10,

S_1 = (∗∗∗001∗110),
S_2 = (∗∗∗∗00∗∗0∗),
S_3 = (11101∗∗001),

tienen los siguientes órdenes:

$o(S_1) = 6$, $o(S_2) = 3$, y $o(S_3) = 8$,

y el esquema S_3 es el más específico.

La noción de orden de un esquema es útil en el cálculo de las probabilidades de supervivencia de un esquema en las mutaciones; esto se analizará luego en esta sección.

La *longitud de definición* de un esquema S, denotado por $\delta(S)$, es la distancia entre la primera y la última posición fija de la cadena. Esta define la densidad de la información contenida en un esquema. Por ejemplo,

$\delta(S_1) = 10 - 4 = 6$, $\delta(S_2) = 9 - 5 = 4$, y $\delta(S_3) = 10 - 1 = 9$.

Nótese que el esquema con una única posición fija tiene una longitud de definición de cero.

La noción de la longitud de definición de un esquema es útil en el cálculo de las probabilidades de supervivencia de un esquema para el crossover; esto se analizará luego en esta sección.

Como se discutió antes, el proceso de evolución simulado de un algoritmo genético consiste de cuatro pasos repetidos consecutivamente:

$t \leftarrow t + 1$
seleccionar $P(t)$ de P(t - 1)
recombinar $P(t)$
evaluar $P(t)$

El primer paso ($t \leftarrow t + 1$) simplemente mueve el reloj de la evolución hacia delante, durante el último paso (evaluar $P(t)$) solo se evalúa la población actual. El principal fenómeno del proceso evolutivo ocurre en los dos pasos restantes del ciclo: selección y recombinación. A continuación se analizará el efecto de estos dos pasos sobre el número esperado de esquemas representados en la población.

Primero se verá el paso de la selección y se ilustrarán todas las fórmulas con un ejemplo.

Considérese el tamaño de la población *tam*_pob = 20, la longitud de las cadenas (por consiguiente, la longitud de los esquemas) m = 33. Considérese, además, que (en el tiempo t) la población consiste de las siguientes cadenas:

v_1 = (100110100000001111111010011011111)
v_2 = (111000100100110111001010100011010)
v_3 = (000010000011001000001010111011101)
v_4 = (100011000101101001111000001110010)
v_5 = (000111011001010011010111111000101)
v_6 = (000101000010010101001010111111011)
v_7 = (001000100000110101111011011111011)
v_8 = (100001100001110100010110101100111)
v_9 = (010000000101100010110000001111100)
v_{10} = (000001111000110000011010000111011)
v_{11} = (011001111110110101100001101111000)
v_{12} = (110100010111101101000101010000000)
v_{13} = (111011111010001000110000001000110)
v_{14} = (010010011000001010100111100101001)
v_{15} = (111011101101110000100011111011110)
v_{16} = (110011110000011111100001101001011)
v_{17} = (011010111111001111010001101111101)
v_{18} = (011101000000001110100111110101101)
v_{19} = (000101010011111111110000110001100)
v_{20} = (101110010110011110011000101111110)

Se denota por $\xi(S, t)$ al número de cadenas en una población en el tiempo t, representadas por el esquema S. Por ejemplo, para el siguiente esquema

$$S_0 = (****111**********)$$

$\xi(S_0, t) = 3$, dado que hay tres cadenas, llamadas v_{13}, v_{15}, y v_{16}, representadas por el esquema S_0. Nótese que el orden del esquema S_0, $o(S_0) = 3$, y su longitud de definición es $\delta(S_0) = 7 - 5 = 2$.

Otra propiedad de un esquema es su fitness en el tiempo t, $eval(S,t)$. Esta propiedad se define como el fitness promedio de todas las cadenas en la población representada por el esquema S. Si hay p cadenas $\{v_{i1},...,v_{ip}\}$ en la población representadas por un esquema S, entonces:

$$eval(S,t) = \Sigma_{j=1..p}\, eval(v_{ij})/p.$$

Supóngase que la selección se realiza mediante selección proporcional. Durante este paso, se crea una población intermedia de tamaño $tam\_pob = 20$, para lo cual se seleccionan veinte cadenas. Cada cadena es copiada cero, una, o más veces, de acuerdo a su fitness. Como se mencionó en la sección anterior, en la selección de una única cadena, la cadena v_i tiene una probabilidad $p_i = eval(v_i)/F(t)$ de ser seleccionada ($F(t)$ es fitness total de toda la población en el tiempo t, $F(t) = \Sigma_{i=1..20}\, eval(v_i)$).

Después del paso de selección, se espera tener $\xi(S, t + 1)$ cadenas representadas por el esquema S. Dado que primero, para una cadena promedio representada por un esquema S, la probabilidad de su selección (en una selección de una única cadena) es igual a $eval(S,t)/F(t)$, segundo, el número de cadenas representadas por un esquema S es $\xi(S, t)$, y tercero, el número de selecciones de cadenas es $tam\_pob = 20$, entonces:

$$\xi(S, t+1) = \xi(S, t) \,.\, tam\_pob \,.\, eval(S,t)/F(t),$$

esta fórmula se puede rescribir, teniendo en cuenta que el fitness promedio de la población es $\overline{F}(t) = F(t)/tam\_pob$, de la siguiente manera:

$$\xi(S, t+1) = \xi(S, t) \,.\, eval(S,t) \,/\, \overline{F}(t) \qquad (1)$$

En otras palabras, el número de cadenas en la población crece de acuerdo al cociente entre el fitness del esquema y el fitness promedio de la población. Esto significa que un esquema con un fitness superior al promedio recibe un número creciente de cadenas en la próxima generación, un esquema con fitness inferior al promedio recibe un número decreciente de cadenas, y un esquema con fitness igual al promedio permanece en el mismo nivel. El efecto a largo plazo de la regla anterior es también claro. Si se asume que un esquema S permanece arriba del promedio por un ε% (por ejemplo, $eval(S,t) = \overline{F}(t) + \varepsilon \,.\, \overline{F}(t)$), entonces reemplazando en (1) queda:

$$\xi(S, 1) = \xi(S, 0)(1 + \varepsilon),$$
$$\xi(S, 2) = \xi(S, 1)(1 + \varepsilon) = \xi(S, 0)(1 + \varepsilon)(1 + \varepsilon),$$
$$\dots$$
$$\xi(S, t) = \xi(S, 0)(1 + \varepsilon)^t,$$

despejando ε queda: $\varepsilon = (eval(S,t) - \overline{F}(t)) / \overline{F}(t))$, esto implica que ε es mayor que cero cuando el fitness de un esquema supera el promedio y es menor que $\varepsilon < 0$ para esquemas que no lo superan.

Esta es una ecuación de progresión geométrica. Ahora se puede decir no solo que un esquema superior al promedio recibe un número creciente de cadenas en la siguiente generación, sino que también dicho esquema recibe un número de cadenas que crece en forma exponencial en las siguientes generaciones.

La ecuación (1) se conoce como ecuación de crecimiento reproductivo de un esquema.

Para el esquema S_0 del ejemplo, dado que hay tres cadenas, llamadas v_{13}, v_{15}, y v_{16} (en el tiempo t) representadas por éste, el fitness $eval(S_0)$ es:

$$eval(S_0,t) = (27{,}316702 + 30{,}060205 + 23{,}867227) / 3 = 27{,}081378.$$

En el mismo tiempo, el fitness promedio de toda la población es:

$$\overline{F}(t) = \Sigma_{i=1..20}\, eval(v_i) / tam\_pob = 387{,}776822 / 20 = 19{,}388841,$$

y el cociente o razón entre el fitness del esquema S_0 y el fitness promedio de la población es:

$$eval(S_0,t) / \overline{F}(t) = 1{,}396751.$$

Esto significa que si el esquema S_0 permanece arriba del promedio, éste recibe un número de cadenas que se incrementa de forma exponencial en las siguientes generaciones. En particular, si el esquema S_0 permanece arriba del promedio por el factor constante 1,396751, entonces, en el tiempo $t + 1$, se espera tener $3 * 1{,}396751 = 4.19$ cadenas representadas por S_0 (lo más probable es 4 o 5). En el tiempo $t + 2$, se espera tener $3 * 1{,}3967512 = 5{,}85$ cadenas, (muy probablemente 6 cadenas.), etc.

La intuición es que el esquema S_0 define una parte prometedora del espacio de búsqueda y está siendo muestreada de una manera que se incrementa en forma exponencial.

Se chequearán estás predicciones para el ejemplo anterior del esquema S_0. En la población en el tiempo t, el esquema S_0 representa tres cadenas, v_{13}, v_{15}, y v_{16}. Después de un proceso de selección, la nueva población queda conformada por los siguientes cromosomas:

v'_1 = (01100111111011010110000110111100 0) (v_{11})
v'_2 = (10001100010110100111100000111001 0) (v_4)
v'_3 = (00100010000011010111101101111101 1) (v_7)
v'_4 = (01100111111011010110000110111100 0) (v_{11})
v'_5 = (00010101001111111111000011000110 0) (v_{19})
v'_6 = (10001100010110100111100000111001 0) (v_4)
v'_7 = (11101110110111000010001111101111 0) (v_{15})
v'_8 = (00011101100101001101011111100010 1) (v_5)
v'_9 = (01100111111011010110000110111100 0) (v_{11})
v'_{10} = (00001000001100100000101011101110 1) (v_3)
v'_{11} = (11101110110111000010001111101111 0) (v_{15})
v'_{12} = (01000000010110001011000000111110 0) (v_9)
v'_{13} = (00010100001001010100101011111101 1) (v_6)
v'_{14} = (10000110000111010001011010110011 1) (v_8)
v'_{15} = (10111001011001111001100010111111 0) (v_{20})
v'_{16} = (10011010000000111111101001101111 1) (v_1)
v'_{17} = (00000111100011000001101000011101 1) (v_{10})
v'_{18} = (11101111101000100011000000100011 0) (v_{13})
v'_{19} = (11101110110111000010001111101111 0) (v_{15})
v'_{20} = (11001111000001111110000110100101 1) (v_{16})

El esquema S_0 ahora (tiempo $t + 1$) representa cinco cadenas: v'_7, v'_{11}, v'_{18}, v'_{19}, y v'_{20}.

Sin embargo, la selección sola no introduce nuevas soluciones para ser consideradas dentro del espacio de búsqueda; solo copia algunas cadenas para formar una población intermedia. De este modo el segundo paso del ciclo evolutivo, la recombinación, tiene la responsabilidad de introducir nuevos individuos en la población. Esto es logrado a través de dos operadores genéticos, el crossover y la mutación. Ahora se discutirá el efecto de estos operadores en el número esperado de esquemas en la población.

Primero se analizará el crossover y se ejemplificará para su mejor comprensión.
Como se discutió al principio de esta sección, una única cadena de la población, por ejemplo

v'_{18} = (11101111101000100011000000100011 0)

está representada por 2^{30} esquemas; en particular, por estos dos esquemas:

S_0 = (****111**************************)

y

S_1 = (111****************************10)

Supóngase, además, que la cadena v'_{18} fue seleccionada para el crossover junto con v'_{13}, y que el punto de cruce generado es *pos* = 20. Es claro que el esquema S_0 sobrevive al crossover, porque por ejemplo, una de sus descendencias aún es representada por S_0. La razón es que el punto de cruce preserva la secuencia '111' en la quinta, sexta y séptima posición de la cadena en uno de sus descendientes:

$$v'_{18} = (1110\mathbf{111}1101000100011 \mid 0000001000110),$$

y

$$v'_{13} = (0001010000\mathbf{1}001010100 \mid 1010111111011),$$

producen:

$$v''_{18} = (1110\mathbf{111}1101000100011 \mid 1010111111011),$$

y

$$v''_{13} = (0001010000\mathbf{1}001010100 \mid 0000001000110).$$

Por otro lado, el esquema S_1 debería ser destruido porque ninguno de los descendientes coincide con éste. La razón es que las posiciones fijas '111' en el principio de la plantilla y las posiciones fijas '10' en el final quedan dispuestas en diferentes descendientes.

Debe quedar claro que la longitud de definición de un esquema tiene una función significante en la probabilidad de su destrucción y supervivencia. Nótese, que la longitud de definición del esquema S_0 es $\delta(S_0) = 2$, y la longitud de definición del esquema S_1 es $\delta(S_1) = 32$.

En general, se selecciona un punto de crossover uniformemente entre $m-1$ posibles puntos. Esto implica que la probabilidad de destrucción de un esquema S es:

$$p_d(S) = \delta(S)/(m-1),$$

y consecuentemente, la probabilidad de supervivencia es:

$$p_s(S) = 1 - \delta(S)/(m-1).$$

Así, las probabilidades de supervivencia y destrucción de los esquemas del ejemplo S_0 y S_1 son:

$$p_d(S_0) = 2 / 32, \qquad p_s(S_0) = 30 / 32,$$

$$p_d(S_1) = 32 / 32 = 1, \quad p_s(S_1) = 0,$$

con lo cual, queda en evidencia el resultado del ejemplo anterior donde S_1 fue destruido.

La aplicación del crossover a un cromosoma depende de una probabilidad p_c, lo cual implica que no todos los cromosomas son alterados por este operador. Esto significa que la probabilidad de supervivencia de un esquema es en realidad:

$$p_s(S) = 1 - p_c \,.\, \delta(S)/(m\text{-}1).$$

Nuevamente, referido al esquema S_0 del ejemplo y con $p_c = 0{,}25$,

$$p_s(S_0) = 1 - 0{,}25 \,.\, 2 / 32 = 63/64 = 0{,}984375.$$

Nótese también, que aun si se selecciona un punto de cruce entre posiciones fijas en un esquema, existe todavía una chance de que el esquema sobreviva. Por ejemplo, si ambas cadenas v'_{18} y v'_{13} comienzan con '111' y terminan con '10' el esquema S_1 debería sobrevivir al crossover (sin embargo, la probabilidad de ese evento es muy pequeña). Como consecuencia de esto, se debería modificar la fórmula de la probabilidad de supervivencia de un esquema:

$$p_s(S) \geq 1 - p_c \,.\, \delta(S)/(m\text{-}1).$$

De este modo, del efecto combinado de la selección y el crossover se obtiene una nueva forma para la ecuación de crecimiento reproductivo de un esquema:

$$\xi(S\,,\, t+1) \geq \xi(S\,,\, t) \,.\, eval(S,t) \,/\, \overline{F}\,(t) \;\; [1 - p_c \,.\, \delta(S)/(m\text{-}1)]. \qquad (2)$$

La ecuación (2) se refiere al número esperado de cadenas que estarán representadas por un esquema S en la siguiente generación en función del número real de cadenas que coinciden con el esquema, el fitness relativo del mismo, y su longitud de definición. Es claro que los esquemas que superan el promedio con una longitud de definición pequeña deberían aun ser muestreados a una velocidad que se incrementa en forma exponencial. Para el esquema S_0:

$$eval(S_0,t))/\,\overline{F}\,(t) \;\; [1 - p_c \,.\, \delta(S_0)/(m\text{-}1)] = 1{,}396751 * 0{,}984375 = 1{,}374927.$$

Esto significa que el esquema S_0 el cual apenas supera el promedio debería aun recibir un numero de cadenas que aumente exponencialmente en las siguientes generaciones: en el tiempo (t + 1) se espera tener 3 x 1,374927 = 4,12 cadenas representadas por S_0 (solo apenas más chico que 4,19, un valor considerado solo con selección), en el tiempo (t + 2) se espera tener $3 * 1{,}374927^2 = 5.67$ cadenas (nuevamente, apenas menor que 5,85).

El siguiente operador que se analizará es el de mutación. Este operador cambia aleatoriamente una única posición dentro de un cromosoma con una probabilidad p_m. El cambio es de un 0 por un 1 y viceversa. Es claro que todas las posiciones fijas de un esquema deben permanecer sin cambios si el esquema sobrevive a la mutación. Por ejemplo, considérese nuevamente una única cadena de la población v'_{19}:

$$v'_{19} = \; (11101110110111000010001111101111 0)$$

y el esquema S_0:

$S_0 =$ (****111**********)

Si la cadena v'_{19} sufre una mutación que al menos cambie un bit, por ejemplo el de la posición 8, entonces su descendiente tiene la siguiente forma:

v''_{19} = (11101110**0**10111000010001111110111**0**)

el cual aún está representado por el esquema S_0. Si la posición seleccionada para la mutación estuviese entre la 1 y la 4 o entre la 8 y la 33, la descendencia resultante seguiría estando representada por S_0. Solo tres bits (quinto, sexto y séptimo – las posiciones fijas del esquema S_0) son importantes, la mutación de alguno de estos bits destruirían el esquema S_0. Claramente el número de estos tres bits relevantes es igual al orden del esquema S_0.

Dado que la probabilidad de la alteración de un único bit es p_m. La probabilidad de supervivencia de un único bit es 1 - p_m. Una simple mutación es independiente de otras mutaciones, de esta manera la probabilidad de que un esquema S sobreviva a una mutación es:

$$p_s(S) = (1 - p_m)^{o(S)}.$$

Nuevamente, referido al ejemplo del esquema S_0 y $p_m = 0{,}01$, p_s quedaría:

$$p_s(S_0) = (1 - 0{,}01)^3 = 0{,}9702.$$

El efecto combinado de la selección, crossover y mutación permite obtener una nueva fórmula de la ecuación de crecimiento reproductivo de un esquema, la cual tiene la siguiente forma:

$$\xi(S, t+1) \geq \xi(S, t) \cdot eval(S,t)/\overline{F}(t) \cdot [1 - p_c \cdot \delta(S)/(m-1)] \cdot (1 - p_m)^{o(S)} \qquad (3)$$

La ecuación (3) indica el número esperado de cadenas que estarán representadas por un esquema S en la siguiente generación en función del número real de cadenas que coinciden con el esquema, el fitness relativo del mismo, su longitud de definición y su orden. Nuevamente es claro que un esquema que supere el fitness promedio, con una longitud de definición pequeña y un orden bajo debería ser muestreado a una velocidad que se incremente exponencialmente.

Por ejemplo para el esquema S_0:

$$eval(S,t)/\overline{F}(t) \cdot [1 - p_c \cdot \delta(S)/(m-1)] \cdot (1 - p_m)^{o(S)} = 1{,}396751 \cdot 0{,}984375 \cdot 0{,}9702 = 1{,}3339$$

Esto significa que el esquema S_0, que supera el fitness promedio, con una longitud de definición pequeña y un orden bajo, debería esperar un número de cadenas que se incremente exponencialmente en las siguientes generaciones. En el tiempo (t + 1) se espera tener 3 $*$ 1,3339 = 4,0 cadenas representadas por S_0 (no muchas meno que 4,19 – un valor obtenido solo con selección, o que 4,12 – un valor obtenido con selección y crossover), en

el tiempo ($t + 2$) se espera tener $3 * 1{,}3339^2 = 5{,}33$ cadenas (nuevamente, no muchas menos que 5,85 o 5,67).
Nótese que la ecuación (3) está basada en la asunción de que la función de fitness f retorna solo valores positivos. Cuando se aplican algoritmos genéticos a problemas de optimización donde la función a optimizar puede retornar valores negativos, se requiere algún mapeo adicional entre optimización y función de fitness.

En resumen, la ecuación de crecimiento (1) muestra que la selección incrementa la velocidad de muestreo de esquemas que superan el fitness promedio, y que este cambio es exponencial. El muestreo en si mismo no introduce nuevos esquemas (no representado en el muestreo inicial t = 0). Por esta causa se introduce el operador de crossover, el cual asegura el intercambio de información en forma aleatoria y estructurada. Además, el operador de mutación introduce una gran variabilidad dentro de la población. El efecto combinado (destructor) de estos operadores sobre un esquema no es significante si el mismo tiene una longitud de definición pequeña y un orden bajo. El resultado final de la ecuación de crecimiento (3) puede ser enunciado como:

Teorema 1 (Teorema de los Esquemas)

Los esquemas con fitness superior al promedio, longitud de definición pequeña y orden bajo, reciben un incremento exponencial de instancias en generaciones subsiguientes.
Un resultado inmediato de este teorema es que los algoritmos genéticos exploran el espacio de búsqueda cerca de esquemas con longitud de definición pequeña y orden bajo, los cuales, posteriormente son usados para el intercambio de información en el crossover:

Hipótesis 1 (Hipótesis de los bloques constructores)

Un algoritmo genético busca desempeños cercanos al óptimo, a través de la recombinación de esquemas con longitud de definición pequeña, orden bajo y alto desempeño, denominados bloques constructores.

Ya se ha visto un ejemplo de un bloque constructor en esta sección:

$$S_0 = (****111**********)$$

S_0 es un esquema corto, de bajo orden, que (al menos en poblaciones tempranas) superó el fitness promedio. Este esquema contribuye en el descubrimiento del óptimo.

Aunque se han hecho algunas investigaciones para probar esta hipótesis [Bethke, A.D.: 1980], para la mayoría de las aplicaciones no triviales se confía generalmente en los resultados empíricos. Durante los últimos 15 años muchas aplicaciones de algoritmos genéticos, que se apoyaban en la hipótesis de los bloques constructores, fueron desarrolladas para diferentes dominios de problemas. No obstante, la hipótesis sugiere que el problema de codificación en un algoritmo genético es crítico para su desempeño, y que tal codificación debería satisfacer la idea de los bloques constructores.

Anteriormente se mencionó que en una población de *tam_pob* individuos de longitud *m* se procesan como mínimo 2^m y como máximo *tam_pob* * 2^m esquemas. Algunos de ellos son procesados de una manera útil: estos son muestreados a una velocidad (deseable) que crece exponencialmente, y no son desestabilizados por las operaciones de crossover y mutación (lo cual puede suceder para esquemas con longitud de definición grande y alto orden).

Holland [Holland, J. H.: 1975] mostró, que al menos tam_pob^3 de ellos son procesados útilmente. A esta propiedad la llamó paralelismo implícito, dado que ello es obtenido sin ningún requerimiento extra de procesamiento o memoria. Esto significa que, en una cierta generación, mientras el GA está evaluando explícitamente las aptitudes de las *tam_pob* cadenas de la población, también está estimando implícitamente las aptitudes promedio de un número mucho mayor de esquemas. Es interesante notar que en una población de *tam_pob* cadenas hay mucho más que *tam_pob* esquemas que las representan.

En esta sección se han presentado algunas explicaciones convencionales de porque funcionan los algoritmos genéticos. Nótese, sin embargo, que la hipótesis de los bloques constructores es solo un artículo de fe. Existen algunos ejemplos en los cuales no se cumple. Por ejemplo, asúmanse que los esquemas cortos de orden bajo:

S_0 = (111********) y S_1 = (*********11).

superan el promedio, pero su combinación

S_3 = (111******11),

es de menor aptitud que el esquema

S_4 = (000******00).

Asúmase, además, que la cadena óptima es s_0 = (11111111111), la cual coincide con S_3. Un algoritmo genético puede tener algunas dificultades en converger a s_0, dado que éste puede tender hacia puntos como (00011111100), debido a que el esquema S_4 generará un número mayor de cadenas que S_3 por tener un fitness mayor.
Este fenómeno se conoce como decepción [Goldberg, D. E: 1989]. La decepción puede desorientar a un algoritmo genético y causar su convergencia a puntos no óptimos porque la combinación de buenos bloques constructores provoca una reducción del fitness en lugar de un incremento.

El fenómeno de decepción está fuertemente conectado con el concepto de *epístasis*, el cual (desde el punto de vista de los algoritmos genéticos) significa que existe una fuerte interacción entre los genes de un cromosoma. En otras palabras la epístasis mide la magnitud con la cual la contribución al fitness de un gen depende del valor de otros genes. Para un problema dado, un alto grado de epístasis significa que los bloques constructores pueden no formarse, por lo tanto, el problema sufrirá del fenómeno de decepción.

2.3.11 Codificación de cromosomas

Los algoritmos genéticos requieren para su funcionamiento la utilización de cromosomas. Un cromosoma codifica un conjunto de parámetros en sus genes, dichos parámetros definen las soluciones potenciales del problema que el algoritmo genético intenta resolver. Los cromosomas generalmente son representados por cadenas simples aunque existen otras estructuras. Para la codificación se pueden usar alfabetos de distintas cardinalidades, siendo el más común el binario compuesto de 0s y 1s.

Por ejemplo para un problema con dos parámetros que toman valores en el rango [0..7] el parámetro p_1 puede ocupar las posiciones de la 0 a la 2 y el parámetro p_2 de la 3 a la 5. El número de bits usado para cada parámetro dependerá de la precisión que se requiera en el mismo, para el caso del ejemplo alcanza con 3 bits para representar los valores en el rango [0..7].

Hay otras codificaciones posibles, usando alfabetos de diferente cardinalidad, como por ejemplo el alfabeto real; sin embargo, uno de los resultados fundamentales en la teoría de algoritmos genéticos, el *teorema de los esquemas* (ver sección 2.3.10), afirma que la codificación óptima, es decir, aquella sobre la que los algoritmos genéticos funcionan mejor, es mediante un alfabeto de cardinalidad 2.

La mayoría de las veces, una codificación correcta es la clave de una buena resolución del problema. Generalmente, la regla heurística que se utiliza es la llamada *regla de los bloques constructores*, es decir, parámetros relacionados entre sí deben estar cercanos en el cromosoma.

Las aplicaciones que utilizan algoritmos genéticos tienen algunos problemas que obstaculizan, si no prohíben, encontrar soluciones óptimas con la precisión requerida. Una de las implicaciones de estos problemas es la convergencia prematura de toda la población; otras consecuencias incluyen la inhabilidad de hacer refinamientos locales y la inhabilidad de operar en presencia de restricciones no triviales.

La representación binaria comúnmente usada en algoritmos genéticos tiene algunos inconvenientes cuando se aplica a problemas numéricos multidimensionales que requieren mucha precisión. Por ejemplo, para 100 variables con dominios en el rango de [-500, 500] donde se requiere una precisión de 6 dígitos después de la coma decimal, la longitud del cromosoma en forma binaria será de 3000. Esto, a su vez, genera un espacio de búsqueda de aproximadamente 10^{1000} puntos. En este tipo de problemas los algoritmos genéticos funcionan deficientemente.

El alfabeto binario ofrece el máximo número de esquemas (ver sección 2.3.10) para el procesamiento genético que cualquier otra codificación [Goldberg, D. E: 1989] y por lo tanto la representación de soluciones en cadenas de bits ha dominado las investigaciones sobre algoritmos genéticos. Esta codificación también facilita el análisis teórico y permite

el diseño y uso de operadores genéticos elegantes. Sin embargo, se demostró empíricamente que puede ser importante experimentar con alfabetos de mayor cardinalidad y con operadores genéticos nuevos. En particular, para problemas de optimización paramétricos con variables sobre dominios continuos, se puede experimentar con genes codificados con números reales junto con operadores genéticos especialmente desarrollados para este tipo de codificación.

En [Goldberg, D. E.: 1990], Goldberg escribió:

"El uso de genes con codificación real o de punto flotante tiene una larga y controversial historia en genética artificial y en esquemas de búsqueda evolutiva. Su uso parece estar en ascenso, lo cual ha sido algo sorprendente para investigadores familiarizados con los principios de la teoría de los algoritmos genéticos ([Goldberg, D. E: 1989], [Holland, J. H.: 1975]), porque algunos análisis parecen sugerir que el mejor procesamiento de esquemas se obtiene usando alfabetos de baja cardinalidad, lo cual es contradictorio ya que otras conclusiones empíricas indican que la codificación real ha trabajado bien en un número de problemas prácticos."

El objetivo principal detrás de las codificaciones en punto flotante o real, es mover el algoritmo genético lo más cerca posible del espacio del problema. Este movimiento fuerza, pero también permite, que los operadores sean más específicos, dado que pueden utilizar características particulares del problema. Por ejemplo, esta representación tiene la propiedad de que dos puntos cercanos en el espacio de la representación están también juntos en el espacio del problema y viceversa. Esto generalmente no es verdadero en la representación binaria, donde la distancia en una representación está definida por el número de posiciones de bits diferentes. Por ejemplo, si se codifican en binario los enteros 5 y 6, los cuales son adyacentes en el espacio de búsqueda, sus equivalentes en binario serán el 101 y e1 110, los cuales difieren en 2 bits en el espacio de la representación. Sin embargo, es posible reducir estas discrepancias usando codificación de Gray.

La representación con codificación Gray tiene la propiedad de que dados dos puntos consecutivos en el espacio del problema difieren solo en un bit en la codificación. En otras palabras, el menor incremento posible del valor de un parámetro corresponde al cambio de un único bit en la codificación.

A continuación se describen ventajas y desventajas del uso de alfabetos de distintas cardinalidades para la codificación de los cromosomas.

2.3.11.1 Alfabetos de cardinalidad pequeña

La teoría fundamental sugiere que alfabetos pequeños son buenos, porque maximizan el número de esquemas disponibles para el procesamiento genético. El cálculo es directo. Considérese un algoritmo genético codificado sobre un alfabeto de cardinalidad k. Dado que hay k+1 esquemas por posición (los k miembros del alfabeto más el asterisco), y cada posición representa $\log_2 k$ bits, hay $(k+1)^{(1/\log 2\ k)}$ esquemas por bit de información de código con cardinalidad k.

Dos teoremas fueron probados basándose en estos resultados.

Teorema 1: hay $n_s = (k+1)^{(1/\log 2\ k)}$ conjuntos de similitudes o esquemas por bit de información para cadenas codificadas usando un alfabeto de cardinalidad k.

Teorema 2: Para un problema dado, las cadenas codificadas con alfabetos pequeños son representativas de un gran número de conjuntos de similitudes (esquemas) que para cadenas codificadas con alfabetos de mayor cardinalidad.

Binario	Octal	Fitness
001	0	22
011	3	8
101	5	11
111	7	3

Figura 2.12 Codificación binaria vs. codificación octal.

Un simple ejemplo ayudará a entender estos cálculos. Supóngase que se toma la decisión de codificar un problema en un alfabeto octal o binario. Una población reducida permitirá comprender la gran cantidad de información que se puede obtener usando alfabetos de baja cardinalidad (Ver Figura 2.12). Explorando las estructuras octales y su valor de fitness, no existe relación aparente. Debido a que cada cadena octal es representativa sólo de sí misma, no se pueden hacer inferencias respecto a cuales de las estructuras podrían ser particularmente prometedoras. En contraste, explorando las estructuras binarias, cada cadena es representativa de un número de subconjuntos de estructuras con similitudes en una o más posiciones. De este modo, se puede especular sobre relaciones de causa y efecto entre buenos componentes y altos valores de fitness. Por ejemplo, se puede pensar que la primera cadena tiene un valor de fitness alto porque termina con dos ceros, o porque termina con un cero, o en realidad, porque las cadenas 000 y 101 son relativamente aptas por el cero que comparten en la posición media. De esta forma, muchas hipótesis pueden ser formuladas respecto a la asociación entre fragmentos de las cadenas y valores de fitness altos, y ésta es la información que luego será recombinada para la potencial obtención de mejores estructuras durante el curso normal de una búsqueda genética.

Este razonamiento y los cálculos previos son directos y apenas abiertos a cuestiones, lo cual parece concluir que se deberían usar alfabetos de cardinalidad baja; sin embargo, hay problemas donde el uso de estos esquemas no es necesario y puede volver ineficiente el proceso de búsqueda.

2.3.11.2 Alfabetos de gran cardinalidad

Aunque el mayor número de esquemas se obtiene usando alfabetos pequeños, hay un número de razones por las cuales un usuario de técnicas genéticas evolutivas podría elegir ignorar esa ventaja:

- Comodidad en la correspondencia de un gen con una variable.
- Menor cantidad de generaciones para obtener soluciones aceptables.
- Reducción de oportunidades del fenómeno de decepción.

La primera de estas razones es más psicológica que técnica, pero muchos usuarios encuentran que una correspondencia uno a uno entre genes y parámetros es más cómoda que la codificación de parámetros como cadenas de bits u otros códigos discretos, lo cual resulta ser desconcertante en ciertas ocasiones.

Otra razón para el uso de alfabetos de mayor cardinalidad es la velocidad. Si se asume un tamaño fijo de la población y un número fijo de alternativas de búsqueda, se puede demostrar teóricamente y empíricamente que los alfabetos de gran cardinalidad convergen a alguna solución más rápidamente que esos codificados sobre alfabetos pequeños. Es importante reconocer que la convergencia rápida es una ventaja a medias porque la calidad de la solución se degrada con el incremento de la cardinalidad. Sin embargo, en algunos casos se puede preferir la convergencia rápida a alguna solución cercana a la óptima.

La última razón por la cual usar alfabetos de mayor cardinalidad es porque éste reduce la dimensionalidad del problema, lo cual a su vez reduce la oportunidad de decepción (Véase sección 2.3.10). La decepción existe cuando los bloques constructores de orden bajo dirigen al algoritmo genético en una dirección alejada de los bloques constructores de orden alto que contienen el óptimo global. Reduciendo la dimensión del problema se puede reducir la oportunidad de decepción porque existirán pocos bloques constructores de orden bajo que hagan confusa la búsqueda. Es cierto que reduciendo la dimensionalidad del problema se reduce la oportunidad de decepción, sin embargo, si el problema tiene más de una dimensión, la decepción puede aún existir y puede causar que la convergencia sea dificultosa.

2.3.12 Convergencia prematura

Los algoritmos genéticos son muy utilizados en problemas de búsqueda y optimización. El método ha mostrado ser eficiente y robusto en un número considerable de dominios científicos, donde la complejidad y cardinalidad de los problemas elegidos son considerados como factores claves. Sin embargo, todavía existen algunas insuficiencias; ciertamente, uno de los mayores problemas generalmente asociados con el uso de algoritmos genéticos es la convergencia prematura a soluciones que representan óptimos locales de la función objetivo. El problema está estrechamente relacionado con la pérdida de diversidad genética de las poblaciones de los algoritmos genéticos, siendo ésta la causa de un decremento en la calidad de las soluciones encontradas. Este factor ha motivado el desarrollo de diversas técnicas que apuntan a resolver, o al menos minimizar el problema; los métodos tradicionales trabajan generalmente para mantener un cierto grado de diversidad genética en la población, sin afectar el proceso de convergencia del algoritmo genético.

Algunos autores [Rocha M. and Neves J. 1996], han analizado varias aproximaciones para evitar la convergencia prematura a óptimos locales. A continuación se describen algunas de estas técnicas.

2.3.12.1 Velocidad de mutación adaptiva

El operador de mutación tiene por objetivo introducir un componente aleatorio en el proceso de búsqueda, con el beneficio de la exploración de nuevos sectores del espacio de búsqueda, lo cual promueve un incremento en la diversidad genética de la población. Por ejemplo, no es asombroso encontrar que uno de los primeros pasos a tomar para mantener la diversidad genética en la población es el incremento de la velocidad de mutación. Sin embargo, un valor alto para este parámetro introduce un cierto grado de ruido en el sistema, lo cual crea serios obstáculos al proceso de convergencia. Por consiguiente, y para superar este fenómeno, se va cambiando el valor de la velocidad de mutación, con la ayuda de una estrategia adaptiva basada en la diversidad genética de la población. Dicha diversidad, es medida en espacios regulares de tiempo, siendo la desviación estándar del valor de fitness de toda la población la magnitud usada para la medición.

El proceso trabaja como sigue: se comienza con un valor inicial para la velocidad de mutación, y en intervalos regulares de tiempo, se mide el valor de la desviación estándar, si es más bajo que un límite predefinido, se incrementa la velocidad de mutación.

2.3.12.2 Técnica de desastre social

La técnica de desastre social fue introducida por Kureichick y colegas [Kur96] para evitar la convergencia prematura a óptimos locales, cuando los algoritmos genéticos son aplicados al problema del viajante. La idea general es diagnosticar la situación de la pérdida de diversidad genética de la población, y en tal caso aplicar un operador catastrófico a ésta. Estos operadores fueron definidos con el propósito de retornar la población a un grado aceptable de diversidad, reemplazando un número de individuos seleccionados, por otros, generados aleatoriamente.

Dos diferentes operadores fueron considerados.

- Empaquetado. De todos los individuos que tienen el mismo valor de fitness, solo uno permanece inalterado, todos los otros son mutados aleatoriamente.

- Día del juicio: Solo el individuo con el mejor valor de fitness permanece inalterado, todos los otros son mutados aleatoriamente.

2.3.12.3 Generación de descendencia aleatoria.

Una de las características de una población que converge a un óptimo local es el gran número de individuos que comparten el mismo material genético. Pero, cuando esta

situación ocurre, hay una gran probabilidad de que el operador de crossover pueda recibir como entrada dos individuos con genotipos iguales. En este caso la recombinación de su material genético será inefectiva, dado que la descendencia generada simplemente será un clon de los padres.

La idea detrás de la generación de descendencia aleatoria es testear el material genético de los individuos antes de la operación de crossover, y si una situación como la anteriormente mencionada es detectada, no se lleva a cabo la operación. En su lugar se genera de forma aleatoria una descendencia o aun dos.

Dos estrategias diferentes son posibles, diferenciándose en el número de descendencia creada. En la primera, el resultado comprende a un individuo generado al azar y otro obtenido por clonación de sus padres. En la segunda, ambos descendientes son reproducidos de forma aleatoria.

2.3.13 Delta Coding

Los algoritmos genéticos tienen algunos problemas para hacer ajustes locales finos, por esta razón los GAs ofrecen soluciones menos precisas a problemas de optimización paramétricos que, por ejemplo las ESs, a menos que la representación de los individuos en los GAs sea cambiada de binario a punto flotante y el sistema provea operadores especializados. Sin embargo, en las últimas décadas se han desarrollado algunas investigaciones que intentan mejorar este aspecto.

Una interesante modificación de los algoritmos genéticos, llamada Delta Coding, fue propuesta en [Whitley et al 1991]. Una de las ventajas de delta coding es su simplicidad, además, es fácil de entender, fácil de implementar y fácil de analizar con la teoría existente de algoritmos genéticos.

La idea de Delata Coding es la búsqueda de modificaciones óptimas de la mejor solución obtenida hasta el momento. En un algoritmo genético convencional con una única población, cuando la población de soluciones candidatas ha convergido, se invoca el proceso Delta Coding y utilizando la mejor solución obtenida, genera e inicializa una población de nuevos individuos denominados Δ-cromosomas. Los Δ-cromosomas tienen la misma longitud que la mejor solución y están compuestos de valores denominados Δ-valores que representan diferencias de los valores (genes) de la mejor solución. La nueva población es evolucionada seleccionando Δ-cromosomas, sus Δ-valores son adicionados a los valores de la mejor solución y finalmente, el cromosoma resultante es evaluado. Los Δ-cromosomas que mejoran el desempeño de la mejor solución son seleccionados para reproducción.

La manera en la cual trabaja Delta Coding, permite explorar el hiper espacio en una vecindad de la mejor solución. Delta Coding puede ser aplicado múltiples veces, con sucesivas Δ-poblaciones representando diferencias de las mejores soluciones previas a cada aplicación.

Delta Coding fue desarrollado por Whitley et al. (1991) para mejorar las capacidades de ajustes locales finos de un algoritmo genético para optimizaciones numéricas. Sin embargo, su potencial para comportamientos adaptivos reside en la facilidad para hacer transiciones de tareas.

2.3.14 Ejemplos específicos de algoritmos genéticos

Mientras el poder de la evolución gana reconocimiento cada vez más generalizado, los algoritmos genéticos se utilizan para abordar una amplia variedad de problemas en un conjunto de campos sumamente diverso, demostrando claramente su capacidad y su potencial. A continuación se presentan algunos de los usos más notables en los que han tomado parte.

Robótica

El torneo internacional RoboCup es un proyecto para promocionar el avance de la robótica, la inteligencia artificial y los campos relacionados, proporcionando un problema estándar con el que probar las nuevas tecnologías -concretamente, es un campeonato anual de fútbol entre equipos de robots autónomos. (El objetivo fijado es desarrollar un equipo de robots humanoides que puedan vencer al equipo humano de fútbol que sea campeón del mundo en 2050; actualmente, la mayoría de los equipos de robots participantes funcionan con ruedas). Los programas que controlan a los miembros del equipo robótico deben exhibir un comportamiento complejo, decidiendo cuándo bloquear, cuándo tirar, cómo moverse, cuándo pasar la pelota a un compañero, cómo coordinar la defensa y el ataque, etcétera. En la liga simulada de 1997, David Andre y Astro Teller inscribieron a un equipo llamado Darwin United cuyos programas de control habían sido desarrollados automáticamente desde cero mediante programación genética, un desafío a la creencia convencional de que "este problema es simplemente demasiado difícil para una técnica como ésa" [Andre, D. and Astro T. 1999].

Para resolver este difícil problema, Andre y Teller le proporcionaron al programa genético un conjunto de funciones de control primitivas como girar, moverse, tirar, etcétera. (Estas funciones estaban también sujetas al cambio y refinamiento durante el curso de la evolución). Su función de aptitud, escrita para que recompensara el buen juego en general en lugar de marcar goles expresamente, proporcionaba una lista de objetivos cada vez más importantes: acercarse a la pelota, golpear la pelota, conservar la pelota en el campo contrario, moverse en la dirección correcta, marcar goles y ganar el partido. Debe señalarse que no se suministró ningún código para enseñar específicamente al equipo cómo conseguir estos objetivos complejos. Luego los programas evolucionados se evaluaron utilizando un modelo de selección jerárquico: en primer lugar, los equipos candidatos se probaron en un campo vacío y, si no marcaban un gol en menos de 30 segundos, se rechazaban. Luego se evaluaron haciéndoles jugar contra un equipo estacionario de postes pateadores que golpeaban la pelota hacia el campo contrario. En tercer lugar, el equipo jugaba un partido contra el equipo ganador de la competición RoboCup de 1997. Finalmente, los equipos que

marcaron al menos un gol contra este equipo jugaron unos contra otros para determinar cuál era el mejor.

De los 34 equipos de su división, Darwin United acabó en decimoséptima posición, situándose justo en el medio de la clasificación y superando a la mitad de los participantes escritos por humanos.

Juegos

Una de las demostraciones más novedosas y persuasivas de la potencia de los algoritmos genéticos fue presentada en [Chellapilla K. and Fogel D. 2001], donde utilizaron un AG para evolucionar redes neuronales que pudieran jugar a las damas. Los autores afirman que una de las mayores dificultades en este tipo de problemas relacionados con estrategias es el problema de la asignación de crédito -en otras palabras, ¿cómo escribir una función de aptitud? Se ha creído ampliamente que los criterios simples de ganar, perder o empatar, no proporcionan la suficiente información para que un algoritmo genético averigüe qué constituye al buen juego.

En este artículo, Chellapila y Fogel echan por tierra esa suposición. Dado sólo las posiciones espaciales de las piezas en el tablero y el número total de piezas que posee cada jugador; fueron capaces de evolucionar un programa de damas que jugaba a un nivel competitivo con expertos humanos, sin ninguna información de entrada inteligente acerca de lo que constituye el buen juego. Es más, ni siquiera se les dijo a los individuos del algoritmo evolutivo cuál era el criterio para ganar, ni se les dijo el resultado de ningún juego.

Química

En un artículo relacionado, [Glen, R.C. and Payne, y A.W.R.. 1995] describen el uso de algoritmos genéticos para diseñar automáticamente moléculas nuevas desde cero que se ajustan a un conjunto de especificaciones dado. Dada una población inicial, bien generada aleatoriamente o utilizando la sencilla molécula del etano como semilla, el AG añade, elimina y altera aleatoriamente átomos y fragmentos moleculares con el objetivo de generar moléculas que se ajusten a los requisitos dados. El AG puede optimizar simultáneamente un gran número de objetivos, incluyendo el peso molecular, el volumen molecular, el número de enlaces, el número de centros quirales, el número de átomos, el número de enlaces rotables, la polarizabilidad, el momento dipolar, etcétera, para producir moléculas candidatas con las propiedades deseadas. Basándose en pruebas experimentales, incluyendo un difícil problema de optimización que implicaba la generación de moléculas con propiedades similares a la ribosa (un componente del azúcar imitado a menudo en los fármacos antivirales). Los autores concluyen que el AG es un “excelente generador de ideas” que ofrece “propiedades de optimización rápidas y poderosas" y puede generar “un conjunto diverso de estructuras posibles”. Continúan afirmando: “Es de interés especial la poderosa capacidad de optimización del algoritmo genético, incluso con tamaños de población relativamente pequeños”.

Ingeniería de materiales

En [Giro, R., Cyrillo M. and Galvão D.S.. 2002], utilizaron algoritmos genéticos para diseñar polímeros conductores de electricidad basados en el carbono, conocidos como polianilinas. Estos polímeros, un tipo de material sintético inventado recientemente, tienen grandes aplicaciones tecnológicas potenciales y podrían abrir la puerta a nuevos fenómenos físicos fundamentales. Sin embargo, debido a su alta reactividad, los átomos de carbono pueden formar un número virtualmente infinito de estructuras, haciendo que la búsqueda de nuevas moléculas con propiedades interesantes sea del todo imposible. En este artículo, los autores aplican un enfoque basado en AGs a la tarea de diseñar moléculas nuevas con propiedades especificadas a priori, comenzando con una población de candidatos iniciales generada aleatoriamente. Concluyen que su metodología puede ser una "herramienta muy efectiva" para guiar a los investigadores en la búsqueda de nuevos compuestos y es lo suficientemente general para que pueda extenderse al diseño de nuevos materiales que pertenezcan virtualmente a cualquier tipo de molécula.

Acústica

En [Sato S. et. al. 2002] utilizaron algoritmos genéticos para diseñar una sala de conciertos con propiedades acústicas óptimas, maximizando la calidad del sonido para la audiencia, para el director y para los músicos del escenario. Esta tarea implica la optimización simultánea de múltiples variables. Comenzando con una sala con forma de caja de zapatos, el AG de los autores produjo dos soluciones no dominadas, ambas descriptas como "con forma de hoja". Los autores afirman que estas soluciones tienen proporciones similares al Grosser Musikvereinsaal de Viena, el cual está considerado generalmente como una de las mejores -si no la mejor- salas de conciertos del mundo, desde el punto de vista de propiedades acústicas.

En [Tang, K.S et. al. 1996] analizan los usos de los algoritmos genéticos en el campo de la acústica y el procesamiento de señales. Un área de interés particular incluye el uso de GAs para diseñar sistemas de Control Activo de Ruido (CAR), que eliminan el sonido no deseado produciendo ondas sonoras que interfieren destructivamente con el ruido. Esto es un problema de múltiples objetivos que requiere el control y la colocación precisa de múltiples altavoces; los GAs se han utilizado en estos sistemas tanto para diseñar los controladores como para encontrar la colocación óptima de los altavoces, dando como resultado una "atenuación efectiva del ruido" en pruebas experimentales.

Ingeniería Aeroespacial

En [Obayashi, S. et al. 2000] utilizaron un algoritmo genético de múltiples objetivos para diseñar la forma del ala de un avión supersónico. Hay tres consideraciones principales que determinan la configuración del ala: minimizar la resistencia aerodinámica a velocidades de vuelo supersónicas, minimizar la resistencia a velocidades subsónicas y minimizar la carga aerodinámica (la fuerza que tiende a doblar el ala). Estos objetivos son mutuamente excluyentes, y optimizarlos todos simultáneamente requiere realizar contrapartidas.

El cromosoma de este problema es una cadena de 66 números reales, cada uno de los cuales corresponde a un aspecto específico del ala: su forma, su grosor, su torsión, etcétera. Se

simuló una evolución con selección elitista durante 70 generaciones, con un tamaño de población de 64 individuos. Al final de este proceso había varios individuos, cada uno representando una solución no dominada del problema. El artículo comenta que estos individuos ganadores tenían características “físicamente razonables”", señalando la validez de la técnica de optimización. Para evaluar mejor la calidad de las soluciones, las seis mejores fueron comparadas con un diseño de ala supersónica producido por el Equipo de Diseño SST del Laboratorio Aeroespacial Nacional de Japón. Las seis fueron competitivas, con valores de resistencia y carga aproximadamente iguales o menores a los del ala diseñada por humanos; en particular, una de las soluciones evolucionadas superó al diseño del LAN en los tres objetivos. Los autores señalan que las soluciones del AG son similares a un diseño llamado ``ala flecha", sugerido por primera vez a finales de los años 50, pero que finalmente fue abandonado en favor del diseño más convencional con forma de delta.

En [Keane, A.J. and Brown, S.M. 1996] utilizaron un GA para producir un nuevo diseño para un brazo o jirafa para transportar carga que pudiese montarse en órbita y utilizarse con satélites, estaciones espaciales y otros proyectos de construcción aeroespacial. El resultado, una estructura retorcida con aspecto orgánico que se ha comparado con un fémur humano, no utiliza más material que el diseño de brazo estándar, pero es ligera, fuerte y muy superior a la hora de amortiguar las vibraciones perjudiciales, como confirmaron las pruebas reales del producto final. Los autores del artículo comentan, además, que su AG sólo se ejecutó durante 10 generaciones, debido a la naturaleza computacionalmente costosa de la simulación, y la población no se había estancado todavía. Haber proseguido la ejecución durante más generaciones habría producido indudablemente mejoras de rendimiento.

Diseño de rutas

En [He, L. and Mort, N. 2000] aplicaron algoritmos genéticos al problema de hallar rutas óptimas en las redes de telecomunicaciones (como las redes de telefonía e Internet), que se usan para transmitir datos desde los remitentes hasta los destinatarios. Esto es un problema NP-hard, un tipo de problema para el que los AGs son “extremadamente aptos y han encontrado una enorme variedad de aplicaciones exitosas en esos campos”. Además, es un problema multiobjetivo, en el que hay que equilibrar objetivos en conflicto como maximizar el caudal de datos, minimizar los retrasos en la transmisión y la pérdida de datos, encontrar caminos de bajo coste y distribuir la carga uniformemente entre los encaminadores o conmutadores de la red. Cualquier algoritmo real satisfactorio debe también ser capaz de redirigir el tráfico de las rutas principales que fallen o estén congestionadas.

3 NEUROEVOLUCIÓN

El aprendizaje y la evolución, desde un punto de vista biológico, son dos formas fundamentales de adaptación. En años recientes ha habido un gran interés en combinar estos dos conceptos con redes neuronales artificiales (ANNs – Artificial Neural Nerworks). Existen diferentes combinaciones entre ANNs y algoritmos evolucionarios (EAs – Evolucionary Algorithms), incluyendo el uso de EAs para evolucionar distintos elementos de las ANNs, como por ejemplo, pesos conexión, arquitectura, reglas de aprendizaje, características de las entradas, etc.

Las redes neuronales artificiales evolutivas (EANNs – Evolucionary ANNs) son una clase especial de redes neuronales artificiales en las cuales la adaptación se logra por evolución en lugar de hacerlo mediante los mecanismos de aprendizaje tradicional. Los algoritmos evolucionarios son usados para realizar varias tareas, tales como la inicialización y el entrenamiento de los pesos de conexión, el diseño de la arquitectura y la adaptación de las reglas de aprendizaje Una característica distintiva de las EANNs es su adaptabilidad a ambientes dinámicos. El aprendizaje y la evolución, hacen que la adaptación a este tipo de ambientes sea mucho más efectiva y eficiente.

En un sentido más amplio, EANNs pueden ser consideradas como un framework para sistemas adaptivos, por ejemplo, sistemas que pueden cambiar su arquitectura y reglas de aprendizaje adecuadamente sin intervención humana.

A continuación se presenta una breve descripción de la estructura y funcionamiento de las redes neuronales artificiales.

3.1 Redes Neuronales Artificiales

Una red neuronal artificial consiste de un conjunto de elementos de procesamiento, también conocidos como neuronas o nodos (Véase Figura 3.1), los cuales están unidos por conectores.
Esta estructura puede ser descripta como un grafo dirigido en el cual cada nodo representa un elemento de procesamiento y las flechas representan conectores entre nodos que indican el sentido en el que circula la información. Cada nodo ejecuta una función de transferencia de la forma

$$y_i = f_i(\text{net}_i - \theta_i)$$

donde

$$\text{net}_i = \Sigma_{j=1..n} w_{ij} x_j .$$

y donde y_i es la salida del nodo i, x_j es la j-ésima entrada al nodo, y w_{ij} es el peso de conexión entre los nodos i y j. θ_i es el umbral (o influencia) del nodo. Generalmente f_i es una función no lineal, como el caso de la función sigmoidea o gausiana.

Los nodos están comúnmente agrupados en estructuras conocida como bloques o capas. Todos los nodos pertenecientes a una capa, realizan el procesamiento de sus entradas de forma análoga. Las capas interiores en una red neuronal se denominan capas ocultas.

El procesamiento comienza con una red en un estado inactivo. Un patrón externo, conformado por un conjunto de señales es aplicado a la capa de entrada. Cada nodo de dicha capa, genera una única señal de salida. Colectivamente, las salidas producidas por todos los nodos son pasadas como señales de entrada a la subsiguiente capa de nodos. Este proceso es repetido, hasta que la última capa produzca una salida.

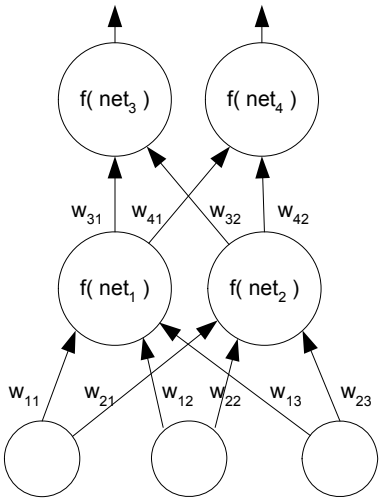

Figura 3.1. Una red neuronal típica con una capa de entrada, una capa oculta y una capa de salida.

La arquitectura de una red neuronal artificial es determinada por su estructura topológica, por ejemplo, la conectividad y la función de transferencia de cada nodo. Con respecto a su conectividad, existen dos clases de redes neuronales. Estas clases se conocen como feedforward y recurrentes. Una ANN es feedforward si existe una forma de numerar todos los nodos en la red tal que no exista una conexión desde un nodo con un número dado a un nodo con un número menor. Es decir, que todas las conexiones van desde algún nodo hacia otros nodos con numeración mayor. Una ANN es recurrente si no existe esa forma de numeración.

3.1.1 Aprendizaje en ANNs

El aprendizaje es el proceso por el cual una red neuronal modifica sus pesos en respuesta a una información de entrada. Los cambios que se producen durante el proceso de aprendizaje se reducen a la destrucción, modificación y creación de conexiones entre las neuronas. La creación de una nueva conexión implica que el peso de la misma pasa a tener un valor distinto de cero y por el contrario, una conexión se destruye cuando su peso pasa a ser cero. Se puede afirmar que el proceso de aprendizaje ha finalizado (la red ha aprendido) cuando los valores de los pesos permanecen estables.

Un criterio para diferenciar las reglas de aprendizaje se basa en considerar si la red puede aprender durante su funcionamiento habitual, o si el aprendizaje supone la desconexión de la red.

Otro criterio suele considerar dos tipos de reglas de aprendizaje: las de aprendizaje supervisado y las correspondientes a un aprendizaje no supervisado. Estas reglas dan pie a una de las clasificaciones que se realizan de las ANNs: redes neuronales con aprendizaje supervisado y redes neuronales con aprendizaje no supervisado. La diferencia fundamental entre ambos tipos reside en la existencia o no de un agente externo (supervisor) que controle el aprendizaje de la red.

3.1.1.1 Aprendizaje supervisado

El proceso de aprendizaje se realiza mediante un entrenamiento, controlado por un agente externo (supervisor, maestro) que determina la respuesta que debería generar la red a partir de una entrada determinada. El supervisor comprueba la salida de la red y en el caso de que ésta no coincida con la deseada, se procederá a modificar los pesos de las conexiones, con el fin de conseguir que la salida se aproxime a la deseada.

Se consideran tres formas de llevar a cabo este tipo de aprendizaje:

- Aprendizaje por corrección de error: Consiste en ajustar los pesos en función de la diferencia entre los valores deseados y los obtenidos en la salida de la red; es decir, en función del error.

- Aprendizaje por refuerzo: Se basa en la idea de no indicar durante el entrenamiento la salida exacta que se desea que proporcione la red ante una determinada entrada. La función del supervisor se reduce a indicar mediante una señal de refuerzo si la salida obtenida en la red se ajusta a la deseada, y en función de ello se ajustan los pesos basándose en un mecanismo de probabilidades.

- Aprendizaje estocástico: Este tipo de aprendizaje consiste básicamente en realizar cambios aleatorios en los valores de los pesos de las conexiones de la red y evaluar su efecto a partir del objetivo deseado y de distribuciones de probabilidad.

3.1.1.2 Aprendizaje no supervisado

Estas redes no requieren influencia externa para ajustar los pesos de las conexiones entre neuronas. La red no recibe ninguna información por parte del entorno que le indique si la salida generada es o no correcta, de este modo, existen varias posibilidades en cuanto a la interpretación de la salida de estas redes.

En algunos casos, la salida representa el grado de familiaridad o similitud entre la información que se le está presentando en la entrada y las informaciones que se le han mostrado en el pasado. En otro caso podría realizar una codificación de los datos de

entrada, generando a la salida una versión codificada de la entrada, con menos bits, pero manteniendo la información relevante de los datos. Algunas redes con aprendizaje no supervisado realizan un mapeo de características, obteniéndose en las neuronas de salida una disposición geométrica que representa un mapa topográfico de las características de los datos de entrada. De esta forma, si se le presentan a la red informaciones similares, siempre serán afectadas neuronas de salidas próximas entre sí, en la misma zona del mapa..

En general en este tipo de aprendizaje se suelen considerar dos tipos:

- Aprendizaje Hebbiano: Consiste básicamente en el ajuste de los pesos de las conexiones de acuerdo con la correlación, así si las dos unidades son activas (positivas), se produce un refuerzo de la conexión. Por el contrario cuando una es activa y la otra pasiva (negativa), se produce un debilitamiento de la conexión.

- Aprendizaje competitivo y cooperativo: Las neuronas compiten (y cooperan) unas con otras con el fin de llevar a cabo una tarea dada. Con este tipo de aprendizaje se pretende que cuando se presente a la red cierta información de entrada, solo una de las neuronas de salida se active (alcance su valor de respuesta máximo). Por tanto, las neuronas compiten por activarse, quedando finalmente una, o una por grupo, como neurona vencedora

3.2 Evolución de Redes Neuronales Artificiales

La evolución en las ANNs ha sido propuesta en tres diferentes niveles: pesos de conexión, arquitectura y reglas de aprendizaje. La evolución de pesos de conexión introduce una aproximación adaptiva y global al entrenamiento, especialmente en el aprendizaje por refuerzo y en el paradigma de aprendizaje de redes recurrentes donde los algoritmos basados en gradientes experimentan con frecuencia grandes dificultades. La evolución de arquitecturas permite a las ANNs adaptar su topología a diferentes tareas sin intervención humana y de este modo provee un método para el diseño automático de ANN. La evolución de reglas de aprendizaje puede ser considerada como un proceso de "aprendizaje para aprender" en ANNs donde la adaptación de reglas de aprendizaje se logra a través de la evolución. Esta también puede ser vista como un proceso adaptivo de descubrimiento automático de reglas de aprendizaje novedosas.

3.2.1 Evolución de pesos de conexión

Las ANNs obtienen sus pesos de conexión por medio de procesos de entrenamiento, estos entrenamientos generalmente tienen como objetivo la minimización de una función de error (como por ejemplo el error cuadrático medio entre el objetivo y las salidas promediadas sobre todos los ejemplos) ajustando iterativamente los pesos de conexión. La mayoría de los algoritmos de entrenamiento, como por ejemplo el Backpropagation (BP) y algoritmos de gradiente conjugado [Møller, M. F.], están basados en la utilización del gradiente descendente. La utilización de BP, ha permitido obtener algunas aplicaciones exitosas en

distintas áreas [Knerr, S. et. al.], pero BP tiene problemas debido al uso de gradiente [Sutton, R. S.]. Este generalmente queda atrapado en mínimos locales de la función de error y es incapaz de encontrar el mínimo global si la función de error es multimodal y/o no diferenciable.

Una forma de sobreponerse a los problemas de los algoritmos de entrenamiento basados en gradiente, es utilizar redes neuronales artificiales evolutivas EANNs. Por ejemplo, mediante la formulación del proceso de entrenamiento, como la evolución de los pesos de conexión en un ambiente determinado por la arquitectura y la tarea que se desea aprender. Los algoritmos evolutivos pueden ser usados efectivamente en la evolución para encontrar un conjunto de pesos de conexión cercanos a los óptimos sin la necesidad de computar información de gradiente. El fitness de una red neuronal artificial puede ser definido acorde a diferentes necesidades. Dos factores importantes que a menudo aparecen en la función de fitness (o error) son el error entre el objetivo y las salidas reales, y la complejidad de la ANN. A diferencia de los algoritmos basados en gradiente, la función de fitness (o error) puede no ser diferenciable o continua, dado que los algoritmos evolucionarios no dependen de información de gradiente. Como consecuencia de que los algoritmos evolucionarios (EAs) pueden usarse en espacios grandes, complejos, no diferenciables y multimodales, se han desarrollado muchas investigaciones sobre la base de la evolución de pesos de conexión, como por ejemplo [Whitley, D. et. al.: 1991].

La aproximación evolucionaria al entrenamiento de los pesos en ANNs consiste de dos grandes etapas. La primera etapa es decidir la representación de los pesos de conexión, por ejemplo, si se representaran en forma binaria o no. La segunda es el proceso evolucionario simulado por un algoritmo evolutivo, en el cual los operadores genéticos tales como el crossover y la mutación tienen que ser elegidos en conjunción con el esquema de representación. Diferentes representaciones y operadores de búsqueda pueden inducir a muy diferentes desempeños de entrenamiento. Un ciclo típico de evolución de pesos de conexión es mostrado en la Figura 3.2. La evolución termina cuando el fitness es mayor que un valor predefinido o la población ha convergido.

1. Decodificar cada individuo (genotipo) de la generación actual en un conjunto de pesos de conexión y construir una red neuronal con los pesos.

2. Evaluar cada ANN computando su error cuadrático medio total entre la salida real y la deseada (otras funciones de error pueden ser usadas también). El fitness de un individuo es determinado por el error. Cuanto más grande es el error, más chico es el fitness. El mapeo óptimo del error al fitness es dependiente del problema. Un término de regulación puede ser incluido en la función de fitness para penalizar pesos muy grandes.

3. Seleccionar los progenitores para la reproducción en función del fitness.

4. Aplicar operadores de búsqueda, tal como crossover y/o mutación a los progenitores para generar la descendencia que formará la nueva generación.

Figura 3.2. Un ciclo típico de evolución de pesos de conexión**.**

3.2.1.1 Representación Binaria

El algoritmo genético canónico [Holland, J. H.: 1975], [Goldberg, D. E: 1989], ha usado siempre cadenas binarias para codificar soluciones, comúnmente llamadas cromosomas. Algunos de los primeros trabajos sobre la evolución de los pesos de conexión en redes neuronales artificiales siguieron siempre esta aproximación [Whitley, D., et. al: 1990]. En este esquema, cada peso de conexión es representado por una cadena de bits de cierta longitud. Una red neuronal es codificada por concatenación de todos los pesos de conexión de la red en el cromosoma.

Una heurística utilizada en el orden de concatenación consiste en agrupar los pesos de conexión que entran y salen a un nodo oculto. Los nodos ocultos en las redes neuronales son en esencia extractores y detectores de rasgos. La disposición alejada de las entradas del mismo nodo oculto en la representación binaria puede incrementar la dificultad de construir detectores de rasgos útiles porque ellos podrían ser destruidos por los operadores de crossover. Generalmente es muy dificultosa la aplicación de operadores de crossover en la evolución de pesos de conexión, dado que ellos tienden a destruir detectores de rasgos durante el proceso evolutivo.

La Figura 3.3 presenta un ejemplo de representación binaria de una red neuronal cuya arquitectura está predefinida. Cada peso de conexión en la red neuronal es representado con 4 bits, la red neuronal entera está representada con 24 bits, donde el peso 0000 indica que no hay conexión entre los dos nodos.

La ventaja de la representación binaria está en su simplicidad y generalidad. La aplicación del operador de crossover clásico (tal como el de un punto o el uniforme – Ver sección 2.3.5) y la mutación a cadenas binarias, es directa. Con esta representación hay poca necesidad de diseñar operadores de búsqueda complejos y a medida. La representación binaria también facilita la implementación de hardware digital de redes neuronales dado que los pesos tienen que ser representados en función de bits en el hardware con una precisión limitada.

Existen diversos métodos de codificación que pueden ser usados en la representación binaria, tal como el uniforme, el Gray, el exponencial, etc. Ellos codifican valores reales usando diferentes rangos y precisiones dado el mismo número de bits. Sin embargo, a menudo es necesario un balance entre la precisión de la representación y la longitud del cromosoma. Si muy pocos bits son usados para representar cada peso de conexión, el entrenamiento podría fallar porque algunas combinaciones de pesos de conexión reales no pueden ser aproximados con suficiente exactitud por valores discretos. Por otro lado, si son utilizados muchos bits, los cromosomas representando grandes redes neuronales se volverán extremadamente largos y la evolución, por esta misma causa, será muy ineficiente.

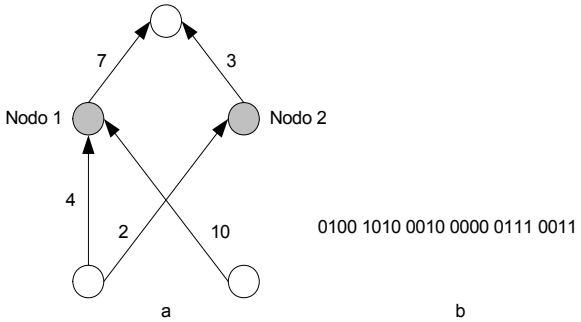

Figura 3.3 (a) Una ANN con pesos de conexión. (b) Una representación binaria de los pesos, utilizando 4 bits para cada uno.

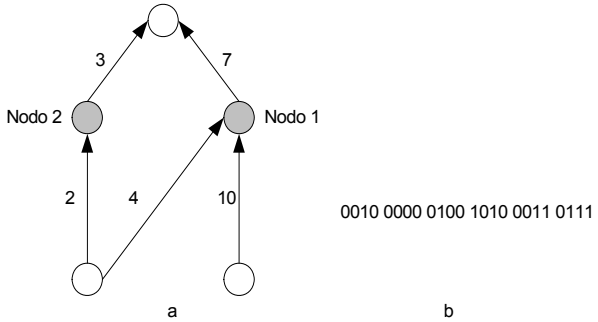

Figura 3.4 (a) Una ANN que es equivalente a la dada en Figura 3.3(a) y Figura 3.3(b) Su representación binaria utilizando 4 bits para cada peso.

Uno de los problemas afrontado por el entrenamiento evolucionario de las redes neuronales es el problema de permutación [Hancock, P. J. B. :1992]. Este es causado por el mapeo muchos a uno de la representación (genotipo) a la red neuronal real (fenotipo) debido a que dos redes neuronales que cambian el orden de sus nodos ocultos difieren en sus cromosomas pero aún siguen teniendo funcionalidad equivalente. Por ejemplo, la red neuronal presentada en la Figura 3.3(a) y la Figura 3.4(a) son funcionalmente equivalentes, pero ellas tienen cromosomas diferentes como muestra la Figura 3.3(b) y la Figura 3.4(b). En general, cualquier permutación de los nodos ocultos producirá redes neuronales con funcionalidades equivalentes y diferentes representaciones de cromosomas. El problema de permutación hace que el operador de crossover sea muy ineficiente e inefectivo en la producción de buena descendencia.

3.2.1.2 Representación Real

Ha habido diversos debates en cuanto a la cardinalidad del alfabeto de los genotipos. Algunos han sostenido que los alfabetos de cardinalidad pequeña, por ejemplo el binario, podrían no ser los mejores. Análisis formales de representaciones no estándar y operadores basados en el concepto de clases equivalentes han dado una sólida base teórica a otro tipo de cadenas más allá de las binarias. Los números reales han sido propuestos [Montana, D. and Davis, L :1989] para representar pesos de conexión directamente. Por ejemplo una

representación con números reales de la red neuronal dada en la Figura 3.3(a) podría ser (4.0, 10.0, 2.0, 0.0, 7.0, 3.0).

Como los pesos de conexión son representados por números reales, cada individuo de la población en evolución será un vector real. En este caso, el crossover y la mutación binaria tradicional no pueden ser usados directamente, sino que deben ser redefinidos especialmente. En [Montana, D. and Davis, L :1989] se definieron un gran número de operadores genéticos a medida, los cuales incorporaron muchas heurísticas acerca del entrenamiento de las redes neuronales. La idea fue retener los detectores de rasgos útiles formados alrededor de los nodos ocultos durante la evolución. Sus resultados muestran que la aproximación del entrenamiento evolucionario fue mucho más rápida que BP (Backpropagation) para los problemas que ellos consideraron.

Una forma natural de evolucionar vectores reales podría ser usando programación evolucionaria (EP) o estrategias evolucionarias (ES) dado que son particularmente adecuadas para tratar optimizaciones continuas. A diferencia de los algoritmos genéticos, el operador de búsqueda primario en EP y ES es la mutación. Una de las mayores ventajas de usar algoritmos evolucionarios basados en mutación es que pueden reducir el impacto negativo del problema de permutación. Por consiguiente, el proceso evolucionario puede ser más eficiente.

3.2.1.3 Entrenamiento evolucionario vs. entrenamiento basado en gradiente

Como se mencionó anteriormente, la aproximación de entrenamiento evolucionario es atractivo porque éste puede manejar mejor el problema de búsqueda global en una superficie extensa, compleja, multimodal y no diferenciable. Este no depende de información de gradiente de la función de error (fitness) y de esta manera es particularmente interesante cuando esta información no está disponible o es muy costosa de obtener o estimar. Por ejemplo muchos investigadores han usado aproximaciones evolucionarias para entrenar redes neuronales recurrentes, redes neuronales de alto orden y redes neuronales difusas. Además, el mismo algoritmo evolucionario puede ser usado para entrenar muchas redes diferentes independientemente de si ellas son feedforwad, recurrentes o de alto orden. La aplicación genérica de la aproximación evolucionaria evita gran cantidad esfuerzo humano en el desarrollo de diferentes algoritmos de entrenamiento para diferentes tipos de redes neuronales.

La aproximación evolucionaria también hace más fácil generar redes neuronales con características especiales. Por ejemplo, la complejidad de una red neuronal puede ser reducida y su generalización incrementada incluyendo un término de complejidad en la función de fitness. A diferencia del entrenamiento basado en información de gradiente, este término no necesita ser diferenciable o continuo.

El entrenamiento evolucionario puede ser más lento para algunos problemas en comparación con variantes rápidas de BP y algoritmos de gradiente conjugado. Sin embargo, son mucho menos sensitivos a condiciones iniciales de entrenamiento. Ellos siempre buscan soluciones óptimas globalmente, mientras un algoritmo de gradiente descendente puede solo encontrar un óptimo local en una vecindad de la solución inicial.

Para algunos problemas, el entrenamiento evolucionario puede ser significativamente más rápido y más confiable que BP.

3.2.1.4 Entrenamiento Híbrido

La mayoría de los algoritmos evolucionarios son más bien ineficientes en búsquedas locales para ajustes finos aunque son buenos en búsquedas globales. Esto es especialmente cierto para algoritmos genéticos. La eficiencia del entrenamiento evolucionario puede ser mejorada significativamente incorporando un procedimiento de búsqueda local en la evolución, por ejemplo, combinando la habilidad en las búsquedas globales de los algoritmos evolucionarios con la habilidad en búsquedas locales para ajustes finos. Los algoritmos evolucionarios pueden ser usados para localizar buenas regiones en el espacio y luego un procedimiento de búsqueda local es usado para encontrar una solución óptima cercana en esa región. El algoritmo de búsqueda local podría ser por ejemplo el Backpropagation, u otros algoritmos de búsqueda aleatoria. El entrenamiento híbrido ha sido utilizado en muchas áreas de aplicación.

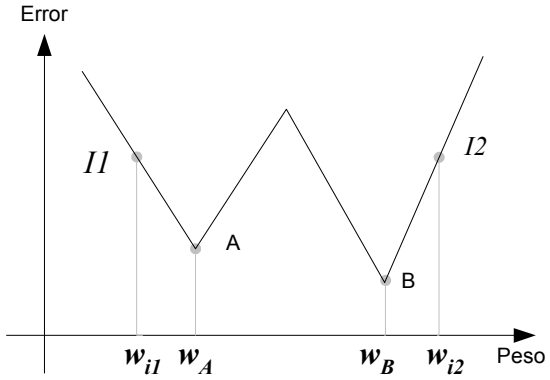

Figura 3.5 Una ilustración del uso de un EA para encontrar buenos pesos de conexiones iniciales que permitan que un algoritmo de búsqueda local pueda encontrar el peso de conexión óptimo global fácilmente. w_{i2} es un peso inicial óptimo porque éste puede conducir al óptimo global w_B usando un algoritmo de búsqueda local.

Muchos investigadores han usado Algoritmos Genéticos para buscar un conjunto de pesos de conexión inicial cercano al óptimo y luego un algoritmo Backpropagation para realizar una búsqueda local a partir de esos pesos iniciales. Sus resultados muestran que la aproximación híbrida GA/BP fue más eficiente que cualquiera de esos métodos utilizados por separado. Trabajos similares sobre evolución de pesos iniciales han sido hechos en redes neuronales de aprendizaje competitivo y redes de Kohonen.

Es interesante considerar el hecho de encontrar pesos iniciales buenos que permitan localizar una buena región en el espacio. Dada una pendiente de atracción a un mínimo local compuesta de puntos, un conjunto de pesos en esa región, pueden converger en el mínimo local a través de un algoritmo de búsqueda local. Si un algoritmo evolucionario puede localizar un punto, por ejemplo, un conjunto de pesos iniciales, en la pendiente de atracción del mínimo global, entonces el mínimo global puede ser fácilmente encontrado

por un algoritmo de búsqueda local. La Figura 3.5 ilustra un caso simple donde hay un solo peso de conexión en la red neuronal. Si un algoritmo evolucionario puede encontrar un peso inicial tal como w_{i2}, debería ser fácil para un algoritmo de búsqueda local llegar al peso óptimo global w_B, aun cuando w_{i2} en sí mismo no sea tan bueno como w_{i1}.

3.2.2 Evolución de arquitectura

En la evolución de pesos de conexión se asume que la arquitectura de una red neuronal artificial está predefinida y fija durante la evolución. En esta sección se discute el diseño de arquitecturas de ANNs. La arquitectura de una red neuronal incluye su estructura topológica, como por ejemplo, la conectividad y la función de transferencia de cada nodo. El diseño de la arquitectura es crucial para la aplicación exitosa de las redes neuronales, porque ésta tiene un impacto significante en las capacidades de procesamiento de información de las mismas. Dada una tarea de aprendizaje, una red neuronal con solo unas pocas conexiones y nodos lineales, puede no ser capaz de realizar toda la tarea debido a su capacidad limitada. Mientras una red neuronal con un gran número de conexiones y nodos no lineales, puede sobreajustarse al ruido de los datos de entrenamiento y fracasar en la adquisición de una buena capacidad de generalización.

El diseño de la arquitectura es aún un trabajo que requiere de la intervención de humanos expertos. Esto depende en gran medida de la experiencia y del proceso tedioso de prueba y error. No hay una forma sistemática y automática para diseñar una arquitectura cercana a la óptima para una tarea dada. Investigaciones en algoritmos constructivos y destructivos representan un esfuerzo hacía el diseño automático de arquitecturas. A grandes rasgos, un algoritmo constructivo comienza con una red minimal (red con el mínimo número de capas ocultas, nodos y conexiones) y agrega nuevas capas, nodos y conexiones cuando es necesario durante el entrenamiento. Mientras que un algoritmo destructivo hace lo opuesto, comienza con una red maximal y borra capas innecesarias, nodos y conexiones durante el entrenamiento. Sin embargo, estos tipos de métodos son susceptibles de quedar atrapados en una estructura óptima local.

El diseño de arquitecturas óptimas para una red neuronal artificial puede ser formulado como un problema de búsqueda en el espacio de arquitecturas donde cada punto representa una arquitectura determinada.

De forma semejante a lo que ocurre en la evolución de pesos de conexión, la evolución de arquitecturas comprende dos grandes etapas: la representación del genotipo y el algoritmo evolucionario empleado para evolucionar las arquitecturas de las redes neuronales. Uno de los puntos claves en la codificación de la arquitectura de una red neuronal, es decidir cuanta información acerca de la arquitectura debería ser codificada en el cromosoma. En un extremo, se pueden codificar en un cromosoma todos los detalles, como por ejemplo todas las conexiones y nodos de la arquitectura. Este tipo de esquema de representación es llamado codificación directa. En el otro extremo, se codifican solo los parámetros más importantes, tales como el número de capas ocultas y el número de nodos ocultos en cada capa. Otros detalles acerca de la arquitectura son dejados para que los decida el proceso de

entrenamiento. Este tipo de representación es llamado codificación indirecta. Después de que un esquema de representación ha sido elegido, la evolución de arquitecturas puede progresar acorde al ciclo mostrado en la Figura 3.6. El ciclo termina cuando una red neuronal satisfactoria haya sido encontrada.

1. Decodificar cada individuo de la generación actual en una arquitectura. Si el esquema de codificación indirecta es usado, más detalles acerca de la arquitectura es especificada por algunas reglas o procesos de entrenamiento.
2. Entrenar cada ANN con la arquitectura decodificada por una regla de aprendizaje predefinida (algunos parámetros de la regla de aprendizaje podrían ser evolucionados durante el entrenamiento) comenzando desde conjuntos diferentes de pesos de conexiones iniciales aleatorios y, si existen, parámetros que controlan el aprendizaje.
3. Computar el fitness de cada individuo (arquitectura codificada) acorde al resultado del entrenamiento anterior y otros criterios de desempeño tal como la complejidad de la arquitectura.
4. Seleccionar progenitores de la población en función del fitness
5. Aplicar operadores de búsqueda a los progenitores y generar descendencia la cual formará la siguiente generación.

Figura 3.6 Un ciclo típico de evolución de arquitecturas

Importantes investigaciones en la evolución de arquitecturas de redes neuronales han sido realizadas en años recientes. La mayoría de las investigaciones se han concentrado en la evolución de las estructuras topológicas. Relativamente pocas han sido hechas sobre la evolución de funciones de transferencia de los nodos, y mucho menos de la evolución simultanea de la topología y de las funciones de transferencia de los nodos.

3.2.2.1 Esquema de codificación directa

Se han tomado dos aproximaciones diferentes en el esquema de codificación directa. La primera separa la evolución de arquitecturas de la de pesos de conexión [Whitley, D., et. al.: 1990]. La segunda aproximación evoluciona arquitecturas y pesos de conexión simultáneamente.

En la primera aproximación, se especifica cada conexión de una arquitectura directamente por su representación binaria [Whitley, D., et. al.: 1990]. Por ejemplo, una matriz puede representar la arquitectura de una red neuronal con nodos, la cual indica la presencia o ausencia de conexiones de un nodo a otro. En realidad, puede representar pesos de conexión con valores reales de nodo a nodo, de modo que la arquitectura y los pesos de conexión evolucionen simultáneamente. Cada matriz tiene un mapeo directo uno a uno a la arquitectura de la red neuronal correspondiente. La cadena binaria que representa la arquitectura, es la concatenación de filas (o columnas) de la matriz. Las restricciones de las arquitecturas que son exploradas pueden fácilmente ser incorporadas en estos esquemas de representación imponiendo restricciones en la matriz, por ejemplo, una red neuronal

feedforward tendrá entradas distintas de cero solo en el triángulo derecho superior de la matriz. La Figura 3.7 y la Figura 3.8 muestran dos ejemplos del esquema de codificación directa de arquitecturas de redes neuronales. Es claro que este esquema de codificación puede manejar tanto redes neuronales feedforward como recurrentes.

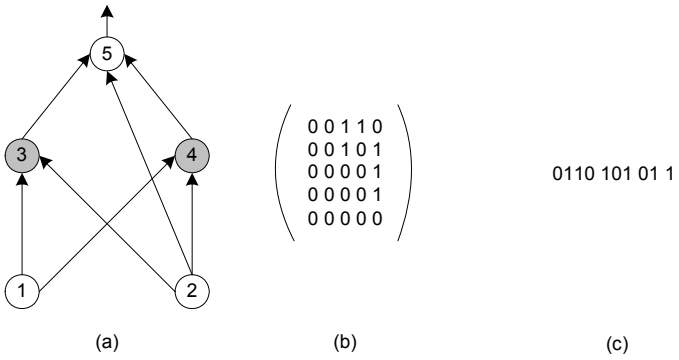

Figura 3.7 Un ejemplo de la codificación directa de una ANN feedforward. (a), (b) y (c) muestran la arquitectura, su matriz de conectividad y su cadena binaria respectivamente. Debido a que solo arquitecturas feedforward son consideradas, la representación binaria solo necesita los datos del triángulo superior derecho de la matriz.

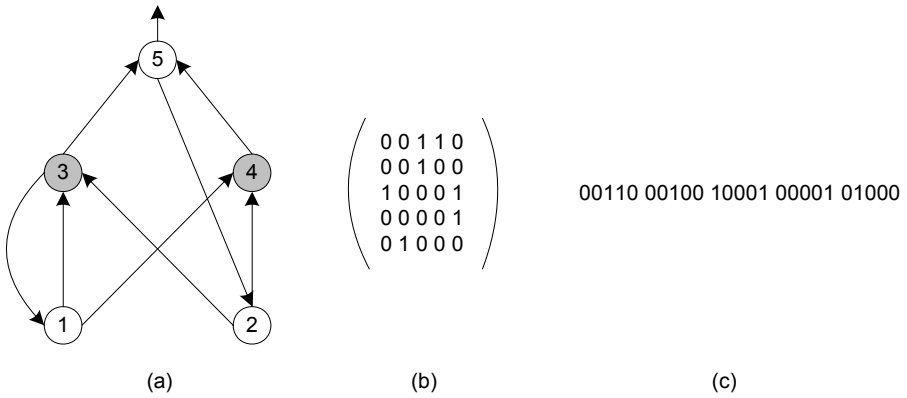

Figura 3.8 Un ejemplo de la codificación directa de una ANN recurrente. (a), (b) y (c) muestran la arquitectura, su matriz de conectividad y su cadena binaria respectivamente.

La Figura 3.7(a) muestra una ANN feedforward con dos entradas y una salida. Su matriz de conectividad está representada en la Figura 3.7(b), donde cada entrada indica la presencia o ausencia de una conexión desde un nodo a otro. Por ejemplo, la primera fila representa las conexiones del primer nodo a todos los restantes. La conversión de esta matriz de conectividad a un cromosoma es directa. Concatenando todas las filas (o columnas) se obtiene

00110 00101 00001 00001 00000.

Dado que la ANN es feedforward, solo son necesarios los datos de la esquina superior derecha de la matriz, lo cual permite reducir la longitud del cromosoma. El cromosoma reducido está representado en la Figura 3.7(c). Luego un algoritmo evolucionario puede ser empleado para evolucionar una población de dichos cromosomas. Para evaluar el fitness de cada cromosoma, es necesario decodificar un cromosoma en una ANN, inicializarla con pesos aleatorios, y entrenarla. El error de entrenamiento será usado para medir el fitness. Es importante notar que la Figura 3.7 tiene un atajo entre la entrada y la salida. Este atajo no plantea un problema para la representación y la evolución. Un algoritmo evolutivo es capaz de explorar todas las posibles conectividades.

La Figura 3.8 muestra una ANN recurrente. Su representación es básicamente la misma que para ANNs de tipo feedforwad. La única diferencia es que no es posible ninguna reducción en el tamaño del cromosoma si se quiere explorar todo el espacio de conectividades. El algoritmo evolutivo utilizado para evolucionar ANNs recurrentes puede ser el mismo que el utilizado para redes feddforward. El esquema de codificación directa descripto arriba es muy fácil de implementar. Es muy adecuado para la búsqueda fina y precisa de arquitecturas compactas, dado que una simple conexión puede ser agregada o removida de la ANN muy fácilmente.

Otra flexibilidad provista por la evolución de arquitecturas se relaciona con la definición de la función de fitness. Virtualmente no hay limitaciones tales como la necesidad de que la función sea diferenciable o continua. El resultado del entrenamiento de una arquitectura tal como el error o el tiempo de entrenamiento puede ser usado en la función de fitness. La medida de complejidad tal como el número de nodos y las conexiones también pueden ser usadas.

Un problema potencial del esquema de codificación directa es su escalabilidad. Una ANN grande requeriría una matriz muy amplia lo cual incrementa el tiempo de cómputo de la evolución. Una forma de reducir el tamaño de las matrices es usar algún conocimiento del dominio. Por ejemplo, si se utiliza una conexión completa entre dos capas vecinas en una ANN feedforward, su arquitectura puede ser codificada solo por el número de capas ocultas y el número de nodos en cada capa, con lo cual se reduce en gran medida el tamaño del cromosoma. Sin embargo, hacer esto requiere de una gran técnica y conocimiento acerca del dominio, lo cual es dificultoso de obtener en la práctica. También se corre el riesgo de perder alguna buena solución cuando se restringe el espacio de búsqueda manualmente.

El problema de la permutación ilustrado por la Figura 3.3 y la Figura 3.4, aún existe y causa efectos laterales indeseados en la evolución de arquitecturas. Debido a que dos ANNs funcionalmente equivalentes que ordenan sus nodos ocultos de forma diferente tienen dos representaciones de genotipos diferentes, la probabilidad de producir una descendencia muy calificada por medio de la recombinación es muy baja.

3.2.2.2 Esquema de codificación indirecto

Para reducir la longitud de la representación de los genotipos de las arquitecturas, varios investigadores han usado el esquema de codificación indirecta. Este método solo codifica en el cromosoma algunas características de una arquitectura. Los detalles acerca de cada

conexión en una red neuronal son o bien predefinidos de acuerdo a un conocimiento previo o especificados por un conjunto determinístico de reglas evolutivas. El esquema de codificación indirecto puede producir representaciones de cromosomas más compactas, pero esto puede no ser muy bueno para encontrar una red neuronal compacta con buena habilidad de generalización.

Las arquitecturas de las redes neuronales pueden ser especificadas por un conjunto de parámetros tales como el número de capas ocultas, el número de nodos ocultos en cada capa, el número de conexiones entre dos capas, etc. Estos parámetros pueden ser codificados en varios formatos en el cromosoma.

Aunque el método de representación paramétrica puede reducir la longitud de un cromosoma binario en la especificación de arquitecturas de redes neuronales, los algoritmos evolucionarios solo pueden buscar un subconjunto limitado del espacio de arquitecturas. Por ejemplo, si se codifica solo el número de nodos ocultos en la capa oculta, básicamente se asume que la red neuronal tiene una arquitectura feedforward con una única capa oculta. En general, el método de representación paramétrica es más adecuado cuando se conoce el tipo de arquitectura que se intenta encontrar.

Un esquema de codificación indirecto muy diferente al anterior, consiste en codificar reglas evolutivas, las cuales son usadas para construir arquitecturas. El cambio de optimización directa de arquitecturas a la optimización de reglas evolutivas ha traído algunos beneficios, tal como una representación de genotipos más compacta en la evolución de arquitecturas. El efecto destructivo del crossover también se reduce debido a que la representación de las reglas evolutivas es capaz de preservar bloques constructores potencialmente buenos que se encuentran lejos. Pero este método también tiene algunos problemas.

Una regla evolutiva es generalmente una ecuación recursiva o una regla generacional similar a una regla de producción en un sistema de producción. El patrón de conectividades de una arquitectura se estructura en una matriz que es construida desde cero. Por ejemplo, se comienza desde un único elemento y luego se aplican repetitivamente reglas evolutivas a elementos no terminales en la matriz actual. Este proceso se repite hasta que la matriz contenga solo elementos terminales (elementos que indican la presencia o ausencia de una conexión), o lo que es lo mismo, hasta que un patrón de conectividad quede completamente especificado. Esto permite definir la arquitectura de una ANN.

La pregunta ahora es como obtener el conjunto de reglas evolutivas para construir una ANN. La respuesta es evolucionarlas. Se puede codificar el conjunto entero de reglas como un individuo o codificar cada regla como un individuo. Luego mediante un algoritmo evolutivo se van obteniendo nuevas reglas que permiten formar nuevas arquitecturas de ANN.

3.2.3 Evolución de la función de transferencia de los nodos

La discusión de la evolución de arquitecturas, hasta aquí, solo trata con la estructura topológica de una arquitectura. La función de transferencia de cada nodo en la arquitectura se ha asumido como fija y predefinida por humanos expertos, sin embargo, la función de transferencia es una parte importante de la arquitectura de una red neuronal y tiene un impacto significativo en su desempeño. A menudo se asume que la función de transferencia es la misma para todos los nodos en una red, al menos para todos los nodos en la misma capa.

Muchos investigadores desarrollaron trabajos al respecto, por ejemplo en algunos casos adoptaron un esquema donde una cierta cantidad fija de individuos en la población inicial usaba una función de transferencia sigmoidea y el resto usaba una función de transferencia gaussiana. La evolución se utilizó para determinar el porcentaje óptimo en el uso de esas dos funciones de transferencia. La función de transferencia sigmoidea o gaussiana en sí mismas no fueron evolucionadas.

Otros autores [Liu, Y. and Yao, X.: 1996], utilizaron programación evolutiva para evolucionar ANNs con funciones sigmoideas y gaussianas. En lugar de fijar el número total de nodos y evolucionar el porcentaje óptimo de utilización de las funciones en diferentes nodos, permitieron que sus algoritmos pudieran cambiar el número de nodos en una red, cada nodo usaba una función de transferencia sigmoidea o gaussiana. El tipo de nodos agregados o borrados era elegido al azar. Buenos resultados fueron reportados con este esquema.

3.2.4 Evolución simultanea de arquitectura y pesos de conexión

Las aproximaciones evolucionarias discutidas hasta aquí, en el diseño de arquitecturas de redes neuronales, evolucionan solo arquitecturas sin tener en cuenta los pesos de conexión. Estos deben ser aprendidos después de que una arquitectura cercana a la óptima fue encontrada, lo cual se hace más notable cuando se usa un esquema de codificación indirecto.

Un gran problema con la evolución de arquitecturas sin los pesos de conexión es que no permite una adecuada evaluación del fitness. La primera causa tiene que ver con la inicialización aleatoria de los pesos, diferentes pesos iniciales elegidos en forma aleatoria pueden producir diferentes resultados de entrenamiento. Por lo tanto el mismo genotipo puede tener fitness muy diferentes debido a los pesos aleatorios iniciales utilizados en el entrenamiento. La segunda causa tiene que ver con el algoritmo de entrenamiento. Diferentes algoritmos de entrenamiento pueden producir diferentes resultados aun para el mismo conjunto de pesos iniciales.

Estos problemas pueden engañar a la evolución como consecuencia de que si el fitness de un fenotipo generado de un genotipo G1 es más alto que el generado de un genotipo G2, no significa que G1 verdaderamente sea de mayor calidad que G2. Para reducir este problema,

una arquitectura tiene que ser entrenada varias veces con diferentes pesos iniciales y finalmente, se debe usar el resultado promedio para estimar el fitness de término medio. Este método incrementa el tiempo de cómputo para la evaluación del fitness en forma drástica. Por esta causa todos los estudios realizados utilizaron pequeñas redes neuronales para la evolución. Es claro que la evolución de arquitecturas sin ninguna información acerca de los pesos de conexión tiene dificultades en la evaluación del fitness en forma precisa. Como resultado, la evolución se vuelve muy ineficiente. Una forma de reducir los efectos de este problema, es evolucionar arquitecturas de redes neuronales y pesos de conexión de forma simultanea. En este caso cada individuo en la población es una especificación de una red neuronal con información completa de los pesos. En estas condiciones solo existe un mapeo de un genotipo a su fenotipo, lo que permite que la evaluación del fitness sea precisa.

Una cuestión importante en la evolución de redes neuronales artificiales es la elección de los operadores de búsqueda utilizados en los algoritmos evolutivos, como por ejemplo el crossover y la mutación. Sin embargo, el uso de crossover parece contradecir las ideas básicas detrás de las redes neuronales, porque el crossover trabaja mejor cuando existen bloques constructores, pero no es claro que ocurra con redes neuronales dado que ellas enfatizan la representación distribuida del conocimiento. El conocimiento en una red neuronal está distribuido entre todos los pesos de la misma. La recombinación de una parte de una red neuronal con otra parte de otra red es probable que destruya ambas.

3.2.5 Neuroevolución simbiótica adaptiva (SANE)

Nuevamente en el campo de la evolución de los pesos de conexión, varias mejoras fueron propuestas por diversos autores y probadas de forma empírica con muy buenos resultados. En particular se analizarán los métodos SANE y ESP, los cuales tienen características similares debido a que trabajan con poblaciones de cromosomas que codifican los pesos de las neuronas de las capas ocultas de una red, pero lo hacen con enfoques diferentes. En realidad ESP es una mejora del método SANE.

En la mayoría de las aplicaciones neuroevolutivas, cada individuo en la población representa una red neuronal completa que es evaluada independientemente de las otras redes. Tratando cada miembro como una solución completa separada, el algoritmo evolucionario focaliza la búsqueda hacia un único individuo dominante. Esto, puede impedir el progreso de la búsqueda en tareas complejas y dinámicas. En contraste, SANE restringe el alcance de cada individuo a una única neurona, mas precisamente a neuronas de la capa oculta. En SANE, las redes neuronales, se construyen combinando neuronas. La Figura 3.9 ilustra la diferencia entre la neuroevolución tradicional y la neuroevolución realizada por SANE.

Debido a que una única neurona no puede realizar una tarea completa, debe optimizar solo un aspecto de la red neuronal y conectarse con otras neuronas que optimicen otros aspectos. La evolución simbiótica, es un tipo de evolución donde los individuos cooperan entre ellos y confían en la presencia de otros individuos para sobrevivir. Por esta causa SANE puede ser caracterizada como una evolución simbiótica.

Una ventaja de la evolución simbiótica es que las especializaciones de las neuronas aseguran diversidad lo cual reduce la convergencia de la población. Una neurona no puede realizar el trabajo de toda la población dado que para lograr buenos desempeños debe haber otras especializaciones presentes. Si una especialización se extiende demasiado, sus miembros no siempre serán combinados con otras especializaciones. De este modo, las soluciones parciales redundantes obtendrán valores de fitness pobres y, por consiguiente, la evolución se volverá en contra de los individuos de especializaciones dominantes. Este esquema es muy diferente de los métodos neuroevolutivos tradicionales que, si bien, pueden encontrar una solución global, con frecuencia encuentran soluciones locales debido a una convergencia prematura.

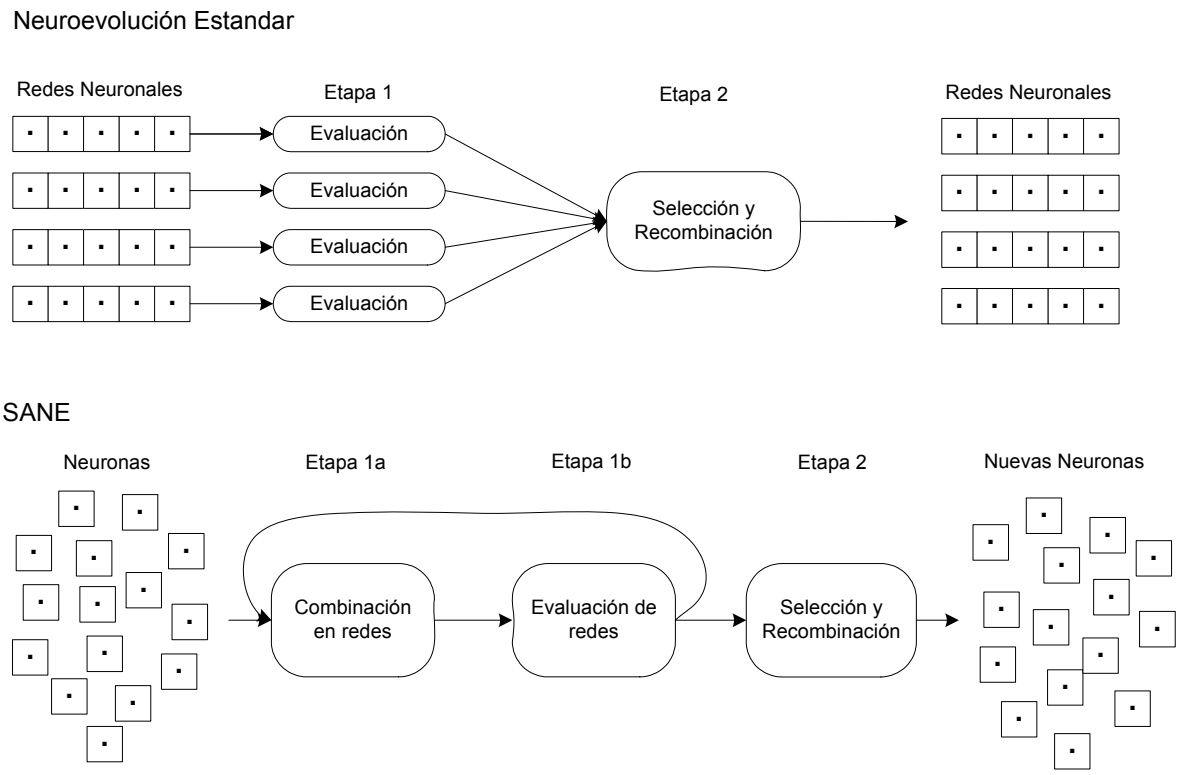

Figura 3.9 Neuroevolución realizada por SANE comparada con la neuroevolución estándar. La aproximación estándar mantiene una población de redes neuronales y las evalúa independientemente. SANE mantiene una población de neuronas y las evalúa en conjunto con otras neuronas. El paso 1 en SANE se divide en dos. Las neuronas son continuamente combinadas con otras y las redes resultantes son evaluadas en una tarea. Cada neurona recibe un fitness en función del desempeño de la red en la cual participó.

Si bien SANE es una buena herramienta, no es suficientemente poderosa para generar redes capaces de resolver tareas complejas. En este caso, se requieren combinaciones útiles de neuronas para lo cual es necesario mantener ese tipo de información. La combinación de neuronas al azar es indeseable por dos razones. Primero, las neuronas pueden ser

combinadas con otras con las cuales no trabajan bien. El segundo problema es que la calidad de las redes varía mucho durante la evolución. Por lo tanto se requiere algún mecanismo para mantener el conocimiento de las buenas combinaciones de neuronas.

El método utilizado por SANE para mantener combinaciones útiles es evolucionar una capa de registros de redes neuronales, también llamados blueprints. La población de blueprints mantiene un registro con las combinaciones de neuronas más efectivas encontradas en la población actual de neuronas y usa este conocimiento como punto de partida para formar las redes neuronales en la siguiente generación. La Figura 3.10 muestra la relación entre el blueprint y la población de neuronas. Cada blueprint mantiene una colección de neuronas que se han desempañado bien en conjunto. SANE utiliza selección y recombinación para explotar este conocimiento, intentando obtener mejores combinaciones en generaciones futuras.

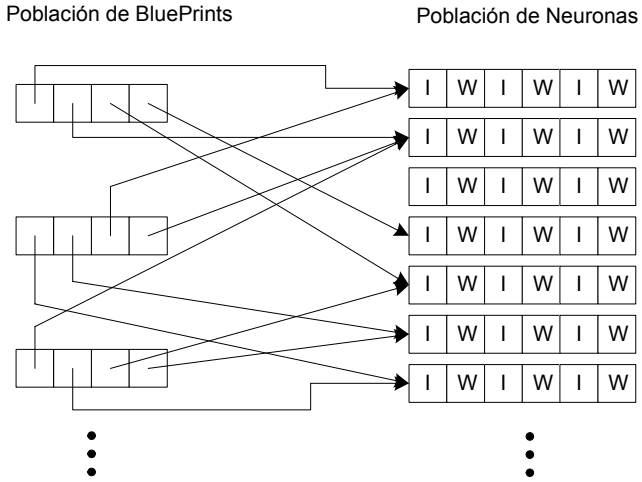

Figura 3.10 El diagrama muestra una población de blueprints relacionados con una población de neuronas. Cada miembro de la población de neuronas especifica una serie de conexiones (etiquetas y pesos) que van a la capa de entrada o a la capa de salida de una red neuronal. Cada miembro de la población de blueprints especifica un conjunto de referencias a neuronas específicas las cuales serán usadas para construir una red neuronal. La población de neuronas representa redes parciales con buen rendimiento, mientras que la población de blueprints mantiene combinaciones efectivas de esas redes parciales.

Los blueprints producen mejores evaluaciones de neuronas y concentra la búsqueda en las mejores redes neuronales. Sin embargo, la ventaja primaria de SANE, es la explotación de las mejores redes encontradas durante la evolución. Los cromosomas de los bluesprints permiten que las mejores redes sean reconstruidas. Además, mediante la evolución de la población de blueprints, se producen recombinaciones de las mejores asociaciones de neuronas, lo cual permiten formar nuevas colecciones de neuronas potencialmente mejores.

La evolución en SANE procede en la siguiente secuencia de pasos:

1. **Inicialización**. Se especifica el número de unidades ocultas *u* en las redes que serán formadas y se crea una población de neurocromosomas. Cada cromosoma codifica los pesos entrada y salida de una neurona oculta mediante una cadena de números reales. Los pesos son elegidos de forma aleatoria. También se crea una población de blueprints que especifican neuronas que formarán una red neuronal.

2. **Evaluación**. En esta etapa, se selecciona un conjunto de *u* neuronas de la población usando un blueprint. Estas neuronas se utilizan para ensamblar la capa oculta de una red feedforward. La red es enviada a una prueba en donde se la evaluará en una tarea y finalmente se la recompensará con un puntaje de fitness. Este puntaje es asignado al blueprint. Este paso de evaluación se repite hasta procesar todos los individuos de la población de blueprints. Luego cada neurona recibe la suma de los fitness de las mejores 5 redes en las cuales participó.

3. **Recombinación**. Después de la evaluación las neuronas son ordenadas de mayor a menor en función de su fitness. Por cada neurona perteneciente al 25% con mayor calificación, se selecciona al azar una pareja del mismo conjunto. De cada pareja se crean dos hijos: uno a través de un crossover de un punto y el otro es una copia exacta de algún padre. Los dos hijos reemplazan a las dos peores neuronas en la población. Como último paso en una generación, se realiza la mutación con una probabilidad del 0.1% en cada posición de cada cromosoma dentro de la población. En la población de blueprints, el crossover intercambia las referencias a las neuronas. La mutación toma cada blueprint y, con una probabilidad del 1%, cambia de forma aleatoria un componente del cromosoma, haciendo que referencia a otra neurona de la población.

4. **Iteración**. Volver al paso 2 si no se alcanzó el objetivo.

La aproximación SANE ha probado ser más rápida y eficiente que otros métodos de aprendizaje por refuerzo [Moriarty, D. E. et. al. 1996]. La razón es que evolucionando soluciones parciales (neuronas) en lugar de soluciones globales (redes neuronales) automáticamente mantiene diversidad en la población. Si un tipo de neurona comienza a predominar en la población, las redes serán a menudo formadas con diversas copias del mismo genotipo. A causa de que las tareas dificultosas generalmente requieren una buena diversidad de neuronas en la capa oculta de las redes, tales redes no pueden realizar su tarea de forma adecuada. Ellas obtienen fitness bajos, y el genotipo dominante será reemplazado por nuevos individuos, lo cual traerá nuevamente diversidad a la población. En las etapas avanzadas de SANE, en lugar de que la población converja alrededor de un simple individuo como sucede con los algoritmos genéticos tradicionales, las neuronas de la población se acumulan en "especies" o grupos de individuos que realizan funciones especializadas dentro del comportamiento objetivo.

3.2.6 Enforced Sub-Populations (ESP)

En Enforced Sub-Populations, como en SANE, las poblaciones consisten de neuronas individuales en lugar de redes completas y para formar una red completa se selecciona un conjunto de neuronas de las poblaciones. Sin embargo, ESP mantiene una población separada para cada una de las u unidades que forman la capa oculta en la red, y una neurona puede solo ser recombinada con miembros de su propia subpoblación. (Véase Figura 3.11)

ESP evoluciona más rápido que SANE por dos razones: Primero, las subpoblaciones que gradualmente se van formando en SANE ya están circunscriptas por diseño en ESP. Las "especies" no tienen que organizarse en si mismas dentro de una única gran población, y su especialización progresiva no es obstaculizada por recombinaciones con elementos de otras especies que generalmente tienen roles relativamente ortogonales dentro de la red. Segundo, a causa de que las redes formadas por ESP siempre consisten de un representativo de cada especialización en evolución, una neurona es siempre evaluada midiendo que tan bien realiza su rol dentro del contexto donde debe funcionar la red. En SANE, las redes pueden contener múltiples miembros de la misma especialización y omitir miembros de otras, con lo cual sus evaluaciones son menos consistentes.

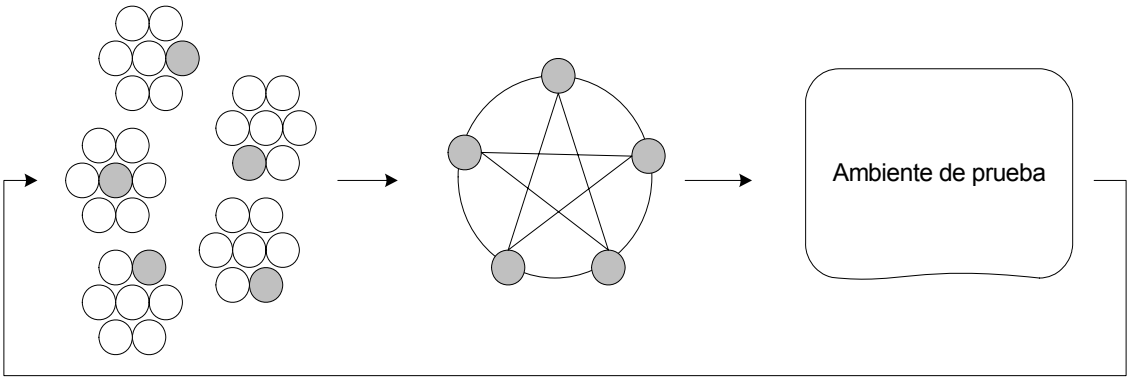

Figura 3.11 El método ESP (Enforced Sub-Populations). La población de neuronas se separa en subpoblaciones mostradas aquí como agrupaciones de círculos. La red es formada mediante una selección de una neurona de cada subpoblación en forma aleatoria.

La contribución principal de ESP, sin embargo, es que esta permite también la evolución de redes recurrentes. El comportamiento de una neurona en una red recurrente es críticamente dependiente de las neuronas a las cuales ésta está conectada. Dado que SANE forma redes por medio de selección aleatoria de neuronas de una única población, una neurona no puede confiarse de que será combinada con neuronas similares en cualquier prueba. Una neurona que se comporta de una forma en una prueba puede comportarse muy diferente en otra, y SANE no puede obtener información precisa acerca del fitness de neuronas recurrentes. La arquitectura de subpoblaciones de ESP permite que la evaluación de la neurona sea más consistente.

La evolución en ESP procede de la siguiente manera:

1. **Inicialización**: Se especifica el número de nodos ocultos *u* que formarán una red y se crea una subpoblación de neurocromosomas para cada uno. Cada cromosoma codifica los pesos de entrada y salida de una neurona con una cadena de números reales seleccionados de forma aleatoria.

2. **Evaluación**: Se selecciona un conjunto de *u* neuronas al azar, una neurona de cada subpoblación, para formar la capa oculta de una red neuronal. La red es enviada a una prueba en la cual es evaluada y recompensada con un puntaje de fitness. El puntaje es agregado al fitness acumulativo de cada neurona que participó en la red. Este proceso es repetido hasta que cada neurona haya participado en un promedio de *t* pruebas.

3. **Recombinación**: Se calcula el fitness promedio de cada neurona dividiendo su fitness acumulativo por el número de pruebas en las cuales participó. Las neuronas dentro de cada subpoblación son clasificadas utilizando su fitness promedio. Cada neurona perteneciente al primer cuarto es recombinada con una neurona de alta clasificación utilizando crossover de un punto y mutación con poca frecuencia. La descendencia resultante reemplaza la mitad de las neuronas con menor clasificación de la subpoblación.

4. **Iteración**: El ciclo de evaluación-recombinación se repite hasta que se encuentre una solución óptima o hasta que se alcance un límite de tiempo.

A medida que la evolución progresa, cada subpoblación perderá diversidad. Este es un problema, especialmente en evolución incremental, a causa de que una población que converge no puede fácilmente adaptarse a una nueva tarea. Para lograr la adaptación a nuevas tareas sin importar la convergencia, ESP es combinada con una técnica de búsqueda iterativa conocida como Delta Coding. Esta técnica, como se explicó en la sección 2.3.13, permite mantener la diversidad genética y mejora el desempeño de un algoritmo genético permitiéndole hacer ajustes finos locales. Sin embargo, su potencial para comportamientos adaptivos reside en que facilita la transición de tareas. Delta Coding provee un mecanismo para hacer transiciones a tareas cada vez más demandantes.

$$t_1 \text{ -}\Delta\text{-> } t_2 \text{ -}\Delta\text{-> } t_3 \text{ -}\Delta\text{-> } \ldots\ t_{n-1} \text{ -}\Delta\text{-> } t_n$$

Cada término "t_i -Δ-> t_{i+1}"(Transición delta) involucra salvar la mejor red *Mejor*(t_i) de la última generación de la tarea que se está evaluando t_i, y luego inicializar una subpoblación de Δ-cromosomas que son entonces evolucionados para adaptar *Mejor*(t_i) a la siguiente tarea t_i+1. Cuando t_i+1 ha sido evolucionada, *Mejor*(t_i+1) (la mejor modificación de Mejor(t_i)) es generada agregando el mejor Δ-cromosoma a *Mejor*(t_i), la solución *Mejor*(t_i+1) será entonces adaptada a t_i+2, y así hasta la última tarea.

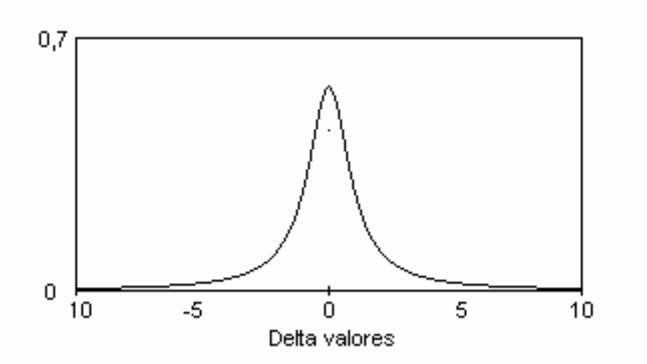

Figura 3.12 Distribución de Cauchy para α = 0,5. La mayoría de los Δ-valores representan pequeñas modificaciones de la mejor solución, pero grandes valores son tambien posibles.

En algunas transiciones de tareas, algunos pesos en la mejor solución pueden necesitar cambiar radicalmente. Esto es especialmente cierto si la nueva tarea introduce nuevos patrones de entrada. Por lo tanto, Δ-valores iniciales deben ser generados usando una función de densidad de probabilidad que concentre la mayoría de los valores en el espacio de búsqueda local mientras que en ocasiones permita obtener valores con una magnitud mucho más alejada. Una distribución que tiene estas propiedades deseables es la distribución de Cauchy, la cual está representada en forma gráfica en la Figura 3.12,

$$f(x) = \alpha / (\pi(\alpha + x^2))$$

Con esta distribución el 5% de los Δ-valores caerán dentro del intervalo [$-\alpha, \alpha$] y el 99.9% dentro del intervalo [318,3 - α, 318,3 + α].

4 APRENDIZAJE Y SISTEMAS MULTIAGENTE

El aprendizaje representa un área desafiante de la inteligencia artificial (AI). La importancia del aprendizaje está fuera de cuestionamiento, pues esta habilidad representa uno de los componentes más importantes del comportamiento inteligente.
Un sistema experto puede realizar extensos y costosos cálculos para resolver un problema. Sin embargo, a diferencia de un ser humano, si se le presenta un problema igual o similar una segunda vez, generalmente no puede recordar la solución y tiene que repetir la secuencia de cálculos. Esto se va a repetir cada vez que el sistema resuelva el problema. La solución obvia para este inconveniente es que el programa aprenda de sí mismo, de su experiencia, por analogías, mediante ejemplos o simplemente diciéndole que hacer.

Aunque el aprendizaje represente un área dificultosa, existen algunos programas que sugieren que no es imposible. Un programa notable es el AM (Automated Mathematician), diseñado para descubrir leyes matemáticas (Lenat 1977, 1982). Inicialmente disponiendo de algunos conceptos y axiomas, AM pudo inducir importantes conceptos matemáticos como la cardinalidad, la aritmética entera, y muchos de los resultados de la teoría actual de los números. AM también conjeturó nuevos teoremas modificando su base de conocimiento.

Programas como el del ejemplo previo, junto a muchos otros, se conocen como programas de aprendizaje máquina. El aprendizaje máquina (ML – Machine learning) es una rama de la inteligencia artificial que está principalmente dedicada a la construcción de programas de computadoras capaces de adquirir nuevo conocimiento o mejorar algún conocimiento ya poseído dentro de algún ambiente particular.

Muchos de los programas de aprendizaje máquina, usan representaciones explícitas del conocimiento y utilizan algoritmos de búsqueda cuidadosamente diseñados para mostrar una cierta inteligencia. Una aproximación muy diferente, intenta construir programas inteligentes usando modelos que imitan la estructura de las neuronas del cerebro humano, tal es el caso de las redes neuronales artificiales; o usando patrones evolutivos tomados de la genética natural, como es el caso de los algoritmos genéticos.

Existe un grupo de problemas, en los cuales se requiere que un grupo de agentes aprenda ciertas habilidades para poder resolverlos. Estos pueden ir desde simples problemas, como por ejemplo el caso de los predadores y presas, hasta muy complejos, como es el caso del fútbol robótico. Desde un punto de vista más general, estos problemas que involucran agentes y requieren mecanismos para coordinar ciertos comportamientos, se conocen como sistemas multi agente.

A continuación se presenta una introducción a los sistemas multi agente, y luego se analizan cuatro métodos de aprendizaje que permiten encarar una solución para este tipo de problemas. Cada método surgió en momentos diferentes y con motivaciones distintas, pero todos pueden ser aplicados a problemas en los cuales existe un grupo de agentes que deben aprender a realizar alguna tarea compleja. En particular el último método de aprendizaje presentado, es un nuevo método propuesto para esta tesis.

4.1 Sistemas Multi Agente

Los sistemas multi agente (MAS – Multi Agent System) pertenecen un una rama de la inteligencia artificial (AI – Artificial Intelligence) que apunta a proveer principios para la construcción de sistemas complejos involucrando múltiples agentes y mecanismos para coordinar comportamientos independientes de los mismos. No hay una definición general aceptada de "agente" en AI, para el propósito de esta tesis, un agente es una entidad con percepción, objetivos, acciones, conocimiento de un dominio y situado en un ambiente. La forma en que un agente actúa, o el mapeo de percepciones a acciones sobre el tiempo, es su comportamiento. Cuando un grupo de agentes en un sistema multi agente comparte objetivos comunes a largo plazo, se puede decir que forman un equipo. Los miembros de un equipo coordinan su comportamiento adoptando procesos cognoscitivos compatibles y también afectando directamente las entradas que perciben otros mediante mecanismos de comunicación. Otros agentes en el ambiente que tienen objetivos opuestos a los del equipo son llamados equipos adversarios.

4.1.1 Aprendizaje por refuerzo en el contexto de un agente

El aprendizaje por refuerzo data de los tempranos días de la cibernética y trabajo en estadísticas, psicología, neurociencia y ciencias de la computación. En los últimos cinco a diez años, ha atraído rápidamente el interés de las comunidades de la inteligencia artificial y aprendizaje máquina. Este método propone una forma de programar agentes mediante un sistema de premios y castigos sin especificar como será realizada la tarea. Sin embargo, existen muchos obstáculos computacionales para que el método sea exitoso.
El Aprendizaje por refuerzo, le permite a un agente aprender un cierto comportamiento en un ambiente dinámico a través de interacciones de prueba y error.

Existen dos estrategias principales para resolver problemas de aprendizaje por refuerzo. El primero es buscar en el espacio de comportamientos para encontrar uno que cumpla con el objetivo en el ambiente. Esta aproximación ha sido tomada por los algoritmos genéticos y la programación genética así como también en algunas novedosas técnicas de búsqueda. El segundo es usar técnicas estadísticas y métodos de programación dinámica para estimar la utilidad de tomar una cierta acción ante un estado determinado del medio.

Modelo de aprendizaje por refuerzo

En el modelo estándar de aprendizaje por refuerzo, un agente es conectado a su medio ambiente vía percepción y acción, como muestra la Figura 4.1

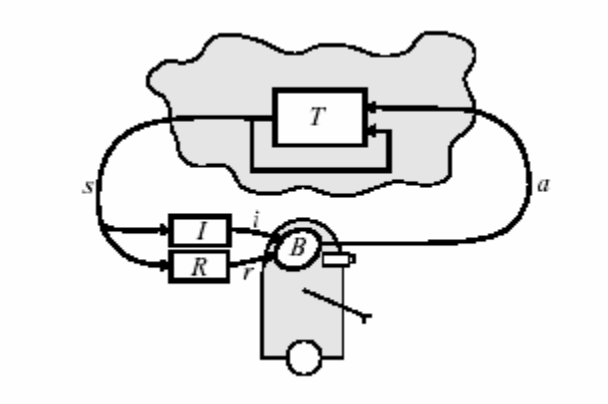

Figura 4.1 Esquema gráfico del aprendizaje por refuerzo.

En cada interacción el agente recibe como entrada alguna información (*i*) *y* alguna indicación del estado actual del medio ambiente (*s*), luego el agente elige una acción (*a*), para generar su salida. La acción cambia el estado del medio ambiente y luego se comunica el valor de esta transición de estado al agente a través de una señal escalar de refuerzo (r). El comportamiento del agente (B), debería elegir acciones que tiendan a incrementar la suma de los valores de la señal de refuerzo durante la evaluación. El agente puede aprender a hacer esto en el transcurso del tiempo a través de evaluaciones sistemáticas de prueba y error, guiado por una amplia variedad de algoritmos.

Formalmente el modelo consiste de:

- Un conjunto discreto de estados del medio ambiente (S).
- Un conjunto discreto de acciones que puede tomar el agente (A).
- Un conjunto de señales escalares de refuerzo, típicamente {0,1}, o valores reales.

La Figura 4.1 también incluye una función de entrada (I), la cual determina como el agente ve el estado del medio ambiente, generalmente esta función es la identidad, es decir que percibe el estado exacto del medio ambiente, pero podría ser cualquier otra función que permita observar solo una parte del medio ambiente.

El aprendizaje por refuerzo difiere del ampliamente estudiado aprendizaje supervisado en diversos aspectos. La diferencia más importante es que no hay presentación de pares de entrada/salida deseadas. En lugar de esto, después de elegir una acción el agente es de inmediato recompensado.

4.2 Aprendizaje por coevolución

En biología, la coevolución, es la influencia mutua en la evolución de dos especies. Cada parte en una relación coevolucionaria ejerce presiones selectivas en la otra, por ello también afecta su evolución. Ejemplos de coevolución incluyen la polinización de las orquídeas por los colibríes. Estas especies coevolucionan porque los colibríes dependen de las flores por su néctar y las flores dependen de los colibríes porque ellos esparcen su polen facilitando su reproducción. La coevolución no implica dependencia mutua.

En el contexto de un agente, la coevolución, consiste en mantener y evolucionar simultáneamente múltiples individuos que realizan diferentes funciones dentro de un ambiente común, estos individuos pueden pertenecer a una o muchas poblaciones. La coevolución puede ser competitiva, en cuyo caso esos roles son adversarios, lo cual provoca que algunos individuos ganen y otros pierdan [Haynes, T. and S. Sen: 1996], o también puede ser cooperativa, en este caso todos los componentes comparten las mismas recompensas y penalidades según los éxitos o fracasos obtenidos [Potter, M. A. and K. A. D. Jong: 2000].

Los tipos de problemas que pueden utilizar coevolución cooperativa son aquellos en los cuales la solución puede ser modularizada naturalmente en subcomponentes que interactúan o cooperan para resolver el problema. Cada subcomponente puede entonces ser evolucionado en su propia población, y cada población contribuye a la solución con sus mejores individuos.

4.3 Aprendizaje por capas

El aprendizaje por capas [Stone, P. and Veloso, M.: 1998] es un paradigma definido como un conjunto de principios para la construcción de un aprendizaje jerárquico que permita solucionar una tarea compleja.

El aprendizaje por capas está definido por 4 principios:

- **Principio 1 (Tratabilidad):** Motivado por el fútbol robótico, el aprendizaje por capas fue diseñado para dominios en los cuales es demasiado complejo aprender un mapeo directo de los sensores de entrada de los agentes a sus actuadores de salida. Se asume que cualquier dominio que tenga restricciones como pueden ser comunicaciones limitadas, tiempo real, ambientes con ruido donde convivan amigos y enemigos, es un dominio demasiado complejo para aprender un mapeo directo de sus sensores a sus actuadores.
  El aprendizaje por capas consiste en dividir el problema en diversas capas de comportamiento y usar técnicas de aprendizaje máquina (ML) en cada nivel. El aprendizaje por capas usa un diseño incremental bottom-up para la descomposición de una tarea en forma jerárquica. Comenzando por comportamientos de bajo nivel, el proceso de creación de nuevas tareas ML continúa hasta alcanzar comportamientos estratégicos de alto nivel que tratan con la complejidad del dominio completo.

- **Principio 2 (Descomposición):** La granularidad de comportamiento apropiada y los aspectos de los comportamientos a ser aprendidos son determinados como una función del dominio específico. La descomposición de una tarea en aprendizaje por capas no es automática. En lugar de esto, las capas son definidas por las oportunidades de utilización de técnicas de aprendizaje máquina en el dominio.
  El aprendizaje por capas, sin embargo, puede ser combinado con cualquier algoritmo para aprender niveles de abstracción.

- **Principio 3 (Aprendizaje):** Las técnicas de aprendizaje máquina son utilizadas como una parte central del aprendizaje por capas para explotar datos que permiten entrenar y/o adaptar todo el sistema.
  El aprendizaje máquina es útil para entrenar funciones que son difíciles de ajustar manualmente y también para la adaptación cuando los detalles de una tarea no son completamente conocidos a priori o cuando ellos pueden cambiar dinámicamente.
  En los casos tradicionales, el aprendizaje puede hacerse en forma pasiva y luego ser congelado en la ejecución de la tarea real. En otros casos, el aprendizaje se realiza en forma activa, dado que el agente necesita adaptarse a situaciones inesperadas pudiendo alterar su comportamiento aún cuando ejecuta su tarea. Como la descomposición de tareas en si misma, la elección del método aprendizaje máquina depende de la subtarea.

- **Principio 4 (Interacciones):** La característica clave del aprendizaje por capas es que cada capa aprendida afecta directamente el aprendizaje en la siguiente capa. Una subtarea aprendida puede afectar la capa subsiguiente de dos maneras: la primera es suministrando una parte del comportamiento usado durante el entrenamiento, y la otra es creando la representación de entrada del algoritmo de aprendizaje. En general, los algoritmos de aprendizaje máquina requieren una representación de la entrada y la salida, un mapeo de entradas a salidas, y ejemplos para el entrenamiento. El objetivo del aprendizaje es generalizar el mapeo desde los ejemplos de entrenamiento que solo proveen salidas correctas para una porción del espacio de entradas.
  Cuando se usa aprendizaje máquina para el aprendizaje de comportamientos, los ejemplos de entrenamiento son generados colocando un agente en una situación correspondiente a una instancia específica de la representación de entrada; permitiéndole a éste actuar; y luego dándole alguna recompensa o indicación del mérito de la acción en el contexto del mapeo deseado. De esta manera, capas aprendidas previamente pueden, en primer lugar, proveer una porción del comportamiento usado durante el entrenamiento, determinando las acciones disponibles o afectando el esfuerzo recibido. En segundo lugar las capas previamente aprendidas pueden también crear las entradas para el algoritmo de aprendizaje afectando o determinando la representación de la entrada del agente.
  Si cada capa aprendida en la descomposición de una tarea afecta directamente a la siguiente capa, entonces el sistema es un sistema de aprendizaje por capas. Sin estás características, un sistema no pertenece al paradigma real de aprendizaje por capas.

En resumen, el aprendizaje por capas es un paradigma de aprendizaje máquina diseñado para permitir que agentes aprendan a cumplir un objetivo en ambientes complejos. El aprendizaje por capas permite definir en forma bootom-up las capacidades de un agente en un dominio complejo. Las oportunidades de utilización de alguna forma de aprendizaje máquina se identifican cuando existe la información o cuando la tarea es imprevisible y las posibles soluciones codificadas a mano son demasiado complejas de realizar. Los comportamientos individuales que se desean aprender son organizados, aprendidos y combinados en forma de capas, cada una facilitando la creación de la siguiente. Los principios de la técnica de aprendizaje por capas están resumidos en la Figura 4.2

1. No es posible aprender un mapeo desde los sensores a los actuadotes directamente.
2. Existe una descomposición jerárquica bottom-up de una tara compleja.
3. Las técnicas de aprendizaje máquina usan datos para entrenar y/o adaptar. El aprendizaje sucede de forma independiente en cada nivel.
4. La salida del aprendizaje de una capa se utiliza como entrada para la siguiente

Figura 4.2 Principios del aprendizaje por capas.

4.3.1 Definición Formal

El aprendizaje por capas puede ser definido formalmente como sigue. Considerar la tarea de aprendizaje que consiste en identificar una hipótesis *h* de una clase de hipótesis *H*. Dicha hipótesis mapea un conjunto de estados *S* a un conjunto de salidas *O*, tal que, basado en un conjunto de ejemplos de entrenamiento, la hipótesis *h* es probablemente (de las hipótesis de *H*) la mayor representante de nuevos ejemplos de entrenamiento no pertenecientes a dicho conjunto.

Cuando se usa el paradigma de aprendizaje por capas, la tarea completa de aprendizaje es descompuesta en capas de subtareas jerárquicas $\{L_1, L_2, \ldots, L_n\}$ con cada capa definida como:

$$L_i = (\vec{F}_i, O_i, T_i, M_i, h_i)$$

Donde:

$\vec{F}_i$ es un vector de entrada con estados relevantes para aprender la subtarea L_i.

$$\vec{F}_i = < F_i^1, F_i^2, ...>. \forall j, F_i^j \in S.$$

O_i es el conjunto de salidas de la subtarea L_i.

$$O_i \subset O.$$

T_i es el conjunto de ejemplos de entrenamiento usados para aprender la subtarea L_i. Cada elemento de T_i consiste de una correspondencia entre un vector de estados $f \in \vec{F}_i$ y una salida $o \in O_i$.

M_i es el algoritmo de aprendizaje máquina usado en la capa L_i para seleccionar un mapeo $\vec{F}_i \rightarrow O_i$ basado en T_i.

h_i es el resultado de correr M_i sobre T_i. h_i es una función de $\vec{F}_i$ a O_i.

Nótese que una capa describe más que una tarea; ésta también describe una aproximación para resolver esa tarea y la solución resultante.

Como fue enunciado en el principio 2 (Descomposición) del aprendizaje por capas, las definiciones de las capas L_i están dadas a priori. El principio 4 (Interacción) utiliza la siguiente estipulación: $\forall i < n$, h_i directamente afecta L_{i+1} en al menos una de tres formas:

- h_i es usada para construir uno o más estados iniciales de F^k_{i+1}.
- h_i es usada para construir elementos de T_{i+1}.
- h_i es usada para recortar el conjunto de salida O_{i+1}.

Queda claro en la definición de $\vec{F}_i$ que $\forall j$, $F_i{}^j \in S$. Dado que $\vec{F}_{i+1}$ puede consistir de nuevos estados construidos usando h_i, la versión más general del caso especial anterior es: $\forall i,j$, $F_i{}^j \in S \cup_{k=1..i-1} O_k$.

Durante el aprendizaje de una subtarea L_i, las subtareas de las capas previas permanecen fijas, con lo cual este método agrega restricciones al proceso de aprendizaje.

4.4 Aprendizaje por capas concurrente

El aprendizaje coevolutivo no provee asistencia humana más allá de la descomposición de tareas en si misma, por este motivo no restringe demasiado el espacio de búsqueda de posibles soluciones. En contraste, el aprendizaje por capas provee una buena proporción de asistencia, restringiendo y guiando el aprendizaje en un espacio de búsqueda más reducido. El aprendizaje por capas concurrente es una aproximación que ocupa un rango intermedio en ese espectro. Este retiene la guía de los ambientes especiales de entrenamiento para las capas inferiores, pero no siempre las deja fijas y pueden seguir adaptándose, lo cual le aporta algo de la flexibilidad del aprendizaje coevolutivo.

Una dificultad del aprendizaje por capas tradicional es que no importa todo el cuidado que se haya puesto en el diseño de los ambientes especiales de entrenamiento de las capas inferiores, siempre están limitados por imperfecciones introducidas por los diseñadores. Siempre existirán, inevitablemente, discrepancias entre los comportamientos que esos ambientes estimulan a aprender y los comportamientos que son óptimos en el ambiente objetivo. El aprendizaje por capas concurrente intenta corregir esas discrepancias permitiendo que ciertas capas inferiores continúen adaptándose mientras las capas más altas están siendo aprendidas. Este nuevo esquema de aprendizaje es consistente con los formalismos existentes del aprendizaje por capas. Solo introduce un poco más de flexibilidad.

El aprendizaje por capas fue originalmente desarrollado para resolver la tarea compleja de aprendizaje del fútbol robótico [Stone, P. 2000]. En la implementación original, el aprendizaje de cada componente fue completado antes de que cualquier capa subsiguiente

fuera aprendida. El aprendizaje por capas concurrente relaja esas restricciones. Cuando se aprende una capa L_i, se selecciona de todas las hipótesis aprendidas previamente algún subconjunto $P \subseteq \{h_1, h_2, ..., h_{i-1}\}$ que se desea que siga aprendiendo en el ambiente actual T_i. El efecto que tales hipótesis tienen sobre T_i, no es fijo a través del aprendizaje de L_i, sino que cambia constantemente mientras esas hipótesis continúan aprendiendo.

En resumen, el aprendizaje por capas concurrente [Whiteson, S. and P. Stone: 2003], mejora el aprendizaje por capas aplicando coevolución en una forma restringida. Por lo tanto, esta aproximación representa un punto medio entre esos dos métodos. Preserva la estructura del aprendizaje por capas pero también ofrece algunas de las flexibilidades de la coevolución.

4.5 Aprendizaje cíclico

El aprendizaje por capas concurrente representa un punto intermedio entre el aprendizaje coevolutivo y el aprendizaje por capas, manteniendo en su estructura características específicas de ambos. Pues si bien el aprendizaje por capas guía el entrenamiento hacia un sector específico del espacio de búsqueda, el método concurrente permite luego explorarlo más allá, lo cual potencialmente permite encontrar nuevas y mejores soluciones.

La coevolución, si bien es muy poco restrictiva y permite encontrar soluciones en cualquier parte del espacio de búsqueda, éste puede ser tan grande que se haría muy difícil sino imposible hallar una solución óptima o simplemente aceptable. Un espacio de búsqueda muy extenso puede ser computacionalmente costoso de explorar.

Cuando en un conjunto de subtareas, cada una es aprendida en un ambiente diseñado especialmente (aprendizaje por capas), se logra una reducción en el espacio de búsqueda y una guía en el proceso de aprendizaje. Si luego se permite que cada subtarea continúe adaptándose cooperativamente con el resto en el ambiente objetivo, como propone el aprendizaje por capas concurrente, nuevamente se expandiría el espacio de búsqueda y esto podría ocasionar que se vuelva difícil o costoso encontrar mejores soluciones.

Una pregunta que surge en este punto y cuya respuesta es la que le da origen al nuevo método propuesto en esta tesis, es: ¿Qué sucedería si en lugar de expandir el espacio de búsqueda directamente, se lo hace en forma ordenada y restringida?. La respuesta quedará al descubierto posteriormente cuando se detallen los resultados de las pruebas realizadas.

Para permitir la exploración del espacio de búsqueda en forma restringida, cada subtarea sigue su adaptación en el dominio objetivo, pero durante el aprendizaje de una, el resto de las subtareas permanecen fijas. La exploración se logra de forma ordenada, debido a que se establece una secuencia en el proceso de aprendizaje de cada subtarea. Cuando termina el aprendizaje de una subtarea se pasa a la siguiente en la secuencia hasta terminar con todas. Cuando termina el aprendizaje de la última subtarea se reinicia el proceso una cierta cantidad de veces, esto hace que se forme un ciclo el cual le da nombre a esta aproximación. El hecho de dejar ciertos componentes fijos mientras otro sigue aprendiendo,

permite que éste último mejore su comportamiento e integración con el resto ya entrenado, lo cual, además, implica un refinamiento en el desempeño de las subtareas en conjunto.

El aprendizaje cíclico permite hacer ajustes finos de comportamiento a los componentes obtenidos por los métodos tradicionales pero en particular a los obtenidos mediante aprendizaje por capas.

4.5.1 Definición formal

Sea L un conjunto de subtareas $L = \{ L_1, L_2, \ldots , L_n \}$ utilizadas en el aprendizaje por capas, sea C el conjunto de controladores obtenidos de dicho aprendizaje $C = \{c_1, c_2, c_3, \ldots, c_n\}$, donde cada c_i resuelve la tarea L_i , y sea E_i un proceso que representa el nuevo aprendizaje de un subcontrolador c_i, entonces el aprendizaje cíclico puede verse como:

$$c_1 = E_1(c_1) \rightarrow c_2 = E_2(c_2) \rightarrow \ldots \rightarrow c_n = E_n(c_n) \rightarrow c_1 = E_1(c_1) \rightarrow \ldots$$

Donde "$c_i = E_1(c_i)$" representa un refinamiento del controlador c_i en un ambiente de entrenamiento, y el signo "–>" indica el orden en el cual se van realizando los entrenamientos.
La Figura 4.3 muestra un esquema gráfico del aprendizaje cíclico.

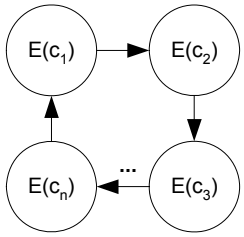

Figura 4.3 Una breve representación gráfica del aprendizaje cíclico. Cada círculo representa el aprendizaje de un subcontrolador y las flechas indican el orden en el cual se van realizando los mismos.

4.5.2 Algoritmo de aprendizaje cíclico

El algoritmo de aprendizaje cíclico permite realizar refinamientos sucesivos de un grupo de subcontroladores que deben utilizarse conjuntamente para resolver una tarea. El objetivo de este método es ir mejorando el desempeño de cada subcontrolador y la interacción con el resto mediante entrenamientos adicionales de cada uno mientras los otros controladores permanecen fijos.

A continuación se detalla en algoritmo de aprendizaje cíclico en pseudo código:

Comenzar *{programa principal}*
Sea C={c_1,c_2,c_3,..,c_n} un conjunto de subcontroladores iniciales.
Sea O el objetivo a alcanzar.
Sea Z la cantidad máxima de ciclos a realizar.
Ciclos = 0 *{hasta ahora ningún ciclo se ha realizado}*
Mientras (objetivo O no sea alcanzado) **y** (Ciclos < Z)
Para cada subcontrolador $c_i \in C$, con i = 1.. n.
$c'_i = Aprendizaje(c_i)$
reemplazar c_i por c'_i en C
Fin Para
Ciclos = Ciclos + 1;
Fin Mientras
Fin *{ programa principal}*

El proceso *Aprendizaje*(c_i) utiliza algún algoritmo aprendizaje máquina para mejorar el desempeño de un subcontrolador. Este proceso retorna un nuevo subcontrolador que reemplazará al subcontrolador original.

Este método de aprendizaje, al igual que los anteriormente expuestos, será utilizado para comparar su desempeño en un dominio de prueba llamado keepaway, el cual será presentado en el capítulo siguiente.

5 DOMINIO DE PRUEBA

El dominio elegido para poner a prueba los distintos métodos de aprendizaje es un juego llamado keepaway. Este es un dominio que está considerado como una subtarea del juego de fútbol robótico [Stone P., 2000]. En keepaway es un juego en el cuál un equipo, los keepers, intentan mantener posesión de la pelota dentro de una región limitada, mientras otro equipo, los takers, intentan ganar posesión de la misma. Cuando sea que el taker se apodere de la pelota o ésta salga de la región de juego el episodio termina y los jugadores son inicializados nuevamente para comenzar otro episodio.

Los parámetros considerados en este juego tienen que ver con el tamaño de la región, el número de keepers, y el número de takers. Para el caso de las pruebas presentadas aquí se utilizó un esquema 3 vs. 1 (3 keepers contra 1 taker) jugando en un campo de forma circular ver Figura 5.1

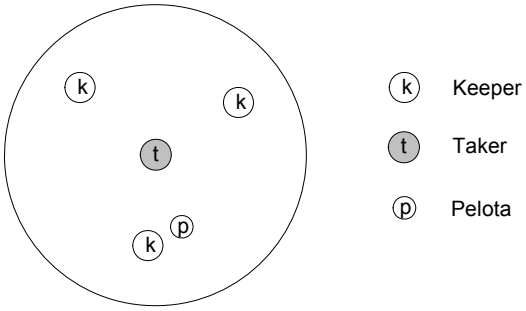

Figura 5.1 Un juego de keepaway 3 vs. 1.

En el simulador de fútbol RoboCup, los agentes típicamente tienen algunas limitaciones en cuanto a las capacidades sensitivas, cada jugador puede ver objetivos dentro de un cono de visión de 90 grados, y la precisión de la ubicación de un objetivo sensado se degrada con la distancia.

Keepaway es un subproblema del fútbol robótico con algunas modificaciones tendientes a facilitar la resolución del mismo. Las principales simplificaciones hechas al juego son que hay pocos jugadores involucrados, cada uno tiene una visión de 360 grados, juegan en un área pequeña y siempre mantienen el mismo objetivo principal, con lo cuál no necesitan hacer consideraciones sobre acciones de defensa o ataque. No obstante, las habilidades necesarias para jugar bien keepaway son también muy útiles en el problema completo de fútbol robótico.

Keepaway es una tarea suficientemente compleja en la cual es muy difícil que las soluciones codificadas a mano pueden producir buenos comportamientos. Sin embargo, también es una tarea suficientemente simple que permite utilizar aproximaciones de aprendizaje máquina para la resolución de la misma.

Muchos investigadores han resuelto con éxito distintas subtareas del fútbol robótico mediante técnicas aprendizaje máquina, pero todavía la resolución del juego completo, en donde 11 jugadores aprenden a jugar exitosamente, parece estar más allá de las modernas técnicas de aprendizaje máquina. Como consecuencia de las distintas técnicas utilizadas y las distintas subtareas resueltas, se hace muy dificultosa la comparación coherente de los resultados obtenidos. Por esta razón se ha propuesto en [Stone, P. and Sutton, R. S.: 2002] el keepaway como dominio de investigación para poder contrastar distintas aproximaciones aprendizaje máquina.

Shimon Whitenson en [Whiteson, S. and P. Stone: 2003] ha utilizado el keepaway como dominio de prueba del aprendizaje por capas concurrente. Pietro en [Pietro, A. D., et. al. 2002], lo utilizó para probar su resolución mediante algoritmos evolucionarios.

El juego Keepaway es un dominio desafiante por las siguientes razones:

1. El espacio búsqueda es demasiado extenso para explorarlo exhaustivamente.

2. El espacio de acciones es continuo.

3. Múltiples compañeros necesitan aprender simultáneamente.

4. Los keepers son relativamente grandes comparados con el área de juego, lo cual hace dificultoso el movimiento y posicionamiento alrededor de la pelota.

5. La pelota no se mueve mucho más rápido que los jugadores, lo cual previene que los keepers puedan hacerse pases rápidamente alrededor del taker.

6. Los keepers no poseen ninguna habilidad para manejar la pelota. Ellos están modelados como simples cilindros y carecen de cualquier forma de posesión de la pelota y movimiento con ella. Si corren contra la pelota, ésta rebota.

Por estas causas, el juego keepaway, requiere comportamientos complejos. Estos van desde el procesamiento de datos de entrada acerca de cada keeper, el taker y la pelota, hasta la toma de decisiones respecto a cual es el mejor curso de acción en cada momento del juego y la adquisición de la habilidad necesaria para realizarla.

Analizado desde un punto de vista más abstracto, el dominio keepaway, así como muchos otros dominios pertenecen a un conjunto de sistemas conocidos como sistemas multi agente. Para más detalles ver sección 4.1.

5.1 Resolución del juego Keepaway

Esta sección presenta las distintas herramientas y tecnologías utilizadas para la resolución de keepaway. Entre ellas se pueden mencionar: técnicas de descomposición de tareas, árboles de decisión, redes neuronales, algoritmos genéticos, neuroevolución con ESP, etc.

El objetivo para la resolución del juego, es obtener un controlador capaz de comandar los jugadores de keepaway de forma exitosa. Dicho controlador estará formado por redes neuronales artificiales evolutivas de tipo feedforward con una capa oculta.

Dado que es muy difícil que una sola red neuronal aprenda a resolver el juego completo, es preciso realizar una descomposición del problema en subproblemas más sencillos de resolver. La descomposición de tareas es un principio general y poderoso en la Inteligencia Artificial que ha sido usado exitosamente en tareas como el Fútbol robótico completo [Stone P., 2000].

Con esta técnica es posible hacer que un problema sea tratable descomponiéndolo en subproblemas menores y fáciles de resolver. En particular si la tarea puede ser dividida en subtareas independientes, cada subtarea puede ser aprendida por separado. De esta manera, es posible aplicar métodos de aprendizaje como por ejemplo el método coevolutivo, por capas, por capas concurrentes y cíclico, cuyas bases teóricas fueron expuestas en el capítulo 4.

La división de tareas utilizada para el keepaway fue propuesta en [Whiteson, S. and P. Stone: 2003] y consiste de las siguientes subtareas: Intercepción, pase, evaluación de pase y posicionamiento. Cada subtarea será realizada por una red neuronal especializada, lo cual elimina el uso de una sola red que resuelva la tarea completa. A continuación se detallan las subtareas propiamente dichas:

- **Intercepción:** El objetivo de esta tarea es simplemente tomar la pelota tan rápidamente como sea posible. La estrategia obvia de correr directamente hacia la pelota, es óptima solo si la pelota no está en movimiento. Cuando la pelota tiene velocidad, un interceptor ideal debe anticiparse al movimiento de la misma.

- **Pase:** En esta tarea, se quiere lograr que el agente patee la pelota en un ángulo especificado. El pase es complicado porque el agente no puede especificar directamente la dirección en que tiene que ir la pelota, en lugar de esto, el ángulo depende de la posición del agente relativa a la pelota. Por esto la tarea de pase requiere de movimientos precisos para aproximar la pelota a la velocidad correcta y en el ángulo correcto.

- **Evaluación de Pase:** El trabajo de evaluación de pase consiste en analizar el estado actual del juego y evaluar la probabilidad de pasar con éxito la pelota a un receptor específico.

- **Posicionamiento:** El comportamiento de posicionamiento se usa cuando un keeper no tiene la pelota y no es receptor de un pase. Claramente, este agente quiere buscar una posición donde pueda recibir exitosamente la pelota. Sin embargo, un comportamiento óptimo de posicionamiento, no solo posiciona a un agente donde es más probable que el pase tenga éxito, sino que también debería posicionarlo donde la recepción del mismo sea estratégicamente más ventajosa en función de oportunidades futuras de pase.

Cada una de las tareas anteriormente expuestas será realizada por una red neuronal. Estas redes en conjunto y entrenadas de forma adecuada, forman el controlador de un jugador de keepaway que será capaz de resolver la tarea en forma completa.

Este esquema de subdivisión de tareas por si solo, no permite que el controlador pueda funcionar. Falta información concreta que defina la lógica de cuando y en que orden evaluar y ejecutar las subtareas. Dado que el KeepAway es un dominio continuo (existen infinitos estados), en cada instante de tiempo el controlador debe tomar decisiones, para esto delega responsabilidades en los subcontroladores, pero le hace falta saber a quien y como.

Para resolver el problema anterior, se utiliza un árbol de decisión (Véase Figura 5.2). Este árbol define una estructura lógica de organización y evaluación de las subtareas. En cada turno, el controlador de un keeper utiliza el árbol de decisión para seleccionar un subcontrolador y delegar en él la responsabilidad de realizar la subtarea para la cual fue entrenado.

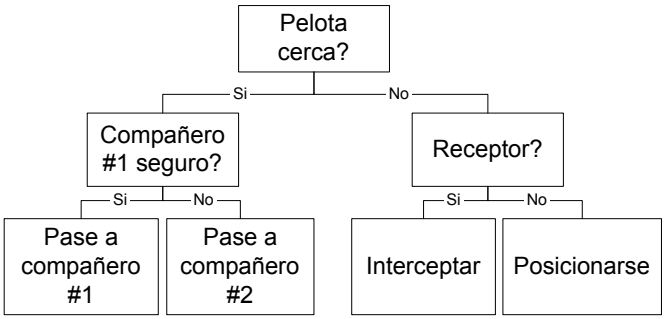

Figura 5.2 Un árbol de decisión para controlar keepers en la tarea de keepaway. El comportamiento de cada una de las hojas es aprendido por una red neuronal a través de neuroevolución. Una red también es evolucionada para decidir a que compañero hacerle un pase. El resto son acciones predefinidas no evolucionadas.

Los comportamientos especificados en las hojas del árbol son realizados por subcontroladores formados por redes neuronales así como también el del nodo "Compañero #1 seguro?". El resto de los nodos tiene un comportamiento fijo.

La lógica utilizada en el árbol de decisión expuesto en la Figura 5.2 es la siguiente: si el agente está a menos de una cierta distancia de la pelota, se utiliza el subcontrolador de evaluación pase, el cual permite saber que compañero tiene más probabilidades de tener una recepción exitosa, y luego con el subcontrolador de pase, se intenta patear la pelota en la dirección del compañero seleccionado como receptor. Por el contrario, si el agente está más allá de una cierta distancia, se evalúa si es el receptor de un pase. Si lo es, utilizando el subcontrolador de intercepción intenta apoderarse de la pelota. Si no es, busca un lugar adecuado para futuras recepciones utilizando el subcontrolador de posicionamiento.

Para el caso de las pruebas realizadas en esta tesis, se fijó la distancia que define si un jugador esta cerca o no de la pelota a una longitud de tres veces el diámetro de un keeper. Es decir, que si la pelota está a menos distancia que la longitud de tres diámetros de un keeper, se considera como cerca, de lo contrario se considera lejos.

Es importante mencionar que todos los nodos de árbol podrían ser controlados por redes neuronales en lugar de tener un comportamiento fijo programado a mano. Incluso el árbol de decisión podría ser reemplazado por una red neuronal capaz de realizar la misma función. Estos escenarios no fueron tenidos en cuenta dado que los entrenamientos adicionales de cada nueva red demandan mucho tiempo y no hace a la finalidad de lo que se pretende exponer.

Como fue mencionado anteriormente, los subcontroladores estarán formados por redes neuronales de tipo feedforward, las cuales serán entrenadas mediante estrategias neuroevolutivas. La Figura 5.3 detalla cada una de las redes neuronales utilizadas.

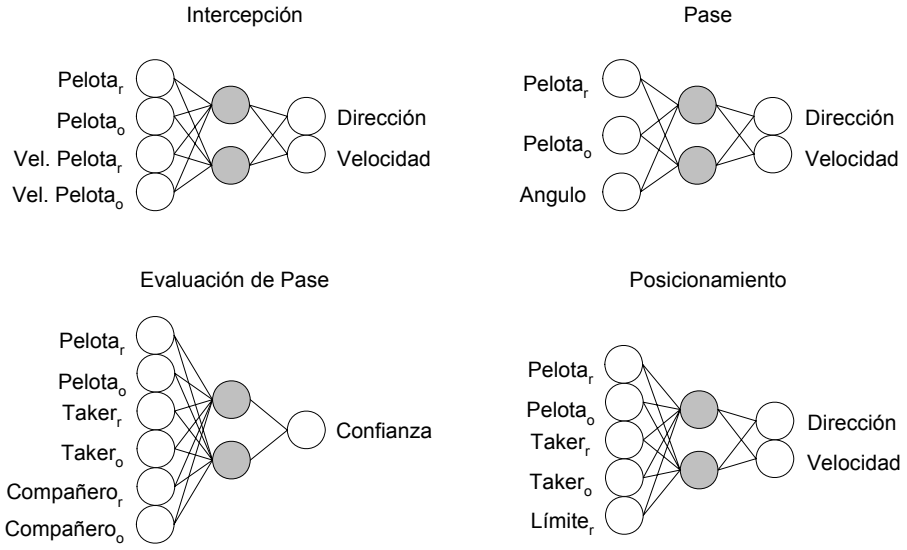

Figura 5.3 Las cuatro redes utilizadas para implementar un controlador de keepaway. Los círculos blancos representan entradas y salidas y los círculos grises representan nodos ocultos.

A continución se describe la estructura de cada una de las redes neuronales artificiales involucradas:

- **Intercepción:** La red de intercepción tiene cuatro nodos de entrada, dos para indicar la posición de la pelota y dos para indicar la velocidad de la pelota. Conjuntamente, tiene dos nodos ocultos y dos nodos de salida los cuales controlan la dirección y velocidad del agente.

- **Pase:** Esta red tiene tres entradas, dos para la posición de la pelota y una que indica el ángulo requerido para el pase. Tiene dos nodos ocultos y dos nodos de salida los cuales controlan la dirección y velocidad del agente.

- **Evaluación de pase:** La red de evaluación de pase tiene seis entradas, dos para la posición de la pelota, dos para la posición del taker, y dos para la posición del potencial compañero. Adicionalmente, tiene dos nodos ocultos y un nodo de salida, este último indica con la baja o alta probabilidad que tiene un compañero de recibir exitosamente la pelota.

- **Posicionamiento:** Esta red tiene cinco entradas, dos para la posición actual de la pelota, dos para la posición actual del taker y una que indica que tan cerca está el agente del límite del campo de juego. Tiene dos nodos ocultos y dos nodos de salida, los cuales controlan la dirección del agente y la velocidad.

Es importante destacar que todas las coordenadas que reciben como entradas las redes están expresadas en coordenadas polares relativas a la posición y dirección del agente (keeper) que está siendo controlado.

Las redes neuronales utilizadas para formar un controlador serán entrenadas utilizando neuroevolución, una técnica de aprendizaje máquina que utiliza algoritmos genéticos para entrenar redes neuronales presentada en el capítulo 3.2.6 [Schaffer, J. D. et. al .: 1992]. En la forma más simple, la neuroevolución codifica los pesos de conexión de una red neuronal para formar un cromosoma, luego una población de estos cromosomas son evolucionados evaluando cada uno de ellos en una tarea y selectivamente reproduciendo los individuos con mejores desempeños a través de operadores de crossover y mutación (Para mas detalles en la evolución de pesos de conexión consultar el capítulo 3.2.1).

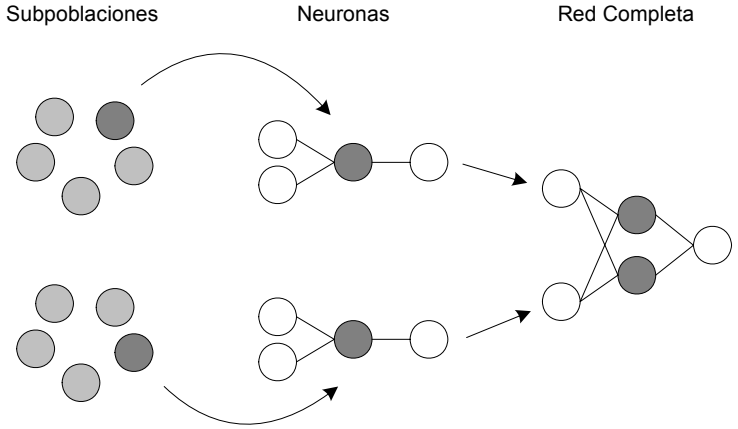

Figura 5.4 Método ESP. Las poblaciones de neuronas son divididas en subpoblaciones. Una neurona es seleccionada de cada subpoblación. Cada cromosomas codifica los pesos que conexión que entran y salen de un nodo oculto. Los cromosomas seleccionados, forman conjuntamente una red que luego será evaluada en una tarea.

Aquí se utilizará una técnica más avanzada de neuroevolución denominada ESP [Gomez, F. and R. Miikkulainen: 1999, 2001, 2003] (Ver sección 3.2.6). En lugar de evolucionar los pesos de conexión de una red completa, ésta evoluciona los pesos de conexión de las neuronas ocultas de la red, para lo cual mantiene varias poblaciones de cromosomas una para cada nodo oculto de la red. Cada neurona es en si misma un cromosoma que codifica

los pesos de conexión que entran y salen de ella. Como ilustra la Figura 5.4, ESP forma redes seleccionando una neurona de cada subpoblación para formar la capa oculta de una red neuronal, la cual luego es evaluada en una tarea. El fitness obtenido es pasado de forma igualitaria a cada una de las neuronas que participaron en la red. Cada población tiende a converger a un comportamiento que maximiza el fitness de la red en la cual ésta participa. La técnica ESP es más eficiente que la neuroevolución tradicional, porque ésta descompone un problema (encontrar un red con alto desempeño) en pequeños subproblemas (encontrar neuronas con alto desempeño). A continuación se detalla paso a paso el algoritmo ESP utilizado para la evolución de las redes neuronales utilizadas en el controlador de un agente de keepaway.

5. **Inicialización**: Se especifica el número de nodos ocultos *u* que formarán una red y se crea una subpoblación de neurocromosomas para cada uno. Cada cromosoma codifica los pesos de entrada y salida de una neurona con una cadena de números reales aleatorios.

6. **Evaluación**: Se selecciona un conjunto de *u* neuronas al azar, una neurona de cada subpoblación, para formar la capa oculta de una red neuronal de tipo feedforward. La red es enviada a una prueba en la cual es evaluada y recompensada con un puntaje de fitness. El puntaje es agregado al fitness acumulativo de cada neurona que participó en la red. Este proceso es repetido hasta que cada neurona haya participado en un promedio de *t* pruebas.

7. **Recombinación**: Se calcula el fitness promedio de cada neurona dividiendo su fitness acumulativo por el número de pruebas en las cuales participó. Las neuronas dentro de cada subpoblación son clasificadas utilizando su fitness promedio. Cada neurona perteneciente al primer cuarto es recombinada con una neurona de alta clasificación utilizando crossover de un punto y mutación con poca frecuencia. La descendencia resultante reemplaza la mitad de las neuronas con menor clasificación de la subpoblación.

8. **Iteración**: El ciclo de evaluación-recombinación se repite hasta que se encuentre una solución óptima o hasta que se alcance un límite de tiempo.

Cuando el rendimiento comienza a estancarse (por ejemplo, el puntaje de la mejor red de cada generación no ha mejorado en 20 generaciones), ESP aplica una técnica de diversificación llamada Delta Coding (ver capítulo 2.3.13) para prevenir la convergencia prematura. Delta Coding selecciona el mejor individuo de una población y lo usa para generar una nueva población. Cada miembro de esta población es una perturbación de la semilla seleccionada. Debido a que la semilla tiene un alto rendimiento, se obtendrán redes óptimas que probablemente sean similares a ésta, pero ocasionalmente pueden ser radicalmente diferentes. Por esta causa, la magnitud de las perturbaciones está basada en una distribución de Cauchy, lo que permite que la mayoría de los nuevos individuos sean muy similares a la semilla y unos pocos sean significativamente diferentes. La diversificación inherente que Delta Coding provee puede mejorar el rendimiento de un algoritmo genético.

Para resolver algunos problemas dificultosos, se puede usar ESP junto con un proceso denominado evolución incremental. En la evolución incremental, un comportamiento complejo es aprendido gradualmente exponiendo al agente a una serie de ambientes de entrenamiento de dificultad creciente. El agente inicialmente aprende a resolver tareas fáciles y avanza a tareas más dificultosas a medida que su rendimiento se incrementa. Por ejemplo, cuando un agente de keepaway tiene que aprender a interceptar la pelota, puede empezar aprendiendo a tomar la pelota cuando está quieta y luego ir incrementando la velocidad de la misma.

A continuación se expondrá como se obtiene cada uno de los subcontroladores por medio de las distintas alternativas de aprendizaje. Cada método será probado en la práctica y finalmente se hará una comparación entre ellos y un análisis final.

5.2 Métodos de aprendizaje aplicados a KeepAway

Esta sección describe en detalle como se aplicó a keepaway cada uno de los métodos de aprendizaje expuestos anteriormente. Si bien todas estas técnicas para resolver keepaway son diferentes, comparten el mismo objetivo general: desarrollar un controlador capaz de comandar a los tres keepers. Esta aproximación es posible porque keepaway es simétrico, es decir, que puede ser jugado de forma efectiva por equipos homogéneos. Dado que los keepers comparten el mismo controlador, ellos tienen el mismo conjunto de comportamientos y las mismas reglas que los gobiernan cuando lo usan. Sin embargo, en un mismo tiempo los tres agentes se comportarán de forma diferente. El hecho de tener agentes idénticos permite que el aprendizaje sea más fácil, debido a que cada agente, además, aprende de la experiencia de sus compañeros.

5.2.1 Aprendizaje por coevolución aplicado

Como se expuso anteriormente, la coevolución es un método que permite que varios componentes de un controlador aprendan a resolver una tarea de forma conjunta en un ambiente común, en este caso el keepaway. Este proceso es particularmente directo cuando se utiliza neuroevolución como método de aprendizaje, porque se realizan varias corridas simultáneas del algoritmo, una para cada componente. Para el caso de keepaway, se utiliza coevolución cooperativa dado que todos los componentes comparten el mismo puntaje de fitness.

El método neuroevolutivo utilizado es ESP Multi Agente [Yong, C. H. and R. Miikkulainen: 2001]. Este método es una extensión de ESP que permite que múltiples componentes coevolucionen cooperativamente, en este sistema cada componente es evolucionado con una corrida separada de ESP en forma concurrente. Por cada evaluación de fitness, ESP Multi Agente forma una red de cada ESP y las evalúa conjuntamente en una tarea, luego cada una de las redes recibe el mismo puntaje cuando la evaluación termina. ESP Multi Agente ha sido utilizado exitosamente en la resolución de tareas de predadores y presas, y es usado también aquí como framework de coevolución para la resolución de keepaway.

Utilizando el árbol de decisión y la descomposición de tareas descriptas anteriormente, la aplicación de coevolución a keepaway es directa. La evolución consiste en aplicar cuatro corridas simultáneas de ESP, una para cada red neuronal que se necesita entrenar. Cada corrida usa subpoblaciones de 100 neuronas y el número de pruebas *t* es 10. En cada evaluación de fitness, se selecciona una red de cada población (por ejemplo, una red de pase, una de intercepción, etc.) y se ensamblan para formar el controlador final del keeper. Mientras transcurre el juego, el árbol de decisión determina como y cuando se usan las redes para controlar cada agente. El puntaje resultante es asignado por igual a cada una de las cuatro redes. El proceso evolutivo se realiza durante 100 generaciones.

Para facilitar el entrenamiento, se utiliza también evolución incremental para controlar la velocidad del taker. Cuando la evolución comienza, el taker puede moverse solo a un 20% de la velocidad máxima de los keepers. Cada red se evalúa en 3 juegos de keepaway y su puntaje (número de pases completos) se suma para obtener su fitness. Cuando el fitness promedio de las redes excede los 20 puntos, se incrementa la velocidad del taker en un 5% de la velocidad de los keepers. Este proceso continúa hasta que el taker alcance la misma velocidad que los keepers, el fitness de la población converja o se alcance la cantidad de generaciones preestablecidas (100).

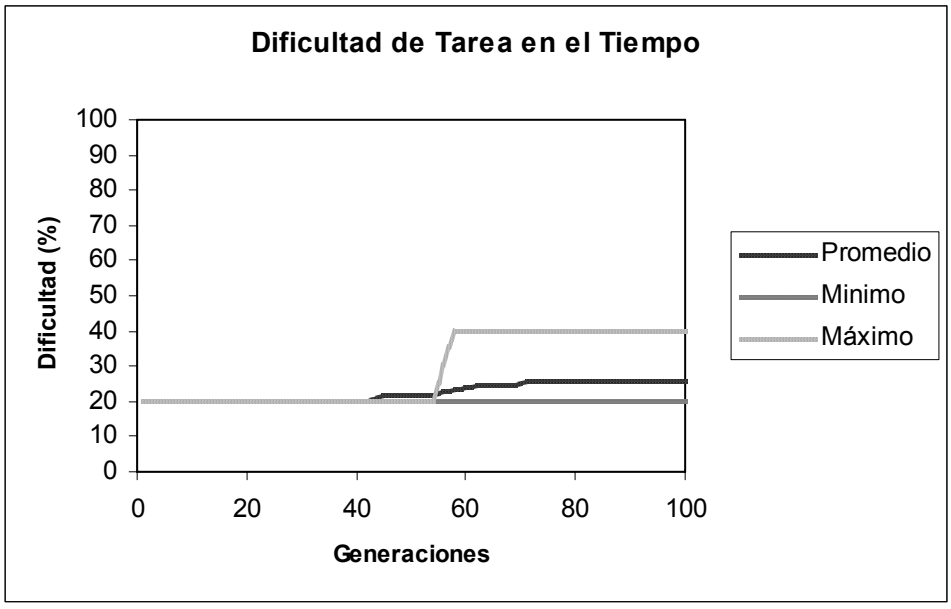

Figura 5.5 Dificultad de tarea alcanzada con el método coevolutivo durante 100 generaciones. La dificultad alcanzada fue promediada sobre 10 corridas del algoritmo, también se muestran las corridas con mayor y menor valor obtenidos.

Resultados

Luego de concluidas las 100 generaciones del proceso evolutivo, no se obtuvieron los resultados deseados (ver Figura 5.6), el controlador no pudo alcanzar el comportamiento adecuado y solo llegó a un 25% de dificultad. Estas pruebas dejan en evidencia que si bien

este método es muy flexible en cuanto a su aplicación, no logra buenos resultados en la obtención de subcontroladores para tareas complejas.

En las secciones siguientes, se exponen otros métodos de aprendizaje que intentan mejorar el desempeño del controlador de un beeper.

5.2.2 Aprendizaje por capas aplicado

El aprendizaje coevolutivo provee un simple y flexible framework para aprender diversas subtareas en forma conjunta. Sin embargo, ofrece poca asistencia al algoritmo de aprendizaje más allá de la descomposición de tareas en si misma y no siempre se obtienen buenos resultados de su aplicación. Para resolver tareas complejas, puede ser útil o necesario aprender las subtareas de una forma más estructurada y secuencial. El aprendizaje por capas es un paradigma jerárquico Bottom-Up que puede ser utilizado para este tipo de problemas. (Véase la sección 4.3 para más información sobre este método)

En la aplicación del aprendizaje por capas a keepaway, la primera etapa es decidir en que orden se deben aprender las subtareas de cada capa y que tipo de algoritmo de aprendizaje máquina es conveniente usar.

La Figura 5.6 presenta el orden utilizado aquí para aprender cada capa. Una flecha de una capa a otra indica que la capa apuntada depende de la que la apunta. Debido a que una capa no puede ser aprendida hasta que todas las capas de las cuales ésta depende hallan sido aprendidas, el proceso de aprendizaje comienza desde la parte inferior, con la intercepción, y se mueve hacia arriba por la jerarquía paso a paso.

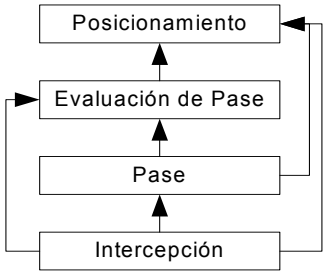

Figura 5.6 Una jerarquía de aprendizaje por capas para la tarea de KeepAway. Cada rectángulo representa una capa y cada flecha indica dependencias entre capas. Una capa no puede ser aprendida hasta que todas las capas de las cuales ésta depende hayan sido aprendidas.

Como se expuso previamente, cada subtarea será realizada por una red neuronal (subcontrolador), el algoritmo de aprendizaje máquina utilizado para entrenar cada red neuronal será neuroevolución ESP debido a las ventajas que esté método posee con respecto a otros métodos de neuroevolución.

A diferencia del aprendizaje coevolutivo, donde cada red neuronal era entrenada sobre el juego completo de keepaway, el aprendizaje por capas requiere de la construcción de un

ambiente especial de entrenamiento para cada red neuronal de cada capa. Por ejemplo para el caso de la primer capa, donde el agente debe aprender a interceptar la pelota, se prepara un ambiente donde la pelota es impulsada en diferentes direcciones y a diferentes velocidades para que el keeper aprenda su tarea.

Si bien esté método permite aprender las tareas de forma más estructurada que el método coevolutivo, tiene como desventaja que requiere un mayor esfuerzo del usuario, dado que hay que construir ambientes especiales de entrenamiento para cada subtarea. Pero desde otro punto de vista, el hecho de aprender las subtareas de forma ordenada permite en ocasiones obtener controladores con mejor desempeño que con el método coevolutivo, sobre todo cuando se trata de la resolución de tareas complejas como es el caso de keepaway.

A continuación se detalla cada una de las capas usando la notación formal del aprendizaje por capas, incluyendo una descripción de los ambientes especiales de entrenamiento para cada una. Como se detalló en la sección 4.3.1, $\boldsymbol{L_i}$ representa una subtarea que se quiere aprender del conjunto de subtareas que surge de la descomposición de tareas. $\vec{\boldsymbol{F}}_i$ es el vector de datos de entrada relevantes para el aprendizaje de la subtarea $\boldsymbol{L_i}$, $\boldsymbol{O_i}$ es el conjunto de salidas de la subtarea $\boldsymbol{L_i}$, $\boldsymbol{T_i}$ es el conjunto de ejemplos de entrenamiento usados para aprender la subtarea $\boldsymbol{L_i}$, $\boldsymbol{M_i}$ es el algoritmo de aprendizaje máquina usado en la capa $\boldsymbol{L_i}$ *y* $\boldsymbol{h_i}$ es el resultado de correr $\boldsymbol{M_i}$ sobre $\boldsymbol{T_i}$

L_1: Intercepción:

$\vec{\boldsymbol{F}}_1$ = {Bola$_r$, Bola$_\theta$, Velocidad de Bola$_r$, Velocidad de Bola$_\theta$ } $\in$ R^4

$\boldsymbol{O_1}$ = {Dirección, Velocidad} $\in$ R^2

$\boldsymbol{T_1}$: Para entrenar del interceptor, la pelota es impulsada hacia el agente en varios ángulos y velocidades. El agente es recompensado en mayor medida cuanto menos tiempo tarde en tocar la pelota.

M_1: Neuroevolución: Se utiliza ESP con subpoblaciones de tamaño 100 para entrenar una red con 4 entradas, 2 nodos ocultos y dos salidas (ver Figura 5.3).

h_1: Una red de intercepción entrenada.

L_2: Pase:

$\vec{\boldsymbol{F}}_2$ = {Bola$_r$, Bola$_\theta$, Angulo Objetivo} $\in$ R^3

$\boldsymbol{O_2}$ = {Dirección, Velocidad} $\in$ R^2

$\boldsymbol{T_2}$: Para entrenar la red de pases, la pelota es nuevamente impulsada hacia el agente. El ángulo en el cual el agente debería patear la pelota es elegido en forma aleatoria. Cuando la simulación comienza, el agente emplea el comportamiento de intercepción aprendido en L_1 hasta que llega cerca de la pelota, en cuyo punto éste cambia al comportamiento de pase en

evolución. La recompensa que recibe el agente es inversamente proporcional a la diferencia entre el ángulo objetivo y la dirección real con la que fue impulsada la pelota.

***M_2*: Neuroevolución**: Se utiliza ESP con subpoblaciones de tamaño 100 para entrenar una red con 3 entradas, 2 nodos ocultos y 2 salidas (ver Figura 5.3).

h_2 = Una red de pase entrenada.

L_3: Evaluación de Pase:

$\vec{F}_3$ = {$Bola_r$, $Bola_\theta$, $Taker_r$, $Taker_\theta$, $Compañero_r$, $Compañero_\theta$} $\in R^6$

O_3 = {Confianza} $\in R$

***T_3*:** Se coloca la pelota en el centro del campo y el evaluador justo detrás de ésta en varios ángulos. Dos compañeros son colocados en lugares opuestos al evaluador, cerca del borde del campo de juego y con ciertas variaciones aleatorias en las posiciones. Un único taker es colocado cerca de la pelota para simular la presión que éste ejerce sobre el que hace el pase. Los compañeros y el taker usan el comportamiento de intercepción aprendido en L_1. En el entrenamiento del evaluador de pase, la red en evolución es utilizada dos veces, una por cada compañero. El evaluador de pase entonces pasa la pelota, usando L_2, al compañero quien recibió la evaluación más alta. Si el pase tiene éxito, el evaluador es recompensado.

***M_3*: Neuroevolución**: Se utiliza ESP con subpoblaciones de tamaño 100 para entrenar una red con 6 entradas, 2 nodos ocultos y 1 salida (ver Figura 5.3).

h_3 = Una red de evaluación de pase entrenada.

L_4: Posicionamiento:

$\vec{F}_4$ = {$Bola_r$, $Bola_\theta$, $Taker_r$, $Taker_\theta$, $Frontera_r$} $\in R^5$

O_4 = {Dirección, Velocidad} $\in R^2$

***T_4*:** El ambiente de entrenamiento para el comportamiento de posicionamiento es un juego real de keepaway. El taker usa el comportamiento de intercepción evolucionado en L_1 y los keepers usan el árbol de decisión descripto en la Figura 5.2 junto con las redes evolucionadas en L_1, L_2 y L_3.

***M_4*: Neuroevolución**: Se utiliza ESP con subpoblaciones de tamaño 100 para entrenar una red con 5 entradas, 2 nodos ocultos y 1 salida (ver Figura 5.3).

h_4 = Una red de posicionamiento entrenada.

Una vez que las cuatro capas han sido aprendidas, pueden ser combinadas con el árbol de decisión para formar un jugador completo de keepaway. La Figura 5.7 muestra los resultados obtenidos mediante la aplicación de este método como se describió anteriormente.

Resultados:

Los resultados obtenidos con este método mejoran los obtenidos con el método coevolutivo, aquí los keepers lograron alcanzar una dificultad de un 45% lo que representa una mejora en el desempeño de aproximadamente un 20% con respecto al método previo.

Si bien esta mejora es buena, no logra alcanzar un nivel aceptable. Recuérdese que el objetivo del entrenamiento es lograr que los takers puedan jugar correctamente contra el keeper corriendo a la misma velocidad que ellos. Aquí no pudieron superar el 50% de la velocidad de los keepers.

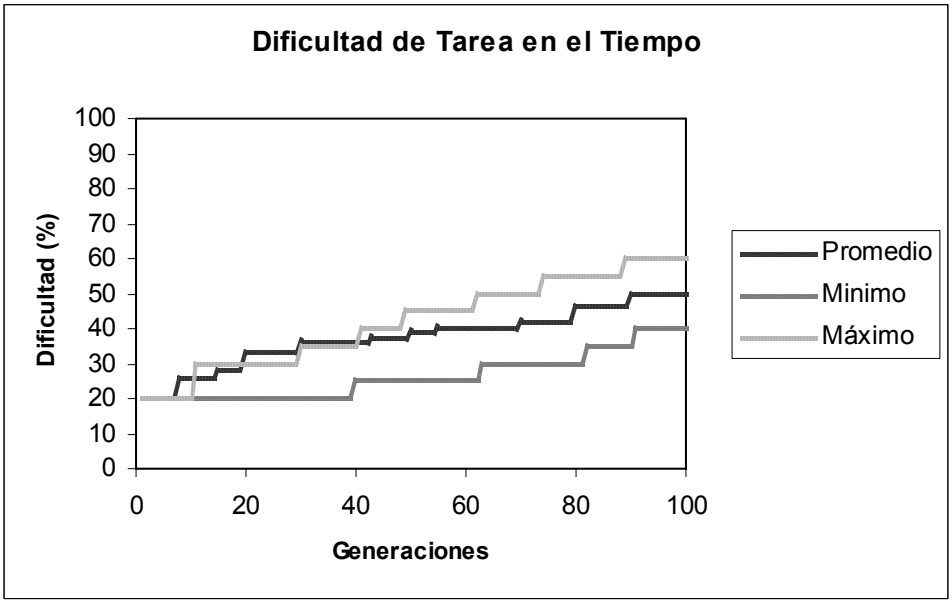

Figura 5.7 Dificultad de tarea alcanzada con el método de aprendizaje por capas durante 100 generaciones. La dificultad alcanzada fue promediada sobre 10 corridas del algoritmo, también se muestran las corridas con mayor y menor valor obtenidos.

A continuación se aplicará un método de aprendizaje más avanzado que combina aspectos del aprendizaje por capas y del aprendizaje coevolutivo.

5.2.3 Aprendizaje por capas concurrente aplicado

El aprendizaje coevolutivo no provee asistencia humana más allá de la descomposición de tareas en si misma, por este motivo no restringe demasiado el espacio de búsqueda de posibles soluciones. En contraste, el aprendizaje por capas provee una buena proporción de asistencia, restringiendo y guiando el aprendizaje en un espacio de búsqueda más reducido. El aprendizaje por capas concurrente es una aproximación que ocupa un rango intermedio en el espectro.

Cuando se está aprendiendo una capas L_i, se seleccionan de todas las hipótesis aprendidas previamente algún subconjunto $P \subseteq \{h_1, h_2, \ldots h_{i-1}\}$ que se quiere que continúen entrenando en el ambiente actual T_i. Para cada $h_k \in P$, se toma la mejor red neuronal de L_k y luego se usa para generar una nueva población antes de que comience el entrenamiento en T_i. Estas nuevas poblaciones continúan aprendiendo junto con una población separada que aprenderá L_i. Por consiguiente, las capas son evolucionadas cooperativamente usando ESP Multi Agente.Sin embargo, cada una de las redes no son agentes separados sino componentes del mismo agente.

Para realizar la evaluación del fitness, se toma una red neuronal de cada población generada con h_k y evaluada en T_i, junto con una red seleccionada de la población que está aprendiendo desde cero L_i. El puntaje resultante es compartido por todas las redes que participaron en la prueba.

Para generar una población en base a los resultados de L_k, se usa una técnica llamada Delta Coding (ver seccion 2.3.13). Debido a que Delta Coding es particularmente adecuada para permitir que las poblaciones se ajusten a cambios repentinos en su ambiente de entrenamiento [Gomez, F. and R. Miikkulainen: 1997], es una excelente forma de generar una nueva población sobre la base de los resultados de una capa inferior.

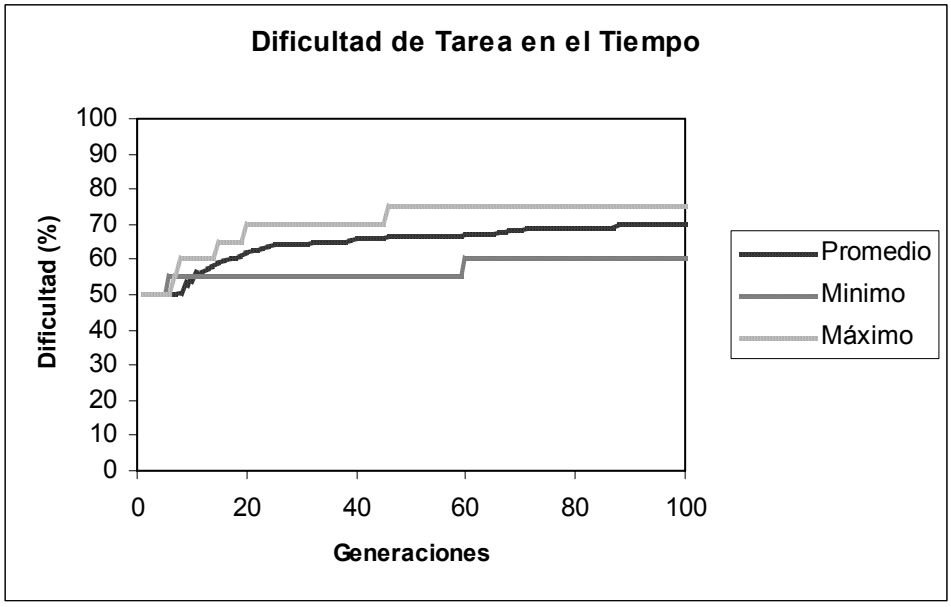

Figura 5.8 Dificultad de tarea alcanzada con el método de aprendizaje por capas concurrente durante 100 generaciones. La dificultad alcanzada fue promediada sobre 10 corridas del algoritmo, también se muestran las corridas con mayor y menor valor obtenidos.

El aprendizaje por capas concurrente provee un framework general para combinar aprendizaje por capas y coevolución pero no especifica que capas deberían continuar su aprendizaje. Por consiguiente, para aplicar aprendizaje por capas concurrente a keepaway, se debe decidir, para cada capa, que otras de nivel más bajo deben quedar fijas y cuales deben ser liberadas para permitir que se sigan adaptando.

En este caso, cada una de las capas más bajas L_1, L_2 y L_3 son entrenadas exactamente como en el aprendizaje por capas tradicional descripto anteriormente. Cualquier componente aprendido anteriormente permanece fijo en estos ambientes de entrenamiento. Sin embargo, cuando comienza el entrenamiento de L_4, se liberan todas las hipótesis h_1, h_2 y h_3 que ya han sido aprendidas con anterioridad. Por consiguiente, cada uno de los componentes de las capas inferiores, que aprendieron inicialmente en su propio ambiente especial de entrenamiento, ahora tiene la posibilidad de hacer ajustes finos a su comportamiento en T_4, que representa el juego de keepaway completo.

Desde otra perspectiva, las tres primeras capas son entrenadas como si fuera un aprendizaje por capas tradicional. Cuando llega el momento de entrenar la última capa, se aplica el método coevolutivo a las cuatro capas, las tres primeras utilizan el conocimiento adquirido previamente y la última parte sin conocimiento alguno.

Resultados:

Como muestra la Figura 5.8, el aprendizaje concurrente mejora los resultados obtenidos con los dos métodos previos. Esto permite concluir que si bien para tareas complejas es conveniente primero aprender subtareas más simples, también es beneficioso que todos los componentes encargados de realizar las subtareas puedan entrenar juntos y así mejorar su desempeño grupal.

La dificultad en el keepaway alcanzada con este método es de un 70%. Es decir, que los keepers pueden jugar de forma aceptable con el taker moviéndose a una velocidad del 65%, pero no pueden lograr tener un juego aceptable cuando el taker se mueve a un 70%.

A continuación se expondrá el método de aprendizaje cíclico propuesto para esta tesis aplicado al juego keepaway.

5.2.4 Aprendizaje cíclico aplicado

El aprendizaje cíclico fue concebido para refinar el desempeño de controladores obtenidos por otros métodos, en particular aquí se utilizarán las redes neuronales obtenidas del aprendizaje por capas. El método será aplicado al keepaway utilizando deferentes números de ciclos y dentro de cada ciclo cada subcontrolador será evolucionado un número fijo de generaciones. También se probará una alternativa en la cual no se establece a priori el número de generaciones que cada subcontrolador puede evolucionar dentro de un ciclo, permitiéndose que este número sea variable.

A diferencia del aprendizaje por capas, es importante destacar que el entrenamiento de cada subcontrolador se realiza en el juego completo de keepaway, no existiendo ambientes especialmente preparados para cada subtarea. Esto permite que las redes neuronales realicen su entrenamiento en el dominio donde finalmente deben desenvolverse.

Al igual que los métodos de aprendizaje anteriores, los subcontroladores c_i están definidos por las redes neuronales de la Figura 5.3. El aprendizaje cíclico utiliza las redes obtenidas del aprendizaje por capas para intentar refinar su comportamiento.
Como se expuso en la sección 4.5, el algoritmo cíclico tiene la siguiente forma:

Comenzar *{programa principal}*
Sea C={c_1,c_2,c_3,..,c_n} un conjunto de subcontroladores iniciales.
Sea O el objetivo a alcanzar.
Sea Z la cantidad máxima de ciclos a realizar.
Ciclos = 0 *{hasta ahora ningún ciclo se ha realizado}*
Mientras (objetivo O no sea alcanzado) **y** (Ciclos < Z)
Para cada subcontrolador $c_i \in C$, con i = 1.. n.
$c'_i = Aprendizaje(c_i)$
reemplazar c_i por c_i' en C
Fin Para
Ciclos = Ciclos + 1;
Fin Mientras
Fin *{ programa principal}*

El proceso *Aprendizaje*(c_i) utiliza algún algoritmo aprendizaje máquina para mejorar el desempeño de un subcontrolador determinado. La elección del algoritmo utilizado aquí para definir este proceso es ESP (Enforced Subpopulation), para más detalles de este método ver la sección 3.2.6.
A continuación se detalla la implementación utilizada para el proceso *Aprendizaje*(c_i):

Proceso Aprendizaje(c_i)

{evoluciona el i-ésimo subcontrolador utilizando ESP}

Comenzar

Generar una subpoblación de neuronas para cada neurona oculta de c_i.

Repetir

Repetir

- Construir c_i' seleccionando en forma aleatoria una neurona oculta de cada subpoblación.
- Evaluar el fitness del controlador compuesto de {c_1,c_2,...,c_i',...,c_n} en el dominio completo del problema.
- Acumular el fitness obtenido en las neuronas ocultas de c_i'.

Hasta (que cada neurona de cada subpoblación haya participado en promedio de diez juegos de keepaway)

Obtener la siguiente población de neuronas para cada subpoblación a través de operadores genéticos.

Hasta (alcanzar un cierto número ***n*** de generaciones) u (obtener un c_i' óptimo)

C = {c_1,c_2,...,c_i',…,c_n} *{reemplazar c_i por el mejor c_i' encontrado hasta el momento}*

Fin *{ proceso Aprendizaje}*

Obsérvese que la condición de fin de la segunda instrucción "**Hasta"** indica que el controlador c_i será evolucionado un número *n* de generaciones. En general ese número se establece antes de comenzar el algoritmo cíclico, con lo cual se logra que cada red neuronal dentro de cada ciclo evolucione un número fijo de generaciones. Esto en algunos casos es ineficiente porque puede suceder que luego de unas pocas generaciones de evolución ya no se observen mejoras en el desempeño del subcontrolador. Bajo este escenario hay dos posibles alternativas a seguir, la primera, consiste en aplicar alguna técnica para promover diversidad genética e intentar evitar la convergencia prematura durante el resto de las n generaciones, para el caso de las pruebas presentadas aquí se utilizará Delta-Coding. En la segunda alternativa, una vez detectado el estancamiento en la evolución, en lugar de aplicar una técnica para promover diversidad genética, y completar las *n* generaciones de evolución, se suspende momentáneamente la evolución del subcontrolador actual y se continua con la ejecución del algoritmo cíclico para el siguiente subcontrolador en la secuencia.

Con la segunda alternativa se consigue que para la misma cantidad total de generaciones empleadas en la evolución de todos los subcontroladores, se realice una cantidad superior de ciclos que para el caso de la primera.

A esta segunda aproximación se la ha denominado aprendizaje cíclico variable, porque no se sabe de antemano la cantidad de ciclos que habrá en la ejecución del algoritmo y tampoco se utiliza como condición de fin del mismo.

Por ejemplo, supóngase que para el caso del juego keepaway se quiere correr el algoritmo cíclico estándar durante 160 generaciones, y se decide que la cantidad de ciclos sea de 4. Esto implica que cada ciclo durará 40 generaciones y que cada subcontrolador evolucionará durante 10 generaciones dentro de cada ciclo. Si hipotéticamente cada subcontrolador luego de 5 generaciones no pudiese mejorar su desempeño, habría otras 5 generaciones desperdiciadas. Utilizando la segunda alternativa bajo este escenario, cada ciclo durará 20 generaciones en lugar de 40 y para la misma cantidad total de generaciones (160), habrá 8 ciclos en lugar de 4.

Las pruebas presentadas aquí se realizaron aplicando el algoritmo cíclico estándar y el cíclico variable durante 100 generaciones. En el caso del cíclico estándar, se hicieron pruebas con 1, 3, y 5 ciclos. Cada prueba se realizó 10 veces y los resultados fueron promediados, para obtener así un resultado más uniforme.

Resultados:

La Figura 5.9 muestra los resultados de la aplicación del algoritmo cíclico durante 100 generaciones con un solo ciclo, es decir, que cada subcontrolador evolucionó durante 100 generaciones en forma secuencial según el orden de tareas establecido dentro del ciclo. La dificultad alcanzada mediante este esquema fue del 71%, resultado muy similar al obtenido con el método de aprendizaje por capas concurrente. La diferencia con este último es que en el entrenamiento de la última capa, todos los subcontroladores son entrenados al mismo

tiempo durante 100 generaciones, mientras que en el método cíclico con un solo ciclo evolucionan durante 100 generaciones pero en forma secuencial.

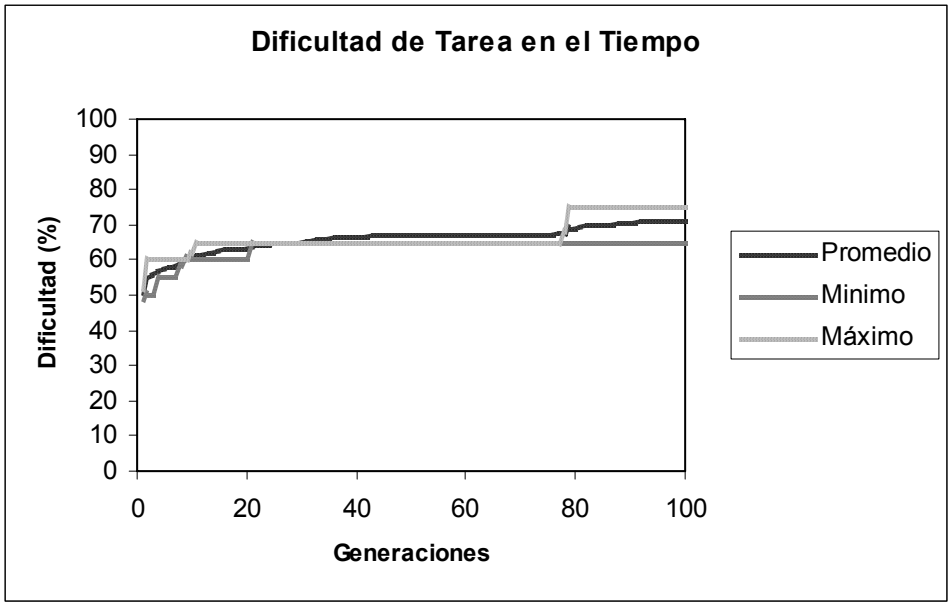

Figura 5.9 Dificultad de tarea alcanzada con el método cíclico durante 100 generaciones y 1 ciclo. La dificultad alcanzada fue promediada sobre 10 corridas del algoritmo, también se muestran las corridas con mayor y menor valor obtenidos.

La Figura 5.10 muestra los resultados obtenidos utilizando el algoritmo cíclico con 3 ciclos, en este caso cada subcontrolador evolucionó durante 100 generaciones pero segmentadas en tres partes, es decir que en cada ciclo los subcontroladores evolucionan en forma secuencial durante 33 generaciones (100 / 3).

La aplicación del algoritmo cíclico con 3 ciclos mejoró el desempeño de los subcontroladores, permitiendo alcanzar una dificultad 74,5%. Si se analiza la aplicación del algoritmo cíclico con un solo ciclo y con tres, se puede notar que en los dos casos cada subcontrolador evolucionó durante 100 generaciones, la diferencia está en la forma en que lo hicieron dada por la cantidad de ciclos. Este factor permite pensar que incrementando el número de ciclos se pueden obtener mejores resultados. Los siguientes resultados tienden a dilucidar esta incógnita.

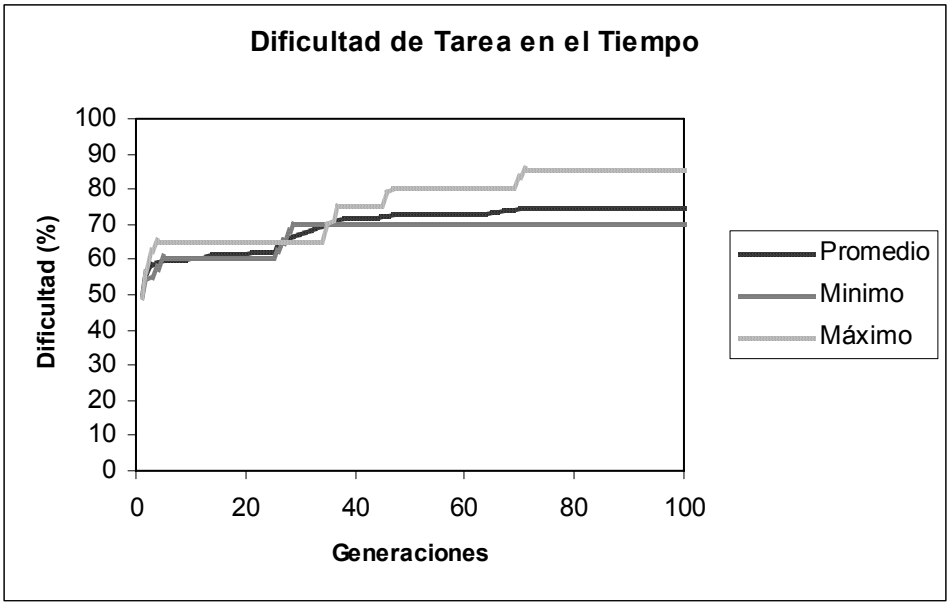

Figura 5.10 Dificultad de tarea alcanzada con el método cíclico durante 100 generaciones y 3 ciclos. La dificultad alcanzada fue promediada sobre 10 corridas del algoritmo, también se muestran las corridas con mayor y menor valor obtenidos.

La Figura 5.11 muestra los resultados obtenidos utilizando el algoritmo cíclico con 5 ciclos, en este caso cada subcontrolador evolucionó durante 100 generaciones pero segmentadas en 5 partes, es decir que en cada ciclo los subcontroladores evolucionan en forma secuencial durante 20 generaciones (100 / 5).

Bajo este escenario, los resultados obtenidos fueron mejores que los resultados previos. Aquí los subcontroladores pudieron alcanzar una dificultad del 80%, lo cual implica que los keepers pudieron jugar aceptablemente con el taker moviéndose a un 75% de la velocidad total permitida. Si bien la dificultad alcanzada hasta aquí no es la ideal, se puede considerar como aceptable.

Nótese que se obtuvo una mejora con respecto al método cíclico con 3 ciclos, y la única diferencia fue que se incremento la cantidad de ciclos en 2. Parecería ser que a medida que se aumentan los ciclos mejora el desempeño de los subcontroladores, el método con 5 ciclos superó al de 3 y el de 3 al de 1.

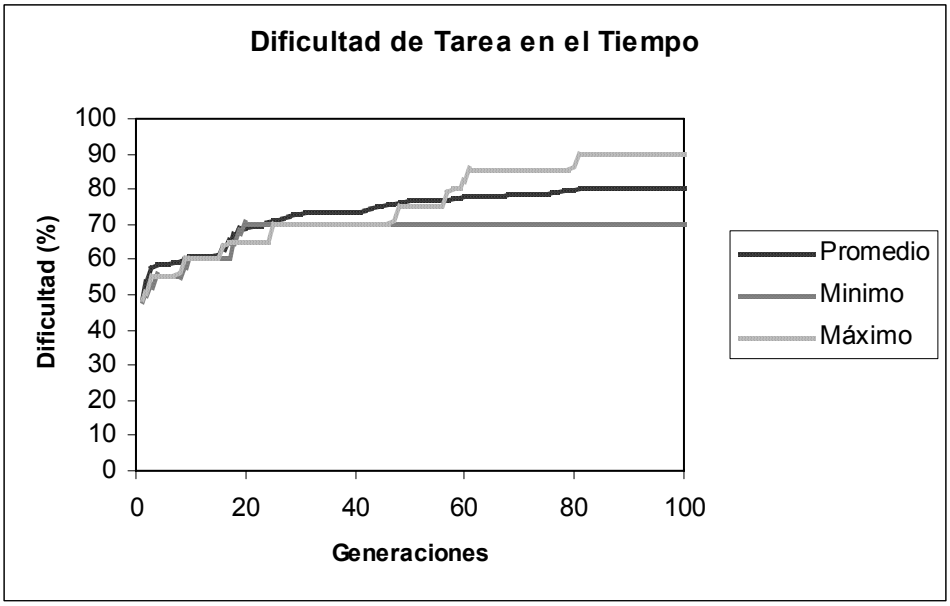

Figura 5.11 Dificultad de tarea alcanzada con el método cíclico durante 100 generaciones y 5 ciclos. La dificultad alcanzada fue promediada sobre 10 corridas del algoritmo, también se muestran las corridas con mayor y menor valor obtenidos.

Finalmente se aplicó el método cíclico variable, la Figura 5.12 muestra los resultados obtenidos. La característica principal de este método es que aumenta en forma considerable la cantidad de ciclos con respecto a los casos anteriores.

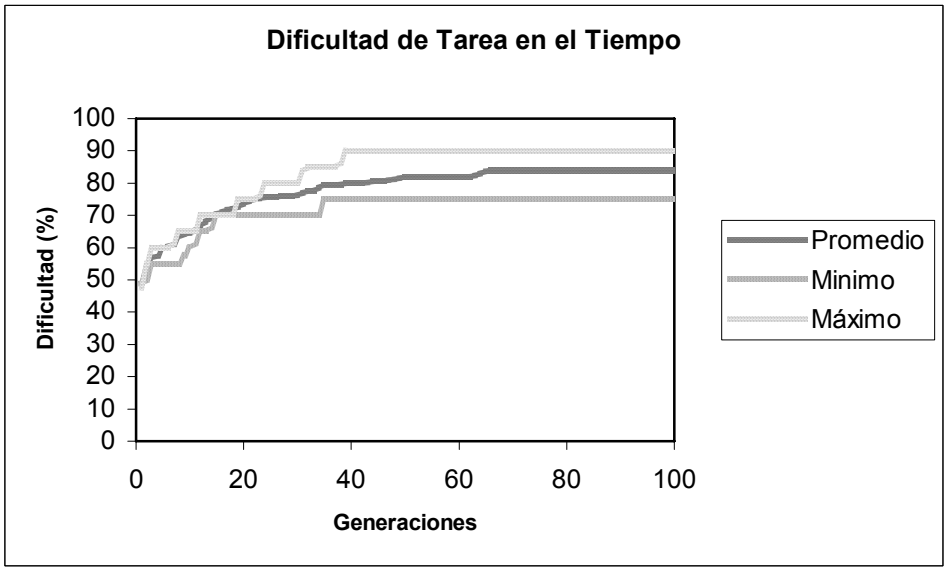

Figura 5.12 Dificultad de tarea alcanzada con el método cíclico variable durante 100 generaciones. La dificultad alcanzada fue promediada sobre 10 corridas del algoritmo, también se muestran las corridas con mayor y menor valor obtenidos.

Cada vez que un subcontrolador no presenta mejoras durante 4 generaciones, el algoritmo comienza con la mejora del siguiente, esto permite distribuir mejor las generaciones de evolución. Por ejemplo los subcontroladores difíciles de mejorar evolucionarán durante un menor número de generaciones que los subcontroladores que tienden a mejorar de forma continua.

Con el aprendizaje cíclico variable se obtuvieron los mejores resultados utilizando la misma cantidad de generaciones que con los métodos previos. La dificultad alcanzada en el juego de keepaway fue en promedio de un 84% con picos de un 90%. Estos resultados ponen en evidencia el impacto positivo del aumento de ciclos debido a la naturaleza del método y a la redistribución de las generaciones dentro de un ciclo a aquellos subcontroladores que mejor los pueden aprovechar.

6 CONCLUSIONES

Los algoritmos de aprendizaje presentados en esta tesis, junto con la utilización de técnicas neuroevolutivas, han demostrado ser herramientas ponderosas para resolver tareas complejas, en particular el juego de keepaway. Cada método en mayor o menor medida, salvo el método coevolutivo, pudo lograr un nivel aceptable en el juego.

La Figura 6.1 presenta los resultados obtenidos de cada método. Como se puede apreciar, el método coevolutivo no logra generar controladores aceptables posiblemente por ser el espacio de búsqueda más extenso que para el resto de los métodos. El aprendizaje por capas, reduce dicho espacio agregando conocimiento humano al proceso. Este último factor permitió que los resultados obtenidos sean mejores que con el método coevolutivo, pero genera el interrogante de sí esa reducción del tamaño del espacio de búsqueda, puede dejar soluciones potencialmente buenas fuera de carrera anticipadamente.

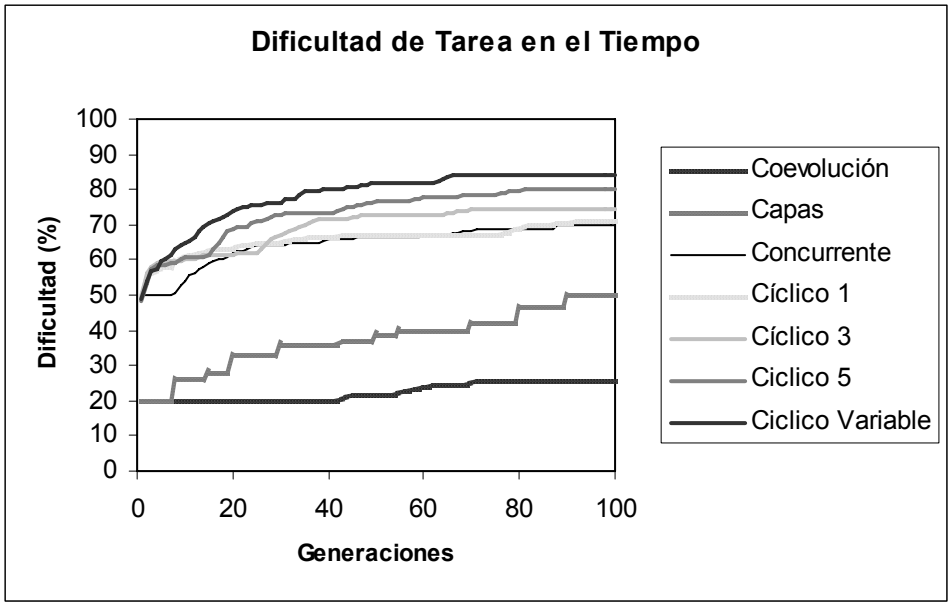

Figura 6.1 Gráfico comparativo de las dificultades alcanzadas por cada uno de los métodos de aprendizaje durante 100 generaciones. La dificultad alcanzada fue promediada sobre 10 corridas de cada algoritmo.

El aprendizaje por capas concurrente representa un punto intermedio entre el aprendizaje coevolutivo y el aprendizaje por capas, agrega conocimiento humano al proceso de aprendizaje pero mantiene algo de la flexibilidad de la coevolución. Con este método se obtuvieron muy buenos resultados generando subcontroladores con desempeños aceptables.

Finalmente el método de aprendizaje cíclico presentado en esta tesis, fue aplicado con diferentes configuraciones al juego de keepaway. En todos los casos, este esquema, mejoró el desempeño de los agentes de keepaway permitiendo concluir que se puede mejorar el

comportamiento de los subcontroladores mediante refinamientos sucesivos en forma cíclica.

Algunas contribuciones de esta tesis son:

1. Verificación que dada una descomposición de subtareas adecuada, la neuroevolción es capaz de resolver una tarea de control compleja en un sistema multiagente.

2. La introducción de un nuevo método, el aprendizaje cíclico, que permite refinar el desempeño de subcontroladores en la resolución de tareas complejas.

3. La obtención de evidencia empírica que permite concluir que el aprendizaje cíclico mejora el desempeño de los subcontroladores en forma proporcional a la cantidad de ciclos aplicados.

4. La conclusión de que para el proceso de aprendizaje, la flexibilidad y la imposición de pocas restricciones pueden no ser del todo buenas y en ocasiones es conveniente guiarlo mediante el conocimiento humano.

Los métodos de aprendizaje representan gran promesa como candidatos para encarar problemas complejos. Agregando conocimiento humano en el proceso, estas técnicas pueden alentar a los algoritmos de aprendizaje actuales a resolver problemas de mayor complejidad.

El esfuerzo manual no tiene por que ser muy costoso. En realidad, las pruebas presentadas aquí, demuestran que los mejores resultados se obtienen cuando las restricciones y guías son aplicadas con moderación.

BIBLIOGRAFÍA

[Andre, D. and Astro T. 1999]. "Evolving team Darwin United." En *RoboCup-98: Robot Soccer World Cup II*, Minoru Asada and Hiroaki Kitano (eds). Lecture Notes in Computer Science, vol.1604, p.346-352. Springer-Verlag, 1999.

[Bäck, T.], "Selective pressure in Evolutionary Algorithms: A Characterization of Selection MEchanisms".

[Bethke, A.D.: 1980], "Genetic Algorithms as Function Optimizers, Doctoral Dissertation, University of Michigan, 1980.

[Box, G. E. P.: 1957] "Evolutionary operation: a method of increasing industrial productivity". *Applied Statistics*, Vol. 6, 81-101.

[Chellapilla K. and Fogel D. 2001] "Evolving an expert checkers playing program without using human expertise." *IEEE Transactions on Evolutionary Computation*, vol.5, no.4, p.422-428 (agosto de 2001).

[Fraser, A. S.: 1957] "Simulation of genetic systems by automatic digital computers". *Australian Journal of Biological Science*, 10, 484-491.

[Giro R., Cyrillo M. and Galvão D.S.. 2002] "Designing conducting polymers using genetic algorithms." *Chemical Physics Letters*, vol.366, no.1-2, p.170-175 (25 de noviembre de 2002).

[Glen, R.C. and Payne, y A.W.R.. 1995] "A genetic algorithm for the automated generation of molecules within constraints." *Journal of Computer-Aided Molecular Design*, vol.9, p.181-202 (1995).

[Goldberg, D. E. : 1990], "Real-Coded Genetic Algorithms, Virtual Alphabets, and Blocking", University of Illinois ar Urbana-Champaing, Thechnical Report No. 90001, September 1990.

[Goldberg, D. E. And Deb, K.], "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms".

[Goldberg, D. E: 1989], "Genetic Algorithms in Search, Optimization, and Machine Learning. Reading, MA: Addison-Wesley, 1989.

[Gomez, F. and R. Miikkulainen: 1997], "Incremental Evolution of Complex General Behavior". Adaptive Behavior 5, 317–342.

[Gomez, F. and R. Miikkulainen: 1999], "Solving Non-Markovian Control Tasks with Neuroevolution". In: Proceedings of the International Joint Conference on Artificial Intelligence. Denver, CO, pp. 1356-1361, Kaufmann.

[Gomez, F. and R. Miikkulainen: 2001], "Learning Robust Nonlinear Control with Neuroevolution". Technical Report AI01-292, The University of Texas at Austin Department of Computer Sciences.

[Gomez, F. J. and R. Miikkulainen: 2003], "Active Guidance for a Finless Rocket Using Neuroevolution". In: E. Cantu-Paz, J. A. Foster, K. Deb, L. D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, K. Dasgupta, M. A. Potter, A. C. Schultz, K. A. Dowsland, and N. J. J. Miller (eds.): Genetic and Evolutionary Computation - GECCO 2003. Chicago, pp. 2084-2095, Springer Verlag.

[Hancock, P. J. B. :1992] , "Genetic algorithms and permutation problems: A comparison of recombination operators for neural net structure specification," in Proc. Int. Workshop Combinations of Genetic Algorithms and Neural Networks (COGANN-92), D. Whitley and J. D. Schaffer, Eds. Los Alamitos, CA: IEEE Computer Soc., 1992, pp. 108-122. pp. 789-795.

[Haynes, T. and S. Sen: 1996], "Evolving Behavioral Strategies in Predators and Prey". In: G. Weiß and S. Sen (eds.): Adaptation and Learning in Multiagent Systems. Berlin: Springer Verlag, pp. 113–126.

[He, L. and Mort, N. 2000] "Hybrid genetic algorithms for telecommunications network back-up routeing." *BT Technology Journa*l, vol.18, no.4, p. 42-50 (octubre de 2000).

[Holland, J. H.: 1975], "Adaptation in Natural and Artificial Systems." The University of Michigan Press, Ann Arbor, MI, 1975.

[Keane, A.J. and Brown, S.M. 1996] "The design of a satellite boom with enhanced vibration performance using genetic algorithm techniques." En *Adaptive Computing in Engineering Design and Control '96 - Proceedings of the Second International Conference*, I.C. Parmee (ed), p.107-113. University of Plymouth, 1996.

[Knerr, S. et. al.] "Handwritten digit recognition by neural networks with single-layer training," *IEEE Trans. Neural Networks*, vol. 3, pp. 962–968, Nov. 1992.

[Kur96] V.Kureichick, A.N.Melikhov, V.V.Miaghick, O.V.Savelev and A.P.Topchy, Some New Features in the Genetic Solution of the Traveling Salesman Problem. In Ian Parmee and M.J.Denham eds. Adaptive Computing in Engineering Design and Control 96(ACEDC'96), 2nd International Conference of the Integration of Genetic Algorithms and Neural Network Computing and Related Adaptive Computing with Current Engineering Practice, Plymouth, UK, March 1996.

[Liu, Y. and Yao, X.: 1996], "Evolutionary design of artificial neural networks with different nodes.", in Proc. 1996 IEEE Int. Conf. Evolutionary Computation (ICEC'96), Nagoya, Japan, pp. 670-675.

[Møller, M. F.], "A scaled conjugate gradient algorithm for fast supervised learning," *Neural Networks*, vol. 6, no. 4, pp. 525–533, 1993. *2nd IASTED Int. Symp. Expert Systems and Neural Networks*, M. H. Hamza, Ed. Anaheim, CA: Acta, 1990, pp. 74–77.

[Montana, D. and Davis, L :1989]."Training feedforward neural networks using genetic algorithms.", in Proc. 11th Int. Joint Conf. Artificial Intelligence. San Mateo, CA: Morgan Kaufmann, 1989, pp. 762-767.

[Moriarty, D. E. et. al.: 1996]. "Efficient reinforcement learning through symbiotic evolution". *Machine Learning*, 22:11–32.

[Obayashi, S. et al. 2000] "Multiobjective evolutionary computation for supersonic wing-shape optimization." *IEEE Transactions on Evolutionary Computation*, vol.4, no.2, p.182-187 (julio de 2000).

[Pietro, A. D., et. al.: 2002], 'Learning In RoboCup Keepaway Using Evolutionary Algorithms'. In: W. B. Langdon, E. Cant´u-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska (eds.): GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference. New York, pp. 1065–1072, Morgan Kaufmann Publishers.

[Potter, M. A. and K. A. D. Jong: 2000], "Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents". Evolutionary Computation 8, 1–29.

[Rocha M. and Neves J. 1996], "Preventing Premature Convergence to Local Optima in Genetic Algorithms via Random Offspring Generation".

[Sato S., Otori K., Takizawa A., Sakai H., Ando Y. and Kawamura H. 2002] "pplying genetic algorithms to the optimum design of a concert hall." *Journal of Sound and Vibration*, vol.258, no.3, p. 517-526 (2002).

[Schaffer, J. D., D. Whitley, and L. J. Eshelman: 1992], "Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art". In: D. Whitley and J. Schaffer (eds.): International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92). pp. 1-37, IEEE Computer Society Press.

[Stone, P. and Sutton, R. S.: 2002], 'Keepaway Soccer: a Machine Learning Testbed'. In: A. Birk, S. Coradeschi, and S. Tadokoro (eds.): RoboCup-2001: Robot Soccer World Cup V. Berlin: Springer Verlag, pp. 214–223.

[Stone, P. and Veloso, M.: 1998], "A Layered Approach to Learning Client Behaviors in the RoboCup Soccer Server". Applied Artificial Intelligence 12, 165-188.

[Stone, P. and Veloso, M.: 2000], "Layered Learning". In: R. L. de Mántaras and E. Plaza (eds.): Machine Learning: ECML 2000 (Proceedings of the Eleventh European Conference on Machine Learning). Barcelona,Catalonia,Spain: Springer Verlag, pp. 369-381.

[Stone, P.: 2000], Layered Learning in Multiagent Systems: A Winning Approach to

[Sutton, R. S.] “Two problems with backpropagation and other steepest-descent learning procedures for networks,” in *Proc. 8th Annual Conf. Cognitive Science Society*. Hillsdale, NJ: Erlbaum, 1986, pp. 823–831.

[Tang, K.S., Man K.F., Kwong S. and He. Q. 1996] “Genetic algorithms and their applications.” *IEEE Signal Processing Magazine*, vol.13, no.6, p.22-37 (noviembre de 1996).

[Whiteson S., et. al.: 2005]. “Evolving keepaway soccer players through task decomposition”. Machine Learning, 59(1):5

[Whiteson, S. and P. Stone: 2003], ‘Concurrent Layered Learning’. In: AAMAS 2003: Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems. pp. 193–200.

[Whitley, D. et. al.: 1991] “Delta Coding: An Iterative Search Strategy for Genetic Algorithms.” In Proceedings of the Fourth International Conference on Genetic Algorithms.

[Whitley, D., Starkweather, T. and Bogart, C.: 1990] "Genetic algorithms and neural networks: Optimizing connections and connectivity", Parallel Comput., vol. 14, no. 3, pp. 347-361,

[Yong, C. H. and R. Miikkulainen: 2001], “Cooperative Coevolution of Multi-Agent Systems”. Technical Report AI01-287, The University of Texas at Austin Department of Computer Sciences. Robotic Soccer. MIT Press.

APÉNDICE 1. HERRAMIENTA DE SOFTWARE UTILIZADA.

Introducción

Esta herramienta fue desarrollada utilizando el entorno de programación Visual Studio.Net 2003, y completamente codificada en el lenguaje C++. Todos los resultados expuestos en esta tesis fueron obtenidos experimentalmente utilizando este software.

Si bien la herramienta en si presenta una complejidad importante de implementación debido a las tecnologías empleadas, su utilización es intuitiva y las posibilidades de uso relativamente pequeñas.

Como funciones principales, la herramienta permite probar cada uno de los métodos de aprendizaje expuestos oportunamente. Dichas pruebas pueden visualizarse en tiempo real a medida que avanza la evolución.

Dos módulos principales componen la herramienta. El primero es una aplicación de consola que es la parte más importante de la misma, debido a que concentra toda la lógica de los métodos de aprendizaje. Este módulo solo, es suficiente para realizar todas las pruebas presentadas aquí. El segundo módulo es una interfaz gráfica de usuario que permite realizar todas las pruebas de forma amigable. Este módulo, agrega funcionalidades de visualización en tiempo real del comportamiento de los controladores y de análisis de resultados arrojados por cada uno de los métodos de aprendizaje.

Debido a que los experimentos neuroevolutivos son costosos en cuanto a recursos de cómputos y pueden extenderse mucho en el tiempo, se decidió hacer una interfaz de usuario optativa lo cual permite mejorar el tiempo de respuesta del módulo principal.

A continuación se expondrá la herramienta de software implementada y se mostrarán sus características principales.

Descripción de la aplicación

La interfaz gráfica se divide en dos tabs principales uno de simulación y otro de resultados. El tab simulación, se divide a su vez en otros dos tabs, uno de aprendizaje y otro de prueba.

En el tab aprendizaje, ver Figura I, se puede seleccionar alguno de los métodos de aprendizaje analizados en capítulos anteriores, así como también parámetros que afectan los mismos. Por ejemplo, se puede elegir el número de generaciones que evolucionaran los subcontroladores, la cantidad de neuronas que conforman cada población, el porcentaje de mutación para cada gen dentro de un cromosoma, el factor de delta coding y velocidad inicial del taker. Para el caso del aprendizaje por capas, se puede seleccionar el número de capa desde la cual continuar el aprendizaje (suponiendo que las capas previas fueron ya

aprendidas). Para el caso del aprendizaje cíclico se puede elegir la cantidad de ciclos que se realizarán durante la evolución de las *n* generaciones elegidas previamente.

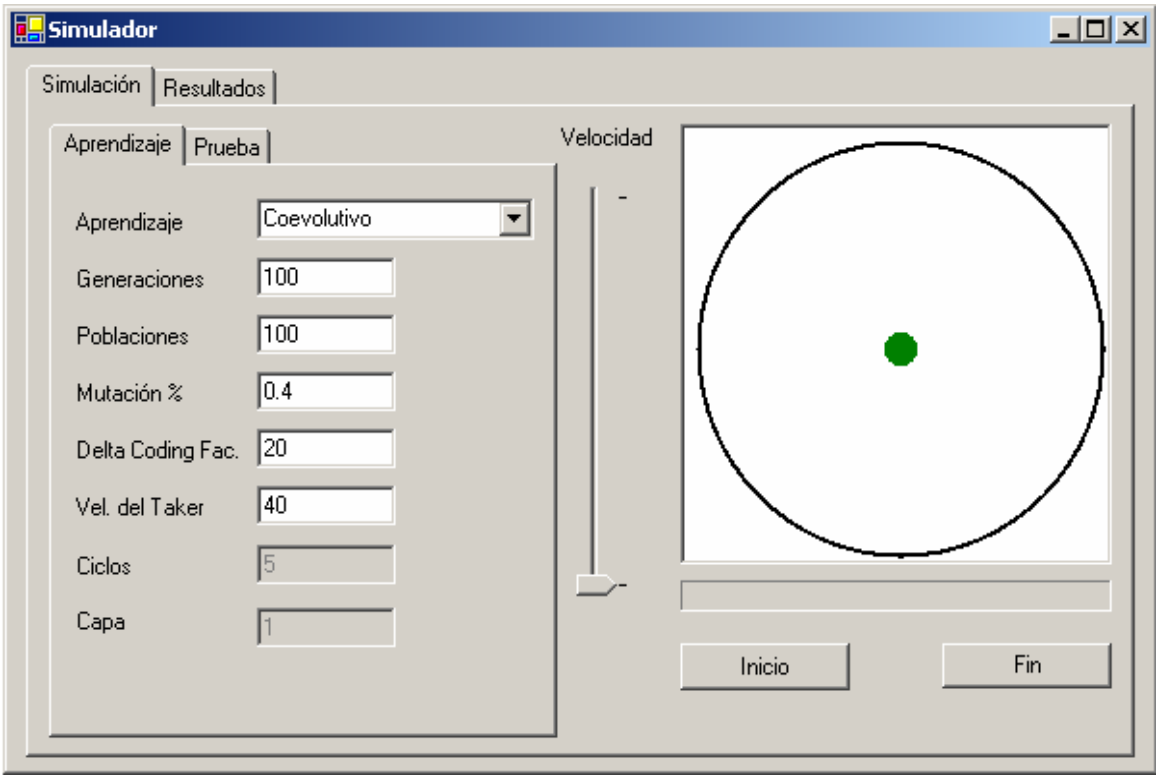

Figura I: Pantalla principal de la herramienta que permite probar los distintos métodos de aprendizaje, y visualizar su evolución en tiempo real.

Como muestra la Figura I, en la parte derecha del tab simulación hay un panel que permite visualizar la evolución de los subcontroladores en tiempo real y probar los mismos luego de finalizado el proceso de aprendizaje. Bajo este panel se encuentra una barra de progreso que indica el estado de avance de la evolución.

Para mejorar el tiempo de respuesta de la herramienta, los entrenamientos se realizan a la máxima velocidad posible la cual está limitada por la capacidad de cómputo del equipo utilizado. Como consecuencia, en general, no es posible apreciar con claridad el funcionamiento de los subcontroladores. Para resolver este problema, inmediatamente a la izquierda del panel de visualización, hay un control que permite cambiar la velocidad de la simulación.

En el caso de las primeras etapas del aprendizaje por capas se podrán visualizar los entrenamientos en ambientes especialmente diseñados para cada tipo de tarea. Por ejemplo, para el caso de la intercepción, se verá un jugador aprendiendo a ir a buscar la pelota. Para el resto de los aprendizajes se podrá observar como van progresando los controladores en el juego completo de keepaway.

Una vez finalizado cualquier tipo de aprendizaje, se puede probar el funcionamiento de los mejores subcontroladores obtenidos. Para este fin, dentro del tab simulación se encuentra un tab de Prueba (Véase Figura II) el cual permite mediante una operación de drag and drop elegir cada subcontrolador salvados previamente en disco.

Una vez seleccionados los subcontroladores a utilizar se puede comenzar la prueba de los mismos y en el panel de visualización de la derecha ver su funcionamiento.

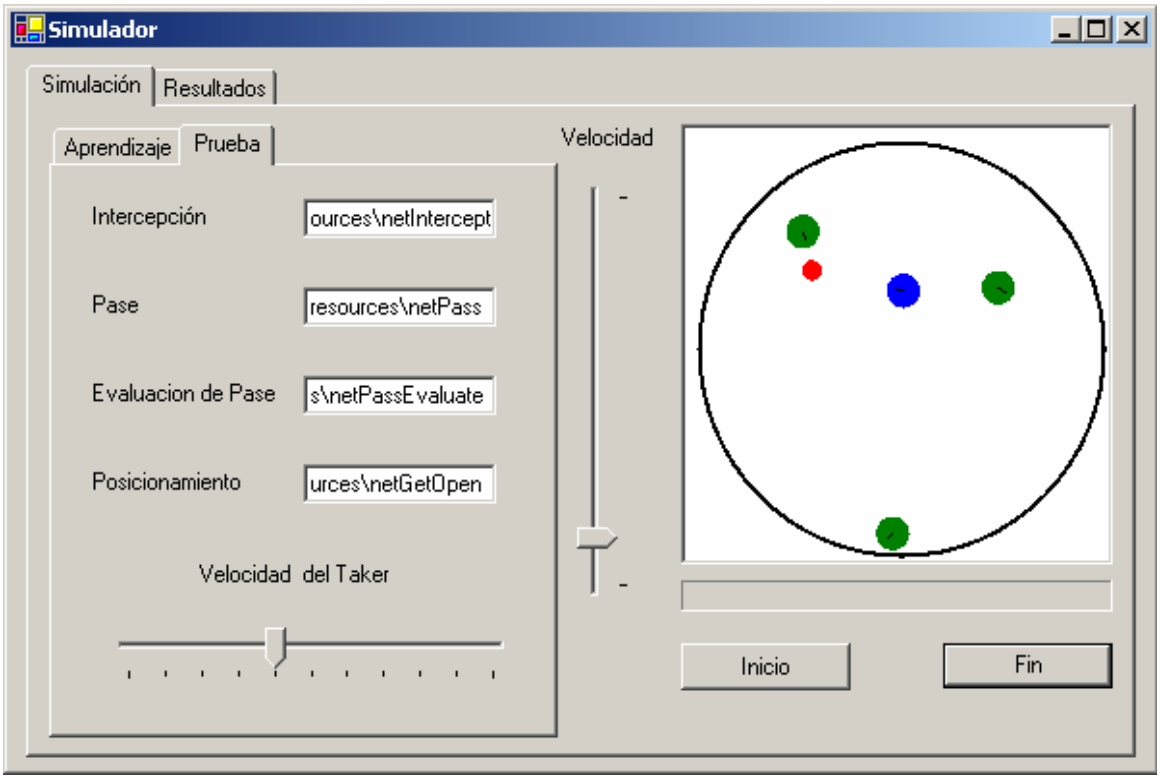

Figura II: La pantalla de prueba, permite cargar los subcontroladores obtenidos por los métodos de aprendizaje y visualizar su funcionamiento en un juego de keepaway

Al igual que en el caso del tab Aprendizaje, se puede modificar la velocidad de la simulación para una mejor apreciación de la misma, pero en el caso de tab de prueba, también se puede variar la velocidad del taker durante la simulación. Esto permite ver como responden los keepers a los distintos grados de dificultad a los que son sometidos.

Es importante destacar que se puede probar cualquier grupo de subcontroladores en cualquier grado de evolución o incluso mezclar subcontroladores obtenidos por distintos métodos de aprendizaje. Esta posibilidad da una gran flexibilidad y permite analizar los resultados desde otro punto de vista.

Todos los métodos de aprendizaje una vez finalizados generan un archivo de texto con los resultados, es decir, la dificultad alcanzada por los jugadores durante la evolución. Para la visualización de estos resultados, la herramienta cuenta con un tab llamado “Resultados”

(Véase Figura III) que está compuesto de dos paneles que permiten visualizar de forma grafica los mismos.

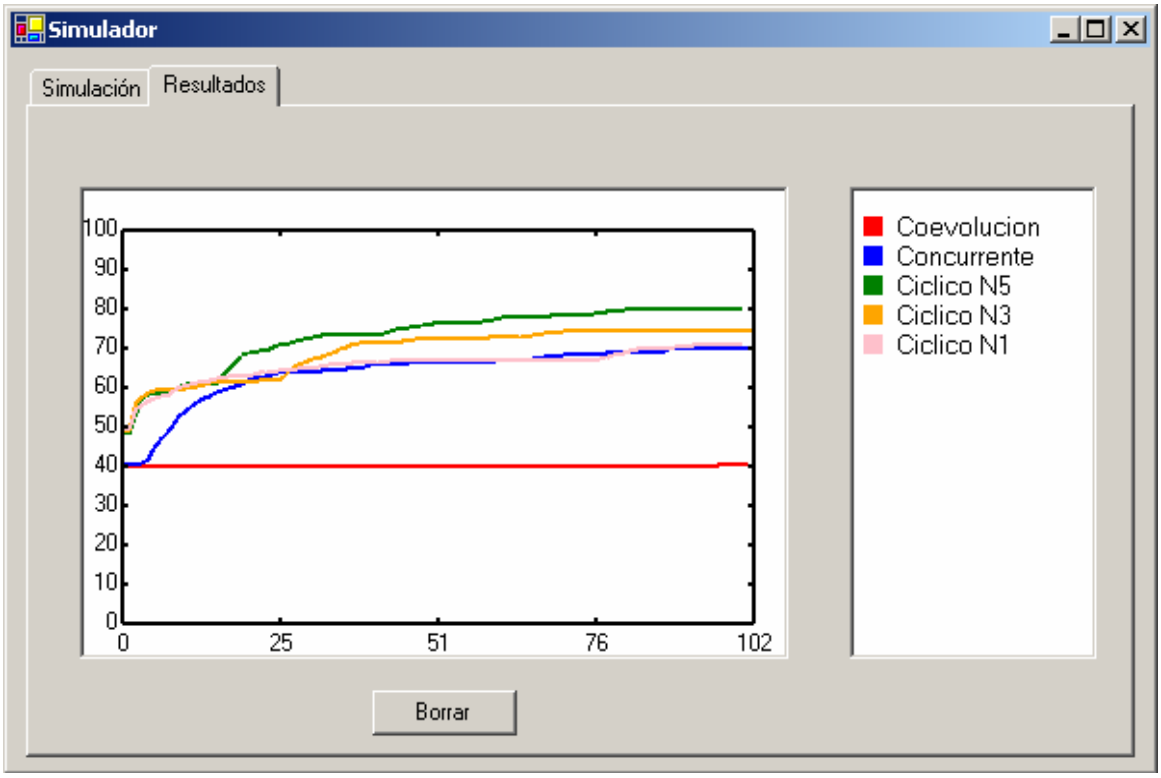

Figura III: La pantalla de resultados muestra en forma gráfica los resultados obtenidos por los diferentes métodos de aprendizaje.

Mediante una operación de drag and drop se pueden seleccionar los archivos de resultados que se desean visualizar. Cada archivo seleccionado muestra el nombre del método de aprendizaje con el cual se obtuvo y genera una línea que indica la dificultad alcanzada durante la evolución. Seleccionando varios archivos se pueden comparar todos los métodos de aprendizaje y sacar las conclusiones respectivas.

APÉNDICE 2. CYCLIC EVOLUTION. A NEW STRATEGY FOR IMPROVING CONTROLLERS OBTAINED BY LAYERED EVOLUTION.

El método denominado "Evolución Cíclica" propuesto por J. H. Olivera y L. Lanzarini y descripto en esta Tesis de Grado fue publicado en el XI Congreso Argentino de Ciencias de la Computación CACIC 2005 realizado en Concordia - Entre Ríos del 17 al 21 de octubre de 2005. Una posterior actualización de este artículo fue seleccionada para ser publicada en el S*pecial Issue on Selected Papers from CACIC 2005* correspondiente al Journal of Computer Science and Technology (JCS&T) *Vol. 5 - No. 4 - December 2005 - ISSN 1666-6038*
http://journal.info.unlp.edu.ar/journal/journal16/papers.html