Universidad de Granada

Facultad de Ciencias



Departamento de Electrónica y

Tecnología de Computadores

# Estudio, aplicaciones y optimización mediante algoritmos genéticos de algoritmos neuronales de cuantización vectorial

TESIS DOCTORAL

Juan Julián Merelo Guervós

Granada, 1994

**Departamento de Electrónica y Tecnología de Computadores**
**Facultad de Ciencias**
**Universidad de Granada**
**18071 GRANADA, Spain**



**D. Alberto Prieto Espinosa**, Catedrático del Departamento de Electrónica y Tecnología de Computadores, y

**D. Julio Ortega Lopera,** Profesor Titular del Departamento de Electrónica y Tecnología de Computadores

**CERTIFICAN**

Que el trabajo de investigación que se recoge en la presente Memoria, titulada *Estudio, aplicaciones y optimización mediante algoritmos genéticos de algoritmos neuronales de cuantización vectorial*, y presentada por **D. Juan Julián Merelo Guervós** para optar al grado de Doctor en Ciencias Físicas, ha sido realizado en su totalidad bajo su dirección en el Departamento de Electrónica y Tecnología de Computadores de la Universidad de Granada.

Granada, 9 de Septiembre de 1994.

Fdo.: Alberto Prieto Espinosa
Catedrático de Universidad

Fdo.: Julio Ortega Lopera
Profesor Titular de Universidad

Dpto. Electrónica y Tecnología de Computadores - Facultad de Ciencias - Universidad de Granada - Campus Universitario Fuentenueva - 18071 GRANADA (Spain) - Tfno. 34-58-243230 Fax: 34-58-243230

# Indice de materias

# Introducción

El objetivo de esta memoria es la presentación de un nuevo algoritmo, que se ha denominado G-LVQ, y la demostración de su utilidad aplicándolo a una serie de problemas de interés en el campo de la clasificación de patrones. Este nuevo algoritmo combina una serie de técnicas conocidas, como los algoritmos genéticos, las redes neuronales, con otras clásicas, como los algoritmos de cuantización vectorial. Con ello se consigue una mejora notable de algunas de sus características. Se han implementado además todos los programas necesarios para su prueba y explotación.

El comienzo de la investigación en redes neuronales tuvo lugar en los años 50, cuando McCulloch y Pitts describieron por primera vez la denominada *neurona formal*. Desde entonces, la investigación en redes neuronales, inspirada en su mayor parte en el sistema nervioso de los mamíferos superiores, ha tenido una gran expansión, ocupando en la actualidad un lugar importante en muchos proyectos de investigación. El campo en el que las redes neuronales artificiales (RNAs) encuentran mayor aplicación es el de reconocimiento de patrones; es precisamente en este campo donde ha convergido con otras técnicas de reconocimiento de patrones clásicas, principalmente basadas en métodos estadísticos, como la cuantización vectorial.

Tanto las redes neuronales como otros métodos estadísticos clásicos adolecen del problema de tener gran cantidad de parámetros libres, que el investigador tiene que ajustar a mano o mediante el método de ensayo y error, probando varias combinaciones hasta obtener buenos resultados. Para ello, evidentemente es necesario evaluar de alguna forma las prestaciones de las redes neuronales, y sistemáticamente encontrar la combinación de parámetros con la cual se obtengan los mejores resultados.

Sin embargo, algunas técnicas de búsqueda sistemática, como los algoritmos genéticos, también basados en procesos naturales, y en concreto, en la teoría de la evolución de las especies por selección natural de Darwin y su base genética, permiten explorar con más eficiencia el espacio de parámetros, hasta dar con una solución óptima. Cuanto menos

restringido esté el espacio de parámetros, más fácil será obtener una solución óptima global, o, en el caso de las redes neuronales, cuantos menos parámetros de la red neuronal se restrinjan, más fácil será hallar el máximo global de prestaciones posible, o al menos una solución suficientemente cercana al mismo.

Hasta ahora, todo acercamiento a la optimización de redes neuronales solía incluir una limitación: la de la estructura de la red neuronal, que se suponía fija. Los casos en que se han utilizado redes neuronales de longitud variable han sido bastante restrictivos (redes neuronales incrementales o evolutivas), y sólo para algún modelo de red neuronal determinado. El algoritmo G-LVQ, que se presenta en esta memoria, supera las limitaciones de otros algoritmos diseñando de forma integral una red neuronal de aprendizaje competitivo, o dicho de otra forma, un diccionario de cuantización vectorial, de forma totalmente automática, incluyendo el número de vectores diccionario y pesos de los mismos. Dado que los objetivos que se pretenden alcanzar en optimización la optimización de la red neuronal son diversos, a saber, una longitud mínima del diccionario (que permita una implementación software rápida y una implementación hardware eficiente), un número de aciertos máximo, y una distorsión mínima, para que el diccionario represente con exactitud las muestras con las que se ha entrenado. No se pueden utilizar los criterios habituales en algoritmos genéticos, ya que en estos es una sóla cantidad la que se optimiza. Se utiliza, por tanto, un método de evaluación *vectorial*.

Considerando el importante aspecto de una posible implementación en hardware paralelo de un algoritmo genético, es interesante contar con un algoritmo que reduzca al mínimo la comunicación entre procesadores (en los cuales se situarán los diferentes genomas, en este caso). En el contexto de la optimización de redes neuronales es poco importante, ya que la mayor parte del tiempo se consume entrenando la red neuronal. Sin embargo, en el caso más general, el algoritmo genético consume la mayor parte del tiempo en la comparación de *fitness* entre diversos elementos de la población. Para acelerar esta etapa al máximo, se utiliza un método de evaluación y selección local, que, sin embargo, no impide el buen funcionamiento del algoritmo genético, y además, se encuadra plenamente dentro de la filosofía de los creadores de los algoritmos genéticos.

Esta construcción teórica no serviría de nada sin una evaluación, que ponga de relieve que los programas elaborados para implementar los algoritmos actúan eficientemente y con

exactitud, y que por otro lado, sirven efectivamente como clasificadores, con una exactitud superior, si es posible, a otros modelos presentes en la literatura. Estos programas deberán cumplir los requisitos exigibles hoy en día a cualquier producto software creado en el ámbito científico: portabilidad, extensibilidad y elegancia, de forma que cualquier otro científico o informático que desee mejorar el producto o personalizarlo no tenga que elaborar desde el principio todas las herramientas necesarias para implementar el programa, sino que pueda aprovechar lo ya realizado.

En los primeros capítulos de esta memoria (Capítulos 1 a 3), se examinará el concepto de cuantización vectorial, y algoritmos que han servido para entrenar cuantizadores vectoriales, tanto en el campo clásico como dentro de los denominados redes neuronales. Dentro de estos algoritmos se encuadran los algorimos de Kohonen, y dentro de ellos se prestará especial atención al denominado Cuantización Vectorial Aprendida o LVQ. Estos algoritmos son susceptibles de una optimización básica, mediante prueba sistemática de parámetros, y usando esta optimización, pueden aplicarse a problemas de extrapolación, como se verá en el capítulo 4.

A partir del capítulo 5 se tratan los algoritmos que pretenden obtener redes neuronales alterando su estructura, tanto mediante técnicas heurísticas como mediante algoritmos genéticos. Se presentan sus limitaciones, y se introduce por primera vez el algoritmo G-LVQ junto con los criterios que se han seguido para su diseño. En el capítulo 6 se examinan las herramientas, programas y lenguajes de programación, usados en el diseño, evaluación y prueba del algoritmo G-LVQ, su adecuación al cometido y justificación de su utilización.

Los dos últimos capítulos, 7 y 8, están dedicados a la prueba y evaluación del algoritmo. Inicialmente, se examina el algoritmo aplicándolo a problemas simples para comprobar que se comporta como un algoritmo genético, y analizar con detenimiento su comportamiento. También se evalúa la interacción entre redes neuronales y algoritmos genéticos y la complejidad temporal del algorimo. Por último, en el capítulo 8 se aplica el algoritmo a una amplia gama de problemas, desde clasificación de grupos linealmente separables, hasta problemas linealmente no separables, y a problemas reales, como clasificación de imágenes de macromoléculas y de "imágenes" de sonar.

# 1 Algoritmos de cuantización vectorial

## 1.1 Introducción

Los algoritmos de Kohonen [Koh82b,90a], en torno a los cuales se desarrolla esta memoria, se pueden encuadrar dentro del grupo de algoritmos denominados de *Cuantización Vectorial*.

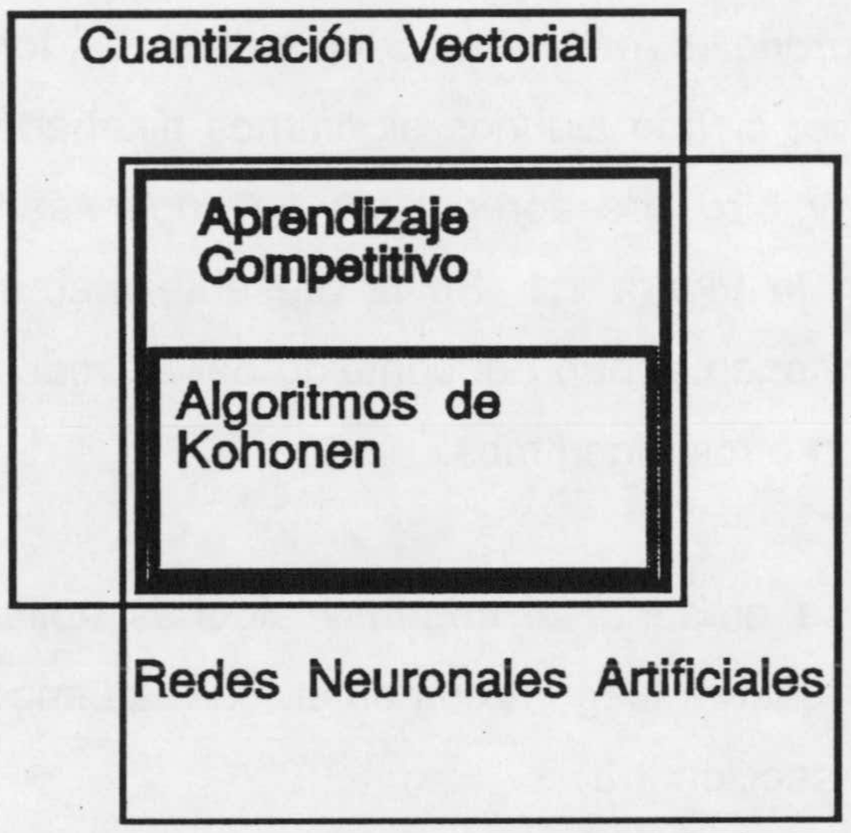



**Figura 1.1** Los algoritmos de Kohonen en el contexto de las RNAs y de los algoritmos de cuantización vectorial.

Esta familia de algoritmos fueron propuestos inicialmente en los años 50 [Llo57] para reducir la cantidad de información a enviar por una línea de transmisión [Mak85]; sin embargo, hoy en día los algoritmos de cuantización vectorial o *cuantizadores vectoriales* se utilizan ampliamente también dentro del campo del reconocimiento automático de patrones. Por ello, gran parte de las herramientas utilizadas para medir la eficiencia de estos algoritmos, así como los tests o pruebas utilizadas para valorar sus prestaciones, suelen provenir de este campo. La eficacia de cualquier nuevo algoritmo de cuantización vectorial propuesto tendrá que ser comparada frente a otros algoritmos clásicos, para justificar su utilización.

Para diseñar un cuantizador vectorial para una señal $x$ es necesario extraer una

muestra estadística de todos los valores posibles que ésta pueda tomar. A partir de esta muestra, se ha de calcular un conjunto de funciones que representen de forma más compacta la señal que la simple enumeración de todos los elementos de la muestra. Habitualmente, esas funciones son distancias a un conjunto de vectores (de cardinalidad inferior a la de la muestra) denominado *diccionario*. Por tanto, el conjunto de funciones que actuará como cuantizador vectorial se tendrá que hallar mediante un entrenamiento a partir de las muestras de entrada conocidas.

A la vez que gran parte de la funcionalidad de los algoritmos de Kohonen se debe a que son cuantizadores vectoriales, también se pueden encuadrar dentro del extenso campo conocido como *Redes Neuronales Artificiales*. Dentro de estos, forman parte de los algoritmos de *Aprendizaje Competitivo*, siendo algunos algoritmos diseñados por Kohonen de ellos de aprendizaje supervisado, y otros de aprendizaje no supervisado. La relación entre estos algoritmos se muestra en la Figura 1.1. En la presente memoria se tratará de situar los diversos algoritmos de Kohonen dentro del contexto de las redes neuronales, señalando los parecidos y diferencias con otros algoritmos.

En este capítulo se presenta el origen y algunas aplicaciones de la cuantización vectorial (sección 1.2); se pasará luego a comentar los más importantes algoritmos clásicos de cuantización vectorial (sección 1.3).

## 1.2 Cuantización vectorial

### 1.2.1 Introducción: reconocimiento de patrones

El objetivo básico de la disciplina conocida como *reconocimiento de patrones* consiste en [Koh90] la categorización de datos vectoriales estocásticos en clases discretas (es decir, en clases tales que la función de pertenencia sea del tipo sí o no; cada elemento puede pertenecer sólo a una clase) de acuerdo con ciertas relaciones métricas simples. Estas métricas tienen en cuenta para la clasificación la distancia de una muestra a todas las demás muestras.

Hay muchas formas tradicionales de resolver un problema de clasificación; se suelen clasificar en dos grupos, *métodos paramétricos*, y *métodos no paramétricos*. En los primeros se supone que se conoce de antemano la forma de la función de densidad de probabilidad *fdp* $p(\vec{x})$, y sólo hay que calcular los parámetros de la misma; por ejemplo, la media y la varianza en el caso de que la *fdp* se pueda expresar como una suma de funciones gaussianas. En los métodos no paramétricos no es necesario conocer la forma de la *fdp* sino que se suele estimar la densidad de probabilidad de forma numérica a partir de una serie de patrones.

La cuantización vectorial es un método no paramétrico, puesto que no supone de antemano ninguna forma de la *fdp*; aunque es cierto que la solución (es decir, el conjunto de vectores que mejor describe la muestra) se podrá hallar más fácilmente si la función tiene cierta forma, por ejemplo, distribución uniforme e igual a $u$ en una zona del espacio y 0 fuera de ella.

### 1.2.2 Definición de cuantización vectorial

Un *cuantizador vectorial* Q [Yai92] es una aplicación del espacio euclídeo $k$-dimensional $\mathbf{R}^k$ en un subconjunto finito $Y = \{y_1, y_2, \ldots y_K\}$ de $\mathbf{R}^k$. Cada $y_j$ se suele denominar *vector diccionario* o *vector código* o *vector de referencia o centro*; en la literatura relacionada con reconocimiento de patrones se les suele llamar patrones o *templates*; a $Y$ se le suele denominar *diccionario o alfabeto* (*codebook*), y a $K$ se le suele denominar tamaño o *número de niveles* del diccionario por analogía con la cuantización escalar (en la cual no se cuantizan vectores, sino cantidades escalares). Para $K = 1$, la cuantización vectorial se reduce a una cuantización escalar (también denominada *algoritmo de Lloyd* [Llo57]).

Con las anteriores definiciones, se cumple

$$\forall \vec{x} \in \mathcal{E} \subset \mathbf{R}^k \; \exists \, i \; / \; Q(\vec{x}) = \vec{y}_i \tag{3}$$

o, dicho de otra forma

$$Q(\vec{x}) = i \tag{4}$$

A cada vector del espacio de entrada se le asigna un vector diccionario; o bien, dicho de otra forma, cada vector diccionario responde a una zona del espacio de entrada $\mathbf{R}^k$,

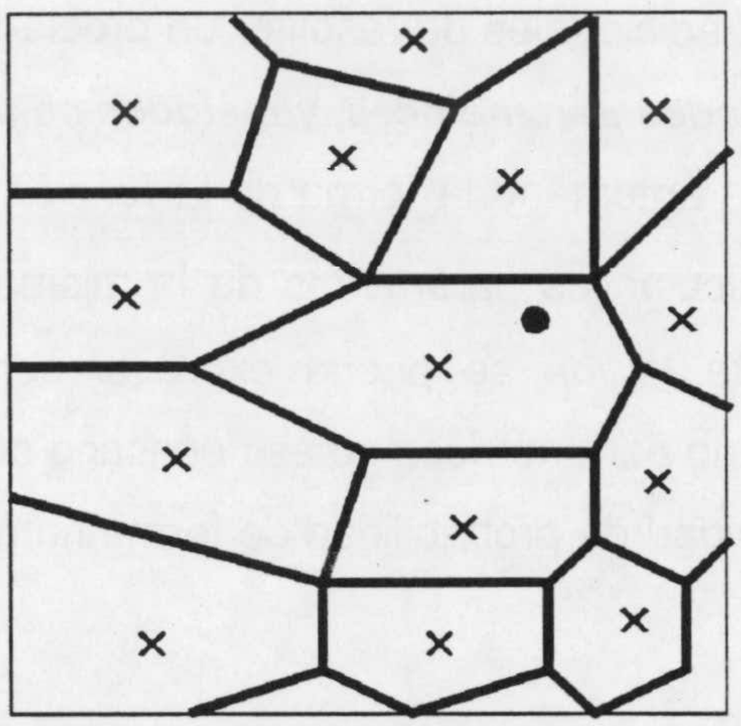


**Figura 1.2** *Teselación de Voronoi,* donde x = centroide de cada celda; el vector • se cuantizaría como el vector representado por la cruz en la misma celda.

provocando por tanto una partición del mismo. $\mathbf{R}^k$ se divide en una serie de células $C_i$ $1 \leq i \leq K$, de forma que si $\vec{x}$ cae dentro de la célula $C_i$, se cuantiza como $\vec{y}_i$, como se puede ver en la Figura 1.2.

$$Q(\vec{x}) = \vec{y}_i \Leftrightarrow \vec{x} \in C_i \qquad (3)$$

Al proceso de diseño del diccionario se le suele denominar *entrenamiento* o *repoblación*.

Cada vez que se sustituye un vector $\vec{x}$ por el valor resultante de la cuantización vectorial $Q(\vec{x})$ se produce un cierto error, denominado *distorsión*, dado por la distancia $d(\vec{x}, Q(\vec{x}))$; las prestaciones de un cuantizador vectorial se miden por la distorsión media,

$$D(Y) = E[d(\vec{x}, Q(\vec{x}))] \qquad (4)$$

entre los vectores de entrada y los vectores cuantizados de reproducción; $Y$ es el diccionario para el cual se calcula la distorsión, y $E[]$ denota el operador esperanza matemática. La ecuación (4), teniendo en cuenta la forma del operador $E$, se puede expresar

$$D = \lim_{M \to \infty} \frac{1}{M} \sum_{n=1}^{M} d(\vec{x}, Q(\vec{x})) \qquad (5)$$

$M$ es el número de vectores que hay en el conjunto de entrenamiento, es decir, $M=card(\mathcal{E})$; $d$ es alguna función de distancia, que suele ser habitualmente la distancia de error cuadrática

$$d(\vec{x},\vec{y}) = \| \vec{x}-\vec{y} \|^2 \quad (6)$$

aunque también se puede tomar cualquier otra distancia, según convenga al problema o a la potencia de cálculo disponible. Otra medida de distancia habitual es $d_1$, donde

$$d_1(\vec{x},\vec{y}) = \sum_{i=1}^{k} |x_i-y_i| \quad (7)$$

Estas dos distancias se denominan medidas de distancia de diferencia, puesto que sólo dependen del vector que une los extremos de los dos vectores, es decir,

$$d(x,y)=d(x-y) \quad (8)$$

Un cuantizador vectorial es tanto mejor cuanto menor es la distancia media entre un vector diccionario y todos los vectores dentro de la zona a la cual *responde*. Un procedimiento de diseño de un cuantizador vectorial consiste habitualmente en la minimización de este error.

Para que un cuantizador vectorial basado sea óptimo, debe de cumplir dos condiciones

[O1]. Debe de utilizar una regla de selección de tipo *vecino más cercano*, es decir

$$Q(\vec{x}) = \vec{y}_i \Leftrightarrow d(\vec{x},\vec{y}_i) \leq d(\vec{x},\vec{y}_j),\ \forall j \neq i,\ 1 \leq j \leq K \quad (9)$$

[O2]. Cada vector código $\vec{y}_i$ debe de ser elegido de forma que minimice la distorsión media en la célula $C_i$; a tal vector se le llama el *centroide* de la célula $C_i$, es decir

$$\vec{y}_i = cent(C_i) \quad (10)$$

### 1.2.3 Origen y aplicaciones

Los procedimientos de cuantización vectorial tienen su origen en los sistemas de transmisión de datos. Su principal objetivo es reducir el número de bits necesarios para transmitir una señal, con lo cual disminuye el tiempo necesario para transmitirla a través del canal. Como es obvio, esta reducción en el número de bits transmitidos debe de intentar no hacer perder calidad al mensaje.

Por ejemplo, si se desea enviar la información digital resultante de la codificación de una señal de voz por un cable telefónico normal, a la señal audio, tras la conversión analógico-digital, se le aplica la *tranformada rápida de Fourier*, de forma que se obtenga un espectrograma, es decir, una serie de vectores, con componentes correspondientes a cada gama de frecuencias. El algoritmo de cuantización vectorial, y, por tanto, la reducción de dimensión, se aplicará sobre estos vectores resultantes.

Se puede demostrar que el conjunto de vectores utilizados en la transmisión de una señal no ocupa todo el espacio (evidentemente, ya que en cada mensaje hay un número finito de vectores), pero incluso si lo ocupase, el oído humano (o cualquier otro medio de recepción) tienen un cierto margen de error, y no pueden apreciar si lo que se recibe es un sonido u otro ligeramente diferente. Por lo tanto, al transmitir esa señal, existe una cierta libertad para cambiar un vector por otro, siempre dentro del margen del error del sensor que lo capte o de las exigencias de un sensor automático.

Para reducir el número de bits a transmitir, se crea un *diccionario* o grupo de *vectores de referencia* $Y = \{\vec{y}_1, \vec{y}_2, \ldots, \vec{y}_n\}$ a partir de una serie de muestras de habla real o de sonido, y se transmite, en vez del vector señal original $\vec{x}$, el número de orden o índice *i* del vector de referencia más cercano, $\vec{y}_i$. A la recepción de la señal, se vuelve a convertir el código del vector recibido, *i,* en una señal audible.

Así se reduce el tiempo de transmisión, pues en vez de transmitir *n* (dimensión del vector señal original) * *m* (número de bits necesarios para representar cada componente del vector), se transmiten sólo *q* (número de bits necesarios para representar el índice del vector de referencia).

Otra aplicación evidente es el reconocimiento automático de patrones. Si a cada uno de los vectores del diccionario se le asigna una *etiqueta A*, y se introduce un vector de entrada $\vec{x}$, al cual "responde" tal vector del diccionario, se dice que $\vec{x}$ pertenece a la clase A, o que ha sido reconocido como A. Se volverá sobre el tema de clasificación automática más adelante. Evidentemente, una versión supervisada o etiquetada de los algoritmos de cuantización es ligeramente diferente, ya que el vector diccionario y la célula a la que responden deben pertenecer idealmente a la misma clase.

## 1.3 Algoritmos clásicos de cuantización vectorial.

Como se ha comentado en el apartado 1.2.1, los algoritmos de cuantización vectorial consisten en un proceso iterativo de diseño del diccionario, de forma que se minimice la distorsión. Uno de estos algoritmos es el que se conoce como *algoritmo k-medias* en la literatura de reconocimiento de patrones [Mak85], aunque también se le denomina algoritmo *LBG* (Linden, Buzo & Gray) [LBG80], y algoritmo de Lloyd generalizado (*GLA*, generalized Lloyd algorithm) [Yai92], o bien algoritmo *ISODATA básico* [Duda73]. El algoritmo divide el conjunto de vectores de entrenamiento en *K clusters* o grupos, de forma que se cumplan las dos condiciones anteriores de optimalidad ([O1] y [O2]).

El algoritmo k-medias únicamente divide el conjunto de muestras de entrada en clases, sin asignar a cada clase una etiqueta, y sin que el usuario le indique en qué clase debe de colocar cada muestra, y, por lo tanto, es un método no supervisado. Sin embargo, sólo necesita una fase posterior de calibración (es decir, de asignación de una etiqueta a cada vector del diccionario), para convertirse en un algoritmo de clasificación.

**Algoritmo K-medias**

Las variables son las siguientes:

- $K$, número de niveles o tamaño del diccionario, es decir, número de vectores de referencia.
- $m$ es el índice de la iteración.
- $M$ es el número de vectores en el conjunto de entrenamiento

[ K1 ] *Inicialización*: Elegir por cualquier método adecuado un conjunto de vectores de referencia iniciales $\vec{y}_i(0)$, $1 \le i \le K$.

[ K2 ] *Clasificación*: Clasificar el conjunto de vectores de entrenamiento $\{ \vec{x}(n),\ 1 \le n \le M \}$ en los *clusters* $C_i$ por la regla del vecino más cercano.

$$\vec{x}(n) \in C_i(m) \iff d(\vec{x},\vec{y}_i(m)) \le d(\vec{x},\vec{y}_j(m))$$

[ K3 ] *Actualizar los vectores de referencia*: m := m + 1; se actualizan los vectores de

referencia de cada cúmulo calculando el centroide de los vectores de entrenamiento de cada cluster

$$\vec{y}_i(m) = \frac{1}{M_i} \sum_{\vec{x} \in C_i} \vec{x}$$

[ K4 ] *Test de terminación*: cuando las distancias de los vectores de cada cluster al centroide del cluster es menor que un umbral determinado, o no varía a lo largo de un número predeterminado de iteraciones, se termina.

---

El algoritmo k-medias, como se puede observar, es un algoritmo que trabaja por lotes; debe de introducírsele todo el conjunto de entrenamiento antes de modificar los vectores del diccionario. Cada vez que se introduce un nuevo vector en el conjunto de entrada, debe recalcularse todo el diccionario, ya que hay que reagrupar de nuevo todas las muestras disponibles y volver a calcular los centroides (pasos K1-K4); esto plantea serios problemas para su utilización en línea [Yai92] o en tiempo real, ya que, no sólo tarda un tiempo considerable (dependiente del tamaño de la muestra), sino que no se puede asegurar de antemano cuánto va a tardar en converger.

Por lo que se acaba de comentar, el algoritmo k-medias, además de ser un algoritmo de cuantización vectorial, forma parte de los denominados *algoritmos de clustering* [Duda73] o de acumulación, puesto que acumula las muestras en una serie de grupos. Un procedimiento de clustering proporciona una descripción de los datos de entrada en términos de grupos o clusters de puntos que poseen similitudes internas fuertes; también es un procedimiento de optimización iterativo, pues halla la solución, ya que se va acercardo progresivamente a la solución en cada iteración.

El algoritmo K-medias permite obtener un estimador de máxima verosimilitud de la media de los clusters, cuando se conozca de antemano cuantos clusters hay y la varianza de los clusters sea similar, es decir, cuando no haya clusters de tamaño muy diferente.

Para estimar lo bueno que es un agrupamiento de muestras en clusters, se suele utilizar alguna función, denominada *función criterio*. Un valor óptimo de esta función criterio indicará que se ha efectuado un clustering óptimo. La función de clustering más utilizada es

el error cuadrático medio entre miembros de una clase, $J_e$, siendo

$$J_i = \frac{1}{M_i} \sum_{\vec{x} \in C_i} d(\vec{x}, \vec{y}_i) \qquad (11)$$

donde $\vec{y}_i$ es el centro de la clase; como se puede observar, $J_i$ es la distorsión D definida anteriormente en la ecuación (4). Por tanto, el problema de clustering se puede reducir a un problema de minimización iterativa de esta función, como en el algoritmo siguiente [Hart73].

**Error Cuadrático Medio**

▸$M_i$ es el número de vectores pertenecientes a la clase $i$

[E1] Seleccionar una partición inicial de las M muestras en grupos y calcular $J_e$ y las medias $\vec{y}_1, \vec{y}_2, \ldots \vec{y}_K$.

[E2] Seleccionar la próxima muestra $\vec{x}$, suponer que pertenece a $C_i$. (i=1,...,K)

[E2.1] Si $n_i = 1$, pasar a E2; si no calcular

$$\rho_j = \begin{cases} \frac{M_j}{M_j+1} \|\vec{x}-\vec{y}_j\|^2 & j \neq i \\ \frac{M_i}{M_i-1} \|\vec{x}-\vec{y}_i\|^2 & j = i \end{cases}$$

[E2.2] Transferir $\vec{x}$ a $C_k$ si $\rho_k \leq \rho_j \; \forall j$.

[E2.3] Actualizar $J_e$, $\vec{y}_i$ y $\vec{y}_k$ (es decir, clase de $\vec{x}$ en la generación y en la generación actual).

[E3] Si $J_e$ no ha cambiado durante $n$ pasos, terminar; si no, ir a E2.

Este algoritmo viene a ser una versión secuencial del K-medias; y por tanto puede ser utilizado en línea. Sin embargo, viene a ser un procedimiento de descenso de gradiente de error, y por tanto puede plantear el problema de verse estancado en un mínimo local; tiene también el serio problema de elección del diccionario inicial que condiciona fuertemente el mínimo de error que se va a alcanzar a lo largo de la ejecución del algoritmo.

En otro algoritmo similar al k-medias, denominado *ISODATA*, [Tou74], se utiliza un procedimiento heurístico para la eliminación y/o añadido de nuevos vectores al diccionario durante el entrenamiento a partir de las muestras de entrada. La filosofía de este algoritmo

se acerca a la de los algoritmos incrementales que se estudiarán más adelante (en el Capítulo 5), sin embargo, su base es puramente heurística.

Estos algoritmos de cuantización vectorial básicos plantean una serie de problemas porque no se puede garantizar la convergencia con conjuntos de muestras *reales* cuya estructura estadística no se adecúe a las hipótesis que se han hecho (es decir, clusters con varianza similar, cuyo número es conocido, y cuyo solapamiento es casi nulo). Además, como simples procedimientos de descenso de gradiente que son (es decir, que tratan de minimizar el error a base de pequeños pasos en los cuales el error disminuye) es muy posible que se vean atrapados en mínimos locales, es decir, que alcancen un supuesto mínimo de error (y por tanto no varíe la distorsión), y termine el algoritmo, aun cuando no se ha alcanzado el mínimo global; esto sucede incluso aunque las hipótesis de aplicación de los algoritmos se cumplan. Admiten, por lo tanto, muchas mejoras, lo cual será la base para la creación de los diversos algoritmos de Kohonen y sus modificaciones.

## 1.4 Conclusiones

Los algoritmos de cuantización vectorial, aunque concebidos inicialmente para codificación o compresión de una señal transmitida por un canal, actualmente son útiles para almacenamiento (por ejemplo, un procedimiento de cuantización vectorial se utiliza en el algoritmo de compresión JPG), y para la clasificación automática de señales previamente convertidas en vectores de números reales.

Una serie de algoritmos clásicos, diseñados a partir de los años 60, se han encargado del diseño de diccionarios, como el *k-medias*, pero presentan una serie de problemas, sobre todo a la hora de trabajar con espacios de entrada difíciles, como el problema de clases no linealmente separables de Hart (que se analizará en detalle en el capítulo 8). Uno de los principales problemas es el de la inicialización, basada casi siempre en reglas heurísticas, y que influye fuertemente en las prestaciones de los algoritmos. El segundo problema se presenta debido a que son procedimientos de optimización locales, y por lo tanto pueden quedarse atascados en un mínimo local. Y por último, salvo en el caso del ISODATA, es necesario saber de antemano en cuantos clusters se va a clasificar la muestra de entrada,

o lo que es lo mismo, el tamaño del diccionario. El algoritmo propuesto en esta memoria tratará de resolver estos tres problemas.

# 2. Mapas autoorganizativos de Kohonen

## 2.1 Introducción

En este capítulo se comienza a tratar de la familia de algoritmos de los que versa esta tesis, el grupo de algoritmos de Kohonen. Este grupo de algoritmos está formado por el mapa autoorganizativo de Kohonen (*Self-Organizing Map,* SOM) [Koh82a], junto con los denominados LVQ1, 2 y 3 [Koh90].

Estos algoritmos son, junto con el perceptrón multicapa adiestrado con retropropagación, los más utilizados dentro del campo de las redes neuronales. Su aplicación principal es, como se ha visto en el capítulo anterior, y dada su función de cuantizador vectorial, la clasificación, aunque hay otras muchas, desde la interpolación con eliminación de ruido hasta el análisis semántico [Scho91, Rit89, Tat89]. Su amplia utilización se debe, sobre todo, a su rapidez, tanto en el entrenamiento como en la posterior fase de explotación (que se diferencia esencialmente de la fase anterior en que todos los parámetros del algoritmo permanecen fijos), y a su simplicidad conceptual, ya que no hay que hacer ninguna suposición sobre la estructura de la muestra de entrada, aunque, claro está, esto no indica que funcione en todos los casos.

Los algoritmos que implementan redes neuronales contienen gran cantidad de parámetros cuyo valor tiene que ser establecido por el usuario; en muchos casos, del valor de estos parámetros dependerá que el algoritmo realice su función adecuadamente. Por tanto, un problema esencial en la teoría de las redes neuronales artificiales será hallar un conjunto de parámetros tal que el algoritmo produzca los resultdos óptimos (por ejemplo, un mínimo error de clasificación, o una distorsión mínima). En esta memoria se irá avanzando desde métodos de optimización rudimentarios (como por ejemplo optimización de un solo parámetro, u optimización heurística), hasta métodos más potentes y generales, como los que se introducirán aquí por primera vez en el capítulo 5.

En la sección 2.2 se hablará del problema que trataba de solucionar el algoritmo SOM;

aunque es totalmente diferente a los problemas abordados por los algoritmos de cuantización vectorial, su solución va a ser similar. Se relacionará también con los algoritmos de cuantización vectorial, vistos en el capítulo 1, y se justificará su inclusión dentro de este grupo.

En la sección 2.3 se expondrá el algoritmo SOM, y a la vista de su desarrollo, se justificará su inclusión dentro de las redes neuronales artificiales; se comentarán también los parámetros que hay que definir a la hora de entrenar y utilizar una red de Kohonen y cómo influyen en sus prestaciones, y su variación sistemática como forma rudimentaria de optimización.

Posteriormente, en las secciones 2.4 y 2.5, se analizarán otros algoritmos sencillos relacionados con la cuantización vectorial: la cuantización vectorial aprendida, o LVQ, ideada por Kohonen [Koh91] y el aprendizaje competitivo en su variante más simple [Rum85]. Por su simplicidad, estos algoritmos serán más flexibles con respecto a la optimización. Una característica los distingue: en general, LVQ es un algoritmo de aprendizaje supervisado, mientras que el aprendizaje competitivo suele ser no supervisado (aunque se puede utilizar también de modo supervisado, como sucede en general con todos los algoritmos de cuantización vectorial).

## 2.2 Planteamiento del problema: mapas topográficos en el cerebro

### 2.2.1 Introducción

En realidad, Kohonen pretendía la resolución de un problema totalmente diferente al presentado inicialmente en el capítulo 1 (es decir, el problema de compresión de la información para su clasificación, almacenamiento o transmisión); sin embargo, su solución es notablemente similar a los algoritmos de cuantización vectorial, y, de hecho, sirve también para este propósito.

Mediante técnicas neurofisiológicas se ha demostrado que en el cerebro se produce una ordenación espacial de las señales procedentes de los sentidos, de forma que cada zona

del cerebro responde a una zona del espacio sensorial correspondiente (entendiendo por *espacio sensorial* el que comprende todos los estímulos posibles, no necesariamente el espacio real; por ejemplo, un espacio sensorial podría estar compuesto por el espacio de todos los vectores tridimensionales que representan un color), y además, zonas cercanas en el espacio sensorial excitan a grupos de neuronas cercanas en el cerebro.

Por ejemplo, todo el sistema sensorial táctil del hombre tiene una correspondencia en el denominado *homúnculo* del cerebro, en el cual diferentes partes del cuerpo tienen su reflejo. La representación de estas zonas se denomina *mapa somatotópico*. También existen *mapas retinotópicos*, que son zonas del cerebro que responden a zonas del espacio con coordenadas (relativas al ojo) similares, y *mapas fonotópicos* correspondientes a diferentes frecuencias auditivas.

Y para lograr un mapa topográfico del espacio de entrada como el anterior hacen falta dos operaciones: por un lado la *reducción de dimensionalidad* de los estímulos de entrada, o proyección en un subespacio de dimensión inferior; y además la *conservación del orden local*, de forma que estímulos similares se apliquen en zonas cercanas en el espacio de salida (el subespacio de dimensión inferior mencionado anteriormente).

### 2.2.2 Reducción de dimensionalidad en reconocimiento de patrones.

La reducción de dimensionalidad consiste en una proyección de estímulos (representados por vectores de muchas dimensiones, como puede ser el vector resultante de una transformación al dominio de las frecuencias de una señal de audio), como el sonido, en el cerebro. El cerebro es aproximadamente bidimensional (aunque en realidad tiene probablemente una dimensión superior a 2, entre 2 y 3), y por lo tanto la percepción produce una proyección a un espacio de dimensión 2, es decir, una *reducción de dimensionalidad* de la señal, al almacenarla o reconocerla en el cerebro.

Esta reducción de dimensionalidad es típica también de los algoritmos de clasificación o de compresión [Duda73][Tou73], que habitualmente lo logran mediante un análisis de componentes principales. Para una señal de dimensión *D,* el análisis de componentes principales o análisis factorial (también llamado *transformación de Karhunen-Loève*) halla las dimensiones del espacio vectorial que representan la máxima varianza, y reduce la señal a

una combinación lineal de *C* vectores, donde *C* < *L*, proyectando efectivamente la señal a un subespacio de dimensión inferior. Otras veces, se halla simplemente una combinación (lineal o no) de las componentes de la señal de entrada; lo cual puede provocar una pérdida de información. Los algoritmos de cuantización vectorial consiguen también esta reducción de dimensionalidad, como ya se ha visto en el capítulo anterior, con una posible pérdida de calidad de la señal, aunque la pérdida de calidad se minimiza si se diseña un diccionario con distorsión mínima.

Las proyecciones a espacios de dimensión inferior suponen que, para que no ocurra ninguna pérdida de calidad de la información, los datos de entrada están situados en una hipersuperficie de dimensión *C*; por ello, no dan soluciones adecuadas para puntos del espacio de entrada situados fuera de esa hipersuperficie. Las transformaciones de proyección son esencialmente lineales; por tanto, algoritmos neuronales, sin necesidad de tener en cuenta ninguna de estas limitaciones, podrán llevar a cabo mucho mejor esta reducción de dimensionalidad. De hecho, en experiencias realizadas [Sau89] con la capa oculta o intermedia del perceptrón multicapa entrenado con retropropagación, se observa una reducción de dimensionalidad análoga a la obtenida con otro tipo de transformaciones, como la de Karhunen-Loève.

La reducción de dimensionalidad descrita se denomina a veces *extracción de características*, o *feature-extraction*, pues extrae las características más sobresalientes de los datos de entrada, que serían los componentes con la máxima varianza (las componentes de un vector que variara poco a lo largo de la muestra no serviría para diferenciar las muestras pertenecientes a una clase de otras muestras). Por eso a veces al mapa autoorganizativo de Kohonen se le denomina *Self-Organizing Feature Map*, SOFM.

Por todo lo dicho anteriormente, la reducción de dimensionalidad se puede llevar a cabo simplemente mediante un proceso de cuantización vectorial.

### 2.2.3 Ordenación de la respuesta a los estímulos.

Para explicar la segunda operación, ordenación de la respuesta a los estímulos en el cerebro, von der Malsburg propuso en 1973 un modelo de autoorganización del cortex visual del cual resultaba una creación de dominios visuales [Mal73]. En este trabajo se presenta un

modelo con una retina de entrada de forma hexagonal, y una *zona cerebral* de salida también de forma hexagonal. En el córtex hay tanto conexioneos excitatorias como inhibitorias. Mediante un proceso de autoorganización similar al hebbiano, en el cual neuronas que se activan simultáneamente ven su conexión reforzada, von der Malsburg prueba que se forman en el *córtex* de salida zonas que responden a la orientación de objetos en el espacio de entrada.

Kohonen se basó en este modelo, aunque simplificándolo mucho (en lo que él llamó *algoritmo atajo*, "*shortcut algorithm*"), y lo generalizó para cualquier tipo de mapa (somatotópico, tonotópico) [Koh82a], de una señal en un espacio de entrada con una dimensión determinada (n= 1, 2, 3,...), a una salida con una dimensión 1 o 2 (a veces 3, pero no es tan común) [Mer91]. En realidad, según [Ack90], la autoorganización es una consecuencia directa de este algoritmo atajo; y no se deduce de las ecuaciones diferenciales presentadas en [Koh89a]. En cualquier caso, el algoritmo logra su cometido, aunque su funcionamiento explícito no esté en demasiada relación con los mecanismos cerebrales en los que se basa.

Para llevar a cabo la ordenación del espacio de salida en dominios sensibles a zonas contiguas, se necesita una noción de "contigüidad" entre los vectores diccionario, de la que carecen los algoritmos clásicos de cuantización vectorial, en los que, en principio, no tiene que haber ninguna relación entre vectores contiguos en el diccionario. Debido a esta vecindad definida cambian, para responder a una zona del espacio de entrada, los vectores del diccionario ganadores, y además, los contiguos, de forma que zonas diferentes del mapa (del diccionario) responderán a zonas diferentes del espacio de entrada, y zonas contiguas del mapa responderán a zonas contiguas del espacio de entrada.

## 2.3 Mapas autoorganizativos de Kohonen: SOM

### 2.3.1 Introducción

Una *red de Kohonen* o *mapa autoorganizativo de Kohonen* (SOM) consiste habitualmente en un plano (como se muestra en la Figura 2.1) en el cual se sitúan una serie de

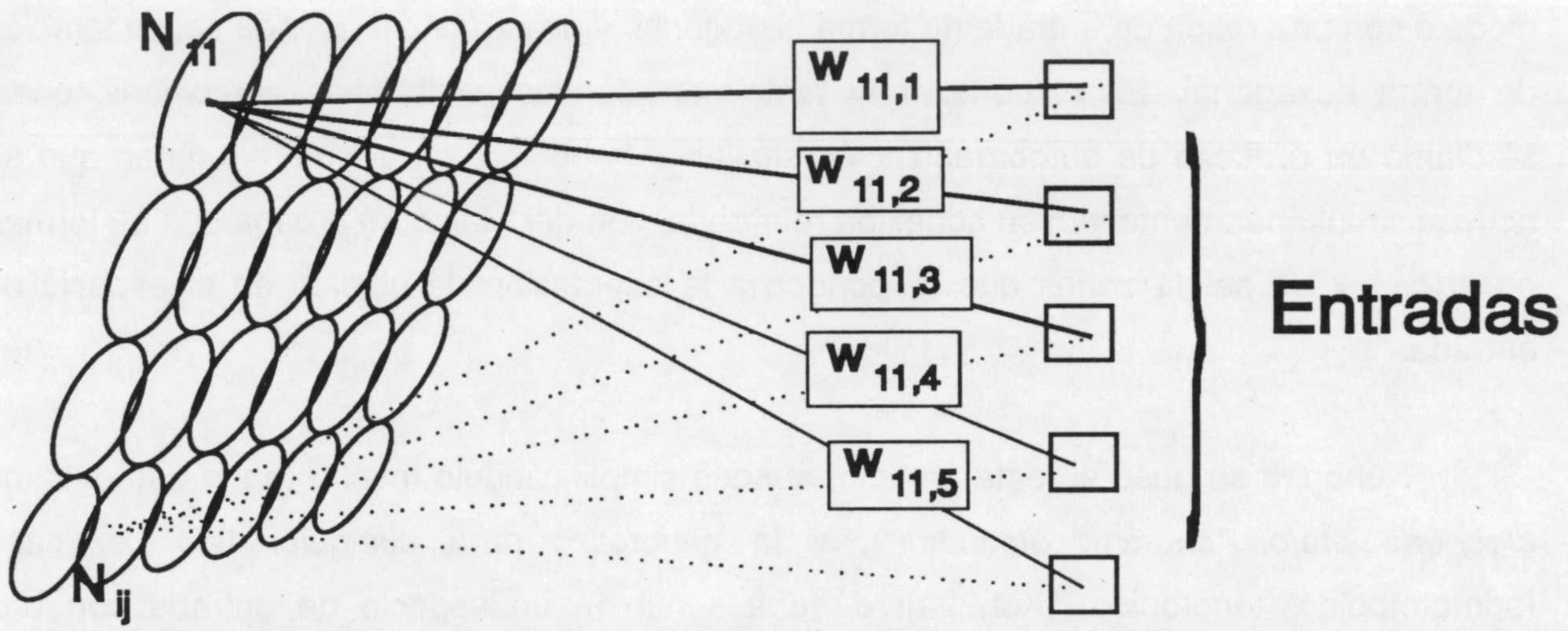



**Figura 2.1** Mapa de Kohonen con 5 entradas (mostradas a la derecha de la imagen) y 6 x 4 salidas; sólo se ilustran los pesos de 2 neuronas (izquierda).

nodos o *neuronas* ($N_{jk}$). Cada neurona está unida por un *peso* $w_{jk.i}$ (como se muestra también en la figura) a cada una de las entradas $x_i$, y además hay una conexión implícita inhibitoria entre cada neurona y las que le rodean. En realidad, es un simple cambio de terminología con respecto a los algoritmos de cuantización vectorial. A cada vector del diccionario se le asignan dos subíndices, *j* y *k*, para hacer explícitas las relaciones de vecindad con los otros vectores diccionario, y se le denomina *vector peso*. Cada neurona será en realidad el vector peso o, dicho de otra forma, el vector código del diccionario.

Por lo tanto, la red de Kohonen es una red monocapa (dado que la salida de las neuronas -que podría asociarse a la distancia del vector peso al patrón de entrada- no sirve como entrada a otras neuronas), con tantas entradas como dimensiones tenga el espacio de entrada. Esta red da como salida la *neurona ganadora,* es decir, aquella neurona para la cual la distancia al patrón de entrada sea mínima.

*Entrenar* esta red consiste en un proceso de modificación de pesos, como sucede en todas las redes neuronales, lo que en este caso se puede designar también como un proceso iterativo de adaptación o diseño del diccionario. El conjunto de vectores de entrenamiento se va presentando continuamente en las entradas de la red (lo cual significa simplemente escoger un vector muestra del conjunto de entrenamiento). Se halla la distancia de estas entradas a los vectores peso, y se elige la neurona cuyo vector peso está a una distancia mínima del vector de entrada. Este vector peso se hará más cercano al vector de entrada. Esta es la idea básica que subyace en todos los algoritmos de Kohonen, inclusive el mapa

A veces se plantea la duda de la inclusión o no de los mapas autoorganizativos dentro de los algoritmos llamados redes neuronales artificiales, dado que la terminología de pesos utilizada anteriormente se puede sustituir simplemente por nociones relativas a vectores de referencia, sin necesidad de hablar de mecanismos neuronales. En realidad, esta inclusión se debe al planteamiento básico del algoritmo, en el cual es necesaria una activación-inhibición lateral de cada neurona a las vecinas con el objeto de hallar ese vector de entrada *más cercano* (paso S2.2) [Koh92]; además, es necesaria alguna forma de interacción implícita para que el cambio de los pesos se propague hasta las neuronas vecinas (paso S2.3). Por último, no hay que olvidar que el mapa autoorganizativo es un modelo de inspiración neurofisiológica, lo cual justifica, finalmente, su inclusión.

### 2.3.3 Comentarios al algoritmo SOM

La gran cantidad de parámetros y funciones presentes en el algoritmo anterior hace difícil elegir unos valores o formas determinadas para las funciones, y que además sean adecuados por el problema. Por ello, diversos autores han propuesto distintas soluciones a este problema, apoyándose en la experiencia. En esta memoria aportamos además nuestra propia experiencia sobre la búsqueda de unos valores óptimos para ciertos parámetros y funciones del SOM [Mer94]. Este método sistemático es similar a otros hallados en la literatura [Sie93], [Rit91].

Para estudiar los diferentes aspectos de los SOM, se ha realizado un programa, KOH, cuyas salidas en pantalla se muestran en este capítulo. Este programa permite variar, de una forma cómoda para el usuario, diversos parámetros, como el número de bits dedicados a los pesos, el número de neuronas del SOM, y el número de iteraciones que se utilizan para el entrenamiento. El programa está realizado integramente en C++ (véase capítulo 6) y comparte elementos comunes con los otros programas mencionados en esta tesis, en el espíritu del C++.

*A. Tamaño de la red*

La mayoría de las veces, el tamaño *N* de la red es fijo y establecido por el usuario, de forma que suele ser un dato del problema, y no un resultado de un ajuste para conseguir un buen valor.

Habitualmente, un mayor número de vectores peso significa una menor distorsión, puesto que se cubre mejor el espacio de entrada. Evidentemente, también significa mucho mayor tiempo de cálculo, pues el tiempo de cálculo que el algoritmo necesita crece linealmente con el número de vectores peso añadidos, ya que se necesita calcular la distancia a todos los vectores peso cada vez que se introduce un vector muestra.

Normalmente el número de vectores peso viene dado por el máximo que puede contener la memoria del ordenador, en el caso de un ordenador compatible PC, utilizando el sistema operativo MS-DOS. Si la memoria no es problema, como en las máquinas UNIX, el tamaño de la red viene determinado por el tiempo que se desea dedicar al entrenamiento del mapa.

El lado del SOM, *M,* es habitualmente mayor que seis. Debido a que las vecindades no se comportan de la misma forma cerca de los bordes, es necesario tener un mapa suficientemente grande para que estos efectos de frontera no perjudiquen la eficacia en la clasificación del mismo. Estos efectos de bordes, advertidos ya en [Koh82a], se deben a que el tamaño de la vecindad en los bordes de los mapas no es el mismo que en el centro.

*B. Inicialización*

Para ***inicializar*** la matriz de pesos, algunas veces es suficiente con asignar a los pesos pequeños valores aleatorios. Esto se suele hacer cuando no se conoce de antemano la distribución de los vectores de entrada; los primeros pasos del algoritmo se encargarán de situar los pesos en los valores adecuados. En ese caso, el paso S1 se convertiría en

Algoritmo SOM -- S1

[S1] $\forall j,k\ 1 \le j,k \le M,\ \forall i\ 1 \le i \le D \quad w_{jk,i} = random(100)/100.0$

que inicializaría todos los pesos con valores situados entre 0 y 0.1 (*random* genera un número aleatorio entre 0 y 100) [Koh84].

A veces lo que se trata es de obtener un conjunto de vectores diccionario que cubran de la mejor forma posible una figura regular, dentro de la cual se supone que los puntos siguen una distribucion uniforme (es decir, la cantidad de puntos en cualquier zona de la figura es siempre aproximadamente la misma). Este tipo de recubrimientos, a veces denominados *teselaciones*, se utilizan sobre todo por su facilidad de evaluación visual (es fácil de ver si los vectores diccionario se distribuyen regularmente sobre la superficie de un cuadrado, por ejemplo).

En este caso, una vía alternativa de inicializar los pesos sugerida por algunos autores [Tat89] consiste en hacer crecer los valores de los pesos con los índices *j, k*, lo cual garantiza de principio una buena aproximación a los valores definitivos de los vectores del diccionario. En este caso,

Algoritmo SOM -- S1

[S1] $\forall j,k \ 1 \le j,k \le M, \ \forall i, 1 \le i \le D \quad w_{jk,i} = k_{k1} * j + k_{k2} * k$

donde $k_{k1}$ y $k_{k2}$ son constantes que dependen de la componente del vector peso, *i*. Con esta fórmula se pueden hacer distribuciones que crezcan o decrezcan con cada dimensión del mapa.

También puede ser conveniente inicializar los pesos con vectores obtenidos aleatoriamente dentro del conjunto de muestra [Kon91]; dado que la clasificación es del tipo *vecino más cercano*, el utilizar vectores ya pertenecientes a la muestra evita que los vectores iniciales queden muy lejos de los vectores de entrada. Este es un método habitual en cuantización vectorial clásica [Tou73]. En este caso

Algoritmo SOM -- S1

[S1] $\forall j,k \ 1 \le j,k \le M, \ \vec{w}_{jk} = \vec{x} \in \mathcal{E}$

*C. Número de iteraciones*

El ***número de iteraciones*** se establece empíricamente, en función de los resultados que se quieran obtener (es decir, del tiempo de CPU que se quiera dedicar o del grado de exactitud que se quiera alcanzar). En general, y como regla heurística, se tiene que presentar todo el conjunto de patrones (vectores) de entrenamiento $\mathcal{E}$ al menos 10 veces, y hasta 20

veces. A cada iteración se le denomina *época*. Kohonen afirma [Koh92] que el número de pasos (es decir, el número de veces que se deben de presentar patrones) debe ser aproximadamente 500 veces el número de unidades de la red, es decir, $M \times M$; es decir, si llamamos $e$ al número de épocas, y $\#\mathcal{E}$ al tamaño de la muestra,

$$e = \frac{500M^2}{\#\mathcal{E}}$$

lo cual da 50 iteraciones para una muestra de tamaño 1000 y 10 neuronas; un poco más que lo que en la presente investigación se ha observado empíricamente.

Por otro lado, hay diversas formas de presentar el conjunto completo de muestras. Los resultados del algoritmo son fuertemente dependientes del orden de presentación de las muestras, y por tanto, cuando las muestras están ordenadas por clases, es conveniente presentarlas en orden aleatorio, calculando éste previamente mediante un algoritmo de *shuffling* (es decir, un algoritmo que baraja el conjunto de entrada, de forma que las muestras se presentan en un orden aleatorio).

La predeterminación del número de iteraciones hace que el SOM difiera fuertemente de los algoritmos de cuantización vectorial, donde la condición de terminación está relacionada con el error de clasificación; esto no está previsto en el SOM. Para tener una condición de terminación equivalente, cabría redefinir el paso [S2] de la forma siguiente

**Algoritmo SOM -- S2**

[S2] Repetir hasta que $J = \sum_{\vec{x} \in S} d(\vec{x}, \vec{w}_{bm})$ es mínima o no varía en un número predeterminado de iteraciones.

Esto, sin embargo, no siempre es posible, ya que significa un rediseño completo del algoritmo. El parámetro de ganancia, $\alpha$, del mapa autoorganizativo depende de la iteración y del número de iteraciones total, determinado de antemano; por tanto, es esencial establecer inicialmente el número de iteraciones para calcular $\alpha$. Para no hacer intervenir el tiempo total de entrenamiento, habría que diseñar una fórmula de cambio de $\alpha$ en la que sólo interviniera $J$. Este parámetro $J$, habitualmente denominado *distorsión*, como hemos visto anteriormente, puede servir sin embargo para evaluar las prestaciones del algoritmo [Mer94].

*D. Distancia*

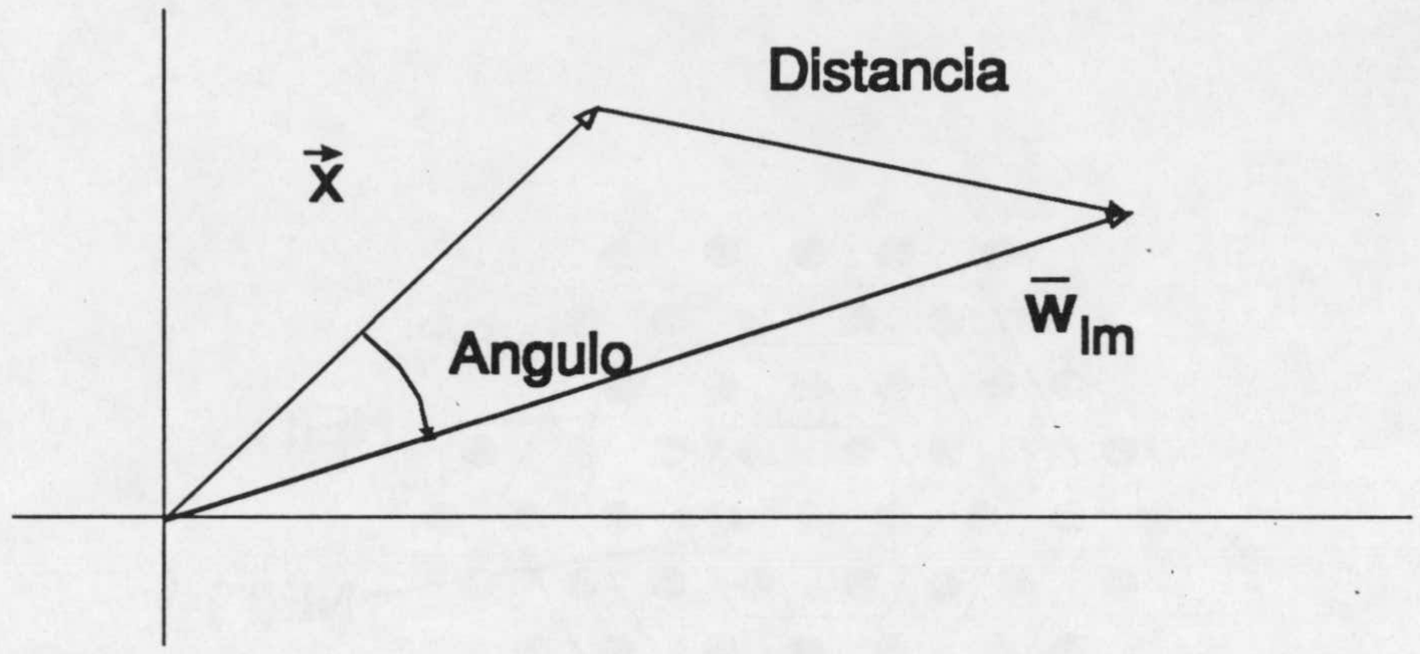



**Figura 2.2** Distancias entre el vector muestra y el peso: ángulo y la distancia euclídea (módulo del vector indicado con "distancia")

Dependiendo de la aplicación, se utilizan diferentes nociones de ***distancia y proximidad*** entre vectores $\vec{x}$ de entrada y vectores peso $\vec{w}_{jk}$ para calcular la neurona ganadora (Figura 2.2).

Los dos tipos de distancia más utilizados son la distancia euclídea y el ángulo euclídeo formado entre la muestra y el vector peso. Este último es menos habitual; y se suele utilizar en el caso de que tanto las muestras como los pesos estén normalizados, lo cual hace que se pierda información, al eliminar una dimensión.

Si se considera $\|$ una distancia arbitraria, la neurona con salida máxima (la que *responde*, o neurona *ganadora*) es tal que

$$\| \vec{x}(t) - \vec{w}_{lm}(t) \| = \min_{jk} \| \vec{x}(t) - \vec{w}_{jk}(t) \|$$

Se puede utilizar cualquier otro tipo de distancia, y de hecho a veces se utiliza la distancia de valor absoluto o *distancia de Manhattan*, que es mucho más rápida computacionalmente que la distancia euclídea. Kohonen [Koh89b] menciona otras distancias, como la *distancia de Mahalanobis*, que tiene en cuenta las matrices de covarianza de las diferentes clases a clasificar. También, por supuesto, se puede utilizar la *distancia generalizada de Minkowski* [Koh92], que equivaldría a una normalización, en caso de que las magnitudes de las diferencias entre las diferentes componentes del vector fueran muy grandes.

*E. Vecindad*

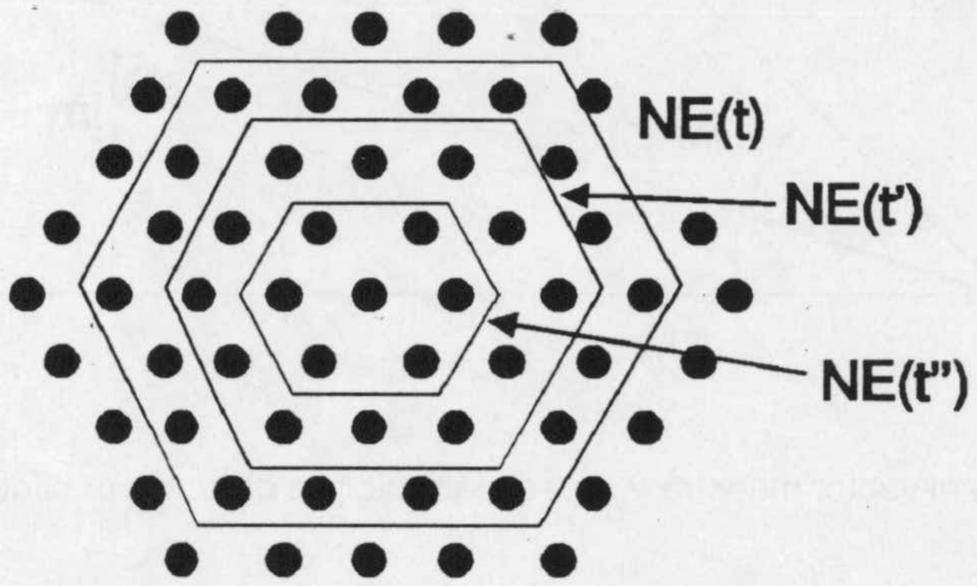



**Figura 2.3** Figura que muestra como decrece la vecindad con el tiempo, para una topología hexagonal. t < t' < t"

La ***vecindad*** $NE_m(t)$ depende evidentemente de la topología de la red, siendo además dependiente del tiempo, en este caso decreciente. Este caso es el más habitual, con el objeto de que la red converja (como se ve en la Figura 2.3).

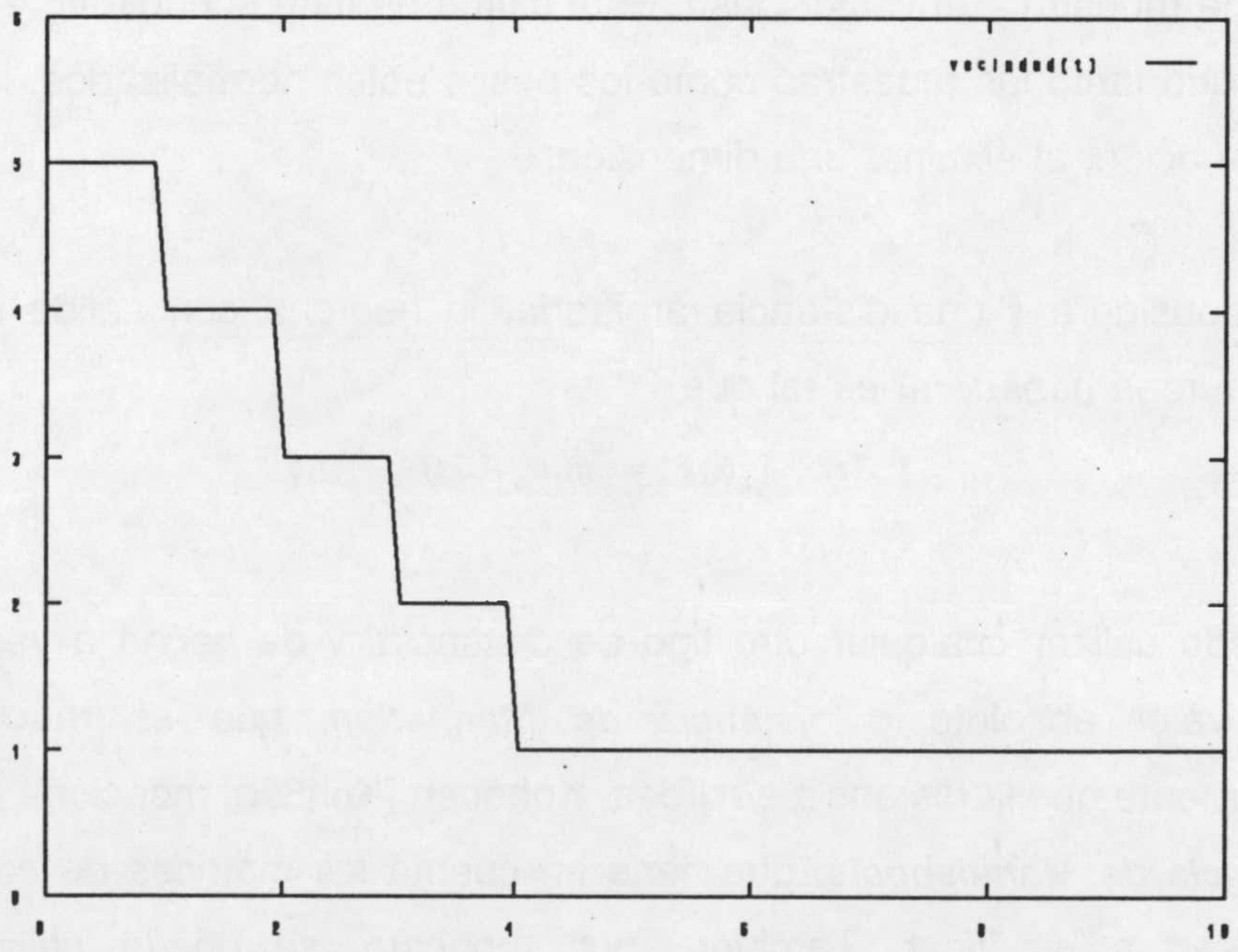



**Figura 2.4** Función de decrecimiento del radio de la vecindad (en ordenadas) con respecto al tiempo (medido en épocas).

Si la vecindad es demasiado grande, la red nunca convergerá (los vectores peso se agitarán como trigales en el viento, sin tomar ninguna dirección definitiva). Si la vecindad es

demasiado pequeña, no se llevará a cabo la autoorganización (ya que no habrá efectos globales sobre todas las neuronas, sino sólo locales). Por ello se suele hablar de dos fases en el proceso de entrenamiento: la fase de *sintonización* (inicial) y la fase de *autoorganización*.

Durante la fase de sintonización la vecindad varía desde aproximadamente la mitad del tamaño de la red (con lo cual, sea cual sea la neurona que gane, se entrenan casi todas), hasta r=1 (como se ve en la Figura 2.4); en este caso la fase de sintonización llegaría hasta la época -presentación completa de toda la muestra- 4; con lo cual se entrenan solamente las neuronas de alrededor de la ganadora, las vecinas inmediatas. Este decrecimiento progresivo de la vecindad evita que se produzcan efectos de mosaico (es decir, diversas zonas del mapa responden a diversas zonas del espacio de entrada, pero sin tener ninguna relación entre sí). La fase de sintonización causa que los vectores pesos de las neuronas tomen valores en el rango del espacio de entrada, y que, de esta forma, todas las neuronas sean capaces de ganar ante alguna muestra. Durante la fase de autoorganización, la competencia entre diferentes neuronas por responder al espacio de entrada provoca la ordenación que se pretende.

La topología de la vecindad también puede influir en las prestaciones. Kohonen utiliza en casi todos sus trabajos una topología hexagonal, como se ve en la Figura 2.2. Esto se debe a que, como demuestra Somhoul et al. [Som73], con esta topología se logra menos distorsión para una distribución estadística uniforme en un cuadrado de lado unidad, que si los vectores del diccionario se distribuyen según cuadrados (es decir, la forma de la celda básica es un cuadrado), como se suele hacer habitualmente. Sin embargo, esta distribución es mucho más fácil de programar, y es la que se encuentra habitualmente en las implementaciones del mapa autoorganizativo. El programa hecho por el equipo de Kohonen permite ambas [Koh92].

Kangas [Kan90] presenta otra serie de alternativas a la vecindad, como los MST (*minimal spanning trees*). Para ello, se crea un árbol de envergadura mínima (MST) en el espacio de entrada, uniendo cada vector peso con los más cercanos en el espacio de entrada. Con este añadido consiguen mejorar algo los resultados (haciendo el entrenamiento más rápido, y pudiéndose utilizar menos neuronas), al coste de complicar enormente la programación del algoritmo y aumentar su tiempo de cálculo. Esta variante se basa en que

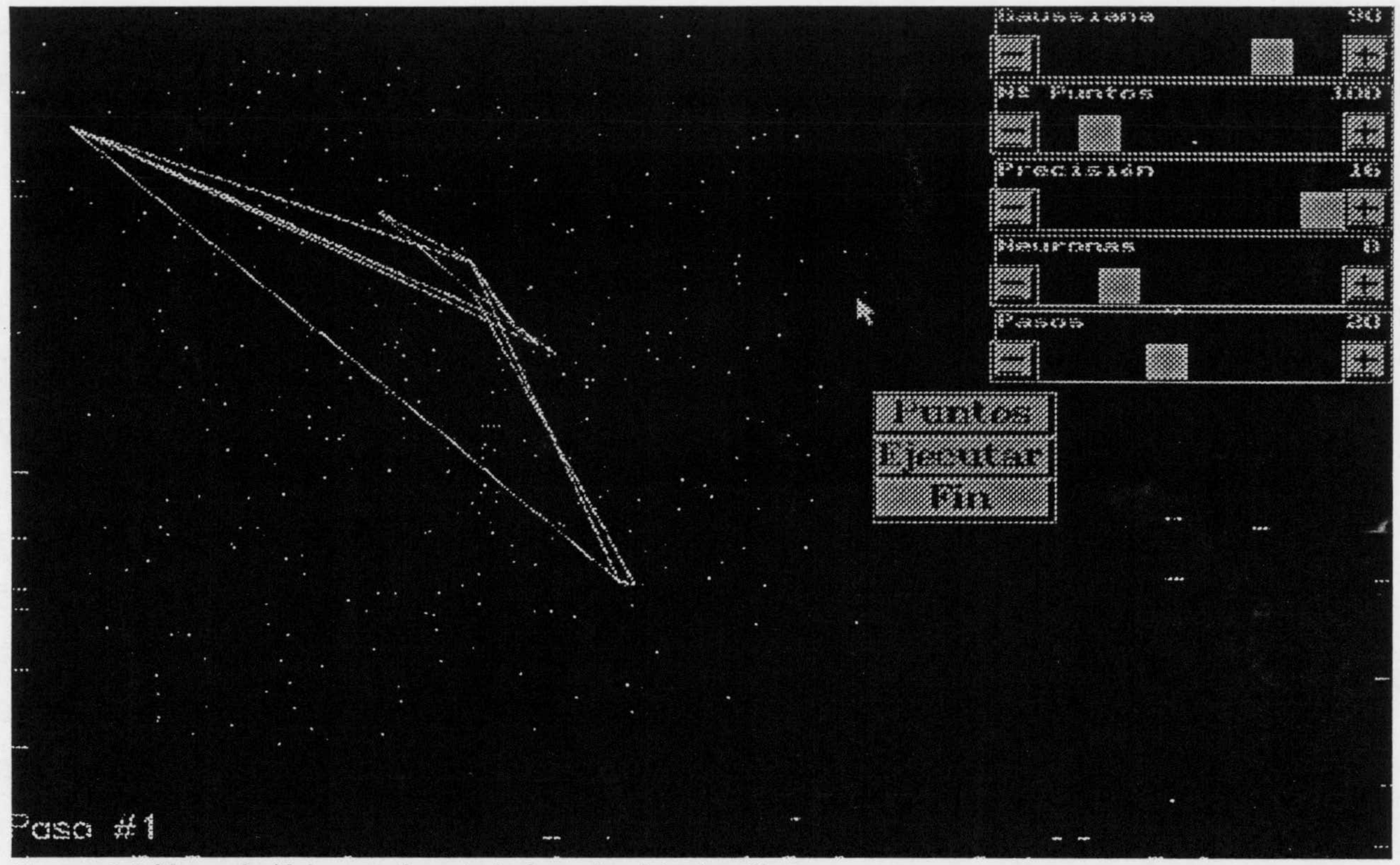



**Figura 2.5** Mapa de Kohonen durante la fase de sintonización.

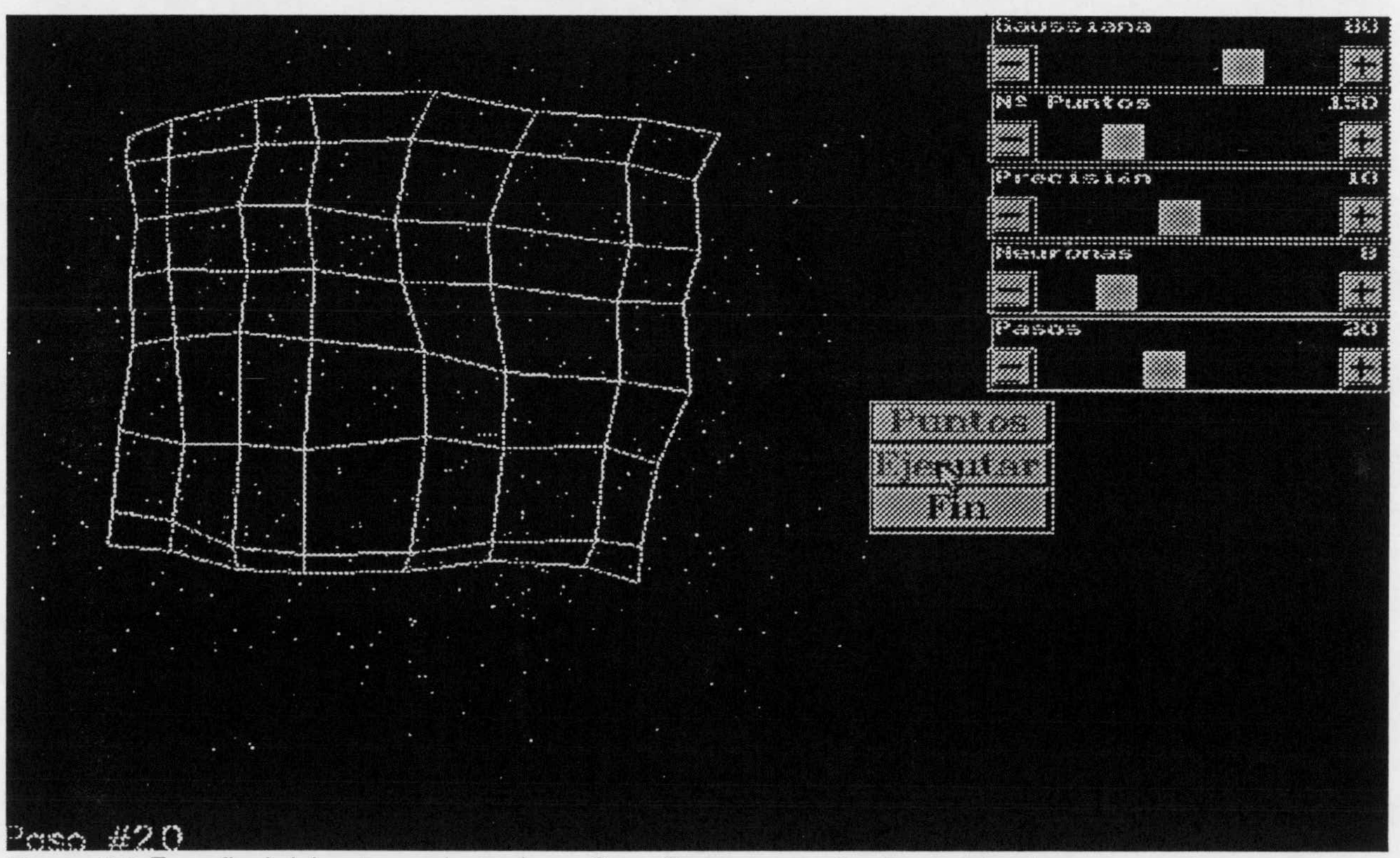



**Figura 2.6** Fase final del entrenamiento de un Som. Programa realizado por el autor.

la vecindad entre neuronas se considera sobre el espacio de entrada, y no sobre el espacio de salida (los índices j,k). Un concepto similar es utilizado en el *gas neuronal* de Martinetz

[Mart93].

En los trabajos en los que el número de neuronas no es fijo [Frit91,93], la topología varía durante el entrenamiento, pudiendo estar cada neurona rodeada de otras neuronas en número variable. En otras implementaciones, y en las iniciales de Kohonen, los autores han utilizado una función llamada núcleo o *kernel,* en la cual el cambio de los pesos depende de la distancia a la neurona ganadora. Es decir,

$$\Delta\vec{w}_{jk} = \alpha(t)\,\zeta(t,d)\,(\vec{x}-\vec{w}_{jk})$$

donde $d = d(\vec{w}_{lm},\vec{w}_{jk})$.

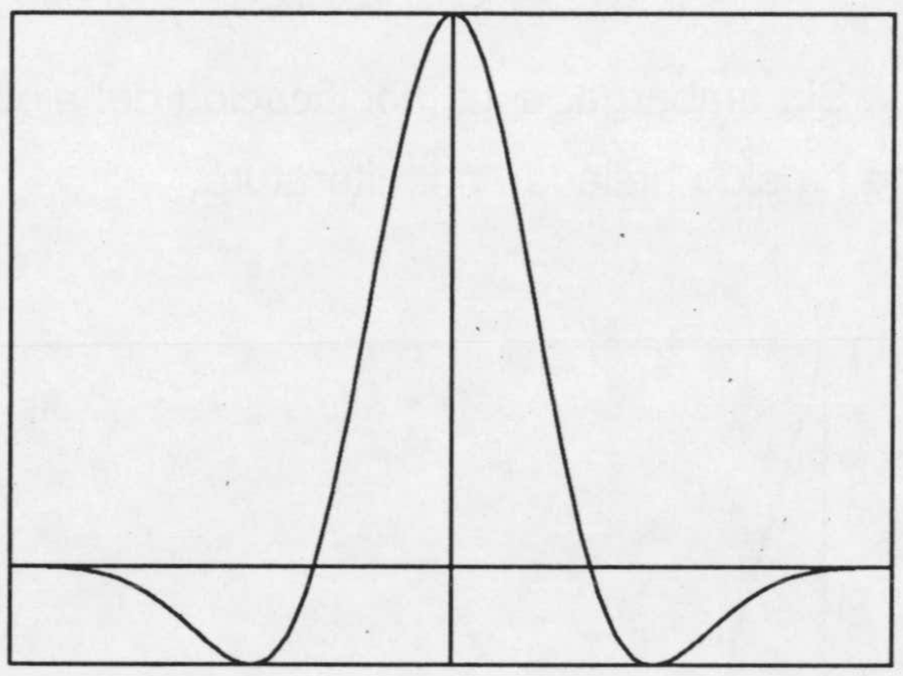


**Figura 2.7** Función sombrero mexicano.

En el SOM básico, se supone una función tipo *sombrero mexicano*, de tal forma que es excitatoria ( es decir, que el cambio de los pesos es en el mismo sentido que el del peso ganador) a corta distancia e inhibitoria a larga distancia, como aparece en la Figura 2.7. Una vez más, esta función no aparece explícitamente en el algoritmo SOM, ya que ha sido estilizada, de forma que vale 1 para la vecindad y 0 fuera.

Luttrell [Lut89] afirma que la existencia de una vecindad se puede deducir a partir de algoritmos jerárquicos o multietapa de cuantización vectorial, en los cuales existen varios niveles de diccionario. Las muestras se utilizan para entrenar el diccionario del primer nivel; los vectores de referencia obtenidos se utilizan para entrenar un diccionario de segundo nivel, y así sucesivamente; con este tipo de algoritmos se lograría también una ordenación análoga a la lograda postulando la función de vecindad.

*F. Modificación de pesos*

Para alcanzar la convergencia, la amplitud de la ***modificación de pesos*** debe disminuir con el tiempo, ya que el número de iteraciones se establece de antemano. Por esto, el vector diferencia entre la entrada y el vector peso se ponderan por una cantidad $\alpha$, similar al término de *ganancia* que se encuentra en otros algoritmos de aprendizaje, que depende del tiempo y decrece con él.

$$\vec{w}_{jk}(t+1) = \vec{w}_{jk}(t)+\alpha(t)[\vec{x}(t)-\vec{w}_{jk}(t)]$$

Una mejora propuesta en la presente memoria es hacer depender $\alpha$ de la distorsión *J*, de forma que

$$\vec{w}_{jk}(t+1) = \vec{w}_{jk}(t)+\alpha(J)[\vec{x}(t)-\vec{w}_{jk}(t)]$$

siendo $\alpha$ creciente con *J*. Sin embargo, esta modificación del algoritmo de Kohonen, aunque propuesto en [Mer91], no ha sido hallado en la literatura.

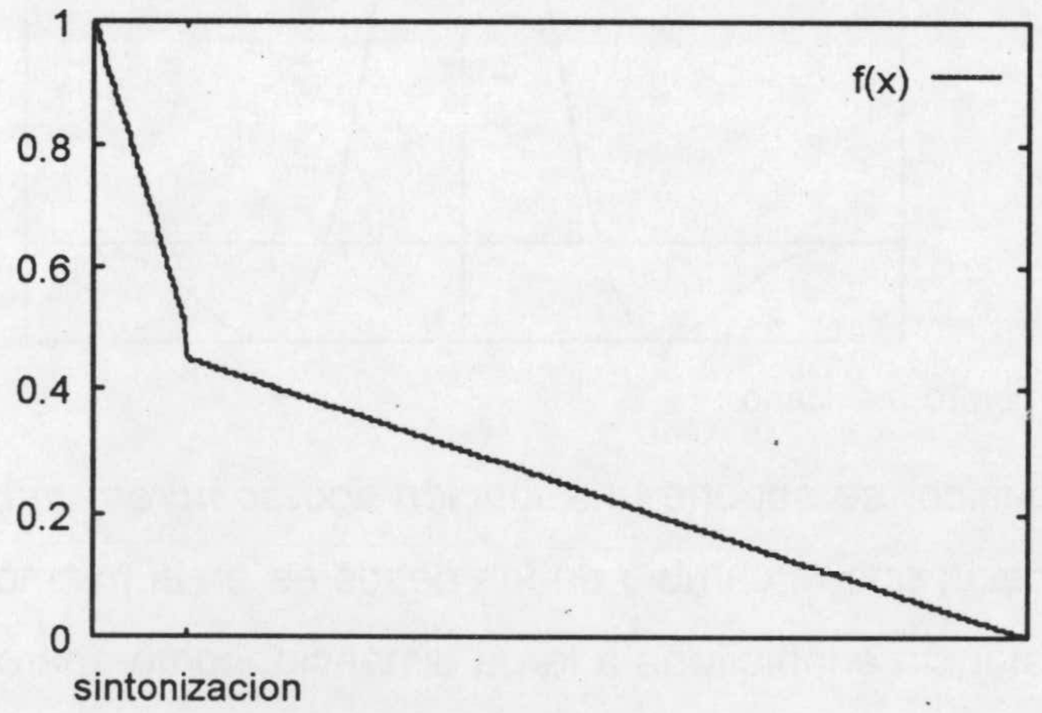



**Figura 2.8** Muestra de la evolución del valor de $\alpha(t)$ con el tiempo.

En la presente memoria se ha comprobado que se obtienen buenos resultados si se le da a $\alpha$ un valor pequeño, invariable durante todo el entrenamiento; aunque quizás lo más razonable es hacer variar la función de ganancia de forma diferente durante las dos fases del SOM: sintonización y autoorganización, de la forma siguiente

$$\alpha(t)=\begin{cases} 1-k_1\dfrac{t}{t_s} & t<t_s \\ k_2-k_3\dfrac{t}{t_{max}} & t_s<t<t_{max} \end{cases}$$

donde $t_s$ es la última iteración de sintonización, y $k_2 = \alpha(t_s)$, como aparece en la Figura 2.8.

## 2.4 Aprendizaje competitivo

En su definición más general, se denomina *aprendizaje competitivo* a aquél algoritmo neuronal en el cual, en cada iteración, sólo aprende la neurona o el grupo de neuronas que, de alguna forma, se acerca más al estímulo o patrón de entrada. La denominación de competitivo viene de que cada neurona compite con todas las demás (o con un grupo de ellas) para reforzar sus pesos, en oposición a los algoritmos *cooperativos*, como el perceptrón multicapa, en el cual cambian todos los pesos de la red cada vez que se presenta un patrón.

En la acepción utilizada en [Rum85][Prie90], el aprendizaje competitivo es una red

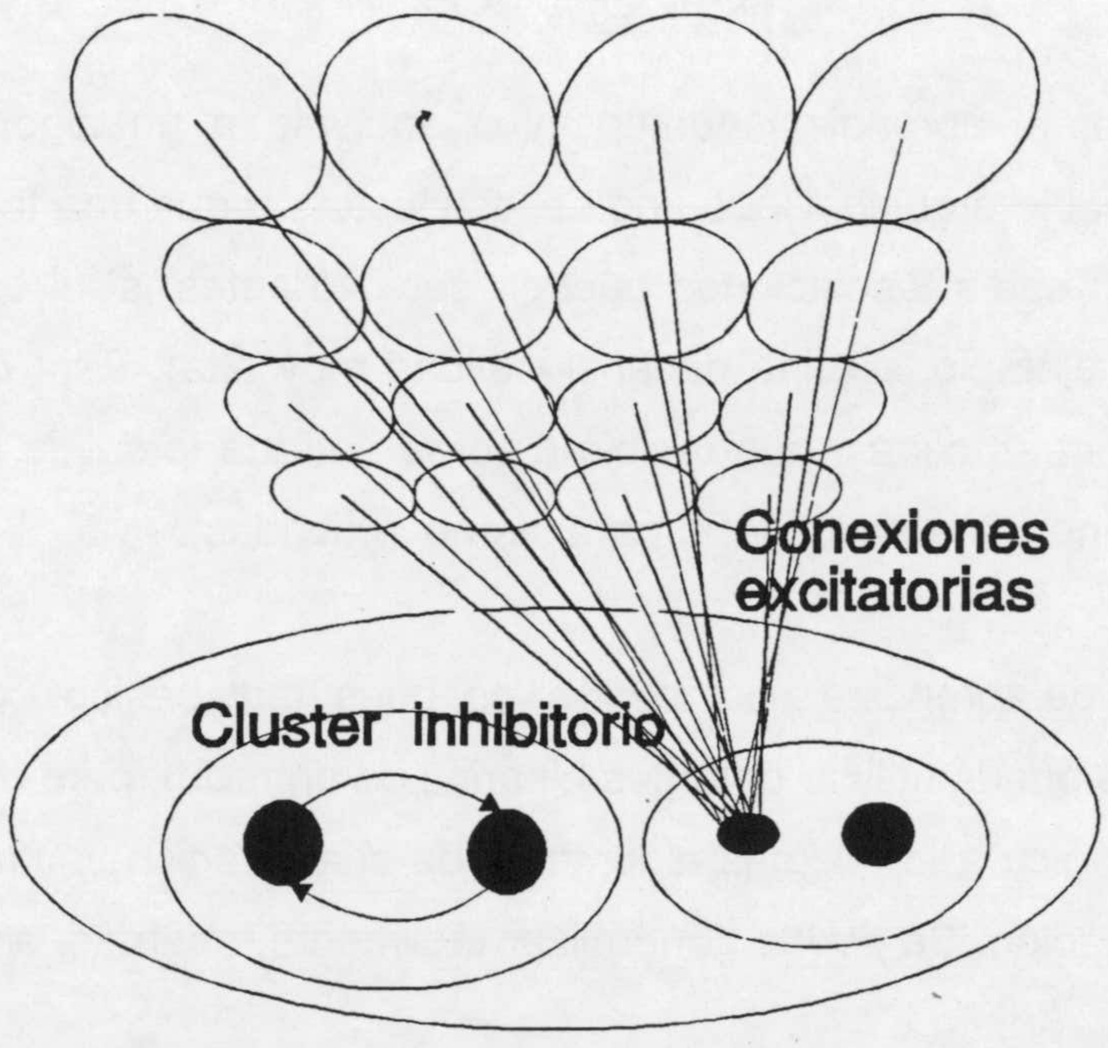



**Figura 2.9** Esquema de una raed de apreneizaje competitivo con 2 clusters y dos neuronas por clusters; y 16 entradas.

neuronal con la arquitectura indicada en la Figura 2.9. Es decir, consiste en una capa de entrada con una serie de células cuyo valor puede ser 0 o 1; no se admiten, en la versión original, valores analógicos. Cada una de estas células está conectada a todas las neuronas de la siguiente capa. Las neuronas de cada capa (excepto la de entrada) están organizadas en *clusters*, de forma que cada neurona de un cluster tiene conexiones inhibitorias con todas las demás neuronas del cluster.

El aprendizaje competitivo sigue las siguientes reglas:

**1.** En cada capa, las unidades se agrupan en una serie de clusters cuya intersección es nula; cada unidad de cada cluster inhibe a las otras unidades dentro del cluster, de forma que cada cluster funciona como *winner-take-all*, la unidad que recibe la entrada mayor logra su valor máximo, mientras que a las demás unidades se les asigna su valor mínimo.

**2.** Cada elemento de cada cluster recibe entradas de las mismas neuronas de la capa de entrada.

**3.** Una unidad aprende si y solo si gana la competición con las otras unidades en su cluster.

**4.** Los estímulos son patrones binarios.

**5.** Cada unidad tiene una cantidad fija de pesos, que se distribuye entre sus entradas; cada unidad aprende transfiriendo peso de sus líneas activas a sus líneas inactivas, es decir,

$$\Delta w_{ij} = \begin{cases} 0 & si\ j\ pierde\ para\ k \\ g\frac{c_{ik}}{n_k} - g w_{ij} & si\ j\ gana\ para\ k \end{cases}$$

En resumen, el aprendizaje competitivo equivale a un algoritmo de clustering o cuantización vectorial jerárquico. Cada uno de los clusters forma una teselación completa del espacio de entrada; estas teselaciones pueden ser diferentes, si el espacio de entrada es muy complejo (por ejemplo, de una dimensionalidad muy alta). Esta complejidad se puede resolver añadiendo otras capas, que recibirán como entrada la salida de las capas iniciales (es decir, 1 para la neurona activada, 0 para las no activadas).

El algoritmo de aprendizaje competitivo es, pues, muy básico, con pocas aplicaciones prácticas (pues sólo puede utilizar patrones binarios de entrada), pero que puede servir como introducción pedagógica a los demás algoritmos de cuantización vectorial, como los que se describen a continuación. Se puede generalizar fácilmente, además, a entradas analógicas.

## 2.5 Algoritmo LVQ

Se puede considerar que los algoritmos LVQ 1, 2 y 3 son versiones supervisadas del mapa autoorganizativo, en los que además la vecindad es nula, por lo que es estrictamente un algoritmo de cuantización vectorial. Como ya se ha visto, el SOM es un algoritmo no supervisado, que sirve para aproximar [Koh90] una serie de vectores diccionario al espacio de entrada. Si el espacio de entrada está dividido en varias clases, habrá un grupo de

vectores diccionario asignado a cada clase.

Si se quiere utilizar un SOM como procedimiento de reconocimiento, habrá que asignar una etiqueta a cada vector diccionario (o peso), de forma que, cuando un vector de entrada $\vec{x}$ quede cerca de un vector diccionario $\vec{w}_{jk}$, se puede afirmar que $\vec{x}$ pertenece a la clase con la cual está etiquetado $\vec{w}_{jk}$. Este etiquetado se lleva a cabo de la forma siguiente

**Algoritmo calibración SOM**

[E1] $\forall \vec{x} \in \mathcal{E}$, calcular $\vec{w}_{lm}$ más cercano a $\vec{x}$.

[E2] Si $\vec{x}$ pertenece a la clase C, se etiqueta $\vec{w}_{lm}$ como perteneciente a la clase C.

En este proceso se pueden presentar una serie de problemas

*i.* Puede haber vectores peso que ganen para varios vectores de entrada pertenecientes a clases diferentes. En este caso se puede llevar a cabo un procedimiento de votación de mayoría, pero en cualquier caso no dará resultados exactos.

*ii.* Puede haber pesos no etiquetados; puesto que pueden no quedar cerca de ningún vector de entrada. Aunque esto se puede eliminar con un buen proceso de entrenamiento, enfatiza aún más la necesidad de hallar sistemas para optimizar el algoritmo SOM.

Debido a los dos problemas anteriores, se hace necesario un proceso de sintonización para utilizar un SOM como reconocedor o clasificador. Los algoritmos LVQ se concibieron inicialmente con este objetivo [Koh88b]. Cuando un SOM se somete a este procedimiento, ya no se tienen en cuenta las conexiones de vecindad; por tanto, se representará a partir de ahora los vectores de peso como $\vec{w}_i$.

Para reducir la tasa de mala clasificación, habrá que modificar ligeramente los vectores del diccionario de forma que respondan solo a una clase; eso significará apartarlos de las zonas de frontera entre dos clases, donde estarán cerca de vectores pertenecientes a ambas. Para ello, una vez etiquetados (mediante el algoritmo de calibración anterior), se volverán a someter al conjunto de entrenamiento; si el vector de entrada se clasifica correctamente, entonces el vector peso se acercará más al mismo, todos los demás vectores pesos permanecerán invariables; y si no, se alejará del mismo, de la forma siguiente

**Actualización pesos LVQ**

[L1] $\vec{w}_c(t+1) = \vec{w}_c(t)+\alpha(t)[\vec{x}(t)-\vec{w}_c(t)]$ si se ha clasificado correctamente, y

[L2] $\vec{w}_c(t+1) = \vec{w}_c(t)-\alpha(t)[\vec{x}(t)-\vec{w}_c(t)]$ si se ha clasificado incorrectamente.

Este proceso se puede mejorar aún más con los algoritmos LVQ2 y LVQ3, que no se van a describir aquí, pero el principal problema que presenta es que tiene que partir de una serie de vectores código que sean ya buenos clasificadores, para mejorarlos. Si se parte de vectores iniciales aleatorios, los resultados no serán buenos, o se tardará mucho en alcanzarlos, ya que la ganancia de aprendizaje $\alpha(t)$ es muy pequeña, del orden del 1%. Por lo tanto, el algoritmo LVQ es muy sensible a las condiciones iniciales. Evidentemente, puede ser necesario optimizar también $\alpha(t)$, y además, no se conoce a priori el número de vectores óptimo para la clasificación.

## 2.6 Conclusiones

En este capítulo 2 se han presentado los algoritmos denominados de Kohonen, así como otro algoritmo similar, denominado *de aprendizaje competitivo.*

La principal de función de estos algoritmos es la de diseño de diccionarios para cuantización vectorial, aunque la mayoría de las veces utilizan una terminología totalmente diferente, tomada del campo de las redes neuronales. Se pueden considerar dentro de estas ya que su origen es la modelización de los mapas topológicos aparecidos en el cerebro, y utilizan mecanismos neuronales, como la difusión del estímulo y la inhibición lateral implícitamente en su funcionamiento.

Los mapas autoorganizativos de Kohonen, además de funcionar como cuantizadores vectoriales, responden de forma ordenada al espacio de entrada, lo cual los hace especialmente útiles en tareas de interpolación. Para mejorar las prestaciones de estos, Kohonen introdujo otro nuevo grupo de algoritmos, los LVQ, que añadían una fase de aprendizaje supervisado al SOM con el objeto de mejorar sus resultados de clasificación. Estos últimos algoritmos son ya puramente algoritmos de diseño adaptativo de diccionario.

El problema que presentan estos algoritmos (junto con otros muchos algoritmos

neuronales) es la gran cantidad de parámetros a optimizar: valores iniciales de los pesos, tamaño de la red (del diccionario), etcétera. En una primera aproximación, se puede optimizar alguno de los parámetros, dejando los otros fijos, o simplemente se pueden probar diferentes combinaciones o aplicar reglas heurísticas hasta dar con los mejores resultados. En esta memoria se propondrán métodos mucho más potentes de diseño de diccionarios, que permitirán optimizar la mayoría de los parámetros de los mismos.

# 3 Análisis teórico de los mapas autoorganizativos (SOM)

## 3.1 Introducción

Como se ha visto en el capítulo anterior, tanto los mapas autoorganizativos como el algoritmo LVQ tienen una gran cantidad de parámetros a sintonizar. Dentro de este amplio rango de variación, el algoritmo sólo funcionará bien para un pequeño conjunto de combinaciones de los mismos, dado un problema determinado. Se deben, por tanto, buscar unas condiciones o imponer unas restricciones sobre el mapa, sus parámetros y el conjunto de entrada, dadas las cuales, el algoritmo converja, y además produzca los resultados apetecidos. La búsqueda de estas condiciones se puede hacer de dos formas: de forma teórica, tratando de hallar resultados matemáticos que garanticen el resultado final, o bien sistemáticas o heurísticas, que indiquen en qué casos funcionará el SOM y en qué casos los resultados no serán buenos.

Por otra parte, desde el punto de vista de una posible implementación VLSI, puede ser interesante estimar cuántos bits son necesarios para representar cada peso, ya que estos deberán almacenarse en el propio chip. La cantidad de bits deberá de ser óptima, pues demasiados pocos pueden provocar que no pueda realizar su tarea adecuadamente, y muchos ocuparán una superficie de silicio excesiva.

El estudio teórico de los mapas autoorganizativos de Kohonen (SOM) está aún bastante poco avanzado, a pesar de que diversos grupos de investigación están dedicados al tema. Los principales son *Kohonen et al.* (Finlandia), *Ritter et al.* (EE UU), *Fort, Cottrell et al.* (Francia), *Geszti et al.* (Hungría). Sin embargo, estos estudios teóricos adolecen del fallo siguiente: sólo ofrecen resultados parciales y las conclusiones que se obtienen no son generalizables, de tal suerte que es muy difícil que lleguen a aplicarse alguna vez a problemas reales, donde seguirá imperando la heurística y la experiencia propia. Ello no ha

impedido, sin embargo, la utilización de los algoritmos de Kohonen en una amplia gama de aplicaciones.

El estudio teórico de los SOM se inició en el año 1982, con un trabajo de Kohonen [Koh82]. Era la época de la justificación teórica de las redes neuronales, más o menos contemporánea a la aparición de los primeros trabajos de Hopfield y los trabajos sobre el perceptrón multicapa con el algoritmo de retropropagación. Había que justificar el nuevo campo ante la comunidad internacional, y se quería demostrar sobre todo que era posible crear un sistema, con ciertas propiedades como extractor de características, memoria asociativa o como clasificador, y cuya convergencia fuera probable en ciertas condiciones.

La prueba de la convergencia se concentra en la búsqueda de una *función de Lyapunov*, o de energía. Si se probaba que esta función representaba adecuadamente la dinámica del sistema, y que además decrecía hasta un mínimo debido al entrenamiento, se podía probar el buen comportamiento de la red neuronal, en el sentido de la convergencia. El probar la convergencia de un algoritmo equivale a probar su terminación (es decir, hallar en qué condiciones termina en tiempo finito), y con ello se prueba que se trata efectivamente de un algoritmo, y no un método sistemático.

Además de probar la convergencia, es necesario probar que los mapas autoorganizativos llevan efectivamente a una autoorganización, es decir, una ordenación de las respuestas en la salida. Este ordenamiento es de tipo local. Es decir, la posición relativa de varios puntos $\vec{x}$ en el espacio D es la misma que la disposición relativa de las neuronas $Q(\vec{x})$ en el espacio S de salida, siendo

$$\mathcal{E} \subset D \equiv \mathbb{R}^d$$

$$S \equiv \{0,\ldots,M\} \times \{0,\ldots,M\}$$

Como se ha visto, un SOM provoca una *teselación* del espacio de entrada, de forma que cada neurona $\vec{w}_m$ responde a una zona del espacio de entrada $\mathcal{E}_m$ que se define así

$$\mathcal{E}_m = \{\vec{x} \mid d(\vec{x},\vec{w}_m) < d(\vec{x},\vec{w}_{jk})\ \forall j,k\}$$

Por tanto,

$$\bigcup_{j,k=1}^{n} \mathcal{E}_{jk} = \mathcal{E}$$

Estas dos ecuaciones establecen la notación utilizada a lo largo del capítulo.

En este capítulo se tratarán de analizar las diversas pruebas teóricas de la convergencia y ordenación global de los SOM con el objeto de conseguir una configuración óptima de los vectores peso desde el punto de vista de la exactitud de la clasificación. En la sección 3.2 se analizarán las pruebas de ordenación, en el caso reducido (espacio de entrada y salida unidimensional); en la sección 3.3 se tratará el problema del cálculo de *funciones de energía*. Finalmente (sección 3.4) se abordará teóricamente el problema, interesante para su implementación hardware, de determinar el número de bits mínimo necesarios en cada uno de los parámetros del mapa de Kohonen.

## 3.2 Ordenación global en mapas de Kohonen

Se considerará en primer lugar el mapa reducido a una sola dimensión, ya que en este caso es fácil e intuitivo definir un orden local y global para todo el mapa, y es el único caso para el cual se han encontrado resultados teóricos que prueben la convergencia [Cot94].

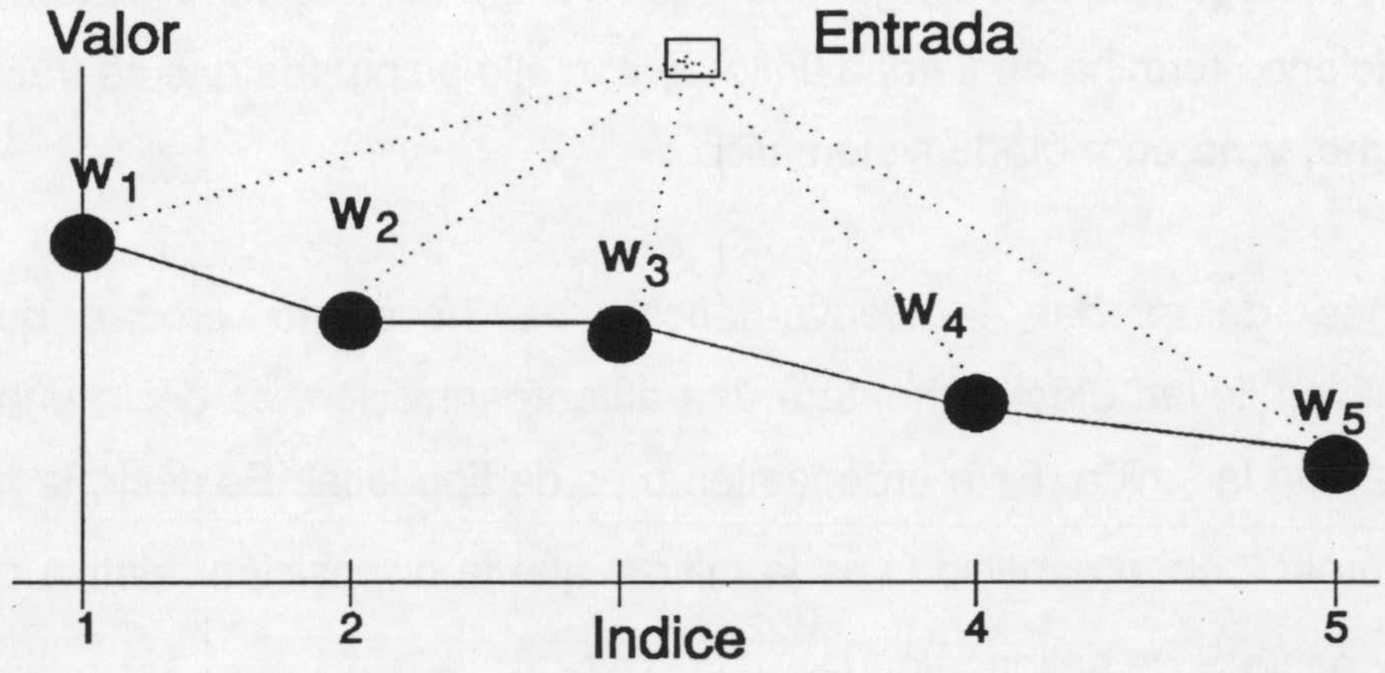



**Figura 3.1** Red de Kohonen en una dimensión; tanto el mapa como las entradas son de dimensión 1.

El primer problema consiste en definir qué es una configuración *ordenada*.

En el caso de un mapa autoorganizativo cuya capa de salida tenga una dimensión $d > 1$, un SOM estará ordenado si, intuitivamente, cada neurona $\vec{w}_{i_1,i_2\ldots i_n}$ (donde cada índice $i_j$ recorre una dimensión) está en el centro de las neuronas cuyos índices varían en ±1 con respecto a ella (por ejemplo, en dimensión $D$=1, $M$=1, el valor del peso -único- de cada neurona $w_i$ estará entre $w_{i-1}$ y $w_{i+1}$, como se muestra en la Figura 3.1).

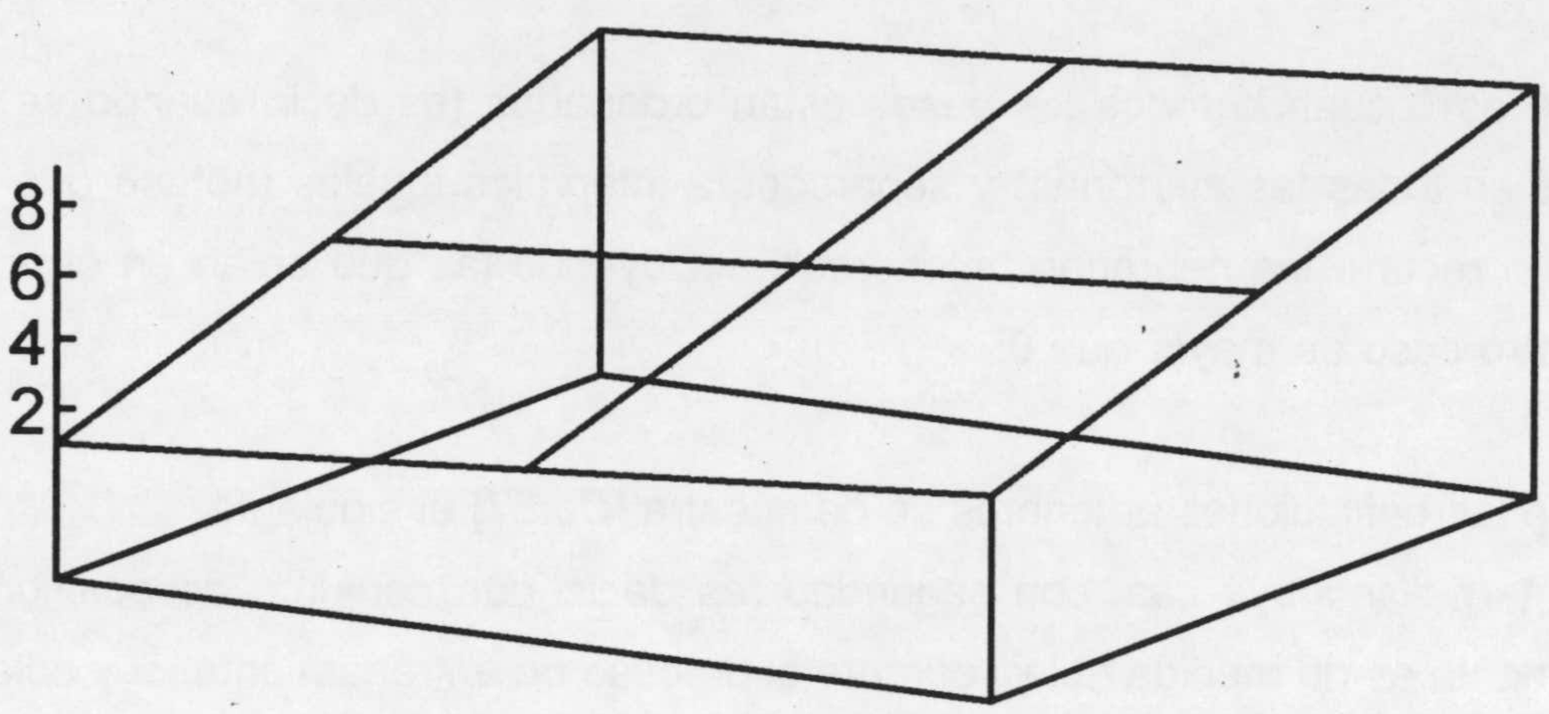



**Figura 3.2** Pesos ordenados en 2 dimensiones; x e y son las coordenadas del peso, z su valor.

En dos dimensiones, hay varias ordenaciones posibles; la idea general es que el valor esté comprendido entre dos neuronas laterales, menor para aquellas que estén situadas "encima", y mayor que las que estén situadas "debajo", como se muestra en la Figura 3.2. Sin embargo, es difícil definir *menor* de una forma significativa para un vector de varias dimensiones; probablemente habría que definirlo para cada una de las dimensiones. El problema de definir un orden en más de una dimensión ha provocado que las pruebas se reduzcan a los SOM de una dimensión.

El ordenamiento es también *global* si esta disposición local se mantiene en todo el mapa (en *M=D=1*, habrá un ordenamiento global si dados i < j < k, $\vec{w}_i < \vec{w}_j < \vec{w}_k$). En el caso más general, el concepto de *ordenamiento global* es más difícil de definir, y habitualmente sólo se define un orden local. Por esto mismo la mayoría de los autores simplifican el problema, para convertirlo en uno donde, si se cumplía el orden local en cada neurona, se convirtiera también en un orden global.

En el *problema reducido*, $M = D = 1$, por lo tanto, se tiene un SOM unidimensional. Además, se usa como espacio de entrada, el intervalo [0,1]. El orden local se cumple si el

valor del peso (único, pues la dimensión del espacio de entrada es 1) de cada neurona es menor que el de la derecha y mayor que el de la izquierda, o viceversa. Si llamamos a los pesos $w_j$, (con un solo índice), se puede definir como función de Lyapunov o de energía que

$$H = \sum_{i=2} ( |w_i - w_{i-1}| - |w_{i+1} - w_i| )$$

trivialmente es 0 cuando todos los pesos están ordenados (es decir, cuando se cumple el orden local en todas las neuronas) y separados a intervalos iguales (nótese que la fórmula anterior sólo recorre las neuronas "interiores", excluyendo las que están en el margen); en cualquier otro caso es mayor que 0.

Con las definiciones anteriores se demuestra [Cot87] el siguiente

**Teorema 1** *H* disminuye casi con seguridad (es decir, que conjunto de ocasiones en los cuales aumenta es de medida nula[1]) durante el proceso de entrenamiento, si y sólo si se dan las condiciones siguientes en los vectores de entrenamiento y en el SOM:

i) *El conjunto de vectores de entrada es infinito*, es decir, contiene todos los elementos del intervalo [0,1].

ii) La función de distribución *F(x)* de las muestras sobre ese intervalo es la *distribución uniforme*, es decir, existe la misma probabilidad de encontrar un punto de entrenamiento en cualquier punto del intervalo. *F(x) = cte.*

iii) Se sigue un proceso de entrenamiento dedicado a eliminar los *picos* de la recta de pesos (es decir, las configuraciones de los pesos en los cuales no se cumple el orden local).

iv) La ganancia $\alpha$ (definida en el capítulo 2) *no varía* durante el entrenamiento.

v) La función de vecindad es constante, e igual a los dos vecinos inmediatos de cada neurona.

Posteriormente, *Bouton & Pagès* [Bou93], probaron que la condición ii) se podía sustituir por cualquier función de distribución, siempre que tuviera soporte [0,1], es decir, que tomara valores dentro de este intervalo. En otros artículos generalizaron el resultado a una clase muy general de distribuciones de entrada [Bou94].

Como indica la condición v), la vecindad tiene que permanecer constante. Sin

---

[1] Lo cual significa que el número de veces en las cuales *H* aumenta es despreciable frente a las ocasiones en las cuales disminuye.

embargo, Erwin et al. [Erw91,92], ampliaron la demostración para incluir todos los algoritmos en los que la función de vecindad disminuyera con la distancia a la neurona ganadora, probando el siguiente teorema [Erw91]

**Teorema 2**: Dado, en una red con $d=1$, cualquier conjunto de pesos $\{w_j(t) \mid j=1,2,...,N\}$ en $t=0$, y una función de vecindad $h(l,m) \equiv h(|l-m|)$ tal que

$$1 \geq h(0) \geq h(1) \geq ... \geq h(N-1) \geq 0$$

donde $N$ es el número total de neuronas, existe una secuencia de patrones $\{x(t) \mid t=1,2,...,T\}, T<\infty$, tal que la aplicación de la regla de actualización del SOM con esta secuencia de valores de $x(t)$ resultará en un conjunto de valores de los pesos $\{w_s(t)\}$ que preservará las relaciones de distancia para $t \geq T$.

Por lo tanto, no se puede afirmar nada en los siguientes casos:

- Función de vecindad arbitraria, o decreciente con el tiempo.
- Factor de ganancia $\alpha$ que disminuya con el tiempo.
- Conjunto de entrenamiento con valores fuera del intervalo [0,1], o con vectores de dimensión M>1.

Si no se da alguna de estas condiciones, que es lo más común, *no se ha probado el teorema que garantice la convergencia del mapa autoorganizativo*. La seguridad de la convergencia proviene sólo de la experiencia, para obtener óptimos resultados hay que utilizar algún algoritmo o procedimiento de optimización del mapa, como veremos más adelante. Por tanto, la optimización en el caso más general de un mapa de Kohonen no se puede hallar por métodos analíticos.

## 3.3 Funciones de energía en mapas de Kohonen

El problema de la convergencia del SOM se puede abordar de otra forma. Se puede tratar de buscar una *función fuerza* para todas las neuronas del mapa o para cada uno de los pesos, de forma que se demuestre que llega un punto en el entrenamiento en el cual la *fuerza* que se ejerce sobre los pesos es nula. El planteamiento anterior se puede enunciar de otra forma, se trata de definir un potencial asociado a la fuerza de forma que el sistema evolucione hacia una región de mínimo potencial. Esta fuerza se debe a la "inmersión" de

cada neurona en un campo, de forma que los pesos tienen cierta tendencia a moverse hacia un mínimo de potencial.

Como en el caso anterior (sección 3.2), el problema de hallar la convergencia se reduce a la búsqueda de funciones de energía globales, que posteriormente hay que minimizar. Una vez hallada esta función de energía, el entrenamiento del mapa se reduce a una minimización de la energía por descenso estocástico (es decir, cambiando el sistema de forma que la energía disminuya en la mayor parte de los casos).

La búsqueda de tales funciones ha fracasado hasta el momento, y algunos intentos como el de *Tolat* [Tol90], cuya forma es similar al término de distorsión definido en el capítulo 2, se ha demostrado que son falsos y mal encaminados (según afirma Erwin et al en Erw91]. La función de energía calculada por Erwin difería en un término referente a los bordes de la función calculada por Tolat. La forma exacta de estas funciones se omite por no ser necesaria para los propósitos de esta memoria.

Considerando estas funciones de energía. La convergencia del SOM presenta estados metaestables, es decir, estados en los que las *fuerzas* sobre los pesos son nulas, y por tanto el sistema se halla en un mínimo energético, pero que sin embargo, son tales que al apartar el sistema ligeramente de ese estado (mediante un cambio estocástico en los pesos), las fuerzas tratan de alejar el sistema del estado anterior, llevándolo a otro estado. Estos estados metaestables no se presentan para todos los tipos de función de vecindad, según demuestra Erwin et al [Erw91].

El hallazgo de las funciones de energía debe de provenir, sin la menor duda, del campo de la estadística, aunque, según afirman Cottrell et al. en [Cot94], el problema radica en la definición de una configuración ordenada para mapas de Kohonen de dimensión mayor que 1. Todas las pruebas de convergencia de basan en teoremas relacionados con los procesos estocásticos y de Markov. Sin embargo, hay que señalar que aunque el problema de probar la convergencia del SOM está abierto, en las aplicaciones prácticas del mismo no se hace uso de los resultados teóricos, puesto que suelen ser muy parciales (por ejemplo, se refieren a SOM de una sola dimensión, y a distribuciones continuas), y, hasta el momento, no existen resultados teóricos referidos al caso más general: distribución arbitrariade las muestras en el espacio de entrada, que además tiene muchas dimensiones, y SOM de más

de una dimensión.

En cualquier caso, la eficacia del SOM como algoritmo de cuantización vectorial y clasificación está probada en la práctica, pues se ha aplicado con éxito a diversos problemas [Mer94][Koh90a], entre los cuales se encuentra, por ejemplo, el estudio del cortejo de las abejas.

## 3.4 El problema de la precisión en redes de Kohonen

Calcular el número de bits necesario para almacenar la información procedente de un canal o en un fichero es una tarea que corresponde a la teoría de la señal o de la información. La mayoría de los canales contienen información redundante, y mediante algún algoritmo de compresión se puede eliminar esta información redundante, manteniendo su contenido informativo, y convirtiendo el contenido del canal en otro con menor cantidad de bits y bytes.

Una red neuronal, en cierto sentido, almacena información sobre las muestras de entrada que se le entregan, codificándola. Sin embargo, no almacena toda la información, sino sólo la necesaria para llevar a cabo un proceso de clasificación, por ejemplo. Se puede considerar el contenido informativo del conjunto de entrenamiento como una cota superior al contenido informativo de los pesos de la red; pero, en cualquier caso, los pesos deberán almacenar mucha menos información. En el caso de una red neuronal, se puede llevar a cabo el mismo proceso. Se puede analizar la entropía de información del conjunto de entrenamiento, y calcular la cantidad de bytes necesaria para almacenarla, que se convertirán mediante algún factor de conversión en el número de neuronas necesarias (y de pesos necesarios).

Sin embargo, esta operación de cálculo del contenido informativo no se puede realizar con la red de Kohonen. La red de Kohonen, al igual que otros modelos neuronales, tratan de abstraer o generalizar a partir de las muestras de entrenamiento, para después reconocer. Esto implica reducir la cantidad de información asociada a los vectores de entrada. La red neuronal, en definitiva, trata de extraer la información relevante del espacio de entrada.

Debido a lo anterior, el cálculo de la entropía de la información asociado al conjunto de muestras de entrenamiento no significa obtener el número de bits necesarios para cada peso (dividiendo el contenido informativo de las muestras de entrada por el número de pesos). Por el contrario, la entropía de información del conjunto de muestras de entrenamiento sería un límite superior, como ya se ha indicado. Además, puede que lo que se pretenda sea generalizar, de tal forma que en los pesos puede haber mucha más información que en el conjunto de entrada.

Desde otro punto de vista, los SOM son funcionan como extractores automáticos de características. Esto implica que "a priori" no se conoce dónde está la información en las muestras de entrada y por tanto no se puede evaluar la catnidad de información de los mismos, considerando comoinformación la codificación o representación de las características extraidas.

En todo caso, lo anterior es aplicable siempre que se pretenda utilizar una red neuronal como clasificador o memoria asociativa. En algunos caso, cuando lo que interesa es que interpole, normalmente los pesos tienen capacidad para almacenar más información que la contenida en el conjunto de entrenamiento [Mer94, And93].

Al no poder tomar este camino analítico para calcular el número de bits por peso, se tendrá que acudir a un análisis cualitativo. La información, a la hora de ser introducida a la red de Kohonen, sufre previamente un proceso de discretización, por el cual se transforma en un número con una cantidad finita de bits; el mayor número admisible estará limitado por la cantidad de bytes de la representación interna. Esto es ya en sí un proceso de cuantización; porque las entradas no tomarán valores en $\mathbf{R}$, sino en un subconjunto finito de él, es decir, para un vector cualquiera $\vec{x} \in \mathbf{R}$, primero sufre una discretización $D(\vec{x}) = \vec{x}' \in A$,, donde $A$ es el conjunto de todos los valores representables dentro del ordenador con la precisión que este admite. Posteriormente, sobre este vector $\vec{x}'$ se aplicará la función de cuantización vectorial $Q$.

Habitualmente, la discretización D significará una pérdida de información. Para que no se produzca esta pérdida de información, se puede aplicar el teorema de Shannon en este caso a frecuencias espaciales, de forma que la frecuencia de muestreo espacial debe de ser

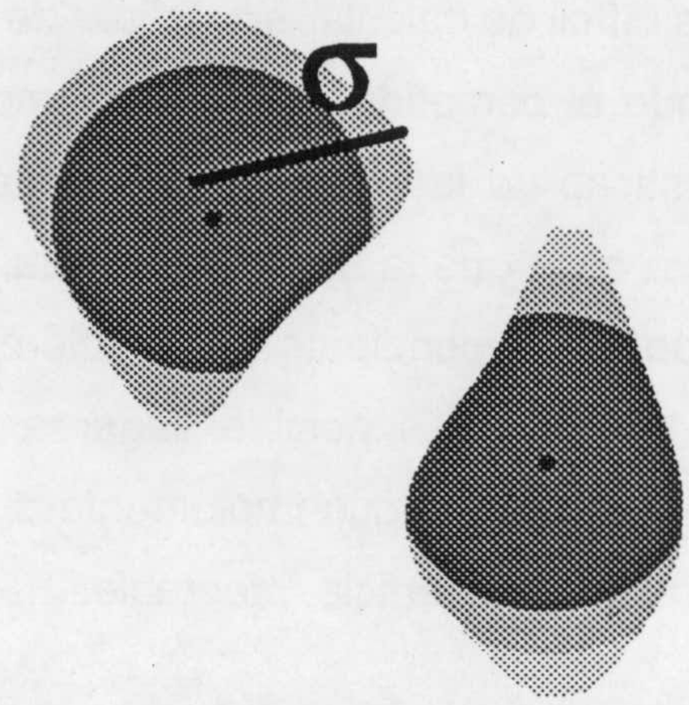



**Figura 3.3** Distribución de puntos de la muestra de entrenamiento, mostrando la anchura de la gaussiana

dos veces mayor que la frecuencia máxima intrínseca de la muestra de entrada. La transformación que sufre una muestra analógica es

$$D(n) = \frac{n - n_{\min}}{n_{\max} - n_{\min}} 2^m$$

donde $[n_{max},n_{min}]$ es el intervalo de entrada analógico, y $m$ el número de bits utilizados para la transformación.

Si suponemos que las muestras de entrada están distribuidas aproximadamente según una gaussiana, su frecuencia fundamental será igual a $\frac{1}{2\sigma}$. La transformación analógico-digital tendrá que hacerse en cuenta teniendo este dato:

[P] $$D^{-1}(1) << \sigma,$$

es decir, que la retícula fundamental en la que se va a dividir el espacio debe de ser mucho menor que el tamaño del cluster de muestras de entrada, como se ve en la Figura 3.3.

Puesto que los pesos deben de compararse con los vectores de entrada para calcular las distancias a los mismos (sea cual sea el tipo de distancia, como se ha visto en la sección 2.3), estos deberán de tener la misma precisión. Dado que, en el caso anterior, el error medio esperado o distorsión será del orden de $\sigma$, se deberán de diseñar los pesos con un número de bits tal que cumpla la condición anterior [P], es decir, que el intervalo mínimo representable con el número de bits utilizado sea mucho menor que el tamaño intrínseco de las muestras de entrada.

Esto, evidentemente, es difícil de calcular sin aplicar de antemano cualquier algoritmo de cuantización vectorial, ya que el cometido de un algoritmo de cuantización vectorial es precisamente extraer características de las muestras de entrada, a partir de las cuales se puede calcular $\sigma$ (que no es otra cosa que la distorsión media, tal como se ha definido en el capítulo 2); y en cualquier caso, es dependiente del problema, puesto que depende de la distorsión del problema. Por tanto, no es posible llegar a ninguna conclusión sobre la precisión de los pesos incluidos en un chip que implemente el mapa autoorganizativo, salvo que debe ser la mayor que permita la superficie integrable.

El utilizar una precisión que haga que se cumpla la condición *P*esta precisión puede provocar, sin embargo, que sea inadecuada para algunos problemas (en los cuales la frecuencia espacial sea mucho menor que lo mínimo que puede discretizarse con los bits que permite el chip), ya que la cuantización a la que se puede ver sometido el espacio de entrada por mor de la transformación analógico-digital puede eliminar gran parte de la información. Las implementaciones VLSI, por tanto, no serán adecuadas para este tipo de problemas, salvo que no importen pérdidas de información.

## 3.5 Conclusiones

El estado actual de la investigación, aunque permite afirmar que el SOM va a alcanzar buenos resultados en el caso unidimensional, no permite afirmar nada en el caso más general de mapas bidimensionales, o en el caso de LVQ, y en el caso de que la dimensión del espacio de entrada sea mayor que 1. La búsqueda de un método de optimización de los resultados deberá de concentrarse en método más heurísticos o de prueba sistemática. Entre ellos, los algoritmos genéticos permiten hallar los mejores resultados; y serán los que se utilicen para optimizar un diccionario entrenado mediante LVQ.

Por otro lado, las representaciones de precisión finita no se consideran adecuadas para la implementación de redes neuronales en hardware. Enfoques que no presenten esta limitación, como la arquitectura MapA [Ort93], se consideran más adecuadas para una implementación hardware de redes neuronales como los mapas autoorganizativos. En cualquier caso, si así se decidiera, el número de bits de representación por peso se podía

considerar también un parámetro a optimizar, pero, evidentemente, la solución sería, siempre, dependiente del problema, lo cual hace que se deseche una solución VLSI de precisión limitada, pero siempre fija, para su implementación.

# 4 Mapas autoorganizativos de Kohonen para clasificación de proteínas

## 4.1 Introducción

En este capítulo se presenta la primera aplicación que trata de optimizar el SOM, siguiendo criterios sistemáticos. En toda aplicación con varios parámetros a ajustar, se utiliza alguna forma de optimización. La más habitual (utilizada, por ejemplo, en el paquete SOM_PAK [Koh92]), consiste en ejecutar varias veces la aplicación con parámetros escogidos al azar, y seleccionar el conjunto de parámetros con el que se obtenga una mejor solución. En la aplicación presentada en este capítulo, se utiliza un método sistemático para hacerlo, y se tiene en cuenta la distorsión para evaluar la exactitud de la clasificación.

La aplicación elegida es la clasificación de proteínas a partir de su espectro de dicroísmo circular. Los resultados obtenidos por el procedimiento aquí propuesto son superiores a los de otros métodos similares, y consume bastante menos tiempo de cálculo y memoria.

En este capítulo se trata, en primer lugar (sección 4.2), de explicar con claridad el entorno del problema (que, como suele suceder muchas veces en el campo de las redes neuronales y del reconocimiento de patrones, cae fuera del dominio habitual de los informáticos). En la sección 4.3 se tratará de hacer un resumen de la relación que ha habido hasta el momento entre redes neuronales y proteínas, así como de otras soluciones clásicas al problema de obtención de la estructura secundaria de una proteína. Y en la sección 4.4 se expondrá la solución aquí propuesta, así como las medidas necesarias para optimizar el SOM.

## 4.2 Estructura de proteínas.

Las proteínas son unos de los componentes básicos de la materia viva. No sólo son los elementos fundamentales en la construcción de las células, sino que también desempeñan importantes funciones catalíticas (enzimas) que son la base del metabolismo celular. No se pretende aquí una revisión en profundidad de las propiedades funcionales de las proteínas, sino presentar y resaltar las características estructurales más relevantes de las mismas. Esta introducción va claramente orientada al objetivo de esta Memoria, donde se propone (más adelante en el capítulo) un método para determinar la conformación de proteínas.

Las proteínas están formadas por la unión a través de un enlace covalente de unas moléculas llamadas aminoácidos, en un orden específico. Un aminoácido es una molécula orgánica que tiene un carbono central, llamado carbono alfa, unido a cuatro radicales o grupos diferentes: un grupo amino (-NH2), un grupo ácido o carboxilo (-COOH), un hidrógeno (-H) y una cadena carbonada (llamada cadena lateral) que es la que diferencia cada aminoácido. Los aminoácidos una vez unidos se denominan residuos y el enlace entre ellos enlace peptídico. Este enlace se logra por la unión del grupo carboxilo de un residuo con el grupo amino del siguiente, formando un enlace N-C. Esencialmente existen 20 aminoácidos distintos, y sus diferentes combinaciones o secuencias dan lugar a la formación de las diferentes proteínas.

Hay que señalar que la estructura de una proteína concreta no es un absoluto, sino un concepto dinámico, que puede cambiar. En principio viene determinada por el medio en que se encuentra y por las condiciones de experimentación. Cuando las condiciones de temperatura, fuerza iónica o acidez del medio, son extremas, pueden desnaturalizar la proteína, adoptando ésta una conformación desordenada, al azar. En lo sucesivo, cuanto se refiera a la estructura de una proteína se referirá a su estructura nativa, es decir, a la que adopta en condiciones fisiológicas.

Se pueden distinguir cuatro niveles de conformación o estructura en las proteínas:

1) *Estructura primaria*, que consiste en la secuencia u orden específico de los resíduos aminoácidos que componen la cadena polipeptídica. Esta estructura es característica de cada proteína y es la que condiciona de un modo u otro las otras conformaciones.

2) *Estructura secundaria*: primer nivel de estructura espacial o tridimensional. Se obtiene

al ubicarse los enlaces que forman el esqueleto de la cadena en una disposición espacial concreta. Las diferentes estructuras se adoptan por giros del propio enlace peptídico N-C y del enlace C-C que une el grupo carboxilo con el el carbono alfa del mismo aminoácido. La libertad de giro viene determinada críticamente por la cadena lateral de cada residuo, de modo que la secuencia de aminoácidos influye en la disposición adoptada. La estructura secundaria se adopta por segmentos o grupos de resíduos. Se distinguen varios tipos, siendo los más importantes: hélice alfa, lámina beta, giros beta y estructura aperiódica o desordenada.

3) *Estructura terciaria*: es la estructura tridimensional que resulta del plegamiento o disposición en el espacio de las diferentes estructuras secundarias. Esta estructura tridimensional determina el tipo de proteína, como globular, fibrosa, etc. La estructura terciaria se mantiene gracias a determinadas fuerzas, como interacciones electrostáticas entre las cadenas laterales de los aminoácidos, presencia de átomos de metales pesados como Fe, Mn, Mg, etc. En general la fuerza que dirige el plegamiento de una proteína en un medio determinado es la resultante de un delicado balance entálpico y entrópico.

4) *Estructura cuaternaria*: es la que se forma al agregar, en una disposición concreta, diferentes cadenas polipeptídicas para formar una estructura funcional. La estructura cuaternaria se mantiene fundamentalmente a través de fuerzas electrostáticas, no covalentes. En definitiva es el resultado del ensamblaje de "proteínas" con su propia estructura terciaria, para formar una estructura más compleja.

Como ya se ha indicado, la estructura tridimensional de una proteina es resultado de dos factores: su secuencia y el medio donde se encuentra. Existen diferentes métodos para estimar la estrucura tridimensional a partir de la secuencia. Pero hay que significar que son métodos aproximados que se basan en el conocimiento de estructuras conocidas por otras técnicas, y que el problema del plegamiento de proteinas (*protein-folding*) no está hoy por hoy ni mucho menos resuelto.

La estructura tridimensional o terciaria se puede hallar mediante cristalización de la proteína, y difracción de rayos X a través de la estructura cristalina resultante. Sin embargo, esto no es fácil, pues no sólo se necesita un alto grado de pureza, sino que además no todas

las proteínas se pueden cristalizar, de tal modo que sólo unas 700 proteínas tienen estructura secundaria conocida a través de esta técnica. Incluso así, el procedimiento de cálculo de estructura secundaria por esta técnica necesita mucho tiempo y esfuerzo por parte de un grupo de investigación. Al cristalizar una proteina, por otro lado, se piede información sobre la conformación de la misma en condiciones fisiológicas, o incluso sobre el cambio en la estructura inducido por la presencia de algún factor externo.

Otra vía alternativa es determinar la estructura de la proteina en disolución. Actualmente existen diferentes técnicas que consiguen aproximar la estructura tridimensional de proteínas, bien calculando su estructura segundaria o bien su estructura terciaria. Una de estas técnicas es la que se va a emplear y estudiar en este capítulo de la memoria. Las proteínas en disolución tienen actividad óptica, es decir, que giran el plano de la luz polarizada (generalmente en el espectro ultravioleta) en uno u otro sentido. Con el giro obtenido para luz polarizada de diferentes longitudes de onda se halla el denominado *espectro de dicroísmo circular*, o *espectro* CD.. Para hallar el mismo es necesario que la proteína se encuentre en estado de alta pureza, pero el grado de pureza no tiene por qué ser tan alto como en el procedimiento de cristalización. Es suficiente habitualmente con una pureza superior al 90%; por tanto muchas más proteínas son susceptibles de ser sometidas a este método. Mediante este método, además, se pueden observar las proteínas en sus condiciones nativas, es decir, en las condiciones en las cuales realizan su función en medio fisiológico.

El interés principal sobre el conocimiento de la estructura tridimensional se basa en que la actividad, es decir, la posible función de la proteína, depende fuertemente de su conformación o estructura tridimensional; hay una rama de la química, denominada *estudios SAR y QSAR*, es decir, *quantitative structure-activity relationship*, dedicada precisamente a este tema.

## 4.3 Redes neuronales aplicadas al reconocimiento de estructura secundaria de proteínas

Como se ha visto anteriormente, hay dos formas principales de deducir la estructura

secundaria: a partir de la estructura primaria y a partir de determindas propiedades físicas de las mismas, tales como las propiedades ópticas en el caso del espectro de dicroísmo circular. Las soluciones, tanto clásicas como neuronales, se dividirán, por tanto, en los dos grupos citados (que se verán respectivamente en los apartados 4.3, 4.3.1 y 4.3.2). También se ha tratado de aplicar redes neuronales, en concreto el SOM, para clasificación de proteínas, pero con un modo de resolución ligeramente diferente, como se analiza en el apartado 4.3.3.

### 4.3.1 Reconocimiento de estructura secundaria de proteínas mediante perceptrones multicapa.

El problema que se trata de resolver mediante un perceptrón multicapa es asignar a cada residuo o aminoácido la estructura secundaria correspondiente. Esto no se puede calcular de antemano; pero para las proteínas para las que se conoce la estructura terciaria, hay programas (HSSP y DSSP) que a partir de la misma predicen la estructura secundaria ($\alpha$, $\beta$ o random) para cada aminoácido. Por tanto, como conjunto de entrenamiento para la red se deben utilizar las proteínas cristalizables para las cuales la estructura terciaria es conocida. En cualquier caso, esto constituye un conjunto de entrenamiento suficientemente grande, ya que en media cada proteína forma una cadena de varios cientos de residuos.

La secuencia o estructura primaria de la proteína se tiene que preprocesar o codificar para introducirla en el perceptrón. El método preferido es codificar cada aminoácido como una secuencia binaria de 21 dígitos [Qian88]; dándole 1 a un valor arbitrario y único para cada aminoácido, y 0 a todos los demás. Por tanto, la red neuronal tiene 21 entradas para introducir la información de cada aminoácido. En este método se supone que la estructura secundaria correspondiente a un aminoácido en una posición determinada depende de los aminoácidos que tiene alrededor, por lo tanto la secuencia se introduce mediante ventanas con un número determinado de aminoácidos (generalmente mayor que 15). El perceptrón multicapa tendrá, por tanto, como mínimo, 15*21 entradas, y la salida corresponderá a la calificación del residuo situado en el centro de la *ventana.* La salida consiste en 3 neuronas, una para $\alpha$, otra para $\beta$ y otra para coil (estructura aleatoria). Se toma como porcentaje de estructura secundaria el correspondiente a la neurona con una mayor activación.

Puede suceder también que proteínas con una secuencia muy parecida, pero que difiera en algún aminoácido, tengan la misma actividad o función (por ejemplo, proteínas

como el *citocromo*, que se encuentran en todo el reino animal, desde los paramecios hasta las células humanas). A estas proteínas se les denomina *proteínas homólogas*, y tendrán para secuencias diferentes (aunque muy similares) la misma estructura secundaria. Si se utiliza en el conjunto de test una proteína homóloga a una proteina del conjunto de entrenamiento, dará buenos resultados probablemente, pero esto no dice nada sobre el éxito del método, ya que el hecho de la homología indica de por sí cuál va a ser la estructura secundaria, proteínas homólogas tienen estructura primaria similar, y por tanto estructura secundaria también similar.

Este método, ideado por Qian and Sejnowski [Qian84], fue utilizado por diversos autores [Kne90, Mus92, Com91, Hol89], con porcentajes de éxito que estaban alrededor del 60%. Se consiguieron ciertas mejoras cambiando el método de codificación, y añadiendo una segunda red [Kne90] para corregir errores comunes en las predicciones de la estructura, pero nunca por encima del 70%.

Rost *et al.* [Rost93] han conseguido optimizar al máximo esta red, reuniendo todas las mejoras introducidas anteriormente por otros autores, y obteniendo los mejores resultados hasta el momento. Para ello, complican el modelo de Qian y Sejnowki, añadiéndole una segunda red que mejora los resultados de la primera (idea introducida inicialmente por [Mus92]) y utilizan un *jurado* o votador de redes neuronales, de forma que se elige como salida (es decir, como estructura secundaria para el residuo) la que elijan la mayoría de 12 redes neuronales entrenadas de la misma forma.

Rost *et al.* codifican la entrada a la red de forma ligeramente diferente a la vista anteriormente; no se codifica una sola proteína, sino un grupo de proteínas homólogas (es decir, de estructura parecida), de forma que cada componente *j* del vector que codifica una posición representa la frecuencia en que en el aminoácido *i* aparece en esa posición. Este es el mejor método que existe actualmente para predicción de la estructura secundaria a partir de la secuencia.

Este enfoque, en general, presenta una serie de problemas. Primero, es muy lento y computacionalmente costoso, tanto para entrenar el perceptrón (por la gran cantidad de pesos de la red, y de muestras), como a la hora de calcular la estructura de una proteína, pues necesita bastante tiempo de cálculo. Por último, no se puede emplear para aquellas proteínas

para las cuales no se conoce la secuencia; aunque esto no es muy limitante, pues la mayoría de las proteínas son secuenciables, sí puede ser una dificultad añadida en el transcurso de una investigación. Tampoco estima la estructura de la proteína en disolución, ni evalúa cambios estructurales.

### 4.3.2 Cálculo de la estructura secundaria de proteínas a partir del espectro de dicroismo circular.

En principio, se han propuesto diferentes métodos para abordar este problema, aunque ninguno de ellos emplea redes neuronales. El espectro CD de la proteína está descrito por un vector de unos 30 a 50 componentes, dependiendo del número de longitudes de onda que se tomen para hallar el mismo. Este método se basa en la suposición de que cada espectro es el resultado único de la estructura secundaria de la proteina. Por tanto, cabría suponer que es el resultado de la adición de espectros producidos por las diferentes regiones o conformaciones $\alpha$, $\beta$ y random.

Por ello, la mayoría de los esfuerzos se han concentrado en hallar el espectro CD de una proteína como una combinación lineal de una serie de espectros ideales $\alpha$, $\beta$ y random. Y estos espectros ideales se han tratado de hallar de dos formas [Yan81]: a partir de proteínas en disolución, que cambian de propiedades dependiendo de las condiciones del medio en el que se encuentran, teniendo un 100% de estructura $\alpha$, $\beta$ o random; o bien tratan de hallar estos espectros ideales mediante combinación lineal de espectros a los cuales le corresponde una cantidad conocida de estructura secundaria [Per91], calculado por otro método, como por ejemplo difración de rayos X.

Sin embargo, la hipótesis indicada en el párrafo anterior es probablememte falsa, puesto que las contribuciones de cada zona no tienen porqué sumarse, sino que puede haber interacciones entre las zonas $\alpha$ y $\beta$ de diferentes partes de la proteína, y, además, no se tiene en cuenta el tamaño de cada zona, que puede también influir en la contribución al espectro. Además, las soluciones que se obtienen adolecen de varios fallos, como dar porcentajes muy diferentes para proteínas muy parecidas [Perc93], y dar incluso porcentajes negativos de estructura secundaria.

### 4.3.3 Clasificación de proteínas mediante mapas autoorganizativos de Kohonen

Mediante redes neuronales, se puede hallar un método para clasificar secuencias de proteínas de acuerdo con su similaridad. En este caso no se trata de reconocer las proteínas, sino simplemente de agruparlas de acuerdo con la similaridad de sus secuencias. Para ello, hay que codificar la secuencia completa para llevar a cabo el entrenamiento, de forma que no se pierda la similaridad. La forma elegida es mediante una matriz de frecuencias de dipéptidos, es decir, de pares de aminoácidos. De esta forma, hay 20x20 elementos de entrada a la red neuronal.

Al entrenar una red de Kohonen con proteínas extraídas de diversos animales, se encuentra que proteínas homólogas, es decir, con secuencias similares, se encuentran en zonas similares; en concreto, la misma proteína para diferentes especies, como el *citocromo c*, se suele encontrar en la misma neurona o en neuronas cercanas; proteinas similares están en neuronas similares [Fer92a,b].

También sucede que la misma proteína, clasificada para muchas especies diferentes, hace que se agrupen en neuronas cercanas las proteínas correspondientes a especies muy relacionadas, lo cual indica un origen evolutivo similar. Esto es también de esperar, ya que las secuencias son similares [Fer92a].

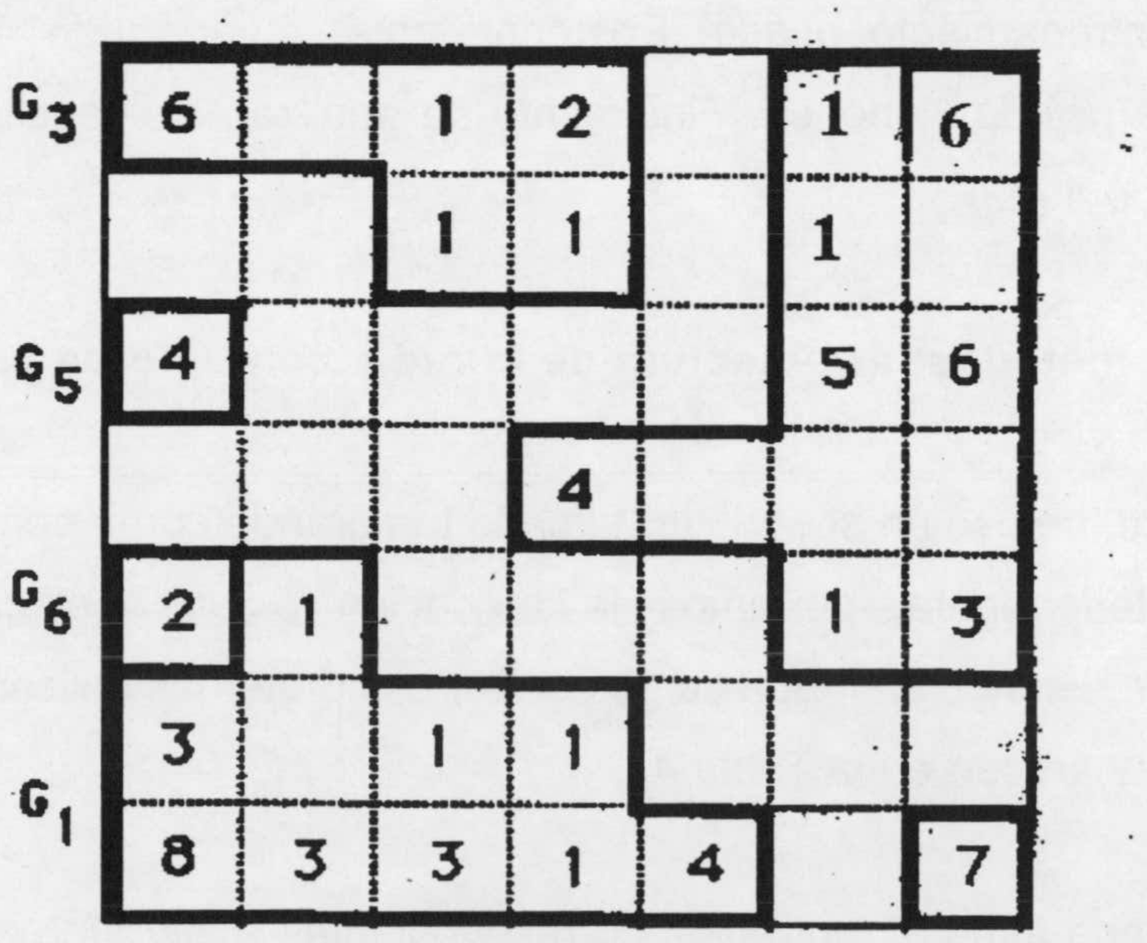



**Figura 4.1** Resultados de la clasificación mostrada en [Fer92]; los números corresponden al número de proteínas para la cual la neurona "gana".

Sin embargo, los gráficos mostrados en [Fer92] (y que se muestran en la Figura 4.1 hacen suponer que no se lleva a cabo el entrenamiento correctamente; ya que si fuera así,

todas las neuronas ganarían para algún elemento del espacio de entrada; sin embargo, los patrones de entrada se ven distribuidos de forma irregular por el mapa.

En este gráfico, además, se puede observar la concentración de muchos vectores en las esquinas del mapa, y, en general, una especie de mosaico (ver capítulo 2, sección 2.3 C), lo cual indica que no ha tenido lugar la autoorganización característica del mapa de Kohonen, y por tanto una ordenación global. Esto no invalida su resultado, pero indica una vez más, como se trata de demostrar en esta memoria, la necesidad de optimizar el SOM para obtener resultados correctos.

## 4.4 SOM propuesto para clasificación de proteinas

En este apartado se presenta nuestra contribución consistente en la implementación y optimización de un SOM para clasificación de proteínas a partir de su espectro CD y el cálculo de porcentajes de estructura secundaria a partir de él.

Se describirá inicialmente (apartado 4.4.1) la estructura de la red utilizada, así como el conjunto de entrenamiento usado. Posteriormente, el apartado 4.4.2 se refiere a la optimización de la red de Kohonen. Finalmente se analizan los resultados de clasificación obtenidos (apartado 4.4.3).

### 4.4.1 Material y métodos: arquitectura de la red y conjunto de entrenamiento

Inicialmente, se usó un SOM de un tamaño tan grande como permitía el PC usado, de 13 por 13. Posteriormente, portamos la aplicación a una estación de trabajo Sun SPARCstation Sun, usando un mapa de 20 por 20. Las proteínas utilizadas y sus porcentajes correspondientes aparecen en la Tabla 4.i.

Cada neurona tiene 32 entradas, correspondientes a los diferentes componentes del espectro CD de las proteínas, eliminando los componentes menos significativos, es decir, aquellos en los que el giro no varía a lo largo del conjunto de todas las proteínas, y es aproximadamente igual a 0. Esto corresponde aproximadamente a los giros por debajo de

200 nm y por encima de 240 nm. Estos espectros aparecen en la figura 2. A este conjunto de entrenamiento se le añadieron otras 6 proteínas cuya estructura era *pura*, es decir, eran todo α, todo β to todo random. Para el cálculo de la estructura secundaria, se tomaron todas menos una, apartada para test; esto se hizo para todas las proteínas. Para la clasificación, se utilizaron todas las proteínas.

**Tabla 4.i** Estructura secundaria de las proteínas utilizadas en el entrenamiento. β1 es lámina β, β2 giros β.

| Proteína | α | β1 | β2 | R |
|---|---|---|---|---|
| Myog | 0.79 | 0.00 | 0.05 | 0.16 |
| LDH | 0.45 | 0.24 | 0.06 | 0.25 |
| Lyso | 0.41 | 0.16 | 0.23 | 0.20 |
| CytC | 0.39 | 0.00 | 0.24 | 0.37 |
| SBNP | 0.31 | 0.10 | 0.22 | 0.37 |
| Papa | 0.28 | 0.14 | 0.17 | 0.41 |
| RNsA | 0.23 | 0.40 | 0.13 | 0.24 |
| Chy | 0.09 | 0.34 | 0.34 | 0.23 |
| Elas | 0.07 | 0.52 | 0.26 | 0.15 |
| CncA | 0.02 | 0.51 | 0.09 | 0.38 |
| Parv | 0.62 | 0.05 | 0.17 | 0.16 |
| AdnK | 0.54 | 0.12 | 0.19 | 0.15 |
| Insu | 0.51 | 0.24 | 0.12 | 0.13 |
| CrbA | 0.37 | 0.15 | 0.26 | 0.22 |
| Thrm | 0.36 | 0.22 | 0.18 | 0.24 |
| Tryl | 0.28 | 0.33 | 0.03 | 0.36 |
| RNsS | 0.26 | 0.44 | 0.13 | 0.17 |

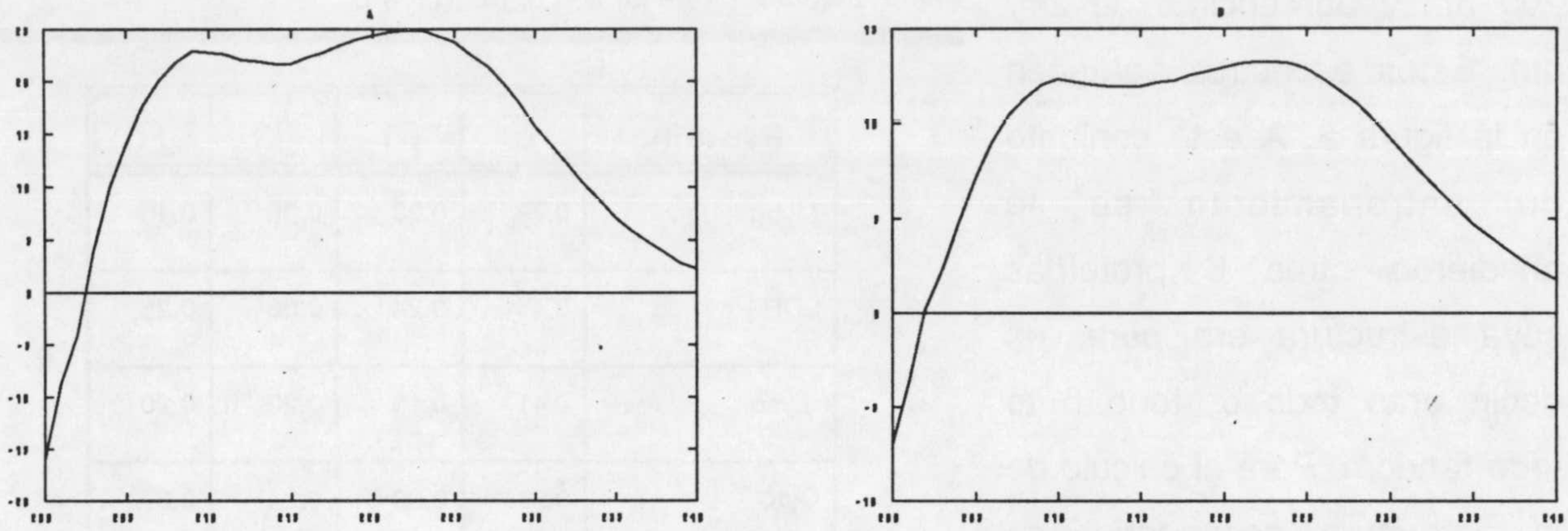


Figura 4.2 a), b) : espectro de dicroismo circular de las proteínas Myog y LDH.

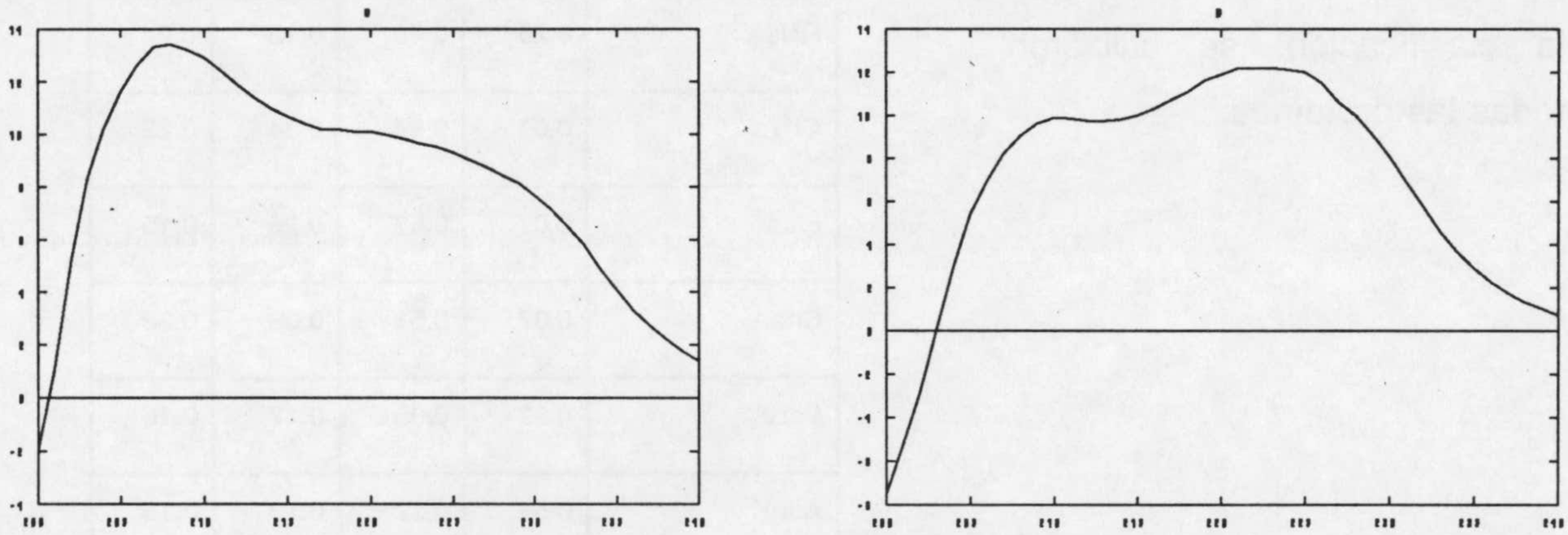


Figura 4.2 c) y d): espectro de dicroismo circular de las proteínas Lys y CytC.

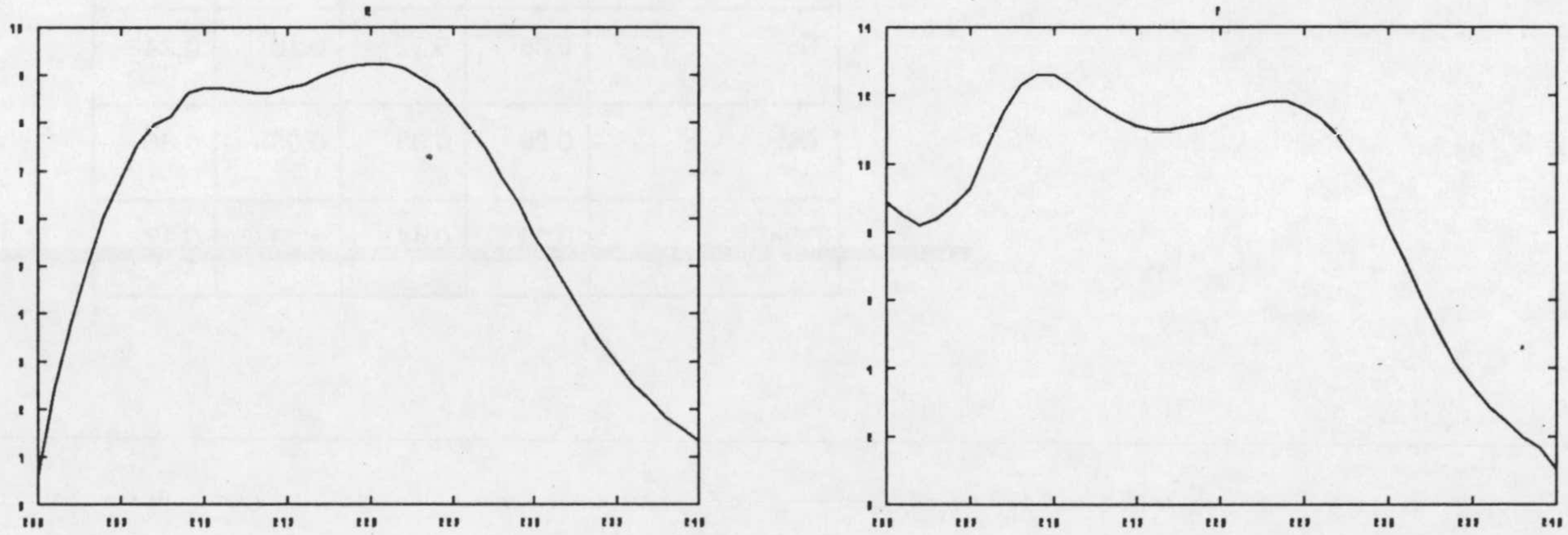


Figura 4.2 e) y f) : espectro de dicroismo circular de las proteínas SBNP y Papa.

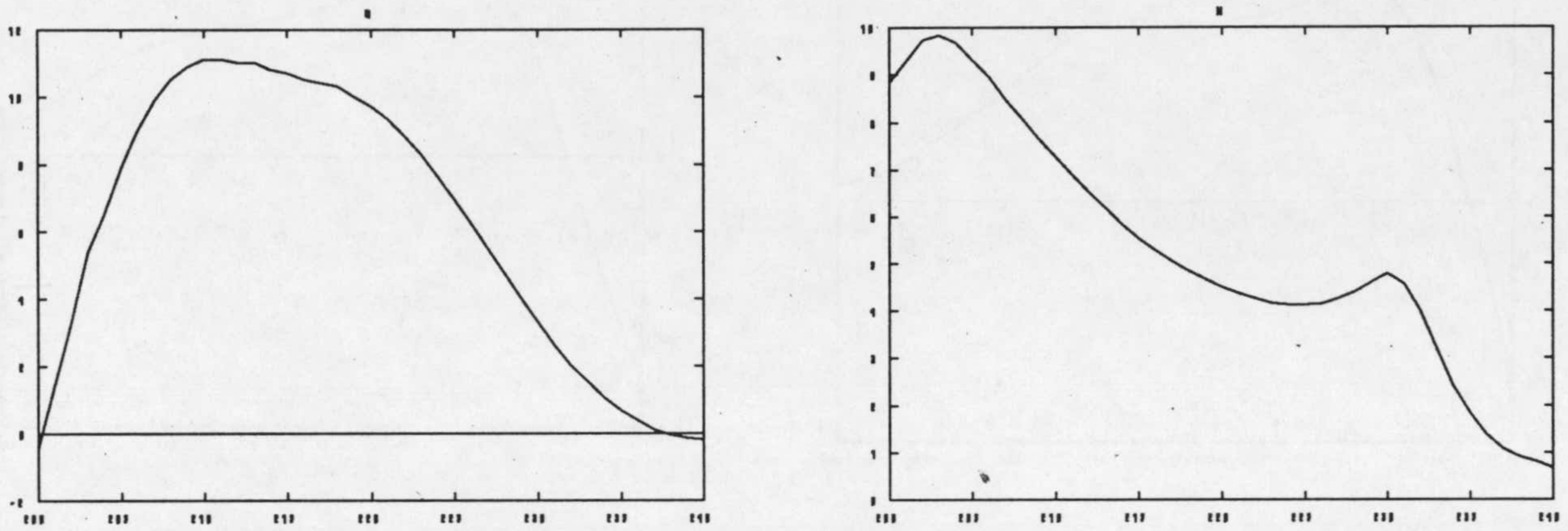


Figura 4.2 g) y h): espectro de dicroismo circular de las proteinas RNsA y Chy.

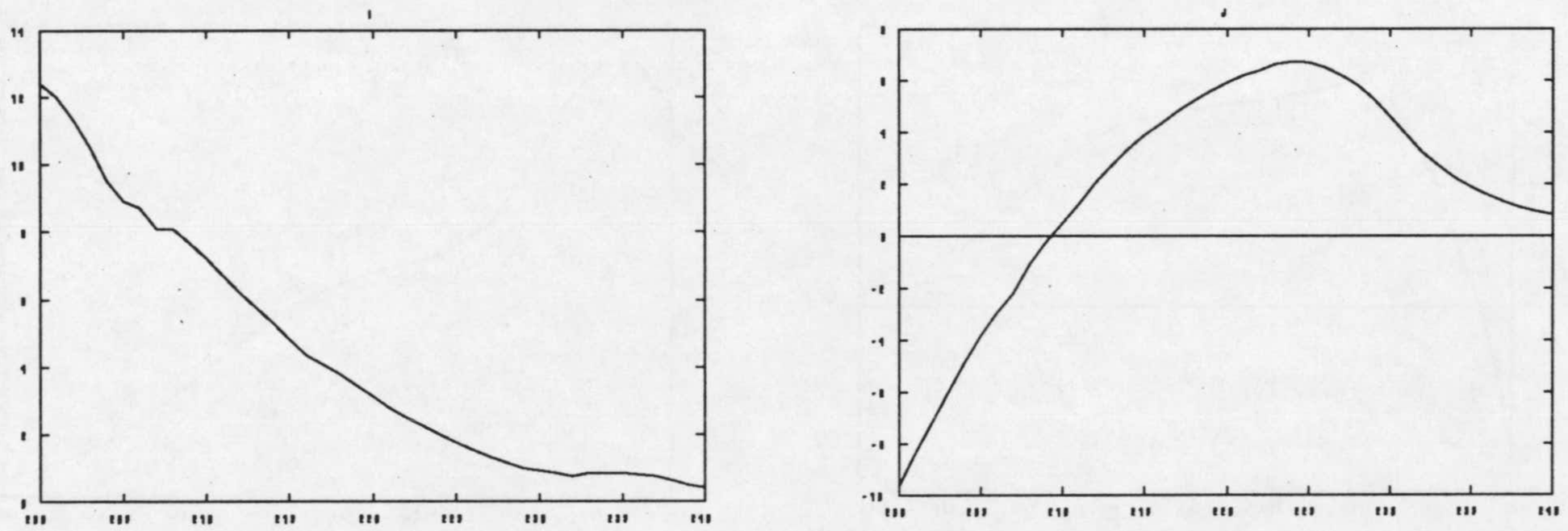


Figura 4.2 g) y h): espectro de dicroismo circular de las proteínas Elas y CncA.

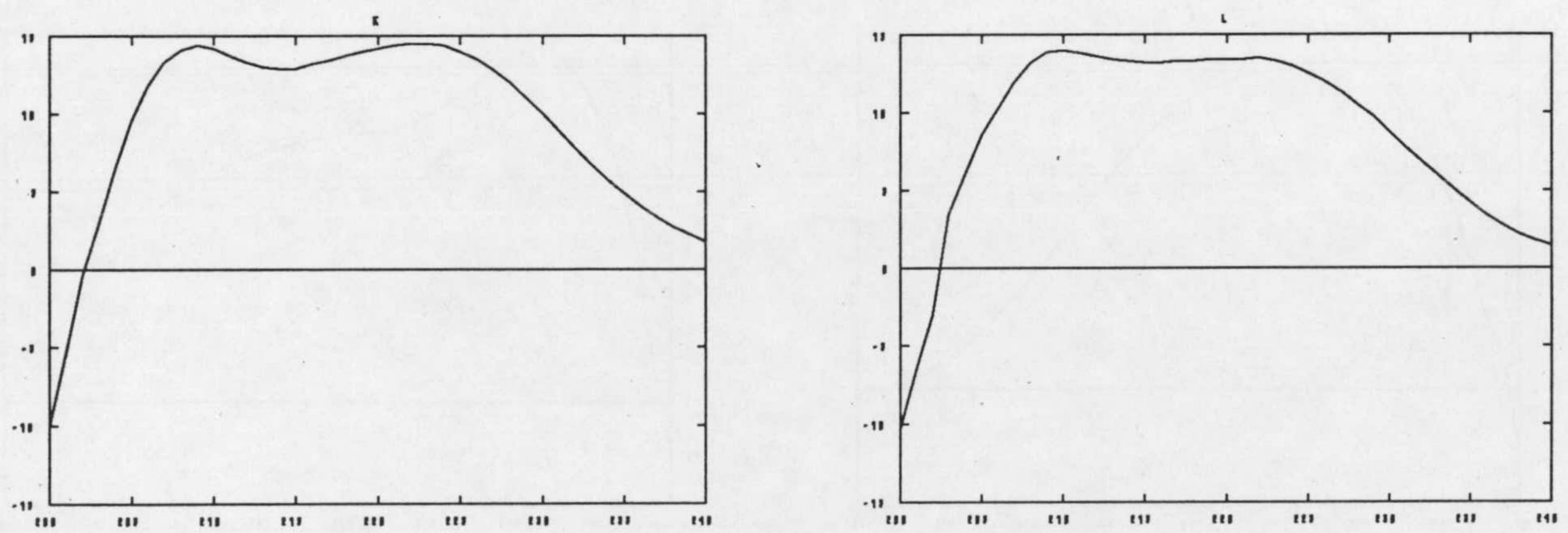
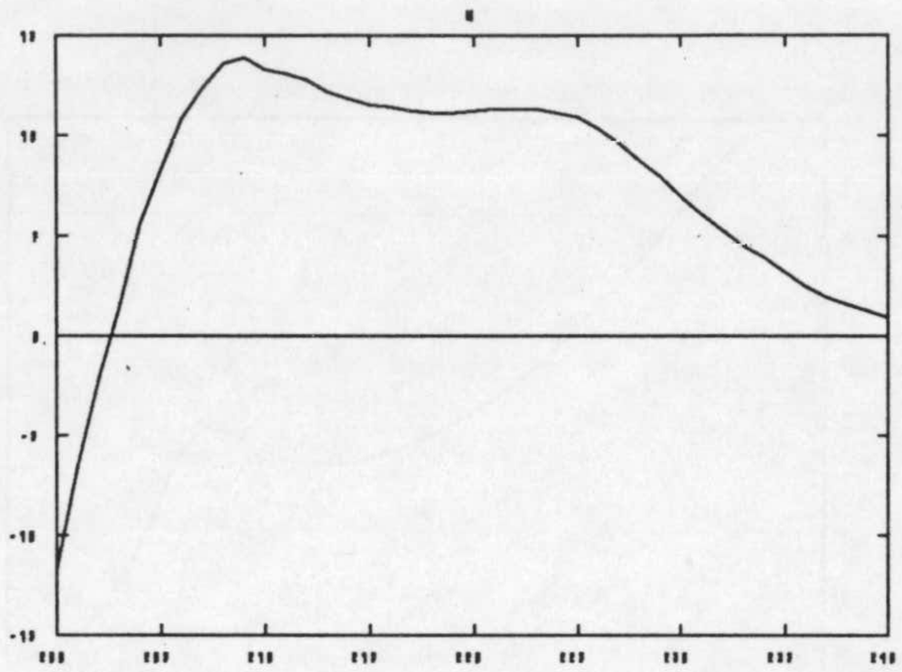


Figura 4.2 i) y j) : espectro de dicroismo circular de las proteínas Parv y AdnK.



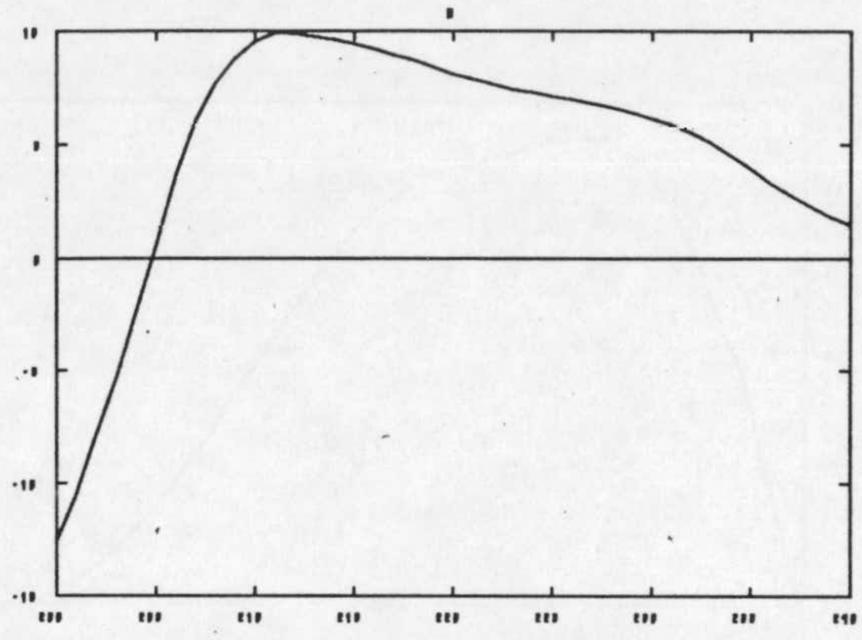
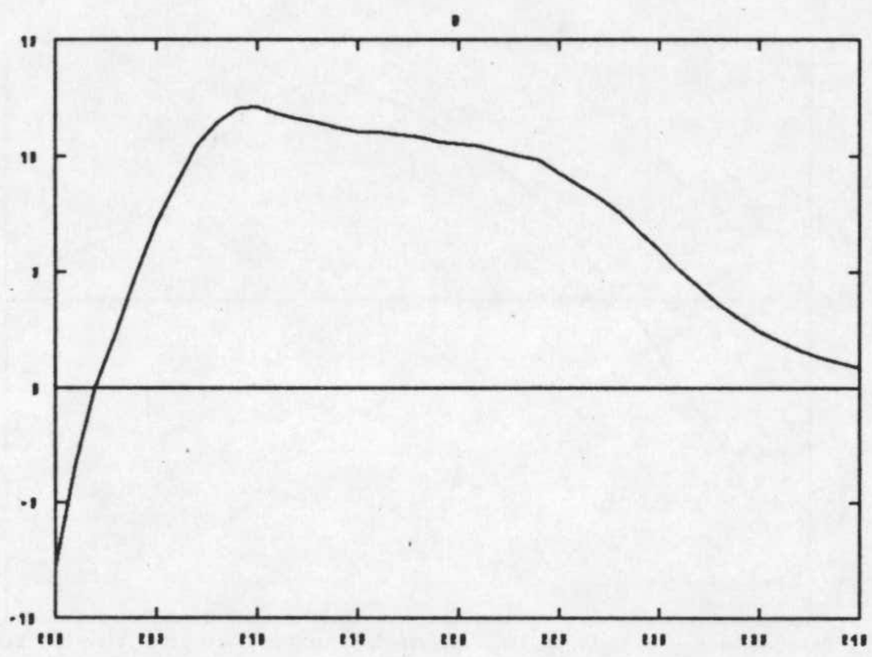


Figura 4.2 k) y l) : espectro de dicroismo circular de las proteínas Insu y CrbA.



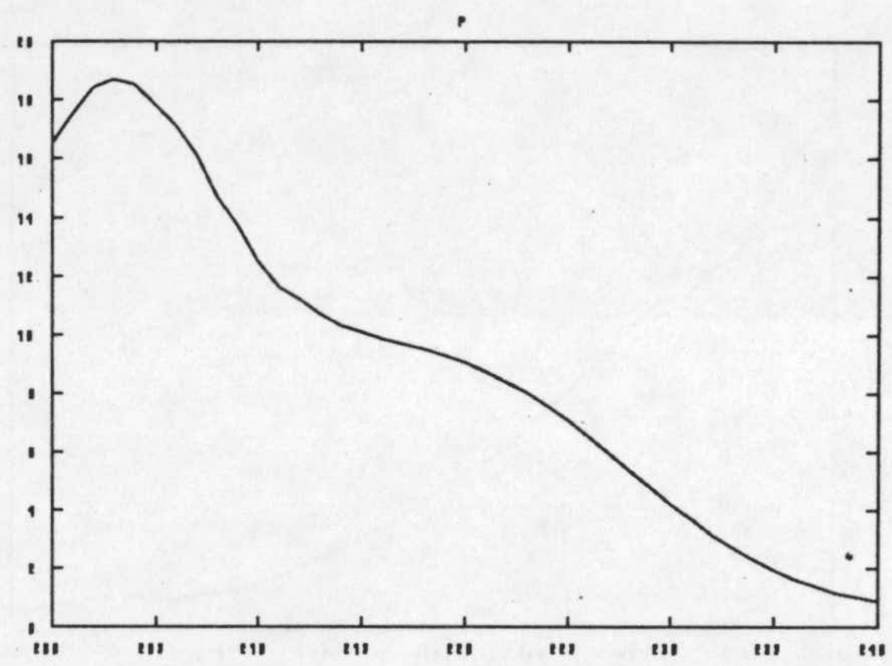
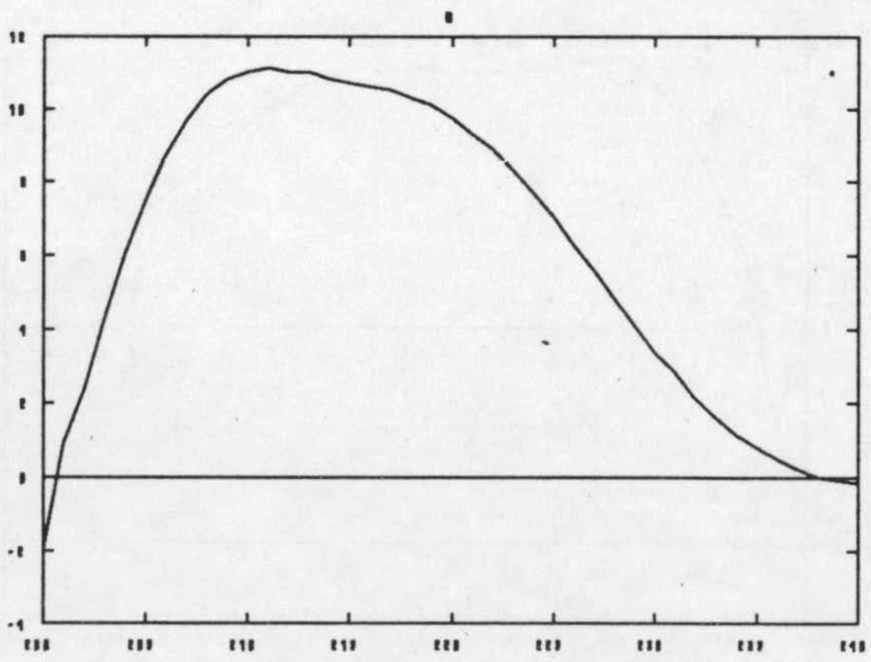


Figura 4.2 m) y n): espectro de dicroismo circular de las proteínas Thrm y Thryl.



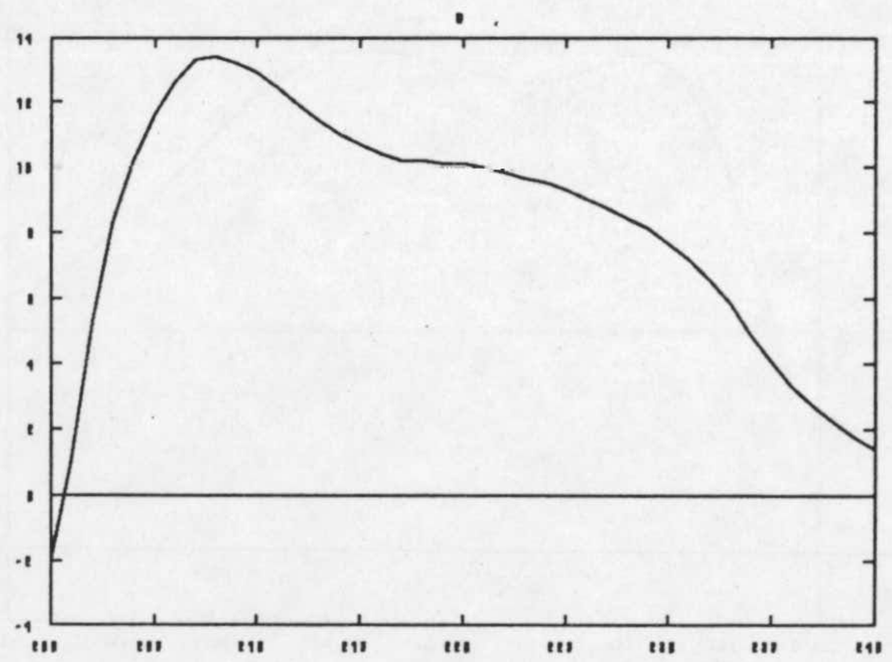


Figura 4.2 o) y p) : espectro de dicroismo circular de las proteínas RnsS, Nucl.

El SOM tiene una celda básica con forma de cuadrado (es decir, cada neurona está rodeada por otras 8 vecinas); la distancia utilizada es la euclídea, y se probaron varias combinaciones de parámetros hasta dar con la combinación más adecuada, es decir, aquella que posiblemente dará la distorsión mínima (y decimos posiblemente ya que este método no garantiza la consecución de un máximo global).

### 4.4.2 Optimización de parámetros

En este experimento se llevó a cabo una optimización básica, que consistió en lo siguiente: dejando fijos todos los demás, e inicializando varias veces con pesos aleatorios, se probó para diferentes valores de varios parámetros, y se halló para cuál de ellos la distorsión era consistentemente mínima. Este procedimiento tiene la suposición implícita de que la superficie de error es más o menos suave, y que además sólo depende de un parámetro, lo cual es probablemente erróneo. Sin embargo, los resultados obtenidos son relativamente buenos.

Hay que tener en cuenta además otro factor. En el capítulo anterior (Capítulo 3) se ha supuesto que la distribución de entrada era continua, o, en cualquier caso, el número de patrones del conjunto de entrenamiento era mucho mayor que el número de neuronas. En este caso no es cierto, ya que el número de patrones es alrededor de 20, mientras que el número de neuronas es de un orden superior, alrededor de 100.

Sin embargo, el procedimiento de entrenamiento se puede también reducir a un procedimiento de cuantización vectorial, pero realizado a la inversa: el adiestramiento de la red procurará que cada uno de los vectores de entrada sea centro de un cluster formado por varios vectores peso de la red. Es decir, en este caso no se trata de diseñar un diccionario cuya distorsión para el conjunto de entrenamiento sea mínima, sino que, dado que el conjunto de entrenamiento es de cardinalidad menor que el diccionario, se trata de crear un conjunto de entrenamiento (con los pesos) cuya distorsión respecto al diccionario formado por las proteínas sea mínima

La distorsión, por tanto, habrá que medirla de otra forma [Mer94]; simplemente teniendo en cuenta que el diccionario será la muestra y el *conjunto de entrenamiento* será

en este caso los vectores pesos del SOM. Por tanto, proponemos utilizar como distorsión

$$D^* = \sum_j \sum_k d(\vec{w}_{jk}, \vec{x})$$

donde $\vec{x}$, en este caso, es el vector del conjunto de entrenamiento más cercano a $\vec{w}_{jk}$, es decir,

$$\vec{x} = \min_{x \in s} d(\vec{w}_{jk}, \vec{x})$$

Los parámetros que se pueden variar son los siguientes

-$k_1$ *and* $k_3$, los dos parámetros que determinan la pendiente del factor de ganancia (Sección 2.3).

-El *tiempo* (es decir, número de iteraciones) dedicados a la sintonización del MAK, como un porcentaje del tiempo total dedicado al entrenamiento. Oscila entre un 10%y un 20%.

-El *radio máximo inicial* de la vecindad; que oscila entre la red completa y la mitad de la red. En el primer caso, todos los pesos cambiarían de valor; en el segundo caso, solo lo harían la mayoría de las veces durante la primera época del entrenamiento (en caso de que la neurona esté cerca del centro, la vecindad incluirá toda la red).

-El *número total de iteraciones dedicadas al entrenamiento*, es decir, el número de veces que se presenta la muestra completa, que es usualmente mayor que 10, y no es necesario que sea mayor que 50.

Durante los experimentos, se nota que el resultado final (es decir, la distorsión), no depende demasiado de los pesos iniciales (siempre que se inicialize la red con valores aleatorios pequeños); y tampoco depende de los parámetros $k_i$, siempre que el factor de ganancia sea estrictamente descendente, y $k_1 > k_3$. Sin embargo, sí depende de los demás factores.

El resultado o mapa final depende fuertemente del tiempo dedicado a autoorganización, y del radio al inicio del entrenamiento, como se puede ver en la Figura 4.3. En esta figura se representan los puntos experimentales correspondientes a la repetición del experimento con diferentes pesos aleatorios iniciales; la curva recorre las medias de esos puntos experimentales. Se ve con claridad la dependencia de la distorsión final con el tiempo dedicado a sintonización del SOM. La distorsión alcanza un mínimo consistente si se dedica el 10% del tiempo de entrenamiento a sintonización del mapa.

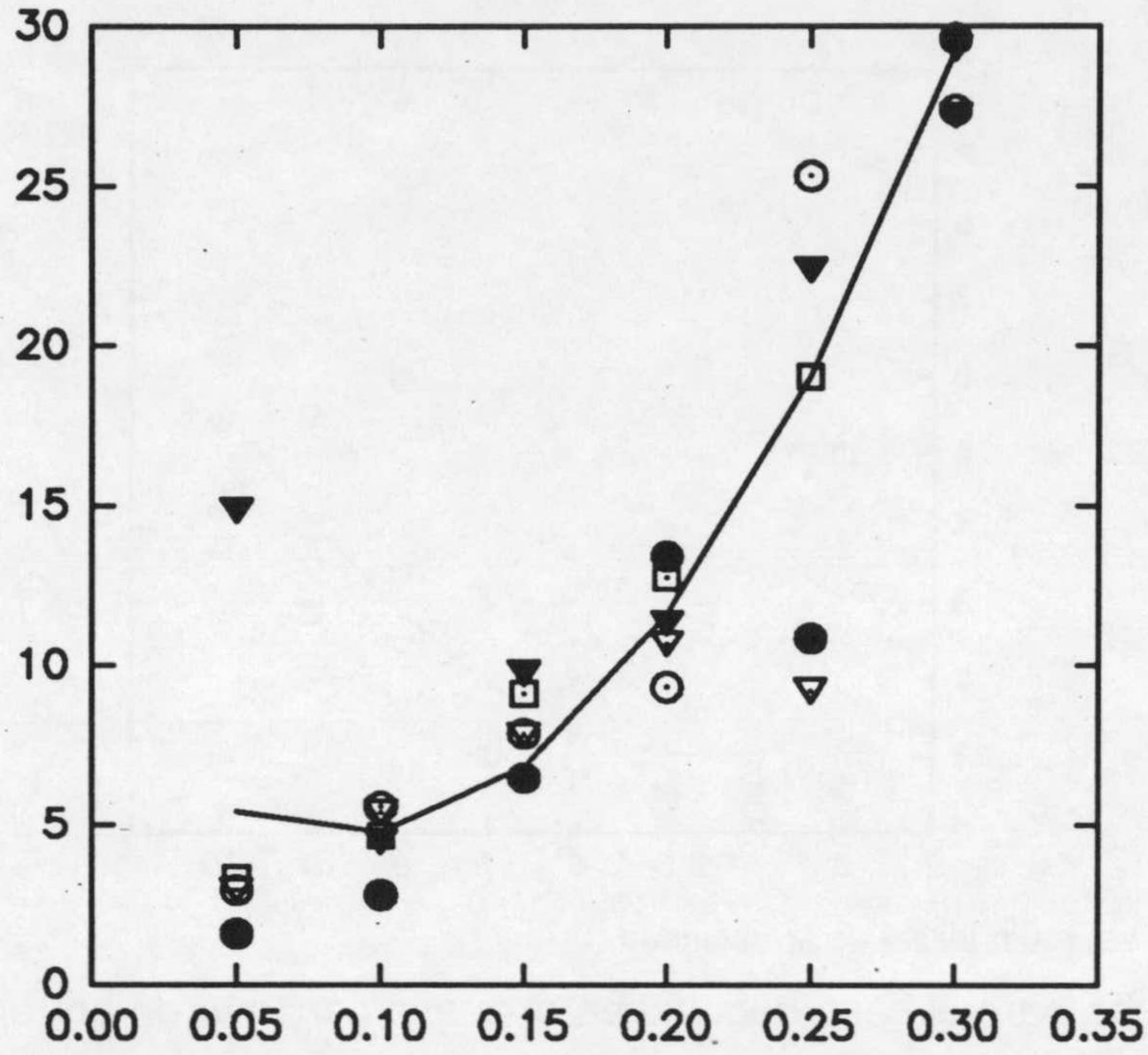



**Figura 4.3** Distorsión vs. proporción del tiempo dedicado a sintonización.

De la misma forma, se puede hallar un mínimo en la distorsión haciendo variar el radio inicial de la vecindad, como se muestra en la Figura 4.4. La curva corresponde a un ajuste de mínimos cuadrados para todos los valores obtenidos; se concluye que el mejor resultado corresponde a un radio igual a 4-6, lo cual equivale a la mitad del tamaño de la red.

Evidentemente, un número mayor de iteraciones supone una distorsión mucho menor; sin embargo, en este caso una distorsión muy pequeña puede no ser aconsejable ya que el mapa perdería la capacidad de generalizar.ç Con 20 iteraciones o épocas, se obtienen resultados suficientemente buenos.

Como ya se ha indicado, en este caso se trata de una optimización bastante rudimentaria. Se trata de explorar un espacio de muchas dimensiones y de gran tamaño, muestreando solamente unos cuantos puntos. De esta forma, se pueden obtener buenos resultados, pero no se puede asegurar que sean los mejores. Deberá, por tanto, de buscarse otro método que sea capaz de obtener los mejores resultados en todos los casos, como el que se presentará en capítulos posteriores.

Aunque el método con el que se han hallado los resultados sí es válido para la optimización de cualquier mapa (aunque no es el mejor método, como se verá), los resultados

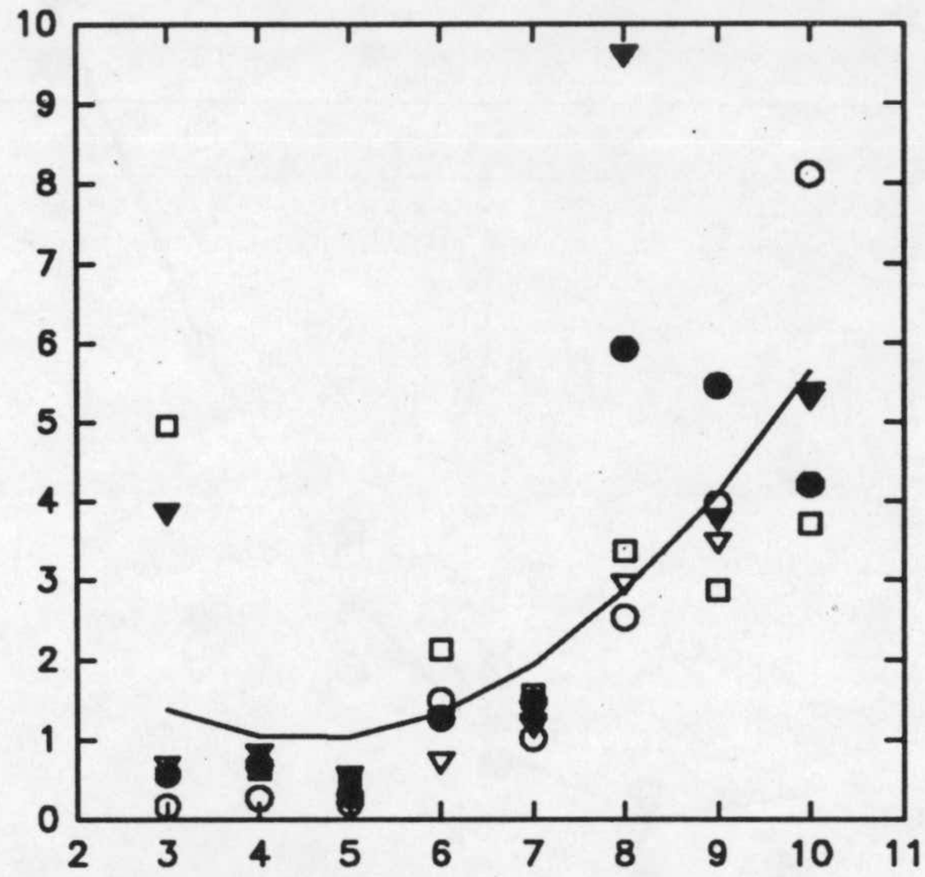



**Figura 4.4** Distorsión vs. radio inicial de la vecindad.

pueden variar mucho para el caso habitual en el que el conjunto de entrenamiento tiene más muestras que neuronas en el mapa.

### 4.4.3 Resultados

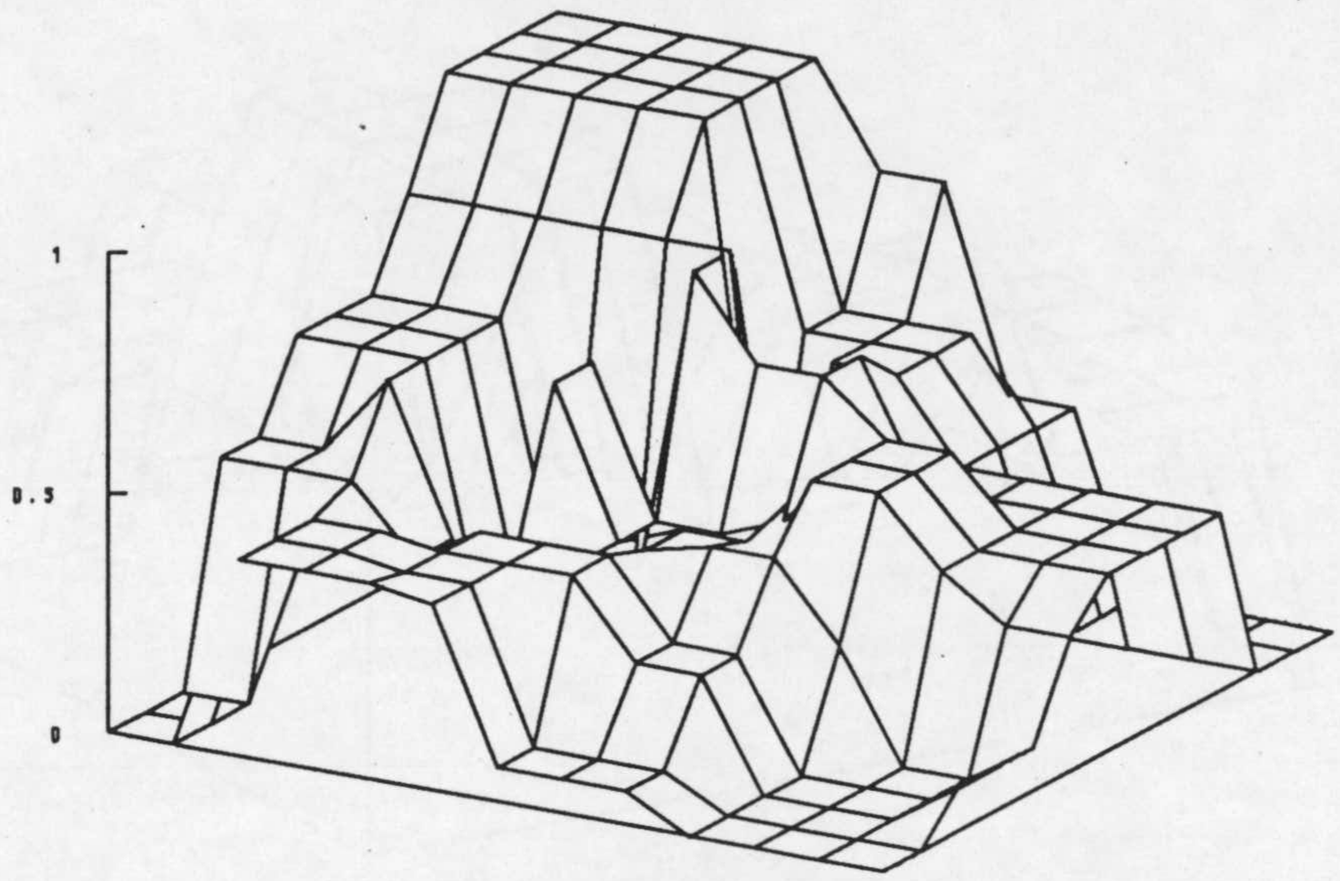



**Figura 4.5** Mapa proteinotópico. Las coordenadas horizontales corresponden a los índices de las neuronas y la vertical al porcentaje de estructura desordenada.

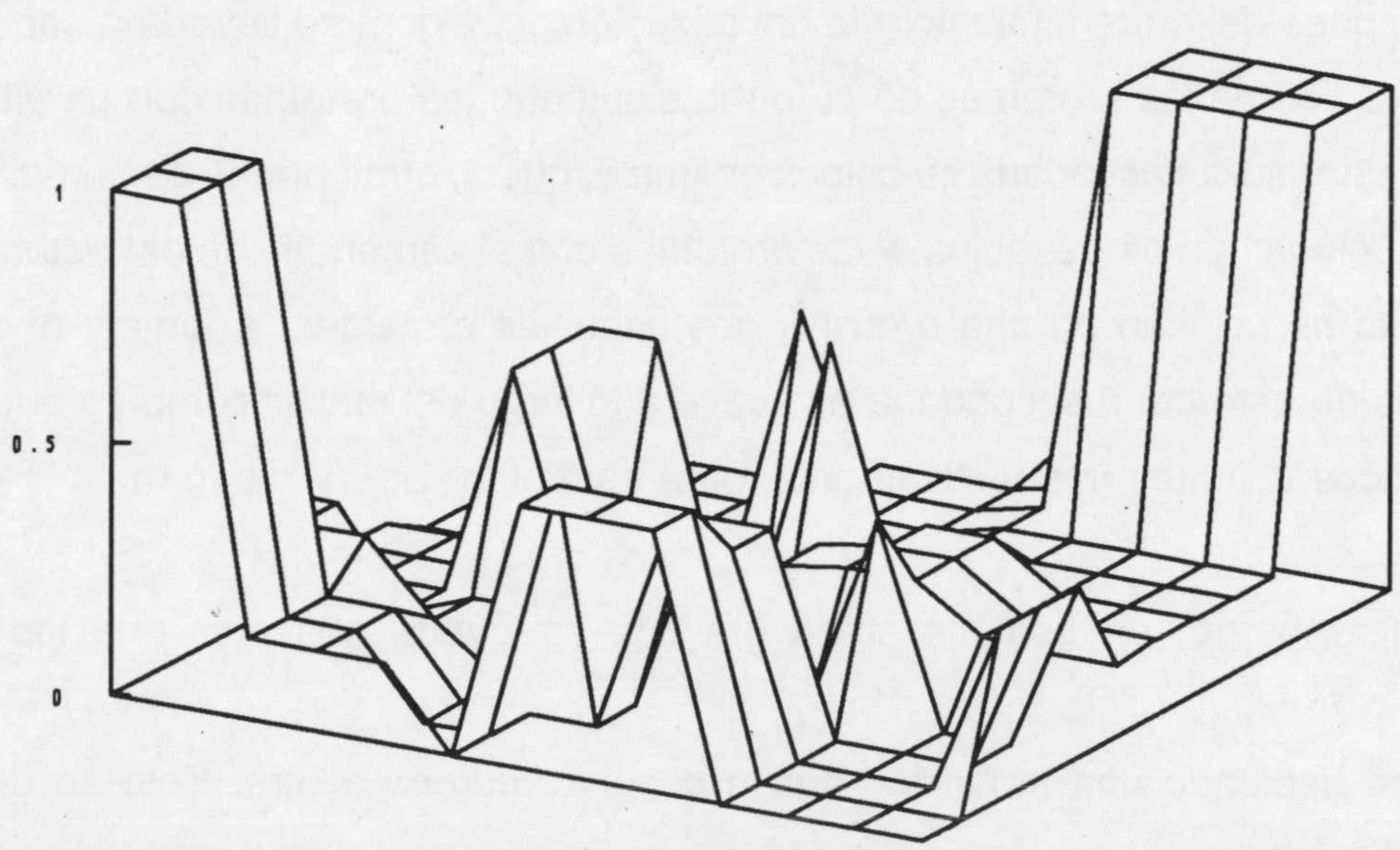



**Figura 4.6** Mapa proteinotópico. Las coordenadas horizontales corresponden a los índices de las neuronas y la vertical al porcentaje de estructura β.

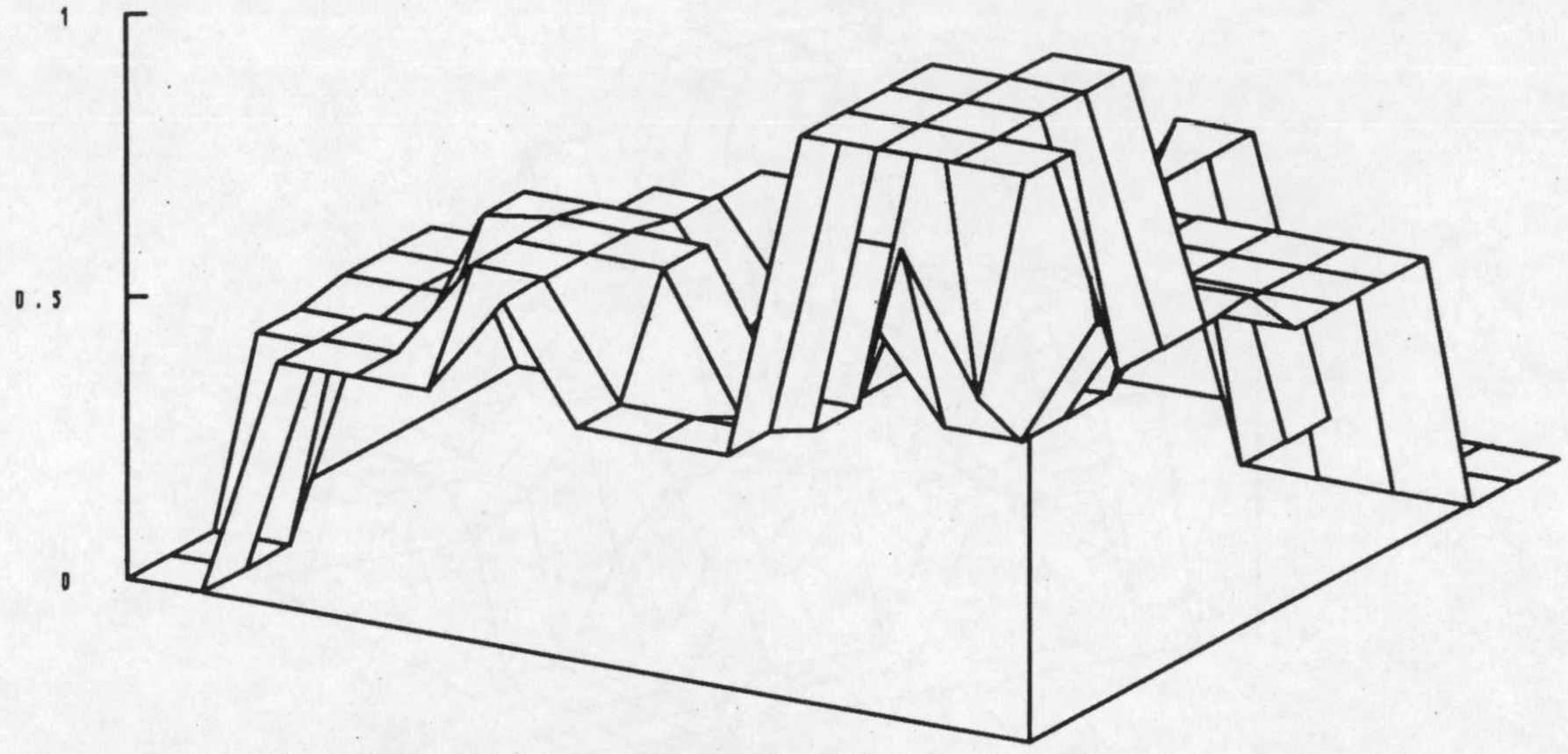



**Figura 4.7** Mapa proteinotópico. Las coordenadas horizontales corresponden a los índices de las neuronas y la vertical al porcentaje de estructura α.

Después del entrenamiento y la optimización del algoritmo llevada a cabo, se obtiene una clasificación de las proteínas de la forma siguiente: las proteínas con un alto porcentaje de alguna estructura secundaria se coloca en una esquina, otras proteínas con alto porcentaje de otra se colocan en otra esquina, y las proteínas con el porcentaje de estructura secundaria restante alto se colocan en una o en las dos esquinas restantes. Además, el descenso de porcentajes de estructura secundaria es suave a lo largo del mapa, como se puede observar en los gráficos adjuntos (respectivamente, para estructura α, β y random).

Sin embargo, en estos mismos gráficos se puede observar que los mapas son bastante

rugosos, no habiendo una transición suave de una neurona a otra. Esto se debe a varios factores: el primero, que en el conjunto de entrenamiento no existen proteínas para todos los valores de los procentajes de estructura secundaria posibles; el segundo, evidentemente, a que la clasificación no es perfecta (es decir, se puede haber alcanzado un estado en el que haya una ordenación no global).

La rugosidad del mapa se se puede evitar con una suavización del mapa (mediante una interpolación, por ejemplo); teniendo en cuenta, para cada vector peso, no sólo aquél

vector del conjunto de entrenamiento que está más cercano, sino también el segundo más cercano. De esta forma se obtiene un mapa de variación mucho más suave, y en el que, además, cada neurona tiene asignado un terceto de porcentajes particular (porque hasta ahora, sólo hay 21 tercetos posibles). A partir de aquí, ya se puede proceder al cálculo de porcentajes para proteínas desconocidas [And93].

## 4.5 Conclusiones

En este capítulo se ha presentado una aplicación novedosa del mapa autoorganizativo de Kohonen, que, en principio, no tiene nada que ver con su cometido principal de cuantización vectorial. En este caso se utilizan más las características de aplicación ordenada de un espacio de entrada a otro de salida del SOM para hallar lo que se ha denominado un mapa proteinotópico. Este mapa proteinotópico mejora la exactitud en el cálculo de los porcentajes de estructura secundaria, y utiliza para hallarlos un método más simple que otros métodos neuronales revisados aquí.

Esta aplicación nos da la oportunidad de acercarnos por primera vez al problema de la optimización de redes neuronales mediante algoritmos genéticos, mediante un enfoque que se puede denominar *sistemático*, y que permite optimizar, dentro de una serie de limitaciones, los resultados de distorsión obtenidos. Es asimismo destacable la elección de la distorsión como indicador de la bondad del ajuste del algoritmo, que se utilizará más adelante en métodos de optimización más potentes.

# 5 Algoritmos evolutivos para optimización de redes neuronales

## 5.1 Introducción

El algoritmo LVQ y el SOM, presentados en el Capítulo 2, se han probado en gran cantidad de aplicaciones, dando buenos resultados[Koh92a]. Sin embargo, el método de diseño del diccionario no garantiza que estos resultados sean los mejores para cada problema, y, además, su implementación y ejecución presentan una serie de problemas. *Primero*, como sucede en cualquier red neuronal artificial, es necesario establecer de antemano una serie de parámetros (en este caso, constante de aprendizaje $\alpha$ y su función de decrecimiento con el tiempo[1]); *segundo*, en el caso del algoritmo LVQ es necesario calcular los pesos iniciales mediante algún otro método de cuantización vectorial, o bien calcularlos aleatoriamente o mediante alguna regla heurística; y *por último*, el número de vectores diccionario es fijo, y ha de establecerse de forma que éstos sean suficientes para llevar a cabo la tarea de clasificación que se quiera encomendar al algoritmo.

El problema de optimización del algoritmo LVQ se puede situar dentro del problema más general de optimización de una red neuronal artificial, del cual se ha visto una primera aproximación en el capítulo 4. Los elementos de una red neuronal artificial son

- los diversos parámetros que afectan el aprendizaje: constantes de aprendizaje o ganancia, radio inicial de la vecindad en el caso de la red de Kohonen, amplitud de la zona de proyección de una neurona a la capa siguiente,
- la conectividad de la red, es decir, la existencia o no de una conexión de una neurona con otra y su valor,
- el algoritmo de aprendizaje en sí, es decir, cómo varían los pesos en función de sus valores anteriores, el tiempo de entrenamiento, y sus valores pre y post-sinápticos, y
- el número de neuronas.

---

[1] Aunque este término es poco susceptible de optimización.

Cualquiera de estos elementos o todos juntos son susceptibles de optimización. Una buena revisión de la optimización de redes neuronales artificiales mediante algoritmos genéticos se puede encontrar en [Yao92].

En general, estos intentos de optimización se encuadran dentro de dos grandes grupos: mediante algoritmos genéticos y mediante métodos constructivos/destructivos, también denominados evolutivos. Los primeros tratan de aplicar un algoritmo genético a uno o varios de los elementos que componen una red neuronal, logrando resultados óptimos [Mon93], [Harp91]. La mayoría de estos algoritmos utilizan genomas con una longitud fija, o una longitud máxima implícita [Mon93], y por lo tanto, no pueden tener en cuenta todos los casos posibles. En general, sin embargo, dan buenos resultados, aunque no son muy abundantes las aplicaciones a algoritmos de cuantización vectorial [Das93][Mon93][Alp93].

Los algoritmos constructivos o destructivos, genéricamente denominados *evolutivos*, generalmente se concentran en la optimización del número de neuronas (o de vectores diccionario) de la red. Habitualmente determinan de forma continua o tras la presentación de una sola o de un conjunto de muestras si se debe añadir o no un vector (o más) a la representación, siguiendo algún criterio de optimalidad. De la misma forma deciden si se debe o no eliminar un vector o neurona. Todos estos métodos incrementales presentan un problema: no garantizan una solución óptima, ya que se basan en alguna regla heurística, o en el establecimiento de algún umbral superado el cual, se elimina o añade un nuevo vector o neurona. Por tanto, no exploran eficientemente todo el espacio de soluciones posibles, en otras palabras, no garantizan una exploración óptima de todo el conjunto de soluciones posibles.

El método presentado en esta memoria aúna las ventajas del LVQ, a saber, velocidad, simplicidad conceptual y de programación, con las prestaciones de los algoritmos genéticos cuya eficacia está garantizada por el teorema de los esquemas. El algoritmo G-LVQ incluye, además, un espacio ilimitado de posibles soluciones, puesto que no se limita el número de vectores diccionario; este espacio se explora de forma eficiente, gracias a los operadores genéticos que actúan sobre la población de diccionarios y sobre cada diccionario.

En la primera sección de este capítulo (5.2) se hablará de los enfoques evolutivos al diseño de redes neuronales, incidiendo una vez más en los aplicados a algoritmos de

cuantización vectorial. La sección 5.3 está dedicado a hacer una breve exposición de los algoritmos genéticos. En la sección siguiente (5.4) se hará referencia a cómo diversos autores han optimizado redes neuronales utilizando algoritmos genéticos, haciendo especial hincapié en la optimización de algoritmos de cuantización vectorial. Se introducirá un nuevo algoritmo de cuantización vectorial, denominado G-LVQ (sección 5.5), que combina aprendizaje neuronal con algoritmos genéticos en el apartado 5.5. En la sección siguiente (5.6) se compara el enfoque de este algoritmo con otros algoritmos similares, para terminar examinando en más detalle el algoritmo G-LVQ, con inclusión de los parámetros necesarios para su ejecución en la sección 5.7.

## 5.2 Algoritmos neuronales evolutivos

Estos algoritmos tienen el gran atractivo de ser capaces de eliminar las neuronas o pesos no necesarios para la red; así como poder calcular dinámicamente las neuronas y pesos correspondientes necesarios. De entre ellos, el más famoso y utilizado es el *Cascade Correlation*, de Fahlman y Lebière [Fahl90], que se utiliza para calcular dinámicamente el número de neuronas de la capa oculta de un perceptrón multicapa entrenado con retropropagación. La mayoría de los esquemas evolutivos se aplican a este algoritmo, y sólo unos cuantos [Das93][Alp91][Harp92] se aplican a algoritmos de cuantización vectorial y redes de Kohonen.

En general, los algoritmos evolutivos se pueden clasificar en dos grandes grupos [Alp91]: aquellos que empiezan con una red pequeña y van añadiendo unidades (de los que se trata en el apartado 5.2.1), y aquellos que comienzan con una red grande, procedente, por ejemplo, de una suposición sobre la distribución de las muestras de entrada o de un entrenamiento anterior, y van eliminando unidades (examinados en el apartado 5.2.2).

### 5.2.1 Algoritmos incrementales

Estos algoritmos comienzan con una red con un pequeño número de neuronas, y van añadiendo otras nuevas hasta que el error alcanza el nivel requerido. Generalmente, añaden unidades cuando alguna unidad previa se activa más de un número determinado de veces,

lo cual significa que la zona del espacio a la cual corresponde está insuficientemente representada en la red neuronal. Otras veces, se añaden agrupaciones de neuronas o subredes cada vez que el error comienza a disminuir más lentamente.

En algunos casos, añadir una neurona nueva representa un problema, pues puede hacer *olvidar* al resto de las neuronas lo que han aprendido; sobre todo, en aquellos algoritmos de aprendizaje *cooperativo*, en el cual cada vez que se introduce un nuevo patrón cambian todos los pesos. Para ello, habitualmente se entrena sólo la nueva neurona, sin alterar los pesos de las demás.

El problema principal que presenta este enfoque es el de crear una red demasiado adaptada al conjunto de entrenamiento, análogo al problema denominado *overfitting* o sobreajuste, que se produce cuando se trata de ajustar una curva a puntos experimentales. El efecto de este *overfitting* es la mala generalización a resultados no incluidos en el conjunto de entrenamiento; y, por tanto, malos resultados con el conjunto de test. Puede decirse que la red se ha limitado a memorizar los patrones de entrenamiento, y no a abstraer o a generalizar a partir de ellos, perdiendo generalidad. Un ejemplo de sobreajuste en el caso del ajuste mediante una curva a un conjunto de puntos se muestra en la Figura 5.1.

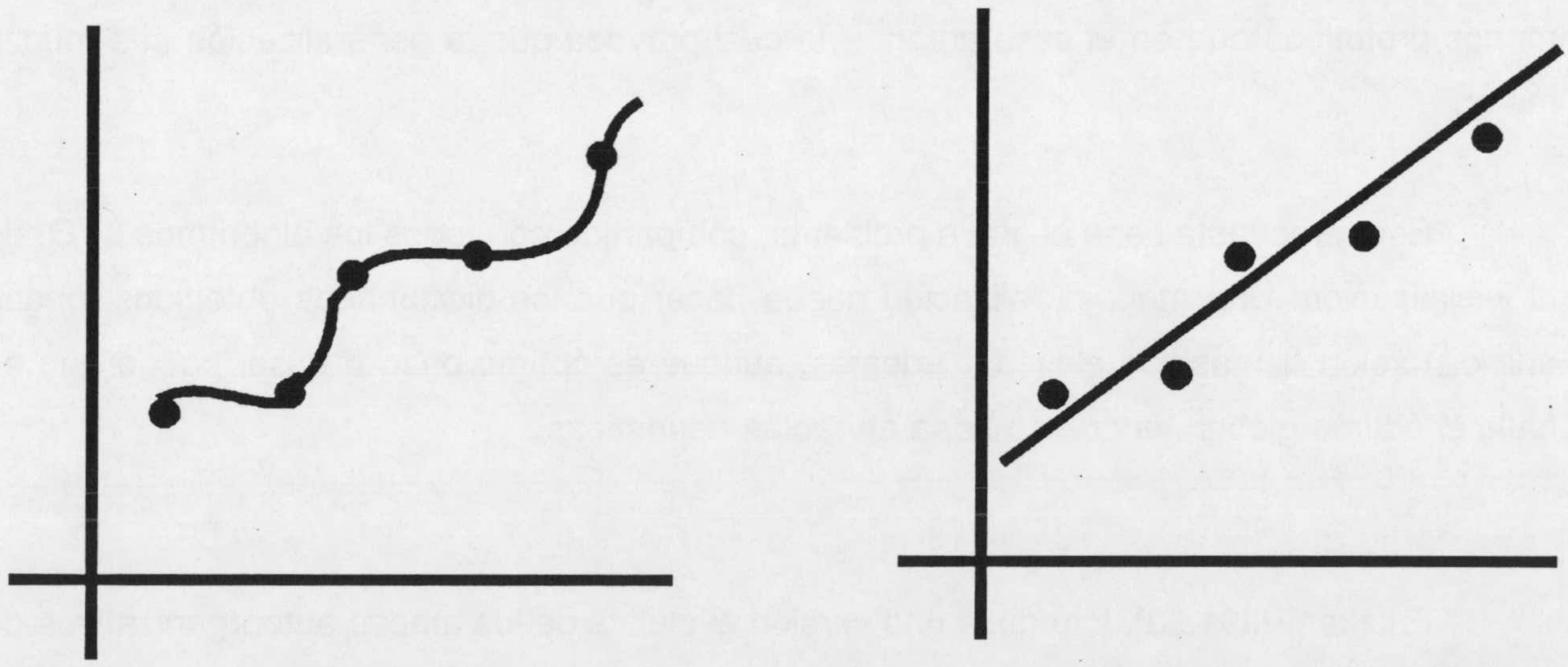


**Figura 5.1** Efectos del sobreajuste de una curva a un conjunto de puntos (izqda); ajuste correcto en la derecha.

Otro problema evidente de este tipo de algoritmos es el crecimiento desmesurado del número de neuronas, hasta el punto que puede resultar poco práctica la implementación software o *hardware* de la red neuronal resultante. Por ello, habitualmente estos algoritmos

van acompañados de otra serie de criterios para eliminar neuronas, como sucede en los algoritmos que veremos en el siguiente apartado.

El enfoque incremental suele ser el preferido en el caso de LVQ o cualquier otro algoritmo de cuantización vectorial [Per93][Poi91][Alp91]. Generalmente se empieza con un número pequeño de vectores peso, y se van añadiendo otros nuevos cada vez que se cumple una condición determinada. En [Poi91] se añade un nuevo vector peso cada vez que una muestra $\vec{x}$ se encuentra mal representada por los vectores peso existentes; es decir, la clase a la que representa $\vec{w}_i=Q(\vec{x})$, no es la misma que la clase a la que pertenece $\vec{x}$, y además el vector peso más cercano está más lejano que un umbral determinado. Si no, se aplica simplemente la regla LVQ.

En [Per93], sin embargo, la evaluación de la necesidad de un nuevo vector peso se realiza sólo después de la presentación del conjunto completo de muestras. Como nuevo vector peso se elige aquella muestra que ha sido clasificada incorrectamente después de una presentación completa, y además es la más lejana de cualquier vector peso perteneciente a la misma clase. También se eliminan vectores peso en el caso de que no hayan sido los más cercanos de ningún vector muestra perteneciente a su clase. En este caso se producen menos prototipos que en el caso anterior, lo cual provoca que la generalización sea mucho mejor.

Esta propuesta tiene el grave problema, compartido con todos los algoritmos LVQ, de la inicialización. Una mala inicialización puede hacer que los diccionarios obtenidos tengan una distorsión demasiado elevada. Además, aunque es óptimo paso a paso, puede que no halle el óptimo global, ya que se basa en reglas heurísticas.

Fritzke [Frit91,93], introduce una versión evolutiva de los mapas autoorganizativos de Kohonen, en el cual se introduce un nuevo vector peso cada vez que el error de alguno de los vectores peso sobrepasa un umbral. El nuevo vector peso se sitúa entre el vector con más error (llamado *oveja negra*), y el vector peso más lejano de los de su vecindad. Esto provoca un cambio en la topología de la red, de forma que la vecindad se define de forma dinámica y sobre el espacio de entrada, no sobre el de salida es decir, los índices asociados

*a priori* a cada vector peso. Este algoritmo está más enfocado a reproducir densidades de probabilidad del espacio de entrada, que a crear diccionarios eficientes; en general crea diccionarios de tamaño excesivamente grande.

### 5.2.2 Algoritmos decrementales

Los algoritmos decrementales se basan en la eliminación de aquellos pesos y/o neuronas que menos contribuyen a la disminución del error; o bien aquellos que están situados en una zona del espacio en la cual la densidad de probabilidad de las muestras de entrada es casi nula o nula, es decir, aquellos vectores diccionario que están lejos de las zonas donde se sitúan los vectores de entrada.

Para decidir qué conexión o neurona eliminar, habitualmente se calcula la variación del error con la presencia o ausencia de la misma. En el caso de las redes neuronales multicapa, esta comprobación se puede realizar al final del entrenamiento, eliminando aquellas conexiones cuyo valor sea prácticamente nulo. Otra alternativa consiste en calcular durante el entrenamiento la derivada del error con respecto a la variación de cada uno de los pesos, y la consiguiente eliminación de aquellos pesos cuya variación no influya en el error.

En el caso de los algoritmos de cuantización vectorial, se elimina un vector peso siempre que no esté cerca (es decir, no *gane*) para ninguna muestra de entrada; lo cual significa que la densidad de probabilidad en esa zona es 0. Esta mejora ya se ha indicado en [Per93], y es bastante trivial; sin embargo, una buena inicialización junto con un buen entrenamiento debían de ser capaces de logar un diccionario que respondiera suficientemente bien al espacio de entrada en general, sin que aparezca algún vector peso que no responda a ninguna zona del espacio de entrada. En [Frit91], se eliminan aquellos pesos que no han ganado durante un número determinado de presentaciones de la muestra.

En general, el enfoque decremental presenta el problema de que, al comenzar con un número grande de neuronas, son bastante ineficientes al principio. Como se ha visto en el apartado 5.2.1, lo más común es combinar ambos enfoques (incremental y decremental) en un algoritmo evolutivo, con criterios para la introducción y eliminación de conexiones.

## 5.3 Algoritmos genéticos.

Los algoritmos genéticos son métodos de optimización basados en los principios de la teoría de la Evolución de las especies de Darwin: selección natural, combinación genética y mutación [Gol89].

Para resolver un problema mediante un algoritmo genético, es necesario crear un modelo del problema, de forma que quede descrito por una serie de parámetros, y posteriormente codificar esos parámetros en un *gen*. Un gen contiene una representación binaria de todos los parámetros necesarios para resolver el problema. Inicialmente, se crea una población aleatoria de estos genes. En cada generación, se evalúa el gen decodificándolo y convirtiéndolo en los parámetros del problema, y se le asigna una puntuación o *fitness* dependiendo de lo que se acerque a la solución que se desea obtener. Por ejemplo, si el problema es hallar el máximo de la función $f(x) = x^2$, el gen codificaría una representación de precisión finita del parámetro x; al decodificarlo, su fitness sería $x^2$ [2].

Posteriormente, aquellos genomas con un *fitness* superior son combinados para generar la nueva población, eliminándose a la vez aquellos genomas con un *fitness* inferior. Esta combinación consiste en una mezcla de material genético mediante una operación denominada *crossover* o entrecruzamiento (Figura 5.2). Además, durante la reproducción, el nuevo genoma puede sufrir mutación (Figura 5.3) con una probabilidad pequeña, del orden del uno por ciento, cambiando un 0 en un 1 o viceversa.

---

[2] Este ejemplo se ha elegido sólo por su simplicidad; evidentemente, es un ejemplo en el cual otros métodos actuarían mucho mejor.

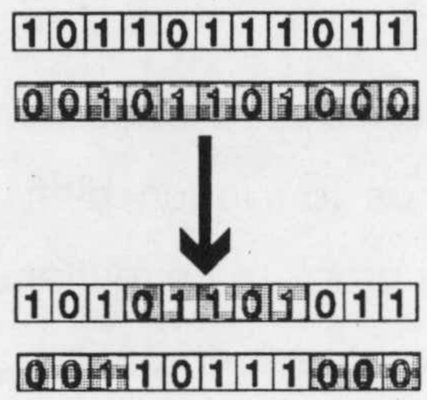
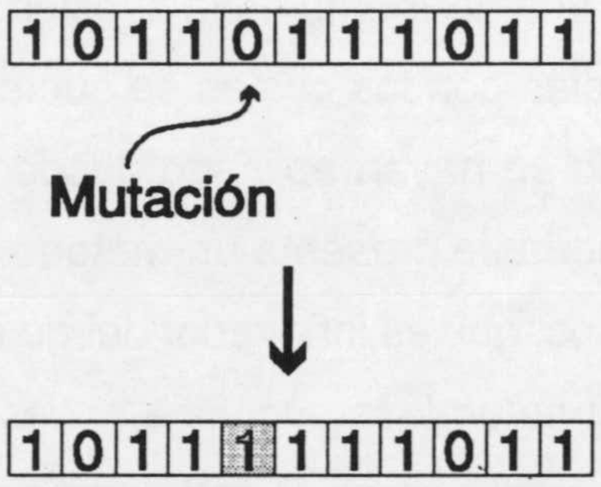



**Figura 5.2** Esquema de la operación del operador *crossover* sobre dos genomas.




Un algoritmo genético se puede expresar de la forma siguiente

**Algoritmo genético**

[G1] Inicializar la población inicial P(0).

[G2] Evaluar P[0].

[G3] Mientras que no se llegue a la última generación hacer

[G3.1] t←t +1;

[G3.2] Elegir los mejores miembros de la población para que se reproduzcan.

[G3.3] A partir de estos elementos de la población, crear nuevos miembros mediante un proceso de entrecruzamiento y aplicación de la mutación al genoma resultante. Estos elementos sustituirán a los peores elementos de la población.

[G3.4] Evaluar P[t].

El resultado matemático más potente que garantiza la eficacia de los algoritmos genéticos es el *teorema de los esquemas*, propuesto por John Holland [Hol75]. Este teorema demuestra que el número de genomas con un fitness superior a la media se incrementa en

cada generación, lo cual implica que la media del *fitness* va a ascender de generación en generación, hasta hallar un óptimo o llegar al final del entrenamiento.

Evidentemente, la eficacia de los algoritmos genéticos viene también apoyada por la experiencia. Los algoritmos genéticos funcionan bien en una amplia gama de problemas, hallando además varios óptimos del problema simultáneamente si se utilizan los operadores adecuados, de forma que se pueden utilizar en optimización multiobjetivo.

## 5.4 Algoritmos genéticos y redes neuronales

Como ya se ha indicado anteriormente en la sección 5.3, los algoritmos genéticos son métodos de optimización generales con los cuales se suelen obtener buenos resultados. No es de extrañar, por lo tanto, que se hayan aplicado desde muy pronto a la optimización de redes neuronales. En esta memoria se presenta un método original que optimiza el algoritmo LVQ mediante algoritmos genéticos, que es innovador desde el punto de vista de los algoritmos de cuantización vectorial, por su naturaleza evolutiva, y desde el punto de vista de las redes neuronales, pues combina el aprendizaje de la red neuronal con la adaptación aportada por el algoritmo genético. Igualmente incluye nuevos operadores genéticos, útiles para trabajar con genomas de longitud variable.

En general, se pueden optimizar todos y cada uno de los elementos de la red neuronal, pero los enfoques presentados más habitualmente por los autores suelen ser los siguientes [Yao92]:

*I)* *Cambio de pesos mediante algoritmos genéticos*: dada una red neuronal cuya estructura está fija, se pueden utilizar los algoritmos genéticos como únicos métodos de cambio de los valores de los pesos. Mediante este procedimiento, se puede hallar una combinación óptima de pesos para el problema tratado; aunque en la mayoría de los casos no es más eficaz que las distintas reglas de aprendizaje de redes neuronales [Whit89]. A veces se puede combinar esta técnica con la eliminación de pesos cuya contribución al error total sea poco significativa, como se hace en [McD93], aunque en este caso se utiliza *programación evolutiva*, una técnica similar

a los algoritmos genéticos.

II) *Cambio de estructura mediante algoritmos genéticos*: que a su vez reviste varias formas. Una de ellas consiste en la utilización de un *mapa de conexiones* (como utilizan Cliff *et al.* [Clif93], y Yaeger en *PolyWorld* [Yae93], así como otra serie de autores), de forma que el genoma describe cómo se tiene que conectar cada neurona con la siguiente, dando lugar a una red de estructura variable, que no tiene porqué estar necesariamente estructurada en capas. Otra forma supone la utilización de una descripción de la red ([Harp91]), que describe sus características generales: número de neuronas, densidad de la conectividad; y deja a un algoritmo de desarrollo especificar la matriz exacta de conexiones. Una última forma utiliza una *codificación gramatical*, utilizada por Gruau [Gru??], Kitano [Kit90] y otros, en la cual el genoma codifica una serie de reglas gramaticales, que indican cómo se tiene que construir y conectar la red en el momento del "nacimiento".

III) *Evolución de las reglas de aprendizaje*. Esta evolución puede ser tan simple como un cambio de los parámetros de aprendizaje (constante de ganancia del perceptrón multicapa, por ejemplo); o una optimización de la fórmula de cambio de pesos a partir de las entradas, salidas y los demás pesos de la red [Chal90],[Das93],[Ben91]. Este enfoque es el menos común.

En realidad, como se puede observar, los distintos enfoques son sólo intentos parciales de optimización de una red neuronal, incluso después de fijada la arquitectura, con lo cual no está garantizada la consecución de un óptimo global. Ciertas características quedan fijas durante la ejecución del algoritmo genético, por lo que, obviamente, éstas no se pueden optimizar.

Habitualmente, además, los algoritmos genéticos utilizan genomas de longitud fija, o de una longitud máxima implícita, con una parte de los mismos sin usar; lo cual limita el tamaño, o la estructura, de la red que se está intentando hallar. En algunos casos se utilizan genomas de longitud fija para dar lugar a estructuras neuronales de tamaño variable; los genomas especificarían reglas de conexión o bien una gramática que generara la red [Gru??]. El utilizar genomas de longitud variable, como hace el algoritmo propuesto en esta memoria, permite superar esta limitación.

El aprendizaje mediante algoritmos genéticos presenta características muy diferentes al aprendizaje mediante redes neuronales. Mientras que los algoritmos genéticos recorren de forma aleatoria el espacio de soluciones, centrándose mediante la selección y el *crossover* en aquellas zonas en los que el *fitness* es máximo, es decir, en la que las prestaciones de las soluciones son mayores, casi todas las redes neuronales artificiales son algoritmos de ascenso de gradiente, que iterativamente van haciendo disminuir el error o incrementando el *fitness* hasta llegar a un máximo. Esto hace que su búsqueda se concentre en una parte pequeña del espacio de soluciones, alrededor de la solución inicial; mientras que los algoritmos genéticos buscan por todo el espacio; siendo posible además en estos últimos que en una población, con las técnicas adecuadas, se encuentren soluciones que se sitúen cerca de varios máximos.

Por tanto, lo ideal es que el algoritmo genético sitúe a la red neuronal cerca de un máximo, para que mediante el entrenamiento neuronal ésta llegue rápida y eficazmente al mismo. Esta regla, ya enunciada por Belew [Bel90], es la utilizada en el algoritmo G-LVQ, que se propone en esta memoria.

## 5.5 Algoritmo G-LVQ

El objetivo principal de este algoritmo es la optimización total de una red neuronal de cuantización vectorial. Esta red neuronal puede utilizar aprendizaje supervisado o no, aunque en las pruebas mostradas en esta memoria se utiliza aprendizaje supervisado. Para ello, hay que resolver principalmente 3 problemas:

- ▸hallar el número de vectores peso óptimo, es decir, los niveles del diccionario,
- ▸asignar correctamente a los mismos las etiquetas de clase, en el caso del aprendizaje supervisado, que es el que se va a tratar,
- ▸hallar valores iniciales óptimos de los vectores peso antes de iniciar el entrenamiento mediante LVQ o SOM; en esta memoria se analizará el entrenamiento supervisado mediante LVQ.

Los algoritmos genéticos clásicos incluyen genomas de longitud fija [Gol89]. Esto supone un serio hándicap a la hora de llevar a cabo una tarea de clasificación, pues se limita

de antemano el tamaño del diccionario. En esta memoria se presenta un método para utilizar genomas de longitud variable, de forma que sea también posible evaluar la aportación de cada gen individual, que codifica un vector peso, a las prestaciones totales del diccionario, y hacerlo que se reproduzca o eliminarlo del diccionario dependiendo de esa aportación.

Como ya se ha indicado (sección 5.3, 5.4, 5.6), un algoritmo genético comienza con la codificación de las soluciones del problema en un cromosoma o genoma. En nuestro caso, estamos tratando de optimizar redes neuronales para cuantización vectorial, luego cada solución será un diccionario. Un diccionario consiste en una serie de vectores de *n*

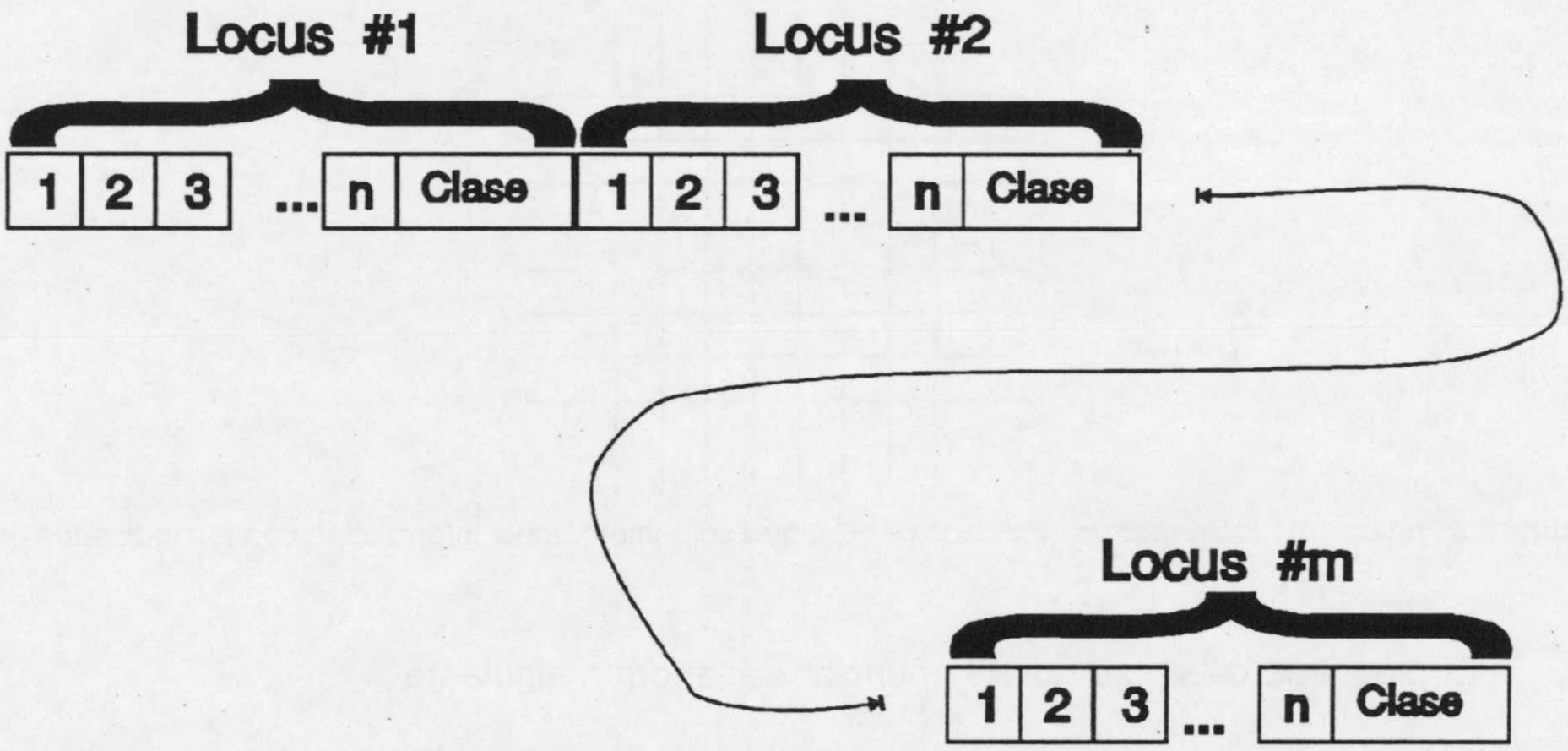



**Figura 5.4** Genoma de longitud variable utilizado en el algoritmo G-LVQ. Cada *locus* está compuesto de un vector con etiqueta.

componentes con una etiqueta de clase. Cada diccionario, por tanto, se codifica en un cromosoma (Figura 5.4). Un cromosoma está compuesto por diversos *loci*, cada uno de los *loci* constituye un vector peso del diccionario más una etiqueta.

A la hora de codificar cada componente del vector, hay que tener en cuenta una serie de consideraciones. Por un lado, tiene que tener la precisión suficiente como para que se explore el espacio de soluciones con exactitud; por otro lado, deben de suficientemente cortos como para que la búsqueda en tal espacio no dure demasiado. Por ello se ha considerado que cada componente del vector peso se represente mediante dos bytes, lo cual da un número de posibles soluciones adecuado ($2^{16}$) para cada componente; y una precisión suficiente, que luego se refinará mediante el entrenamiento LVQ. Por otro lado, cada etiqueta

de clase se codifica con un sólo byte. Para hallar la etiqueta de clase se divide este byte módulo el número de clases. Ello permite trabajar hasta con 256 clases diferentes, que, en los problemas abordados, han sido suficientes.Por lo tanto, cada cromosoma contiene

*(2x(#componentes de cada peso)+1)x#vectores del diccionario*

bytes, y, como se ha indicado, el número de vectores del diccionario varía para cada solución. Evidentemente, el número de componentes de cada peso es igual a la dimensión del espacio de entrada.

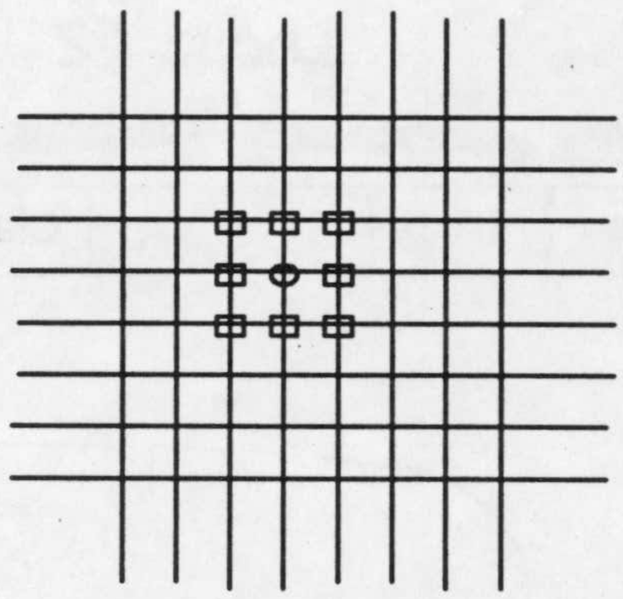


**Figura 5.5** Retícula de los genomas (indicado por O), que sólo intercambia información con los indicados por □.

El algoritmo G-LVQ se puede enunciar de la forma siguiente

**Algoritmo G-LVQ**

[G1] Generar una población inicial de tamaño $P$, con genomas de tamaño diferente.

[G2] Evaluar y generar la población siguiente durante $G$ generaciones

[G2.1] Calcular el fitness de cada uno de los genomas en una tarea de clasificación.

[G2.2] Comparar el fitness de cada elemento con el fitness de los $2^{c+1}$ vecinos, donde $c$ es la dimensión del hipercubo.

[G2.3] Si el fitness es el mayor de todos los vecinos, no hacer nada. Si no, sustituir el genoma por el producto del emparejamiento del genoma con mayor fitness de la vecindad y cualquier otro tomado aleatoriamente de la misma vecindad, aplicando los operadores genéticos de mutación y crossover.

[G2.4] Con una probabilidad dada, y para un diccionario elemento de la población determinado, calcular cual es el vector que más veces ha ganado en la ejecución anterior, y crear un nuevo vector peso idéntico al anterior salvo

mutaciones (duplicación). De la misma forma, evaluar cuál es el que menos ha ganado, y eliminarlo con una determinada probabilidad (eliminación).

Este algoritmo pretende resolver tres problemas fundamentales:

i) La optimización global del algoritmo LVQ, combinando una creación de vectores peso inicial con el entrenamiento mediante LVQ.

ii) La no restricción del tamaño del diccionario, lo cual le da características de proceso evolutivo abierto.

iii) Adaptación a la paralelización del algoritmo en una máquina con arquitectura de tipo hipercubo.

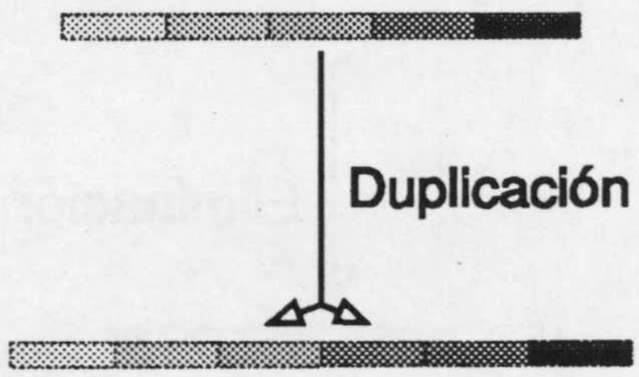



**Figura 5.6** Duplicación de un *locus* al cromosoma

Para trabajar con genomas de longitud variable, se introducen por primera vez en esta memoria dos operadores genéticos: *duplicación* (Figura 5.6), *eliminación* (Figura 5.7), e *incremento aleatorio*; con ello se pretende superar a los algoritmos incrementales y decrementales (examinados en las secciones 5.2, 5.3, 5.4, 5.6 y 5.3, 5.4, 5.6), hallando el tamaño correcto independientemente de cuál sea este.

Estos operadores hacen que el algoritmo genético utilizado en G-LVQ opere a dos niveles. El primer nivel actúa sobre la población de diccionarios, creando diccionarios que globalmente minimizan la distorsión, y que además tienen un nivel máximo de aciertos. Utilizando operadores genéticos clásicos: *crossover,* mutación y selección, se van hallando en generaciones sucesivas poblaciones de diccionarios con un nivel de aciertos creciente.

El segundo nivel actúa dentro de cada diccionario, cubriendo con más precisión las regiones en las que hay un sólo vector y la densidad de probabilidad es alta mediante la creación de un nuevo vector en esa área, con una cierta probabilidad, y la eliminación de

vectores en las áreas de densidad de probabilidad nula. En este caso los individuos de la población sobre la que actúa el algoritmo genético son los vectores del diccionario. El algoritmo genético actuaría en este caso sobre una población de tamaño variable, eliminando a elementos con un *fitness* mínimo, es decir, un número de aciertos mínimo, y que por tanto están lajos de la distribución de las muestras de entrenamiento, y dupalicando con mutación aquellos slementos cuyo fitness es alto, y que por tanto, están situados en una zona con una alta densidad de muestras de entrenamiento (alta densidad de probabilidad). El operador de incremento aleatorio introduce nuevos elementos en la población, sobre los cuales actuará, evidentemente, la selección y serán eliminados en caso de no tener un *fitness adecuado)*

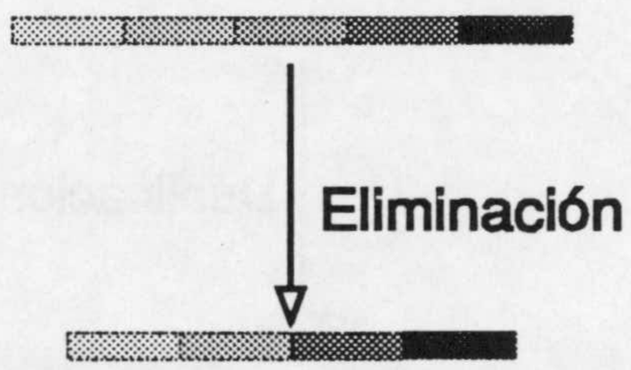



**Figura 5.7** Eliminación de un *locus* del genoma

Como se puede observar en los pasos G2.2 y G2.3, la selección y reproducción actúa sólo en un pequeño entorno definido alrededor de cada genoma (como se indica en la Figura 5.5). Habitualmente, la reducción de algoritmos genéticos a una población pequeña tiene un efecto nefasto sobre los resultados, denominado *inbreeding* o *consanguineidad*. Sin embargo, en este caso no se presenta el problema de consanguineidad, ya que sólo se restringe el emparejamiento *en cada generación*, porque en general los genomas pueden viajar por libertad por la retícula definida. De hecho, lo que se observa en las pruebas efectuadas sobre el algoritmo (presentadas en el capítulo 7) es que un aumento de fitness local *viaja* por la retícula, es decir, en sucesivas generaciones se observa que un aumento de *fitness* local se va propagando por toda la retícula hasta conseguir un aumento global del fitness.

Este algoritmo tiene en cuenta la posible implementación hardware en una arquitectura de tipo hipercubo, como se verá a continuación. Con el objeto de reducir al mínimo posible el tráfico entre procesadores, y teniendo en cuenta los consejos de Goldberg sobre el diseño

de algoritmos genéticos [Gol89b], todo el tráfico en la red de procesadores se realiza entre procesadores vecinos (Figura 5.5). Al reducir el tráfico de la red a procesadores vecinos, se reduce la cantidad de tiempo que el algoritmo tiene que emplear en intercambio de información. Si se conservara la forma habitual de un algoritmo genético, tendría que compararse el fitness de cada genoma con todos los demás, lo cual dificultaría bastante su implementación en paralelo.

Un esquema similar, en el cual los operadores genéticos se aplican sólo a una subpoblación forma por los vecinos en el retículo, ha sido ya presentado por Y. Davidor [Dav91]. La principal intención del mismo, sin embargo, es la preservación de la variabilidad genética, no la posible implementación en arquitecturas paralelas. Los resultados presentados indican que este esquema logra lo que pretende, preservar la variabilidad genética, y, además, consigue buenos resultados de convergencia global de la población. La principal diferencia del esquema de Davidor con el presentado en esta memoria es el método de selección utilizado, que en nuestro caso es elitista (es decir, que preserva siempre una parte de la población); en el caso de Davidor es estocástico.

## 5.6 Algoritmos genéticos y LVQ

Aunque los algoritmos genéticos han sido aplicados a LVQ [Mon93] y el SOM [Harp90], ninguno ha logrado resolver los tres problemas que resuelve el algoritmo G-LVQ, presentado en las secciones anteriores. En cualquier caso, la proyección de la población sobre una retícula y la consiguiente especialización del operador de selección, puede ser muy específica y enfocado a la implementación paralela, pero la no resolución de los dos primeros problemas, a saber, creación de un buen conjunto de pesos iniciales y búsqueda de la longitud correcta del diccionario, no garantiza encontrar una solución global.

En general, la regla de cambio de pesos (expuesta en el capítulo 2) no suele optimizarse. Algunos autores [Das93], sin embargo, tratan de encontrar una regla óptima de cambio para la red de Kohonen, resultando, como era de esperar y suele dictar la experiencia, que la regla original es ya bastante óptima, no justificando la variación de la misma o la utilización de otra. En cualquier caso, sólo se presenta el resultado de una sola

ejecución del algoritmo genético, sin contrastar entre diferentes ejecuciones del mismo, lo cual puede restar validez a los resultados expuestos en [Das93].

En el primer caso de aplicación a LVQ, Monte [Mon93], utiliza un algoritmo genético clásico (SGA), para minimización de la distorsión de un diccionario, utilizando el algoritmo genético para diseño de un diccionario, solo y en combinación con LVQ. El problema al que lo aplica es la cuantización vectorial de muestras de habla, para lo cual tiene que utilizar diccionarios de gran tamaño, ya que los diccionarios usados en clasificación de muestas de habla son habitualmente de 1024 niveles. Esto trae consigo un largo entrenamiento y una gran población. El número de vectores del diccionario es fijo, lo cual limita la generalidad del método. Sin embargo, los resultados obtenidos mejoran bastante los obtenidos con un algoritmo clásico (k-medias, visto en el Capítulo 1), así como con cualquiera de los algoritmos por separado, algoritmos genéticos y LVQ. El problema, claro está, es que el tiempo necesario para obtener esos resultados mediante algoritmos genéticos crece en muchos órdenes de magnitud con respecto a los algoritmos clásicos, mientras que el error se reduce sólo a la mitad. Este es un problema habitual cuando se trabaja con algoritmos genéticos

Harp et al. [Har91], proponen un sistema que combina los SOM y los algoritmos genéticos. Para ello, se codifican en un gen diversos parámetros de la red de Kohonen (la pendiente de la función que da la constantes de aprendizaje, el tamaño de la red en un rango pequeño, y la vecindad inicial), en un enfoque similar al expuesto anteriormente en el Capítulo 4, pero con la seguridad de obtener soluciones óptimas (dentro de la variación de parámetros establecido) que el teorema de los esquema provee a los algoritmos genéticos. En el genoma se incluye el rango de los pesos iniciales, una variable binaria que indica el tipo de vecindad, y si los pesos iniciales están ordenados o no (en ciertos problemas, como se indicó en el Capítulo 2, comenzar con pesos ordenados simplifica el entrenamiento ). El tamaño de la red es fijo en el ejemplo propuesto y de 5x5. Este pequeño tamaño puede causar serios problemas en el entrenamiento de la red, ya que en los extremos de la red, se producen unos efectos de bordes, según indica Kohonen en [Koh82].

El enfoque presentado en esta memoria combina las mejores características de los algoritmos genéticos aplicados a diseño de diccionarios, con el enfoque evolutivo propio de los algoritmos comentados anteriormente en la sección 5.2. Entre las características de los algoritmos genéticos se encuentran la búsqueda eficiente en el espacio de soluciones,

concentrándose en las zonas de tal espacio donde se hallan las mejores; y del enfoque evolutivo se toma el cambio de tamaño de la red neuronal, que permite hallar la red neuronal de tamaño óptimo.

## 5.7 Disección del algoritmo G-LVQ

En este apartado se tratará de llegar a un mayor grado de detalle en la descripción del algoritmo G-LVQ. Se analizará cada uno de los pasos del algoritmo visto en la sección 5.5, indicando los parámetros necesarios para su implementación y ejecución. Los valores de los parámetros indicados se basan en la experiencia; en el capítulo siguiente se hará una prueba más exhaustiva de los parámetros necesarios para cada tipo y tamaño de problema.

### 5.7.1 Inicialización

La inicialización llevada a cabo en el paso G1, incluye saber el *tamaño de la población*. Generalmente el tamaño de la población se establece de forma heurística, aunque hay una regla de Holland que indica que el número de individuos en la población debe ser proporcional a la longitud del genoma [Holl75]. Según esta regla, el tamaño de la población en el algoritmo G-LVQ debería variar durante el entrenamiento, pero lo que se ha decidido, ante la dificultad de la creación de poblaciones de tamaño variable, es utilizar el tamaño mayor que permita una ejecución eficaz según las características del ordenador.

En algunos casos, este tamaño puede ser demasiado grande, en el sentido de que el entrenamiento puede durar demasiado. Una regla adecuada es utilizar retículas con un lado inferior a 10 para problemas con pocas clases (menos de 5), y un lado mayor para un número de clases mayores.

Sin embargo, el tamaño de la población (o, de modo equivalente, el lado de la retícula) debe de venir dado por la experiencia. Hay algunas reglas heurísticas que indican que el algoritmo genético no se está comportando bien; por ejemplo, si el *fitness* medio de la población oscila mucho, y no se alcanza el *fitness* máximo deseado, a pesar de que la población se ha quedado estancada y no aumenta el fitness, probablemente se haya

producido el problema de la consanguineidad. En ese caso, si es posible, se aumentará el tamaño de la población. En el capítulo siguiente se indicarán los valores que han dado buenos resultados para los problemas tratados.

Aparte de su tamaño, que es fijo, la población inicial se genera de modo totalmente aleatorio. La longitud inicial de los genomas se establece aleatoriamente, con un tamaño comprendido entre las tres cuartas partes del número de clases y las siete cuartas partes del mismo. Esta regla, que es también heurística, permite que se comience con unos diccionarios pequeños, y que la búsqueda se desarrolle avanzando hacia diccionarios de longitud superior (aunque no siempre sucede esto, como se verá en el capítulo siguiente). Evidentemente, un diccionario deberá de contener al menos un vector por clase, aunque en la población inicial no se hace ninguna provisión de este tipo, para que la búsqueda genética no se halle sesgada. De esta forma, es el propio algoritmo genético el que va hallando las etiquetas de clase adecuadas y asignándolas a los vectores diccionario correspondientes.

### 5.7.2 Cálculo del fitness y reproducción

En los pasos G2.1 y G2.2 del algoritmo se calcula el *fitness* de cada diccionario en una tarea de clasificación y se compara con sus vecinos en la retícula. Este paso es el crítico en el algoritmo, pues consiste en un entrenamiento mediante G-LVQ del diccionario definido inicialmente en el genoma, mediante la presentación de una muestra completa de entrenamiento, y su posterior evaluación sobre la misma muestra, sin cambio de pesos.

Para hallar diccionarios óptimos, nos interesan tres cantidades:

- el número de aciertos en la clasificación, o *victorias* de cada uno de los vectores del diccionario, que tienen que *competir* con los demás vectores a la hora de *responder* a una muestra de entrada, como se explicó en el Capítulo 2,
- la longitud del diccionario, pues, para un número de aciertos determinado, nos interesará obtener un diccionario con un número mínimo de niveles o vectores, y, por último,
- la distorsión, definida en el Capítulo 1, que indica lo bien que los vectores del diccionario representan las muestras de entrenamiento; puede haber una distorsión alta con un número de aciertos alto en el caso de que el problema sea simple, con pocas clases.

Por tanto, mientras que en los algoritmos genéticos clásicos el *fitness* es una sola cantidad, en el caso del G-LVQ habrá tres cantidades que expresen las prestaciones de un diccionario, y habrá que definir un criterio para comparar el *fitness* de dos diccionarios.

Algunos autores [Scha84,85] han apuntado a la posibilidad de usar más de un parámetro como *fitness*; sin embargo, lo que hacen en realidad es dividir un sólo parámetro de *fitness* en varios subparámetros que hay que optimizar por separado, dándoles a todos la misma importancia; en concreto, en un sistema de clasificación, se consideraban criterios de *fitness* independientes la puntuación obtenida en cada una de las clases.

Habitualmente, el orden de prioridad que se toma es el siguiente (denominado HLD, de *hits* -aciertos-, longitud y distorsión):

- el primero es un grado de aciertos máximo,
- el segundo una longitud mínima,
- y el tercero una distorsión mínima.

De esta forma, para un mismo número de aciertos, se considera más óptimo el diccionario que tenga una mínima longitud; y en el caso de que el número de aciertos y la longitud sean iguales, se considerará más óptimo el diccionario con una menor distorsión. De esta forma se van optimizando por orden estos tres parámetros.

Este orden se puede cambiar, lo cual influirá decisivamente en las prestaciones del algoritmo. El orden por defecto es número de aciertos, longitud del diccionario y distorsión; en problemas especialmente difíciles (como los que se presentarán en los Capítulos 7 y 8) se puede utilizar el orden número de aciertos, distorsión y longitud del diccionario. En casos excepcionales se optimiza primero la distorsión, lo cual habitualmente tiene el efecto de incrementar excesivamente el tamaño del diccionario. Sin embargo, a veces la optimización del número de aciertos en primer lugar hace que se llegue a un mínimo local; la optimización en primer lugar de la distorsión hace que se consiga, aparte de una optimización de la misma, una mejora del número de aciertos, a costa de un tamaño del diccionario no óptimo.

En cuanto a la reproducción, llevada a cabo en los pasos G2.3 y G2.4 se realiza un emparejamiento clásico; a destacar que se trataría de un procedimiento elitista o de tipo *steady-state* [Gol93], ya que en cada generación permanecen unos pocos miembros de la

población: aquellos cuyo fitness es superior al de todos sus vecinos. El crossover es *uniforme*, es decir, que con una probabilidad dada, se intercambian bits de los dos genomas en los cuales se lleva a cabo la reproducción.

### 5.7.3 Incremento y decremento de la longitud

Estos operadores se aplican en los pasos G2.4, y solamente una vez por generación a cada genoma, con una probabilidad fija que debe de establecer el usuario. Para aplicar estos operadores, se genera un número aleatorio entre 0 y 1, y se aplican si el número generado no supera la probabilidad correspondiente. Cada diccionario, como se verá en el capítulo siguiente, almacena información sobre el número de aciertos, es decir, clasificaciones correctas, de todo el diccionario, así como el número de aciertos o *victorias* de cada uno de los vectores del diccionario.

Para aplicar el operador duplicación se calcula para el diccionario elegido cuál es el vector con un número mayor de victorias, o de aciertos, y se crea una nueva copia, que se muta con una cierta probabilidad de mutación igual a la probabilidad de mutación del genoma; esta mutación puede afectar tanto al valor del vector como a su etiqueta de clase. El operador de incremento aleatorio se aplica creando un nuevo diccionario, que con respecto al anterior, contiene un nuevo vector con valor y etiqueta aleatoria. En el caso del operador de eliminación, se calcula cuál es el vector del diccionario con un número menor de aciertos, y se elimina. En cada generación, sólo se puede aplicar uno de estos operadores, ya que el cálculo del número de victorias no sería significativo en el caso de que hubiera un vector nuevo o ya se hubiera eliminado uno.

Como en el caso de cualquier otro operador genético, la utilización de estos operadores genéticos afecta sólo a un miembro de la población, que, en caso de dar una solución peor que la ya existente en su vecindario, será eliminado. Por tanto, permiten una exploración eficaz y paralela del espacio de soluciones, incluyendo ya en este soluciones de diferente tamaño.

### 5.7.4 Número de generaciones

El número de generaciones *G* indicado en el paso G2 es un parámetro heurístico, si

bien, dado que el entrenamiento completo se puede y debe repetir varias veces, para observar si los resultados obtenidos son consistentes, es aconsejable que si se ha alcanzado en un número pequeño de iteraciones el máximo, la segunda vez se limite el entrenamiento a ese número de iteraciones. En este criterio G-LVQ coincide, obviamente, con el usado en los algoritmos genéticos clásicos.

El número de generaciones durante las cuales se ejecuta el algoritmo genético depende fuertemente del problema, aunque para los casos que se han estudiado en esta memoria, donde la longitud de los diccionarios era hasta de 15 o 20 vectores, en 400 generaciones se suele hallar el óptimo, como se verá en el Capítulo 7.

## 5.8 Conclusiones

En este capítulo se han presentado diferentes enfoques evolutivos, basados en algoritmos genéticos y basados en reglas heurísticas, que tratan de diseñar redes neuronales de estructura variable y adaptada al problema. Estos algoritmos adolecen de una serie de fallos.

Los algoritmos LVQ y otros algoritmos de cuantización vectorial no siempre obtienen un óptimo dentro del rango de variación de los parámetros inciales. Por tanto, se propone optimizar los algoritmos de cuantización vectorial mediante algoritmos genéticos, variando los pesos iniciales y sus etiquetas. Además, no se conoce de antemano el número de vectores diccionario necesarios para hallar un buen equilibrio entre un número de aciertos máximo y una longitud aceptable del diccionario que no haga demasiado costosa su implementación hardware o software. Para solucionar esto, se proponen una serie de operadores genéticos que alteran la longitud de los genomas, y por tanto, de los diccionarios basándose en los mismos principios de los algoritmos genéticos: mutación y selección.

Los algoritmos tradicionales tratan de optimizar una sola cantidad (por ejemplo, la distorsión o la exactitud de clasificación) a expensas de otras, como el tamaño del diccionario. El algoritmo G-LVQ propuesto utiliza un vector de criterios de optimalidad, de forma que se optimiza con preferencia el número de aciertos, para seguir con la longitud y con la distorsión.

Este orden de evaluación lo puede variar el usuario.

Los algoritmos genéticos tradicionales utilizan un control centralizado, ordenando los genomas por orden de *fitness* para asignar probabilidades o posibilidades de reproducción. Esto equivale a un control centralizado, que, aparte de no existir en la Naturaleza, impide una implementación hardware eficaz. Se propone por tanto un esquema de evaluación del *fitness* local, en el cual el diccionario entrenado desarrollado a partir de cada genoma compara su eficacia con aquellos más cercanos. Esta medida, sin impedir el correcto funcionamiento del algoritmo genético (como se comprobará en el capítulo siguiente), facilita sobremanera la implementación en una arquitectura hardware de tipo hipercubo.

# 6. Implementación de los algoritmos

## 6.1 Introducción

A la hora de implementar un nuevo algoritmo, como el que se presenta en esta memoria, así como evaluarlo y analizar sus resultados, es necesario elegir una serie de herramientas que sean las más adecuadas para el proceso de diseño, implementación, prueba y evaluación. Estas herramientas son lenguajes de programación, arquitecturas computacionales en las cuales se pueda ejecutar el programa con una serie de exigencias, como velocidad y capacidad de memoria, y utilidades de análisis y presentación gráfica para visualizar los resultados.

En cuanto a la implementación software del algoritmo, cabe exigir una serie de características que se están convirtiendo hoy en día en imprescindibles en el diseño de programas complejos. Estas características son la *portabilidad*, la *elegancia* y la *extensibilidad*.

Una de las características más importantes es la *portabilidad*. Dada la gran cantidad de arquitecturas hardware y software existentes, crear un programa para una sola cierra un amplio abanico de posibilidades, y reduce forzosamente la utilidad del programa, sobre todo en un entorno científico. Por ello, tanto el lenguaje de programación como las herramientas utilizadas deben de poder ejecutarse en una amplia variedad de plataformas; las plataformas mínimas exigibles deben de ser los ordenadores compatibles PC ejecutando MS-DOS, y diversos modelos de estaciones de trabajo ejecutando versiones diferentes de UNIX. Los ordenadores que usan alguna de las dos arquitecturas citadas constituyen la mayoría de las existentes hoy en día. Es, por tanto, deseable, que el código fuente del programa que implementa el algoritmo presentado en esta memoria (Capítulo 5) pueda compilarse generando ficheros ejecutables válidos tanto en PCs como estaciones de trabajo ejecutando alguna versión de UNIX. Por eso se ha elegido el lenguaje C++, cuya estandarización es casi

total a lo largo de los diversos ordenadores utilizados es mínima. De hecho, los programas escritos para implementar el algoritmo G-LVQ y programas auxiliares, con una mínima variación, sobre todo en algunas funciones de librería que no son comunes, han sido compilados y probados en estaciones de trabajo Sun, en una estación de trabajo Silicon Graphics Indigo, y en PCs sin ninguna dificultad. Lo mismo ocurre con las demás herramientas: el lenguaje PERL y la herramienta gráfica GNUPLOT. Con todos ellos se puede trabajar tanto en el PC desde MS-DOS como desde *Windows*, en la Sun desde el interfaz gráfico *OpenWindows*, y en la Indigo desde *WorkPlace*.

Asimismo, en el código de los programas actuales, la *elegancia* es una exigencia fundamental; el algoritmo debe de expresarse en un programa lo más comprensible posible, puesto que durante el ciclo de vida de un programa, el código fuente puede pasar por muchos autores diferentes; sobre todo si se pretende que tanto el programa como su código fuente queden en el dominio público, como va a ocurrir con los programas diseñados para esta memoria. Lenguajes de programación implícitamente elegantes, y que por tanto, obliguen a pensar estructuradamente, hacen que los programas que se escriban sean más legibles y compactos.

Un algoritmo, aunque útil en sí, debe de poder ser utilizado desde otros programas, que hagan uso de él, y debe de admitir la posibilidad de ser alterado, en caso necesario, o extendido, de forma que se pueda aplicar a problemas diferentes. El lenguaje de programación elegido debe de permitir, por tanto, una implementación de algoritmos *extensible* y reutilizable, para que el código ya escrito pueda volver a utilizarse mediante simples extensiones, en un enfoque similar al de los elementos de una caja de herramientas. Un algoritmo debe de entenderse como una herramienta; y la extensibilidad debe de permitir que sea añadido, en el contexto de un programa, a otras herramientas, o bien modificado para aplicarlo a un problema particular.

En este capítulo se describen las bases de las herramientas de programación utilizadas. Para comenzar, en la sección 6.2 se hace referencia a la programación orientada a objetos, paradigma de programación relativamente nuevo en el que se encuadra el lenguaje utilizado en esta memoria, el C++; la utilidad de este lenguaje para programación de redes neuronales ha sido ya notada y destacada por varios autores. En la sección 6.3 se hace una breve introducción al lenguaje PERL de proceso de textos, con ayuda del cual se han generado

los conjuntos de datos sintéticos utilizados en esta memoria, y analizado los datos resultantes. Posteriormente (sección 6.4) se explican las estructuras de datos y del programa utilizado para implementar el algoritmo G-LVQ, así como una serie de técnicas complementarias y herramientas (sección 6.5). El esquema de la utilización de las diversas herramientas se muestra en la Figura 6.1.

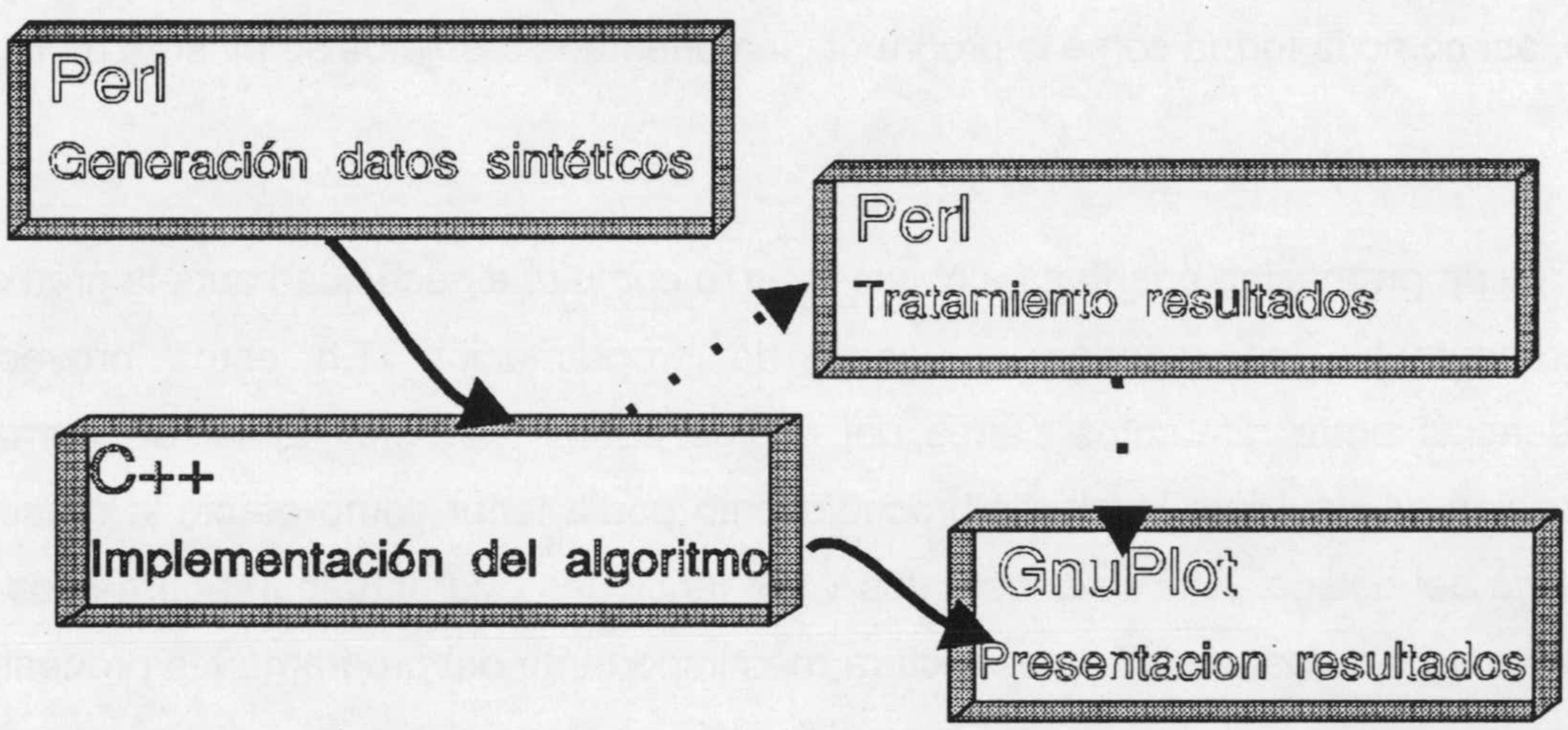



**Figura 6.1**Esquema de la utilización de los diversos lenguajes y herramientas para implementación y evaluación. Líneas punteadas indican caminos alternativos.

## 6.2 Programación orientada a objetos y C++

La *programación orientada a objetos* [Elli90][Eck91][Mast93], es un paradigma de programación relativamente nuevo, que data de principios de los 80. Se habla de un nuevo *paradigma*, puesto que se sitúa fuera del paradigma *procedural* tradicional, en el cual la estructura fundamental es el procedimiento, y fuera también del nuevo paradigma *funcional*, cuyos lenguajes sirven para expresar problemas, encargándose el lenguaje de la resolución de los mismos. El primer lenguaje que se puede denominar *orientado a objetos* es el *SmallTalk*, aunque actualmente el lenguaje orientado a objetos más utilizado es el C++ [Elli90].

La programación orientada a objetos en general se tratará en el apartado 6.2.1;

mientras que las diversas herramientas que combinan tal paradigma de programación con las redes neuronales serán el objeto del apartado siguiente (6.2.2).

### 6.2.1 Programación orientada a objetos en C++

Aunque el campo de la programación orientada a objetos es muy extenso, tanto como cualquier otro paradigma de programación, se pretende dar aquí una breve introducción al mismo, así como la forma como la programación orientada a objetos se plasma en el lenguaje C++.

La programación orientada a objetos surgió como una necesidad ante la gran dificultad que presentaban los grandes proyectos de programación. En estos proyectos, las dependencias entre diferentes partes del código podían ser muy altas, de forma que la modificación de una sola función o procedimiento podía tener como efecto la reescritura de gran parte del código. A la vez, los datos y las funciones que actuaban sobre ellos estaban totalmente separados; siendo la estructura más importante del programa los procedimientos. Estos procedimientos podían causar la modificación no deseada de alguna variable global con la consiguiente dificultad de depuración.

Para limitar al máximo las interdependencias y evitar efectos colaterales, los lenguajes orientados al objeto proponen la división de un problema en varias partes, de forma que cada parte es totalmente independiente de las otras; y las modificaciones y depuración se pueden reducir a cada una de esas partes. Cada parte, que se denomina *clase*, consiste en una serie de datos (es decir, variables) junto con las funciones o procedimientos que actúan sobre ellos, llamadas *métodos de la clase*. Este concepto de clase contribuye también a la creación de tipos de datos abstractos; cada clase es un tipo de dato abstracto, y cada *instancia* de esa clase, o copia de la clase con unos valores de inicialización particulares, es un *objeto*.

Los datos utilizados por el programador, por tanto, están *encapsulados* dentro de la clase, separándose completamente cada clase y sus contenidos del resto de las clases. Una clase presenta al exterior solamente un *interfaz*, mediante el cual puede interaccionar con el programa principal y el resto de las clases, y que es la forma mediante la cual el programador puede utilizar la clase desde otras clases. Debido a esta encapsulación, es factible un cambio total de la implementación de la clase sin cambiar el resto del programa, puesto que el

interfaz no cambia, y no afecta, por tanto, al resto del proyecto.

El código resultante de una clase puede adaptarse a nuevas necesidades sin necesidad de ser reescrito; las características de una clase pueden ser *heredadas* por otra u otras clases. Por ejemplo, se puede definir la clase siguiente (las palabras remarcadas en **negrita** son palabras clave del lenguaje C++), que ilustra el concepto de *herencia*. Estas clases son simplemente declaraciones, y para ser útiles tendrían que formar parte de un programa que las utilizara. La que se muestra a continuación declara una `Figura` genérica, así como un procedimiento para cambiar la posición de la figura.

```
class Figura {
protected:
      int x,y;                          // posición del objeto
public:                                 // definición de métodos de la clase
void  Move( int _x, int _y )  // cambiar de posición el objeto
}
```

Se puede declarar otro objeto, por ejemplo `Circulo`, que herede tanto las variables como los métodos de esta clase ya definida

```
class Circulo: public Figura {        // hereda de la clase anterior
      int rad;                        // x e y son heredadas, rad es el radio
public:
void  incrementarRadio ( int _radplus ) { rad += _radplus; }
}
```

de forma que tanto las variables como los procedimientos de la clase *base* pueden ser utilizados por la clase *derivada*. La clase `Circulo` tendrá tanto sus propios métodos y variables como los métodos y variables de su clase antecesora, `x, y, Move;` por tanto un objeto de la clase `Círculo` será una `Figura` que además tendrá una serie de métodos y parámetros propios. La herencia se puede expresar en la relación *es-un-tipo-de*, en el caso anterior, `Circulo` es-un-tipo-de `Figura`. En general, la clase derivada es-un-tipo-de clase base.

En el ejemplo anterior el interfaz ofrecido por la clase `Circulo` consiste en dos métodos, `Move,` que ha sido heredado, e `incrementarRadio`. El programador que utilizara esta clase sólo podría acceder a sus variables internas (`x, y` y `rad`) mediante estos dos métodos, es decir, sólo de esta forma podría alterar el estado de un objeto perteneciente a esta clase. Evidentemente, la definición de esta clase no está completa; harían falta, por ejemplo, funciones para establecer los valores iniciales de las variables de instancia. Estas funciones se suelen denominar *constructores*. También se necesitarían una serie de funciones

para acceder a valores internos de los objetos.

Todas las clases pertenecientes a la misma *jerarquía*, es decir, descendientes de la misma clase ***base***, pueden compartir el mismo interfaz con el resto de las clases, aunque la implementación sea diferente. El *polimorfismo* permite que un método se pueda implementar de forma diferente a lo largo de la jerarquía de clases; y que el compilador incluya en el programa ejecutable código que averigüe en cada instante en qué lugar de la misma se halla. Al llamar a un método de este tipo, la propia clase se encarga de ejecutarlo de la manera adecuada. En el ejemplo siguiente se muestra la diferente implementación de una función que muestra la figura geométrica en el caso de las dos clases, el punto (denominada `Figura`) y el `Círculo`[1]

```
virtual void Circulo::Draw() {
      circle( x, y, rad );
}
//... resto de la implementación
virtual void Figura::Draw() {
      putpixel( x, y );
}
```

De la misma forma, una función cualquiera declarada en un programa en C++ se puede llamar con parámetros de tipos diferentes; escribiendo implementaciones diferentes para llamadas a la función con tipos diferentes, ya que, a diferencia del lenguaje de programación C, el C++ es un lenguaje fuertemente tipificado, lo cual significa que hace comprobación de los tipos de llamada a funciones en tiempo de compilación. Esto permite una gran compacidad de código, pudiendo, por ejemplo, redefinir (es decir, *sobrecargar*) el operador suma de forma que se puedan sumar matrices, vectores, o cadenas alfanuméricas. La traslación del algoritmo al programa, si se cuenta con las clases adecuadas y los operadores sobrecargados, es muy directa.

El ejemplo siguiente muestra como se puede implementar en C++ la regla de actualización de pesos de los algoritmos de Kohonen. En esta forma ha sido utilizada por lor programas creados para implementar el algoritmo G-LVQ

```
//-----------------------------------------------------------------
void LabelDict::train( int _delta, LblWeight _sti ) {
//-----------------------------------------------------------------
```

---

[1] Tanto `putpixel` como `circle` son funciones genéricas de dibujo; en la forma indicada se puede encontrar, por ejemplo, en la librería gráfica del Borland C++.

```
        float tmp = alpha *_delta;
        (*refVectors[lastWinner]) +=
                    (_sti - (*refVectors[lastWinner]) )*tmp;
}
```

Estas dos líneas de programa actualizan los vectores peso, multiplicando la diferencia entre el vector peso `(*refVectors[lastWinner])` ganador y el vector de entrada `_sti` por una constante, y añadiéndosela al valor anterior del vector peso. En esta función, el operador - y el += están sobrecargados, de forma que en realidad se realizan las operaciones correspondientes sobre vectores con etiqueta (tipo LblWeight) cuya definición se verá a continuación en el apartado 6.4.1. También el operador * está sobrecargado, de forma que realice la operación de multiplicación de un vector por un escalar, en vez de actuar, como sucede en C, con datos escalares.

Todas estas características del C++ lo hacen ideal para el diseño y programación orientada al objeto de redes neuronales (igual que sucede con otros muchos campos), como ya han notado ciertos autores [][Mas93], y es el enfoque que se ha tomado en esta memoria.

### 6.2.2 Programación orientada a objetos y redes neuronales

Dado que la programación orientada a objetos es uno de los paradigmas dominante en programación en la actualidad, no es de extrañar que se encuentren varios ejemplos de utilización del mismo para la programación de redes neuronales, incluso en un mismo congreso [Tre93][Fue93][Drei93][Tro91][Lin93].

Generalmente, casi todos estos intentos tratan de crear una herramienta de programación y generación de redes neuronales de propósito general basada en un lenguaje orientado a objetos, generalmente C++, de forma que tanto un usuario sin conocimientos de programación (que usará la herramienta a través de su interfaz para el usuario) como un programador (que podrá utilizar los objetos definidos y extenderlos mediante los mecanismos de herencia) puedan usarlo con comodidad. En algunos casos, se crea un generador de aplicaciones en C++ [Drei93], a partir de otro lenguaje de más alto nivel que refleja la naturaleza orientada al objeto del mismo.

Rara vez se menciona la utilización de lenguajes orientados al objeto para aplicaciones particulares, o para crear un nuevo algoritmo, sin embargo, las facilidades ofrecidas para un programador en este caso son mucho mayores que las de cualquier otro lenguaje de programación, como se ha indicado en la introducción. De esas facilidades se han aprovechado los programas diseñados para implementar el algoritmo presentado en esta memoria.

En cuanto a la jerarquía de clases utilizada para implementar redes neuronales en c++, es muy similar en todos los casos, creando un objeto base, que suele ser una capa (`Layer`), y construyendo a partir de ahí los demás elementos, de control y de entrada y salida. En el caso del algoritmo G-LVQ, presentado en esta memoria, el elemento básico de la red son los pesos, ya que en el caso de la red neuronal con aprendizaje LVQ no existe el concepto de capa. Además, nuestro enfoque añade el objeto que implementa un gen o cromosoma, para utilización de un algoritmo genético, como se verá a continuación.

## 6.3 PERL

A la hora de crear conjuntos de datos sintéticos, así como procesar los datos generados por una aplicación, se hace necesario un lenguaje que, siendo tan potente como el C, sea tan fácil de utilizar y tan directo como el *BASIC*, y que, a la vez, esté especialmente preparado para el tratamiento de grandes ficheros de datos. Un lenguaje que reune estas características es el PERL [Wall91].

PERL es un lenguaje interpretado optimizado para el análisis de ficheros de texto arbitrarios, extrayendo información de los mismos, e imprimiendo informes basados en esa información. El lenguaje se ha diseñado de forma que sea práctico, es decir, fácil de utilizar, eficiente y completo, en vez de elegante o mínimo, lo cual contradice una de las reglas enunciadas en la introducción, aunque en este caso, no es tan necesario, ya que los programas en PERL son típicamente sólo de unas docenas de líneas. Combina alguna de las características del lenguaje C, y de los lenguajes SED, AWK y *SH, que se hallan habitualmente en el entorno UNIX. Además, la sintaxis es parecida a la del C, con lo cual el aprendizaje es rápido si ya se conoce ese lenguaje.

PERL no limita el tamaño de los datos con los que puede tratar; puede utilizar ficheros tan grandes como la memoria principal (y, a veces, la virtual) permita. Admite la definición y uso de matrices asociativas, es decir, matrices cuya clave de acceso no es un número que indica una posición, sino una cadena alfanumérica. Sus prestaciones no descienden con el aumento del tamaño de las matrices. Otra de sus características es el análisis rápido de grandes ficheros de datos, buscando expresiones regulares. Una *expresión regular* es una forma de expresar una multitud de cadenas alfanuméricas diferentes utilizando letras, números y símbolos que actúan como comodines o como operadores sobre los mismos. Por ejemplo, `a[1..10]` se referiría a las cadenas alfanúmericas `a1, a2,..., a10`.

Además, PERL es un poco más rápido que las utilidades estándar de UNIX mencionadas anteriormente (como AWK o SED), y permite desarrollar programas en menos tiempo que el C, ya que utiliza un entorno de desarrollo mínimo, compuesto por un editor y el intérprete/compilador/depurador. Cuando se llama al intérprete de PERL, éste compila el programa a una representación interna; en esa fase se detectan errores sintácticos, y si los hay, se detiene la compilación del programa. La forma compilada se ejecuta posteriormente paso a paso.

Todas estas características hacen que sea ideal para el desarrollo rápido de programas; tanto para generación de conjuntos de datos sintéticos como para exploración de conjuntos grandes de datos y modificación o alteración de los mismos.

Por ejemplo, el programa siguiente

```
#-----------------------------------
# Se llama de la forma siguiente. Argumentos #
#      0 - número de puntos por nube
#      1 - nombre del fichero de salida
#-----------------------------------
srand;
open( CIRCOUT, '>'.$ARGV[1]);open( CIRCOUT1, '>'.$ARGV[1].".1");
open( CIRCOUT2, '>'.$ARGV[1].".2");
for ( $i = 0; $i < $ARGV[0]; $i ++ ) {
      #generar punto aleatorio, como radio y ángulo
      $radio = rand;
      $angulo = rand(2*3.141595);
#     si es el círculo central, imprimirlo como clase 1; también
#     en el archivo correspondiente; el concéntrico es la clase 2
      $px = $radio * cos( $angulo );$py = $radio * sin( $angulo );
      if ($radio < 0.7) {
            $class = 1;
```

```
            print CIRCOUT1 $px."\t".$py."\n";
        } e lse {
            $class = 2;
            print CIRCOUT2 $px."\t".$py."\n";
        }
        print CIRCOUT $px."\t".$py."\t".$clase."\n";
}
close CIRCOUT;close CIRCOUT1;close CIRCOUT2
```

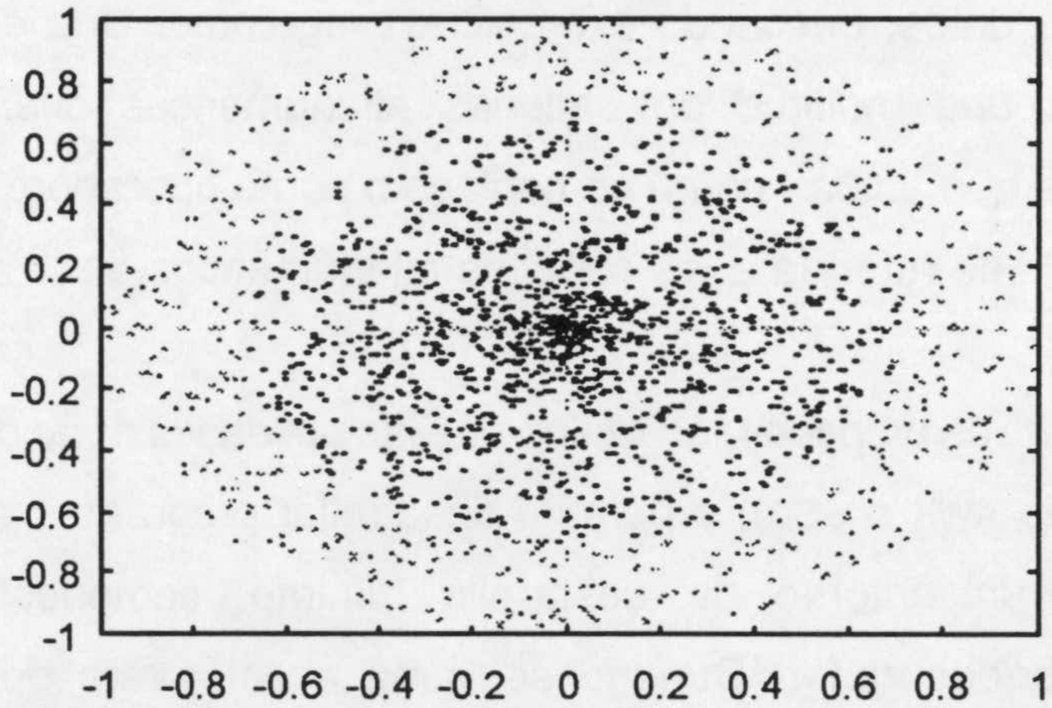



**Figura 6.2** Distribución de puntos en dos clases (puntos claros y puntos oscuros) generada por el programa `circconc.pl`.

genera el fichero que se presenta en la Figura 6.2. La principal diferencia, en este caso, con un programa en C, es la no necesidad de declaración de variables, ni de función `main()`; lo cual hace que el programa sea mucho más compacto. El programa abre los ficheros necesarios en las primeras líneas, para entrar en un bucle que genera puntos aleatorios, y los asigna a una u otra clase dependiendo de si están situados dentro de un círculo de radio 0.7 o fuera del mismo; finalmente cierra los ficheros.

El lenguaje PERL permite utilizar todas las facilidades del lenguaje de comandos del sistema operativo, sea UNIX o MS-DOS, ya que las incluye como órdenes propias, de forma que se pueden utilizar todas las facilidades de manipulación de ficheros. En el caso del MS-DOS, permite la utilización de utilidades no presentes en el lenguaje de macros. En UNIX se pueden utilizar *pipes*, es decir, redirigir la entrada estándar de un programa[2], de forma que los comandos para un programa los puede suministrar otro programa escrito en PERL, de esta forma, se pueden manejar programas como el GNUPLOT desde un programa en PERL. Esta facilidad ha sido utilizada para representar fácilmente los resultados obtenidos de las

[2] Para una explicación más completa de los *pipes*, ver el capítulo 7.

ejecuciones del algoritmo G-LVQ, que se presentarán en los capítulos siguientes.

## 6.4 Implementación de redes neuronales y algoritmos genéticos en C++

A continuación se analiza la estructura del programa que implementa el algoritmo presentado en esta memoria. En el algoritmo indicado hay dos estructuras de datos principales: el *genoma*, que contiene la información necesaria para crear el diccionario, y el *diccionario*. Se analizará cada una de ellas por separado; la estructura que implemente el diccionario en el apartado 6.4.1, y la que implementa el genoma en el apartado 6.4.2.

Dado que se está trabajando con un algoritmo genético, se hace necesaria una clase que gestione la población de genomas de la forma más transparente al usuario posible, así como la más general. Esta clase se describe en el apartado 6.4.3. La política de selección de población, descrita en el capítulo 5, sección 5.5, es un caso particular de esta memoria, y por tanto, se implementará en otra clase que descienda de la clase anterior; además, se trata de una población de diccionarios, lo cual tendrá que ser tenido en cuenta también. Esta clase se describe en el apartado 6.4.4.

### 6.4.1 Clase diccionario

Como ya se ha visto anteriormente en el Capítulo 2 de esta memoria, un diccionario está compuesto por un conjunto de tamaño variable de vectores referencia, no estando la dimensión de los vectores establecida de antemano. Cada uno de los vectores puede tener o no una etiqueta de clase, aunque en el caso que se estudia en esta memoria, todos los vectores del diccionario tienen una etiqueta de clase, ya que se trabaja con entrenamiento supervisado.

El elemento base de la jerarquía de clases es el vector. Esta clase base se denomina `Weight`, ya que, como se ha visto en capítulos anteriores, un vector se equipara en el caso de LVQ al conjunto de pesos que corresponden a la misma neurona. En esta memoria se han utilizado vectores de números reales de doble precisión, que corresponden al tipo de datos `double` del C++, de forma que todos los cálculos se hagan con la máxima precisión, aún al

coste de una menor velocidad y una mayor ocupación de la memoria. El vector básico utilizado por el algoritmo G-LVQ tiene una longitud que se determina en tiempo de ejecución, para no restringir al usuario a una dimensión de vector determinada.

A este vector se le añade una etiqueta, para formar la clase `LblVector`. Es decir, la clase `LblVector` hereda de la clase `Weight`, ya que un *vector-con-etiqueta* es-un-tipo-de vector. Esta clase se usa para representar los vectores utilizados en el entrenamiento y la prueba de los algoritmos; ya que ambos son vectores con una etiqueta de clase.

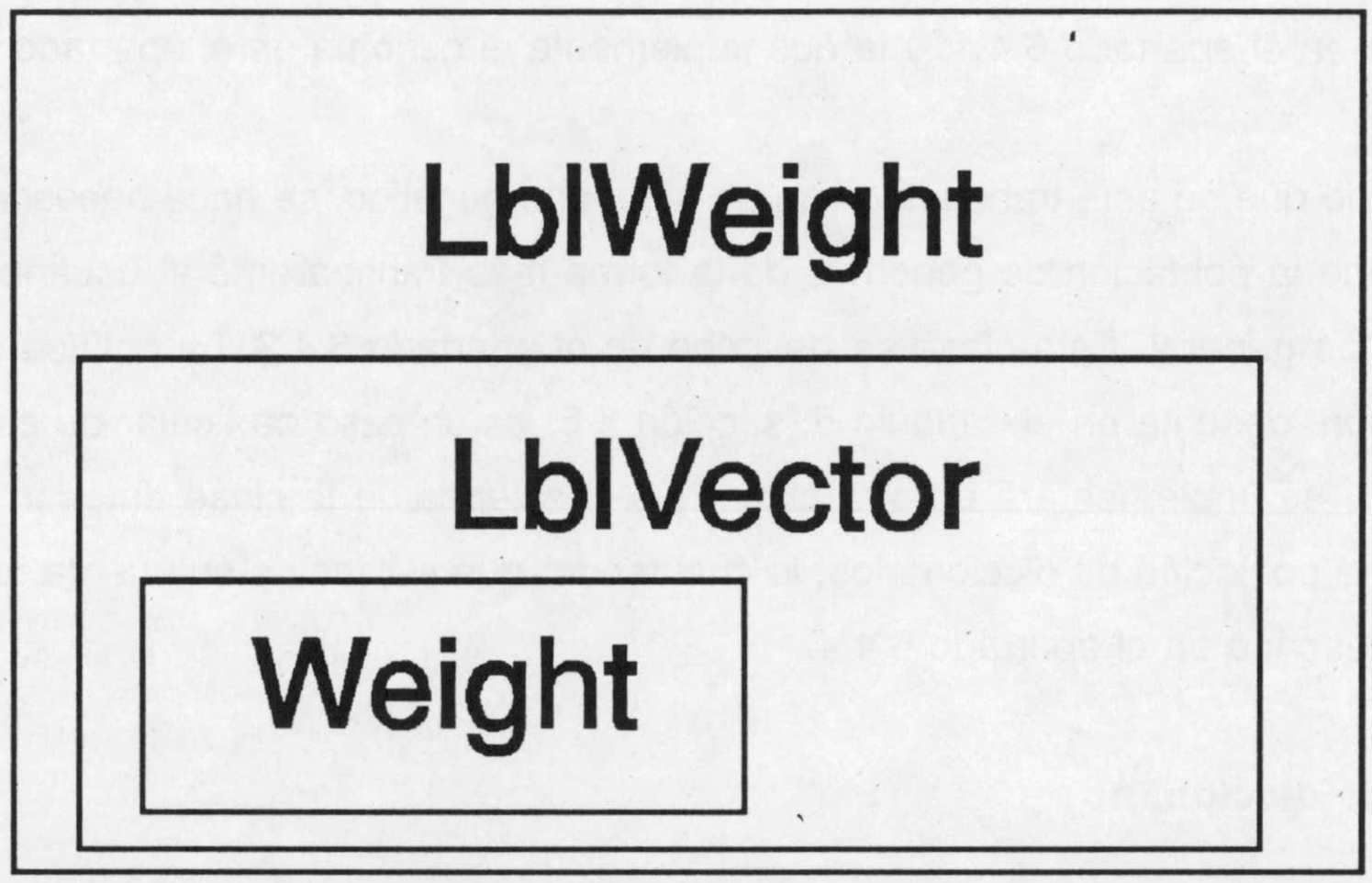



**Figura 6.3** Jerarquía de clases. Las clases más externas heredan de las más internas.

Por último, durante el entrenamiento es necesario guardar una serie de variables que indicarán cuán bien se ha comportado cada vector durante el mismo; por ejemplo, cuantas veces ha sido el vector ganador. A la clase que incluye esta información, heredando el resto de la anterior, se le denomina `LblWeight`, y sirve para representar los vectores de un diccionario. En todos los casos anteriores, cada elemento hereda del anterior, como se indica en la Figura 6.3.

Un diccionario está compuesto por muchos vectores, cada uno de ellos con etiqueta y contadores; el tamaño del diccionario, también denominado número de niveles del diccionario, será también variable, y dependiente del genoma que representa los valores iniciales de cada diccionario. La clase que lo implementa, denomina `LblDict`, incluye por tanto varios objetos del tipo `LblWeight`, aparte de variables que contienen información global

sobre el diccionario, como el índice del último vector ganador, la distorsión total del diccionario y el número de aciertos del mismo.

### 6.4.2 Clase gen de longitud variable

Esta clase contiene toda la información relacionada con el gen, y todos los métodos necesarios para implementar parte de los operadores genéticos: crossover y mutación; aparte de duplicación y eliminación de loci.

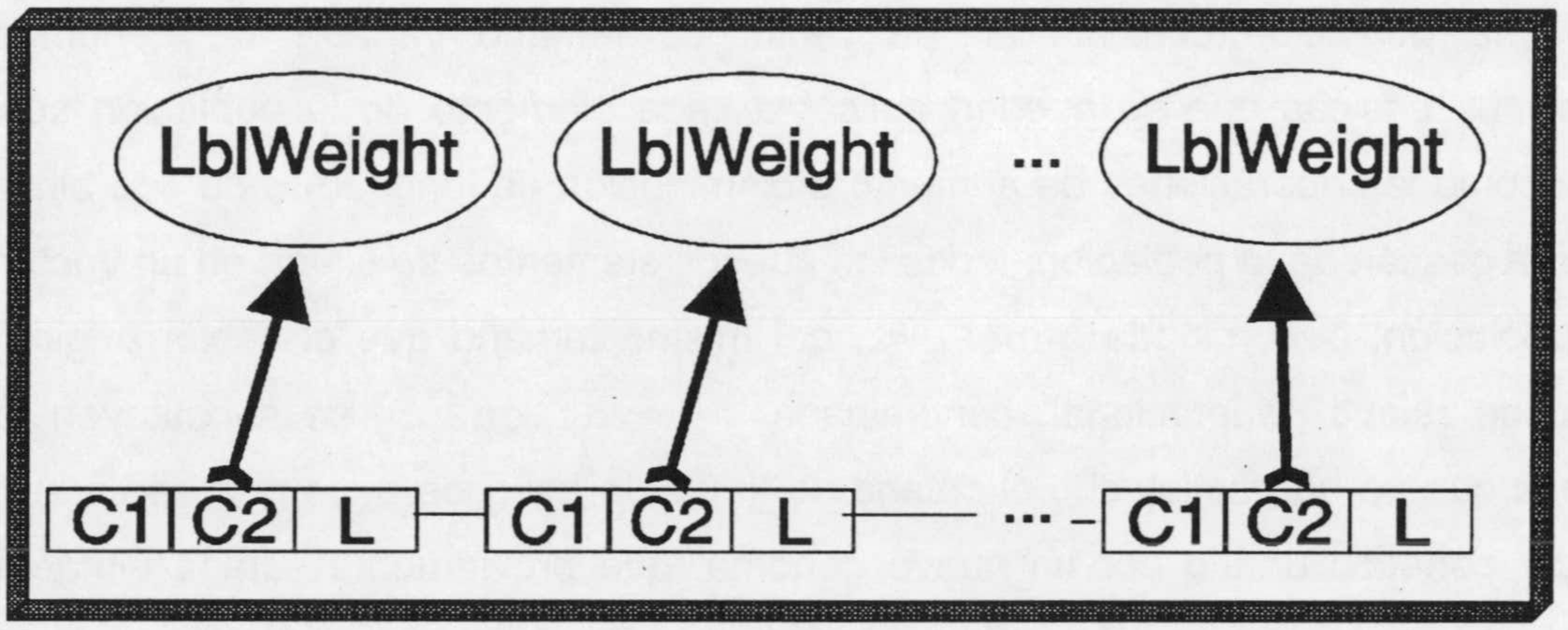



**Figura 6.4** Estructura del objeto `LblDict` y genoma utilizado para generar cada vector del diccionario, clase genVar.

Los operadores genéticos actúa sobre objetos de esta clase, denominada `genVar`; por tanto, las funciones que llevan a cabo tal labor son métodos de esta clase. Los objetos de tipo `LabelDict` se inicializarán a partir de objetos de esta clase, puesto que los vectores peso iniciales de cada diccionario, antes del entrenamiento, se calculan directamente a partir del genoma, correspondiendo cada elemento del genoma, denominado *locus*, a un vector con etiqueta del tipo `LblWeight`.

Esta clase se ha diseñado de forma que sea de propósito general, es decir, que su utilización no tiene porqué estar restringida al algoritmo presentado en esta memoria, ya que incluye todos los operadores genéticos a nivel de genoma necesarios para ejecutar un algoritmo genético; sirve también para utilizarla en genomas de longitud fija; basta con no

utilizar los operadores de longitud variable, y la provisión para utilizarlos no supone una pérdida de eficiencia. Hasta ahora, ninguna implementación de dominio público de algoritmos genéticos [Schr92][Gref90] incluía la posibilidad de utilizar genomas de longitud variable.

### 6.4.3 Clase población

Para facilitar la implementación de algoritmos genéticos con genomas de longitud variable, se ha diseñado e implementado una clase que permite trabajar de forma general con una población de genomas, sin tener en cuenta los criterios de selección utilizados ni la función de fitness asociada a cada genoma. A esta clase se le ha denominado `Population`.

Una población consiste en un vector de tamaño variable de genomas; y las operaciones básicas que se pueden aplicar a cada elemento de la población son de tipo unario (como las operaciones de aumento o disminución de longitud) o de tipo binario. Para facilitar la gestión de la población, todos los nuevos elementos se sitúan en un vector gemelo de la población, denominada `genePool`, del mismo tamaño que el vector original. En un método de relevo generacional, denominado `generationSub`, se substituyen todos los genomas que no han satisfecho el criterio de fitness o aquellos a los que se ha aplicado un operador genético unario por un nuevo genoma, que previamente estaba almacenado en `genePool`. Esta estructuración de la población permite utilizar cualquier política de selección, sea *elitista* o de *ruleta* [Gol75]; en nuestro caso, la estrategia o política de selección utilizada, como se ha indicado en el capítulo 5, es elitista, puesto que sólo una parte de la población cambia de generación en generación.

Al igual que la clase anterior, esta clase es de propósito general, ofreciendo un interfaz que simplifica enormemente la definición de una población de genomas y la utilización sobre los mismos de algoritmos genéticos.

### 6.4.4 Clase población de diccionarios

En esta clase se implementan o utilizan todos los aspectos del algoritmo diseñados para esta memoria, y por tanto, mientras que la clase anterior era de propósito general, esta clase es específica. Este diseño hace uso de la *herencia*, que, como se ha indicado en el apartado 6.2.1, es una de las características diferenciales de la programación orientada a

objetos.

Visto de otra forma, esta clase, denominada `PopDict`, incluye todas las características *fenotípicas* del algoritmo, puesto que como tales se pueden clasificar los diccionarios, desarrollados a partir de un genoma. Una de estas características es el *fitness*, que es un resultado de evaluar ese genoma. Por ello, esta clase incluye todo lo relacionado con los diccionarios: inicialización a partir del genoma, entrenamiento, prueba, y almacenamiento de estadísticas.

En caso de utilizar la clase `Population` para aplicar este algoritmo a otro problema, habría que hacer una clase similar a esta, que heredara de la misma, pero incluyendo todas las estructuras fenotípicas necesarias para calcular el *fitness*.

### 6.4.5 Estructura general del programa

El programa general, que incluye todas las estructuras anteriores, se puede expresar algorítmicamente de la forma siguiente

**Programa G-LVQ**

1. Leer fichero de parámetros del algoritmo.
2. Inicializar población.
3. Leer conjunto de entrenamiento.
4. Repetir un número de generaciones determinado:
   - 4.1. Calcular el número de aciertos y distorsión de cada uno de los elementos de la población.
   - 4.2 Aplicar algoritmo genético G-LVQ.
   - 4.3 Grabar estadísticas.
5. Probar los mejores elementos de la población sobre el conjunto de test.

La estructura y relación entre objetos se muestra en la Figura 6.5. Y, dado que la implementación se ha realizado totalmente de forma modular, el programa principal es el siguiente

```
// ------------------------------ G-LVQ-E.CC ----------------------
// Implementation of the g-lvq-d algorithm using several previously
// defined classes
```

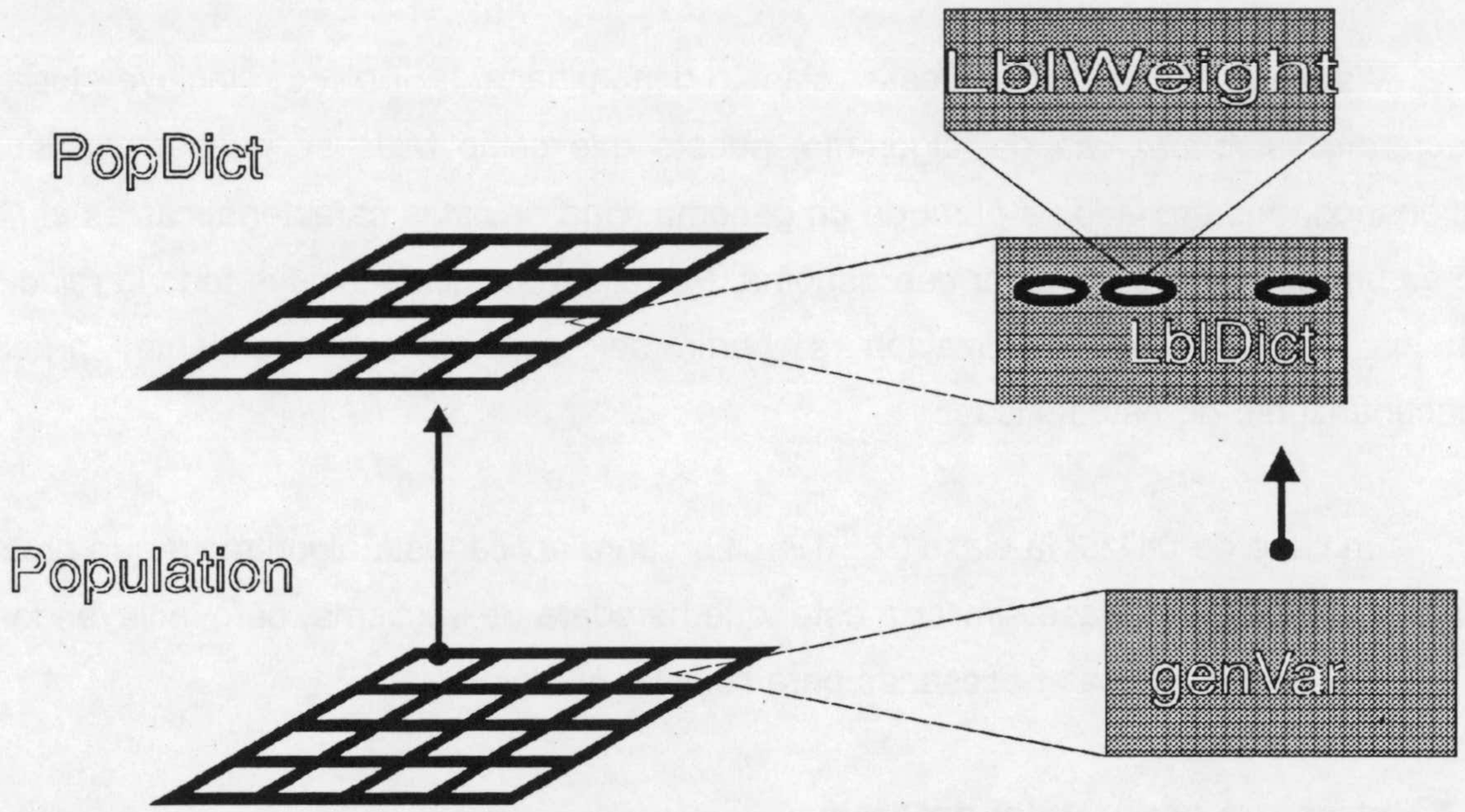



**Figura 6.5** Objetos principales utilizados en el programa g-lvq y relación entre ellos. Un triángulo indica inclusión, y la flecha indica un objeto se desarrolla a partir de otro.

```
// References: JJ's Thesis

// LOG
// 22-Mar-93 - creacion
// 4-Abr-93  - correcion de algunos errores tipograficos y caligraficos;
mas comentarios

#ifndef __BCPLUSPLUS__
#ident "@(#) g-lvq-e.cc by J. J. Merelo"
#endif // Borland C++ does not swallow this useful directive

// Main file -- implementation of  the g-lvq algorithm

#include "general.hpp"          // general and system-dependent defs
#include "popdict.hpp"          // Population of dictionnareis

//---------------------------------------------------------------------
main( int argc char **argv)
//---------------------------------------------------------------------
{
    if ( argc <= 1 ) {
      printf( "Parameters file is missing\n" );
      exit(INCORRECT_PARAMETERS);          // #defed in pop.hpp
    }

    printf( "G-LVQ By J. J. Merelo, %s\n", __DATE__ );

    PopDict thisPopulation( argv[1] );//generation of initial population
    unsigned Generations = thisPopulation.getGens();
```

```
for ( unsigned k = 1; k <= Generations; k ++ ) {

  printf( "------- Generation #%d -------\n", k );

  thisPopulation.resetAllVectors();
  if ( k%10 == 0) {
      thisPopulation.initBaldwin();
      thisPopulation.testBaldwin();
        // tests on the training sample and computes best
        // remember to closeBaldwin at the end
  }
  thisPopulation.trainVectors(); // trains all dictionnaries

  // now tests after training
  thisPopulation.resetAllVectors();   // sets hits & dist to 0
  thisPopulation.testVectors();
  thisPopulation.printOnFiles(k );
  // saves statistics every 10 generations

  //application of genetic algorithm
  //generation substitution and development of new dictionnaries
  if ( k != Generations ) { // not the last time no use
        thisPopulation.applyGA();
        thisPopulation.generationSub();
        thisPopulation.developNew();
    }
    if ( k %10 == 0 ) {      // Baldwin effect is tested every 10 steps
        thisPopulation.closeBaldwin();
  }

  thisPopulation.saveRes();
          // saves some of the best dictionnaries and tests them
  return 0;

} // main
```

Como se puede observar, este programa principal es prácticamente una traducción directa del algoritmo anterior. Comienza con la inicialización de la población a partir de los parámetros contenidos en un fichero, para luego repetir el resto del algoritmo tantas generaciones como haya indicado el usuario. En la parte interior del bucle, cada diez generaciones se evalúan los diccionarios sin someterlos a un entrenamiento, para intentar probar el efecto Baldwin (que se verá en el capítulo siguiente), se realiza el entrenamiento de todos los diccionarios, y posteriormente su prueba para asignar un vector de *fitness* a cada uno de ellos. A continuación se aplica el algoritmo genético, incluyendo los nuevos operadores genéticos aportados en esta memoria, se substituyen los genomas por su descendencia y se convierten estos genomas en diccionarios. Por último, después de terminar

el bucle, se salvan los resultados y se comprueban los diccionarios ganadores sobre el fichero de test, introducido también previamente por el usuario.

Para ejecutar el programa, por tanto, el usuario sólo tiene que editar los ficheros de configuración, que incluyen los parámetros siguientes:

- Nombre del fichero de entrada y de test.
- Parámetros de duplicación, incremento aleatorio y decremento de la longitud del genoma.
- Grado de crossover y grado de mutación.
- Número de generaciones durante los cuales se va a ejecutar el programa.
- Lado de la "rejilla de procesadores" en la cual se va a ejecutar el algoritmo; en la práctica es la raíz cuadrada de la población.
- Precedencia de evaluación, HDL (número de aciertos seguido por la distorsión) o HLD (número de aciertos seguido por la longitud)

El nombre de este fichero se suministra en la línea de órdenes al programa, de esta forma

```
kal-el:/giga1/redlocal/jj/progs/koh% g-lvq-d phart
```

. También es necesario que estén presentes en el mismo directorio, evidentemente, los ficheros que contienen las muestras de entrenamiento y de test. Del directorio en el que se halla el programa debe de colgar otro directorio, llamado DATA, en el que se graban todos los ficheros que produce el programa. El esquema de flujo de los datos y los ficheros necesarios se muestran en la Figura 6.6.

El programa utiliza solamente librerías estándar de C y c++, y ha sido compilado y ejecutado con éxito en las siguientes plataformas:

- Ordenador compatible PC, con microprocesador 486DX a 50 MHz, ejecutando sistema operativo MS-DOS 6.0, utilizando el compilador Borland C++ 3.1.
- Estación de trabajo compatible Sun SPARCstation 10, con microprocesador SuperSPARC a 40 MHz (y otras versiones más lentas), utilizando el compilador de dominio público de C++ de la *Free Software Foundation,* G++.
- Estación de trabajo Silicon Graphics Indigo R4000 con gráficos *Entry*, con microprocesador R4000 a 100MHz internos/50 Mhz externos, utilizando el compilador

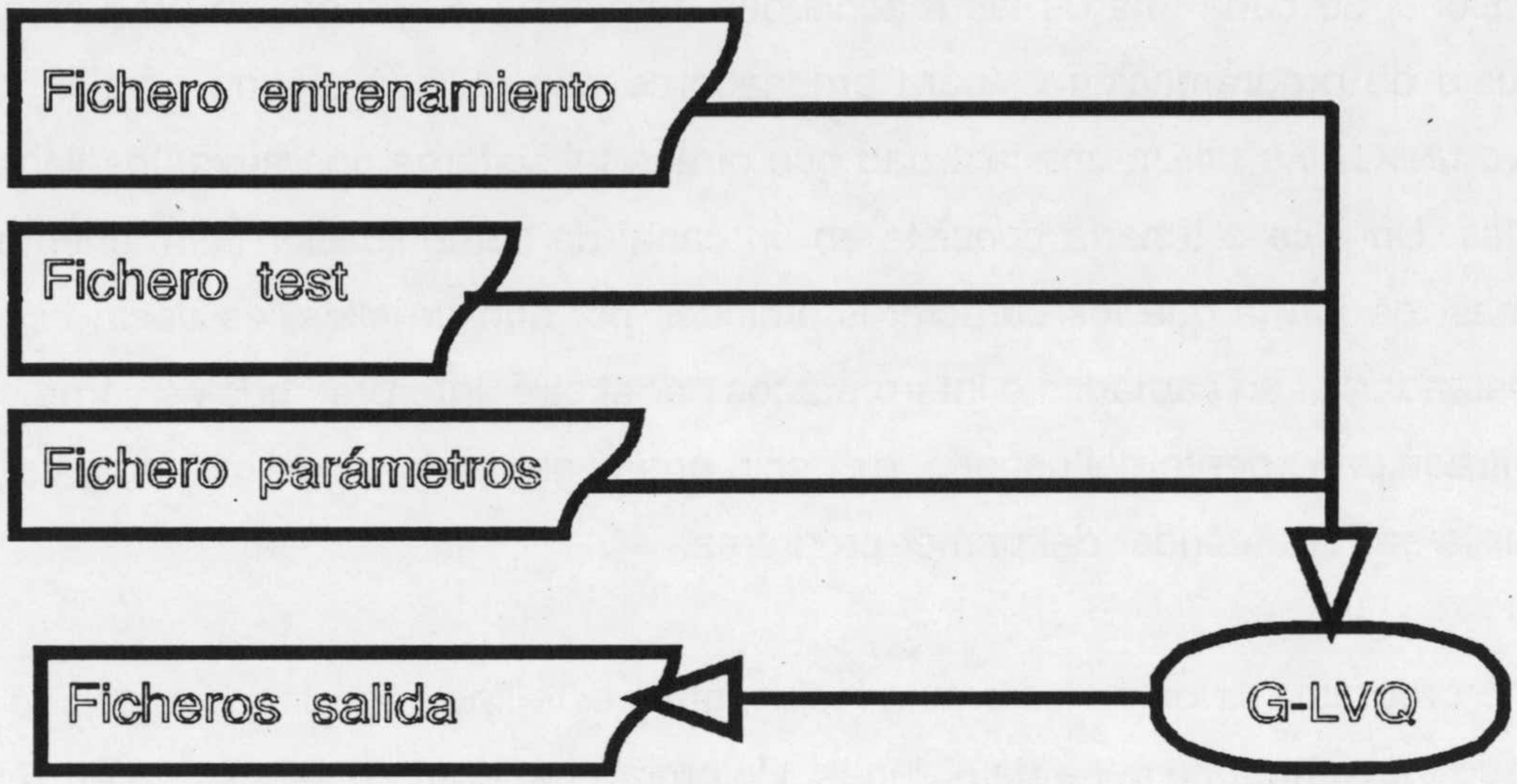



**Figura 6.6** Esquema de los ficheros necesarios para la ejecución del programa `g-lvq`

de dominio público de C++ de la *Free Software Foundation,* G++.

Evidentemente, en el que presenta mayores prestaciones es en la tercera plataforma.

## 6.5 Técnicas de programación y herramientas utilizadas

### 6.5.1 Control de aplicaciones mediante *pipes* en UNIX

Durante la ejecución del programa, se hace necesario a veces presentar gráficamente los resultados, o bien ver la evolución temporal de alguna magnitud de generación en generación. Para llevar a cabo esta labor se ha elegido el programa GNUPLOT, que realiza una gran variedad de presentaciones gráficas bi- o tridimensionales. Con ello se enfatiza la utilización de utilidades estándar como herramientas, en vez de construir o escribir todas las herramientas necesarias para llevar a cabo un trabajo.

Para apreciar la evolución temporal de las magnitudes en los diferentes elementos de

proceso en los que se sitúan diccionarios y genomas (como se ha descrito en el capítulo 5), se ha elaborado un programa, DIAS, que muestra en sucesión cada 10 generaciones un mapa tridimensional de cada una de las magnitudes anteriores. El programa DIAS está escrito en el lenguaje de programación C, para ordenadores que utilicen alguna versión del sistema operativo UNIX. DIAS utiliza una facilidad que ofrece tal sistema operativo, los llamados *pipes* o tuberías. Un *pipe* o tubería consiste en un canal de comunicación permanente entre dos programas, de forma que los caracteres emitidos por uno de ellos, es decir, escritos en su salida estándar, sean captados e interpretados por el otro, interpretándolos como si se tratara de la entrada procedente del usuario; es decir, que el segundo programa toma como entrada estándar la salida estándar del primer programa.

El programa GNUPLOT se presta fácilmente a este tipo de utilización, debido a que está controlado mediante una serie de órdenes. Un programa, como DIAS, escrito en C, C++ u otro lenguaje como PERL (capítulo 6), puede utilizar esta facilidad para abrir una tubería a una copia del programa GNUPLOT, e ir enviándole órdenes.

En efecto, el programa DIAS abre un *pipe* o tubería al programa GNUPLOT, y le va enviando órdenes para abrir y trazar en secuencia los ficheros producidos anteriormente por el programa G-LVQ-D. Sin embargo, los ficheros producidos por este programa no están exactamente en el formato que requiere la orden de GNUPLOT que traza puntos en tres dimensiones. El formato de salida se pretende que sea claro para inspección visual, de forma que las cantidades están alineadas en filas y columnas. Este formato no es el que admite el comando de trazado en tres dimensiones del programa GNUPLOT, denominado SPLOT, por tanto, los datos grabados en ficheros por el programa G-LVQ-D son previamente procesados por un programa escrito en AWK, una utilidad estándar de proceso de textos del sistema operativo UNIX.

### 6.5.2 Presentación automática de genomas

Para la representación automática de los genomas, presentados en diversas figuras a lo largo del capítulo, se ha diseñado e implementado un programa, PSGEN3.PL, en el lenguaje PERL, que tomando como entrada alguno de los ficheros con los genomas producidos por el programa G-LVQ-D, de implementación del algoritmo G-LVQ, lo convierte en el programa en lenguage PostScript encapsulado que produce esa ilustración, pudiendo ser impreso o

incluido, como en este caso, dentro de otro texto. Los genomas aparecen en una o dos columnas; y su tamaño está escalado para que llenen una página.

## 6.6 Conclusiones

A lo largo del ciclo de vida de un programa, diseño, implementación, ejecución, presentación y análisis de resultados, el programador tiene que elegir una serie de herramientas que hagan este proceso lo más eficiente posible. Dado el entorno computacional multiplataforma presente en los laboratorios y empresas actuales, se hace conveniente que estas herramientas y lenguajes puedan servir para todas o al menos la mayoría de las plataformas. De esta forma, el programa diseñado aumenta su utilidad, al estar disponible para un gran número de usuarios pontenciales. Desde el punto de vista del posible científico que desee utilizar con facilidad y con un mínimo de esfuerzo de programación las facilidades que da la implementación de un algoritmo, el lenguaje en el que se diseñen los mismos debe de ser elegante, y los programas escritos en el mismo fácilmente ampliables.

Durante el diseño e implementación de un programa, se ha elegido la programación dirigida a objetos, también denominada programación orientada al objeto (del inglés *object-oriented programming),* y un lenguaje que se puede encuadrar dentro de este paradigma, el lenguaje C++. Este lenguaje, en sus diversas implementaciones y en su versión ANSI 3.0, se halla disponible en una amplia gama de plataformas. Sus características le hacen ideal para la programación de proyectos extensos, y para crear bibliotecas de funciones que puedan ser usadas y ampliadas por otros programadores.

Por otro lado, durante la prueba del y análisis de sus resultados, otros programas multiplataforma, como el GNUPLOT, utilizado en sus versiones IRIX, MS-DOS, SUNOS y Windows 3.1, y el lenguaje PERL, también disponible en esas plataformas, han hecho posible la generación de conjuntos de entrenamiento, la comparación de ficheros de respuestas generados con los programas con las etiquetas de los de entrenamiento, y el formateo de los mismos para un fin determinado.

Se considera, por tanto, que las herramientas utilizadas son las más adecuadas para

emprender el tipo de proyectos complejos con los que se suele enfrentar un científico computacional hoy en día, sobre todo en el campo de las redes neuronales.

El resultado obtenido en la implementación del algoritmo G-LVQ es tanto un programa utilizable directamente por el usuario para resolver sus problemas de clasificación, como una serie de clases programadas en C++ de propósito general, que incluyen capacidades no disponibles hasta el momento, disponibles para el programador.

# 7. Comportamiento del algoritmo G-LVQ

## 7.1 Introducción

Una vez presentado el algoritmo G-LVQ en el capítulo anterior, es esencial evaluar si, por un lado, el algoritmo hace lo que pretende, y, por otro lado, si los programas diseñados y escritos para implementarlo realizan la tarea encomendada correctamente y con eficiencia.

Según establece la disciplina de la *Ingeniería del Software* [Prie89], una de las fases en el desarrollo de un programa es la *prueba*. La prueba de un programa implica normalmente dos aspectos:

*a) Verificación* o comprobación de la realización correcta de lo que se pretendía, lo cual equivale al control de calidad en las otras ramas de la ingeniería; en el presente caso se trata de comprobar que el programa es una fiel implementación del algoritmo.

*b) Validación* o comprobación de la adecuación del producto final. En el presente caso consiste en comprobar que el programa realiza de una forma eficiente el diseño de un cuantizador vectorial, mejorando otras alternativas posibles, como LVQ u otros algoritmos neuronales evolutivos, como los presentados en el capítulo 6.

El algoritmo G-LVQ introduce varias innovaciones con respecto a otros algoritmos de cuantización vectorial. La primera es la adecuación para una implementación en arquitecturas de tipo hipercubo, o de tipo malla de procesadores. En este tipo de implementación, se sitúa cada genoma y el diccionario correspondiente en un elemento de proceso autónomo, donde se realiza todo el proceso de entrenamiento, y todo el intercambio de información, en el momento de la selección y reproducción, se hace con los elementos de proceso vecinos (distantes uno, o, como máximo, 2 tramos de la red de interconexión). No se pretende crear una máquina específica para el entrenamiento de diccionarios mediante el método G-LVQ; sino adecuar el algoritmo genético a su ejecución en máquinas paralelas o masivamente paralelas de tipo hipercubo (como la Connection Machine CM-5).

El restringir el emparejamiento a genomas alojados en los procesadores vecinos de la malla de procesadores, constituye, evidentemente, una restricción de las posibilidades de emparejamiento, que sin embargo está de acuerdo con la regla "Desconfíad de la autoridad central", enunciada por Goldberg [Gol89b]. El hacerlo así acerca más el comportamiento del algoritmo genético utilizado por G-LVQ al comportamiento real de la Naturaleza, en la cual no hay un "evaluador global" de adecuación de un ser vivo, sino que la supervivencia del mismo depende de factores locales de entorno y de los congéneres que lo rodean. Sin embargo, casi todos los algoritmos genéticos confían en la comparación del *fitness* de un individuo con todos los demás de la población, aunque esta comparación pueda hacerse de un modo indirecto. Este aspecto se ampliará en el apartado 7.2.

G-LVQ utiliza genomas de longitud variable, que, al decodificarse, dan lugar a redes neuronales de diferente tamaño, como las estudiadas en el capítulo 5. Los operadores *decremento* e *incremento,* también denominados *duplicación* y *eliminación,* y su vínculo al número de aciertos del vector correspondiente, es decir, al número de veces que ese vector ha ganado para el conjunto de entrenamiento, son introducidos por primera vez en esta memoria, y, por tanto, tendrán también que evaluarse. Existen en la literatura otros algoritmos genéticos, como los algoritmos genéticos *messy* presentados en [Gol91], que utilizan genomas de longitud variable. En este caso, sin embargo, dan lugar a estructuras de tamaño fijo.

Estos nuevos operadores genéticos deben comportarse de tal forma que la longitud de los genomas, y por tanto del diccionario, se adecúe al problema, y no se reduzca excesivamente, o aumente demasiado, en este último caso con el consiguiente incremento en el consumo de recursos del sistema. Se verá el comportamiento de la longitud de los diccionarios a lo largo de la ejecución del algoritmo genético en el apartado 7.3, así como la influencia de diversos valores de los parámetros de incremento y decremento de longitud. Otro elemento a evaluar es la sinergia obtenible con la combinación de algoritmos genéticos y LVQ: comprobar hasta qué punto es necesaria, y cómo ayuda el algoritmo LVQ a hallar mejores soluciones que las que el algoritmo genético encontraría solo. Este aspecto se tratará en el apartado 7.4. Y por último, parece necesario realizar un apunte sobre las necesidades computacionales del algoritmo. Es sabido que los algoritmos genéticos necesitan habitualmente de una gran cantidad de recursos. Por tanto, en el apartado 7.5. se incluirá una evaluación de los requisitos temporales del mismo, para diversos tamaños del problema.

## 7.2 Algoritmo genético restringido espacialmente

Como se ha visto en el capítulo anteriore (Capítulo 5), cada gen en la posición *i,j* compara su *fitness* con los genes de las posiciones vecinas $i \pm 1, j \pm 1$ en el momento de seleccionar los progenitores de la generación siguiente. Si su *fitness* es inferior al *fitness* máximo de la vecindad, se sustituye por un gen resultado de entrecruzar el de mayor *fitness* con otro aleatorio del conjunto de ocho vecinos. De esta forma, se restringe el emparejamiento a pequeñas poblaciones, pero por otro lado se consigue una autoorganización similar a la que consigue el algoritmo de Kohonen: diversas zonas de la retícula compiten por imponer su tipo de solución a todo el mapa.

Para evaluar la eficacia del esquema de restricción espacial, se ejecutó el algoritmo sobre una población de 15x15 genomas/diccionarios, repartidos en sendos elementos de proceso. El problema consistía en clasificar puntos bidimensionales pertenecientes a un número pequeño de clases. En algunos casos se ha intentado clasificar una muestra con 10 clases diferentes, y en otros casos muestras con 7 clases diferentes.

El programa creado para implementar G-LVQ, denominado G-LVQ-D, crea a intervalos de 10 generaciones, una serie de ficheros que contienen el número de aciertos para el diccionario/genoma presente en cada procesador en esa generación, la longitud del mismo y la distorsión tras el entrenamiento. Evidentemente, el *fitness*, la longitud y la distorsión del diccionario que se ejecutará en cada elemento de proceso van cambiando a lo largo del entrenamiento, según va cambiando el genoma/diccionario residente en el mismo.

Para apreciar la evolución con el tiempo, se ha elaborado un programa, DIAS (capítulo 6) que va mostrando en secuencia los ficheros. Inicialmente, se ha evaluado la evolución del número de aciertos para un problema de clasificación en 10 clases; cada una de las clases tiene 40 elementos. En las figuras 7.1 a 7.3 se representa el número de aciertos, sobre un máximo de 400, alcanzados por cada diccionario en generaciones sucesivas: la primera, la generación número 20, y la última generación (número 200). En general, las buenas soluciones se expanden en unas cuantas generaciones a todo el retículo, como se puede ver en las figuras 7.1 a7. 3. Este resultado ha sido también observado por otros autores, como

Davidor [Dav91], para su algoritmo genético celular. En la secuencia de ilustraciones, se representa la evolución del número de aciertos (sobre una muestra de tamaño 400, para un problema de clasificación de 10 clases) a lo largo del entrenamiento.

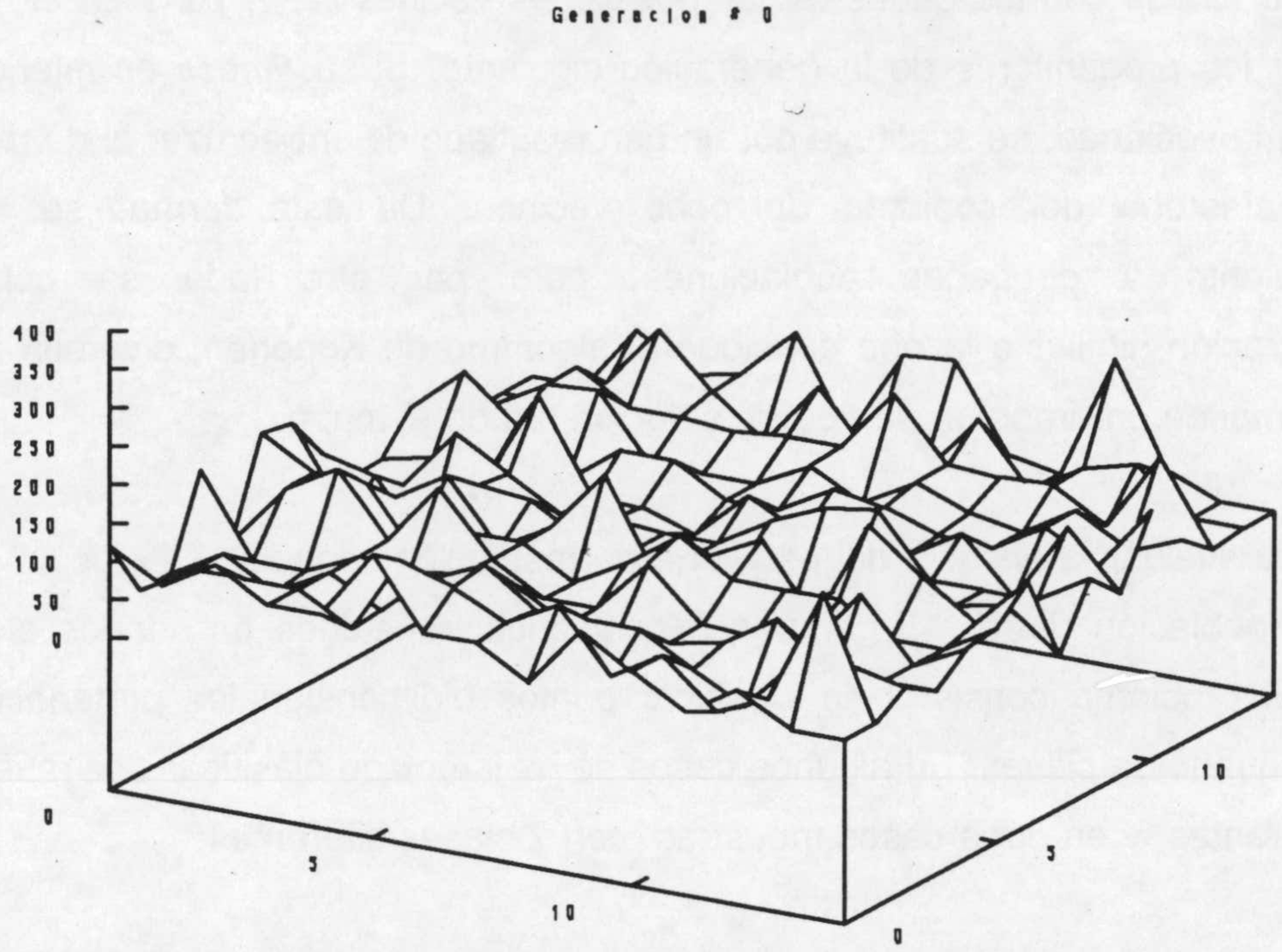



**Figura 7.1** Distribución del número de aciertos por elemento de proceso, al principio del entrenamiento. En ordenadas el índice del genoma; z es el número de aciertos sobre 400.

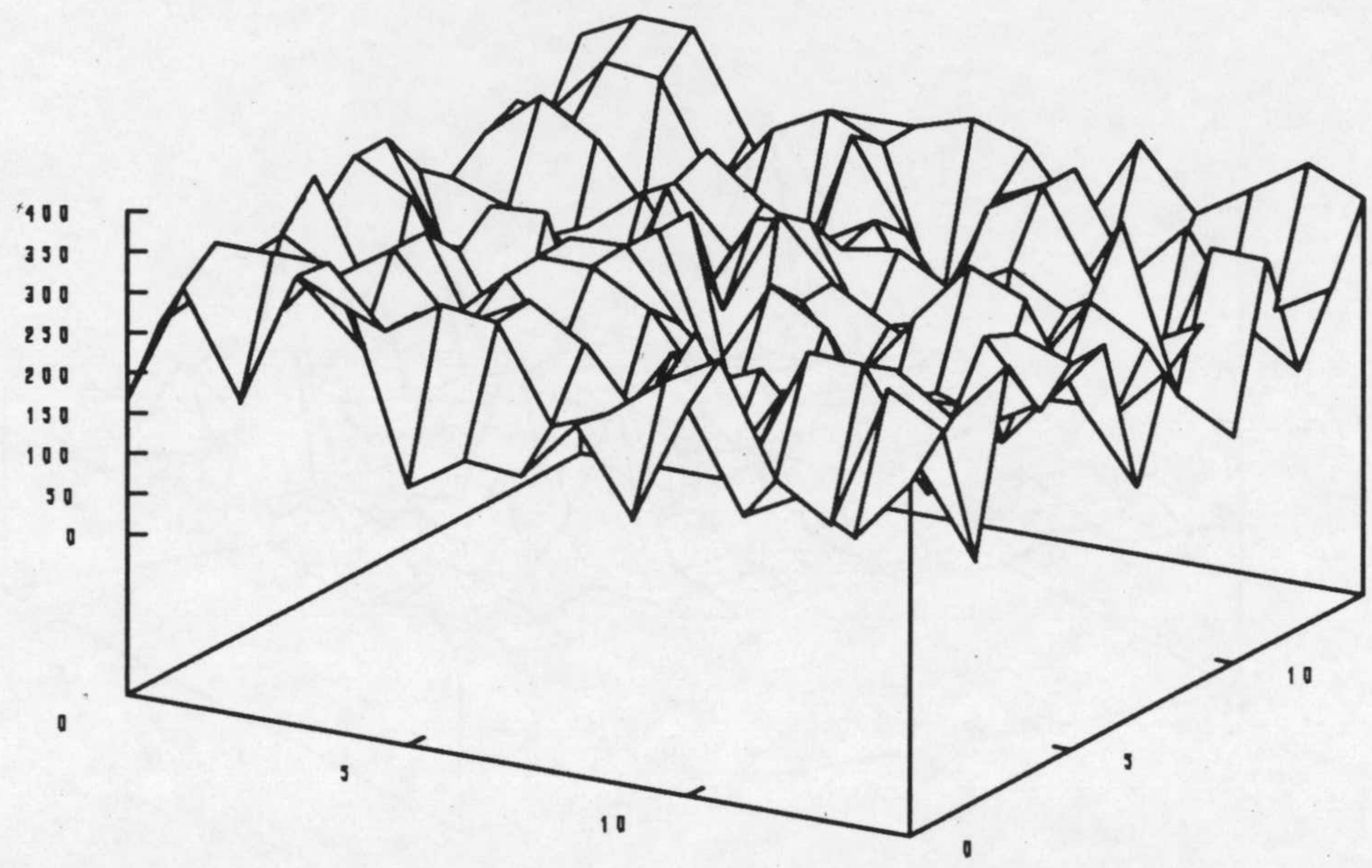



**Figura 7.2** Distribución del número de aciertos por elemento de proceso, en la generación número 20.

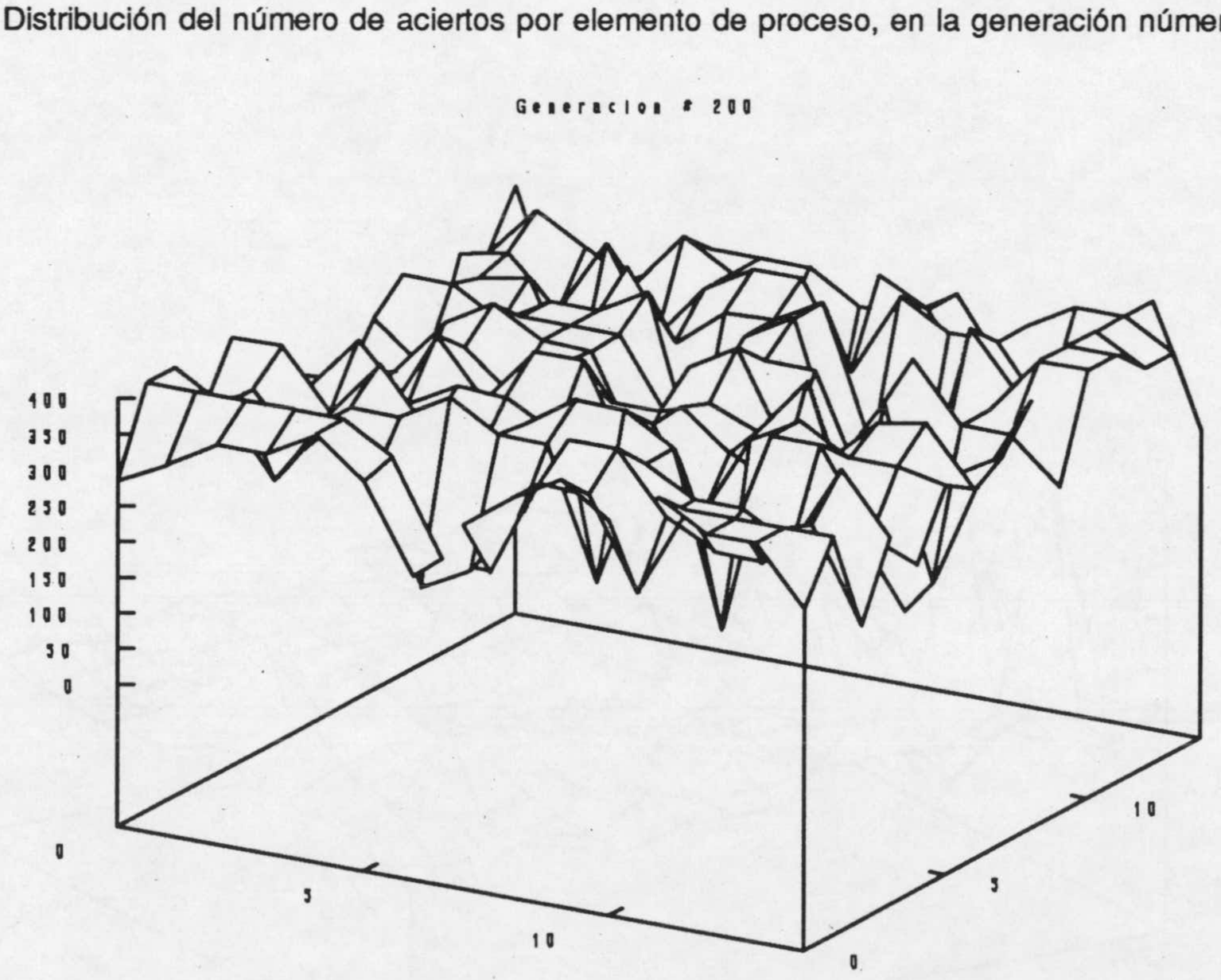



**Figura 7.3** Distribución del número de aciertos por elemento de proceso, al final del entrenamiento.

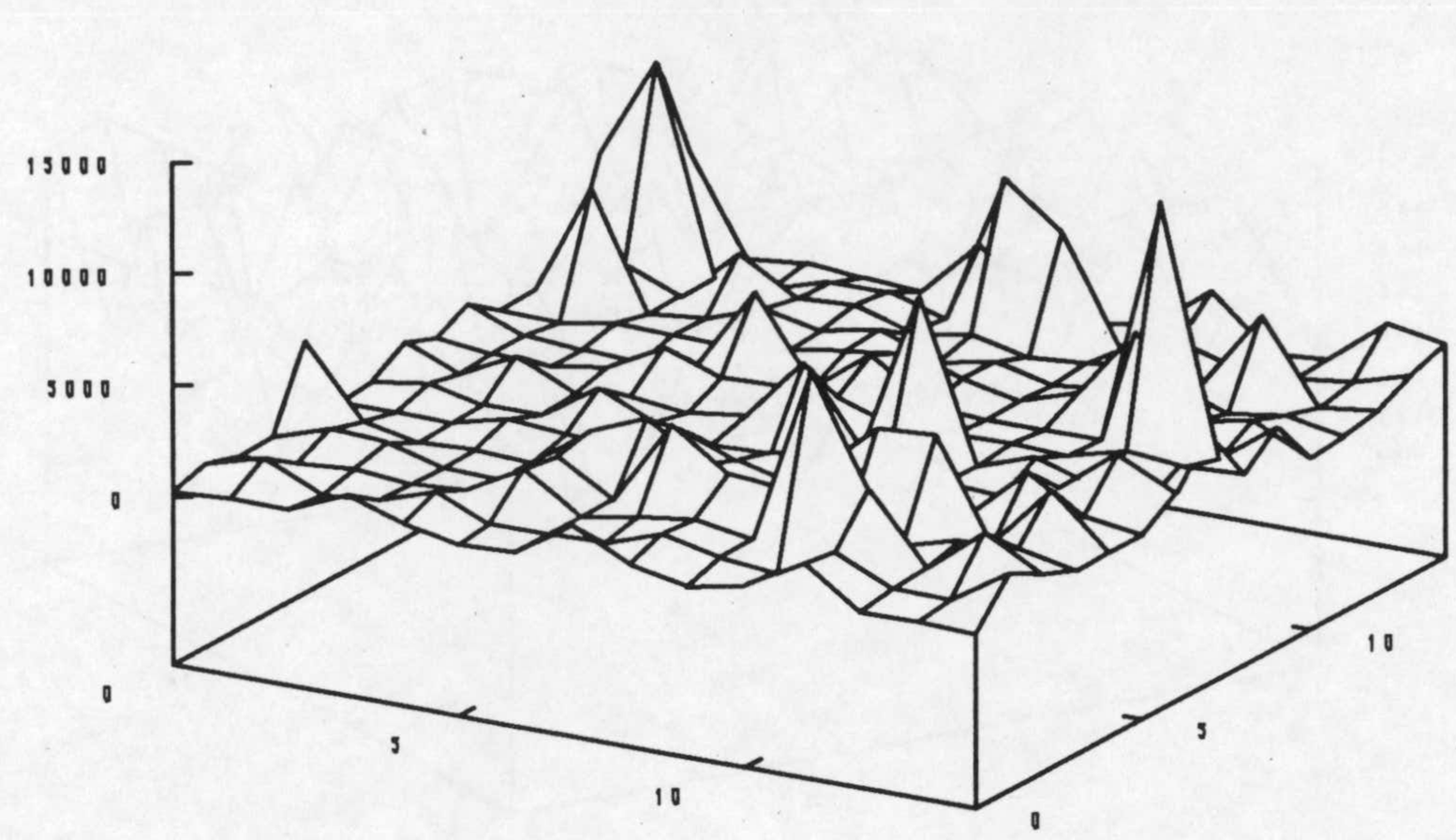



**Figura 7.4** Distorsión inicial

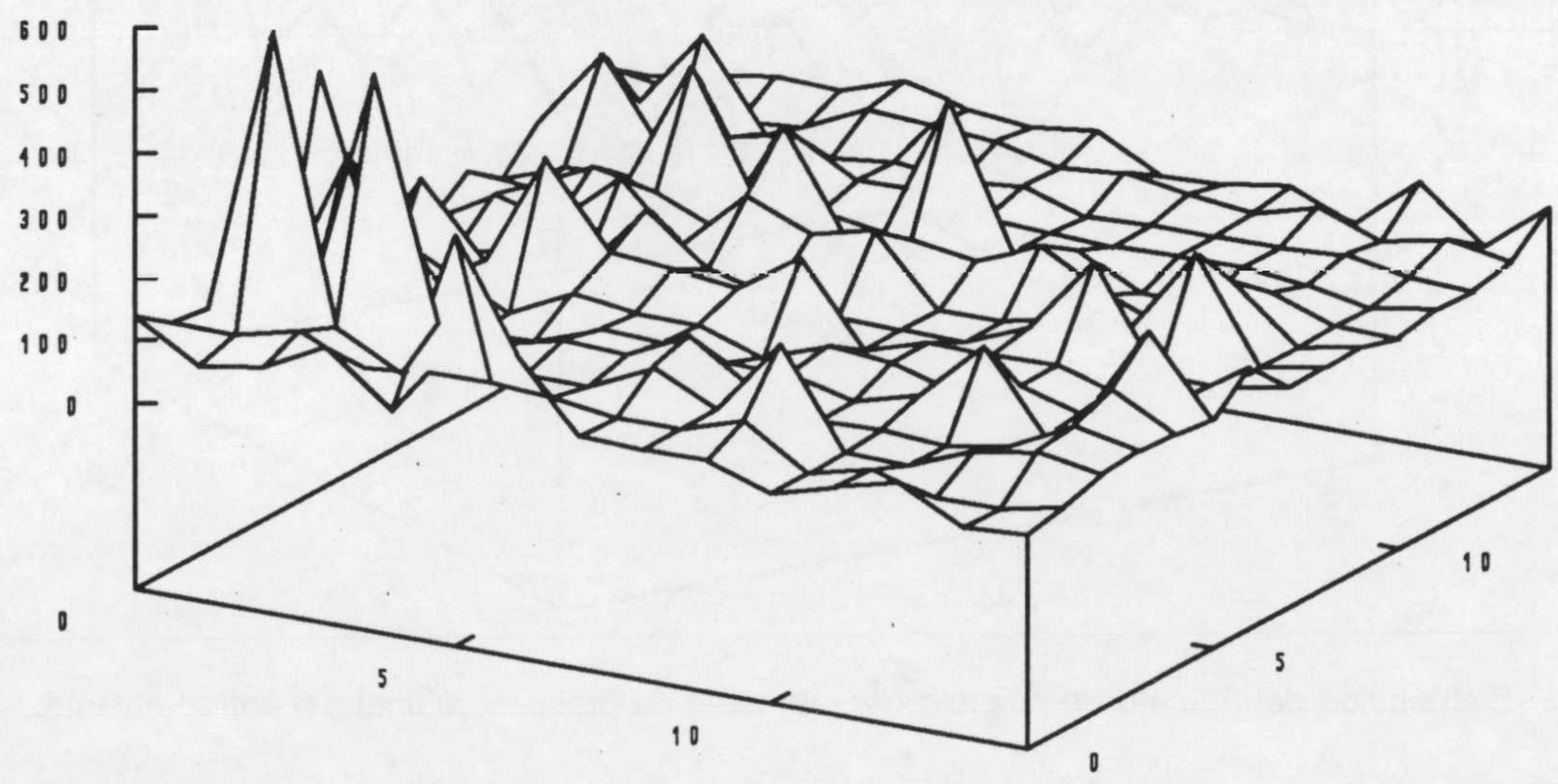



**Figura 7.5** Distorsión en la generación número 20.

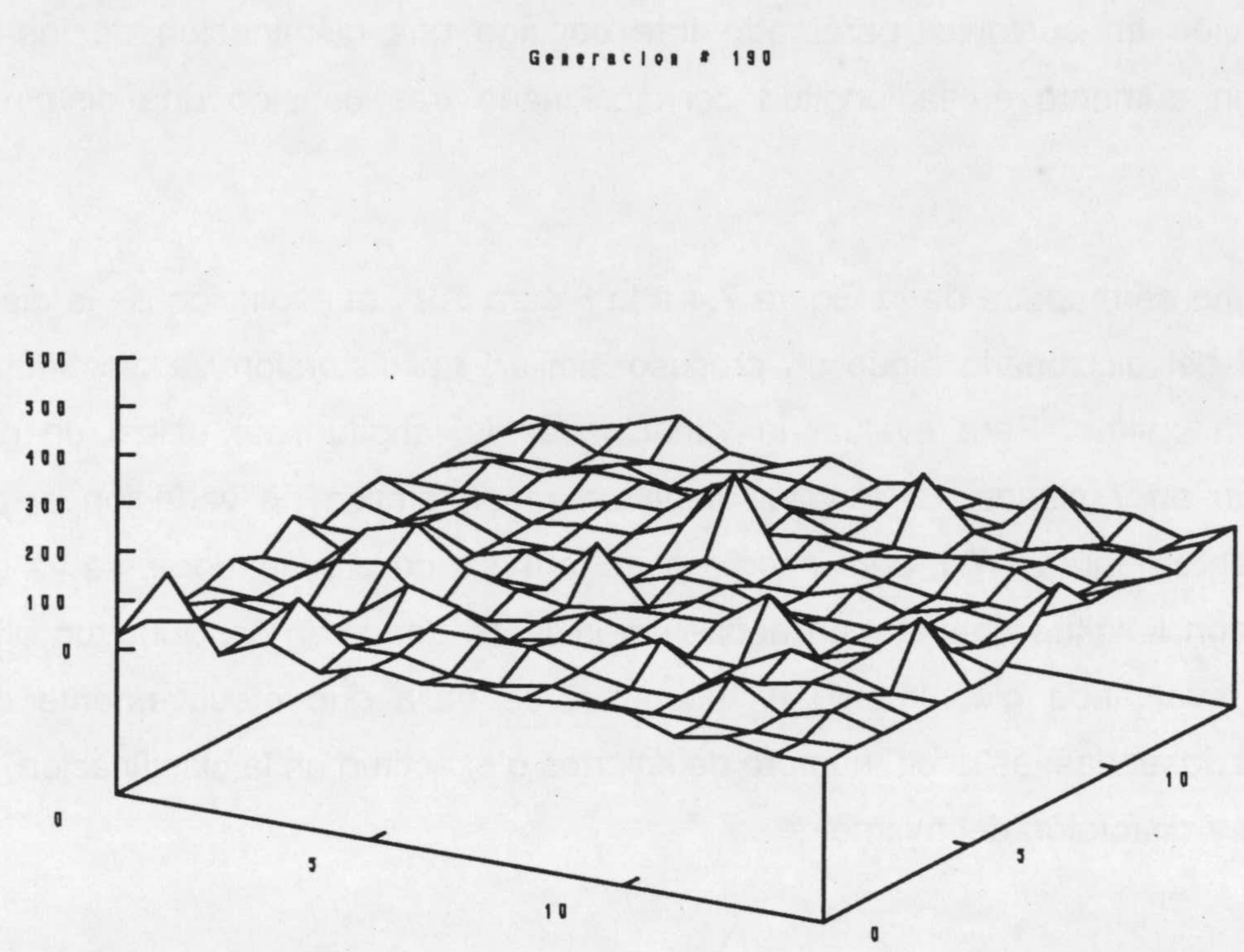



**Figura 7.6** Distorsión al final del entrenamiento.

Como se puede observar en la secuencia de gráficos (Figura 7.1 a Figura 7.3 ), al principio el número de aciertos es uniformemente bajo. Según va transcurriendo el entrenamiento, el nivel medio de aciertos crece, de forma que en la generación número 20 ya hay algún genoma que ha alcanzado un grado de aciertos cercano al 100%; mientras que al terminar el entrenamiento (Figura 7.3 ) son ya varios los elementos de proceso (en donde se sitúan los genomas que se desarrollan en redes neuronales) cuyo porcentaje de aciertos es del 100%. Asimismo, como se puede comprobar, la restricción espacial del emparejamiento de genomas no impide el comportamiento correcto del algoritmo genético, ya que progresivamente se van optimizando los diccionarios hasta alcanzar los valores deseados.

El orden en el que se evalúan los tres parámetros que se quieren optimizar (número de aciertos, longitud del diccionario y distorsión), implica que se optimicen también por ese orden a lo largo de la ejecución, en generaciones sucesivas. En primer lugar, en las generaciones iniciales, se busca un diccionario con el número óptimo de aciertos; en segundo lugar se optimiza la longitud del diccionario, hallando el diccionario con menor longitud para un número de aciertos dado, y por último se minimiza la distorsión. A veces, sin embargo,

una alteración en cualquier parámetro trae consigo una disminución de los otros. Por ejemplo, un aumento en la longitud del diccionario trae consigo una disminución de la distorsión.

Como se muestra de la Figura 7.4 a la Figura 7.9 , la evolución de la distorsión y de la longitud del diccionario sigue un proceso similar. La distorsión va disminuyendo hasta alcanzar un óptimo. Para evaluar la variación de la longitud, se utilizó un problema de clasificación en 7 clases. La longitud tiene un rango amplio de variación al principio del entrenamiento (Figura 7.7 ), pero poco a poco la malla de procesadores se va poblando de genomas con longitud óptima, con pequeñas islas de genomas con longitud diferente. Por otro lado, evaluando globalmente el algoritmo, se halla que efectivamente optimiza las cantidades deseadas, es decir, número de aciertos o exactitud en la clasificación, longitud del diccionario y distorsión del mismo.

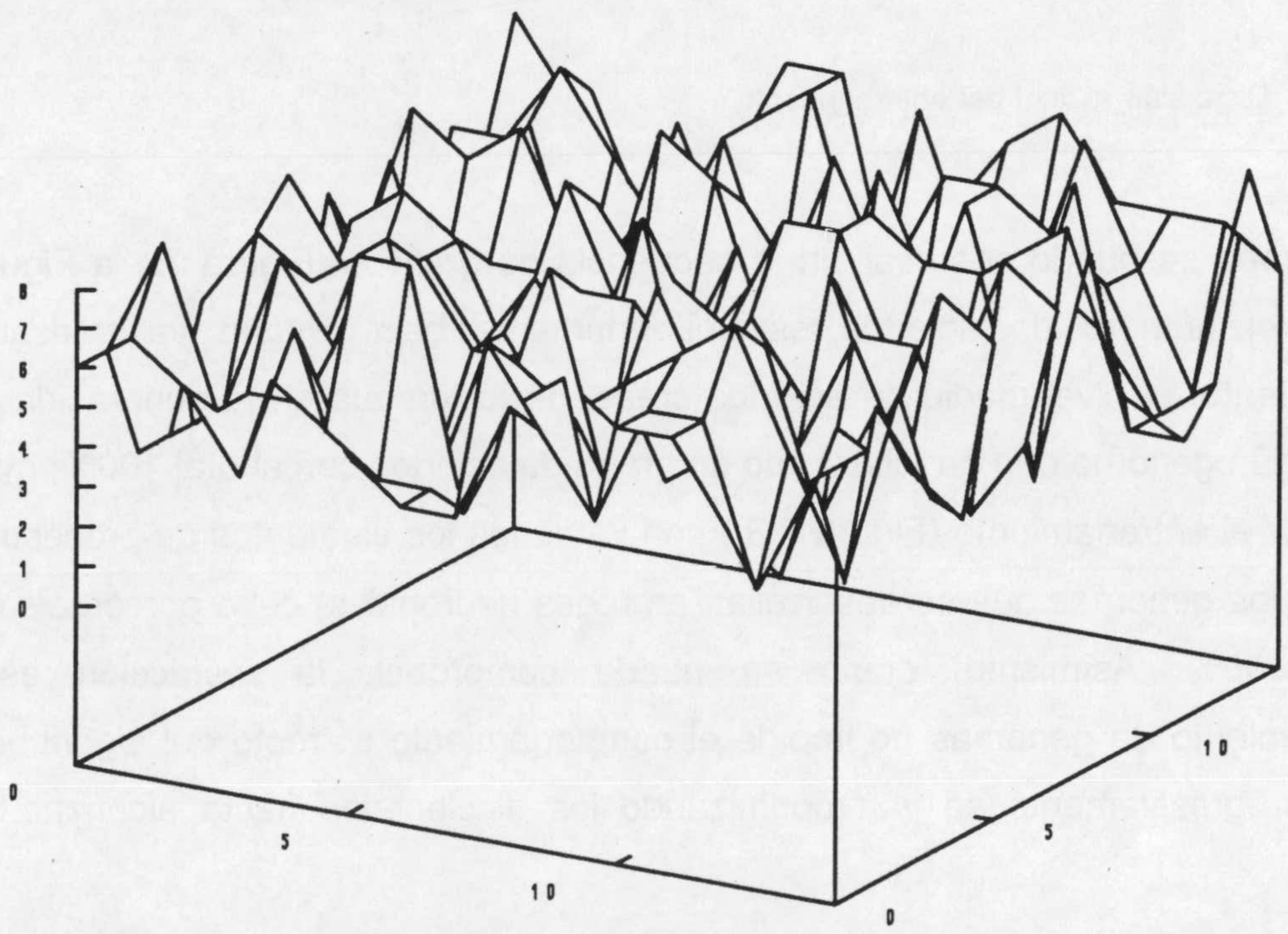



**Figura 7.7** Longitud de cada diccionario al principio del entrenamiento.

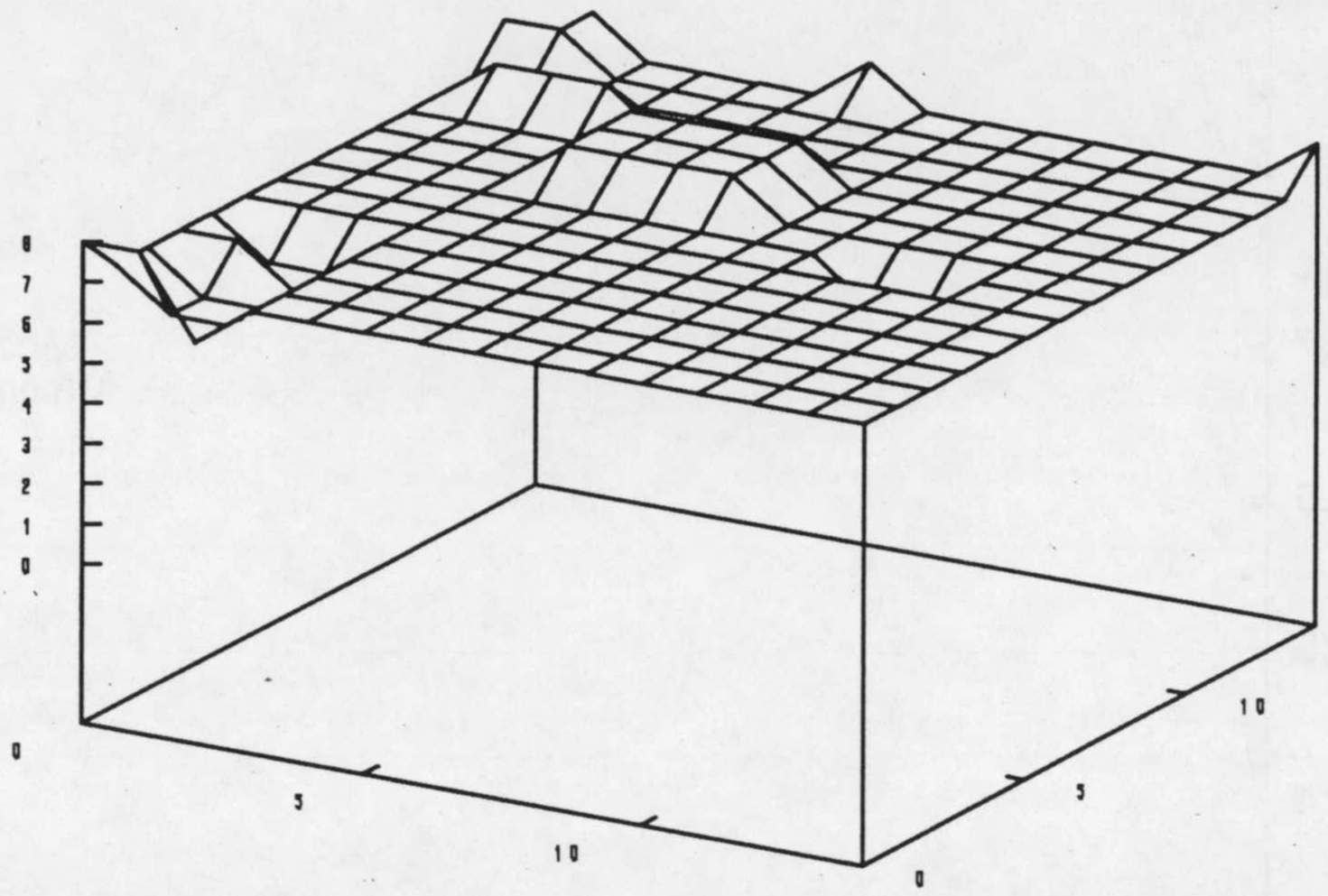



**Figura 7.8** Longitud del diccionario, en la generación número 10.

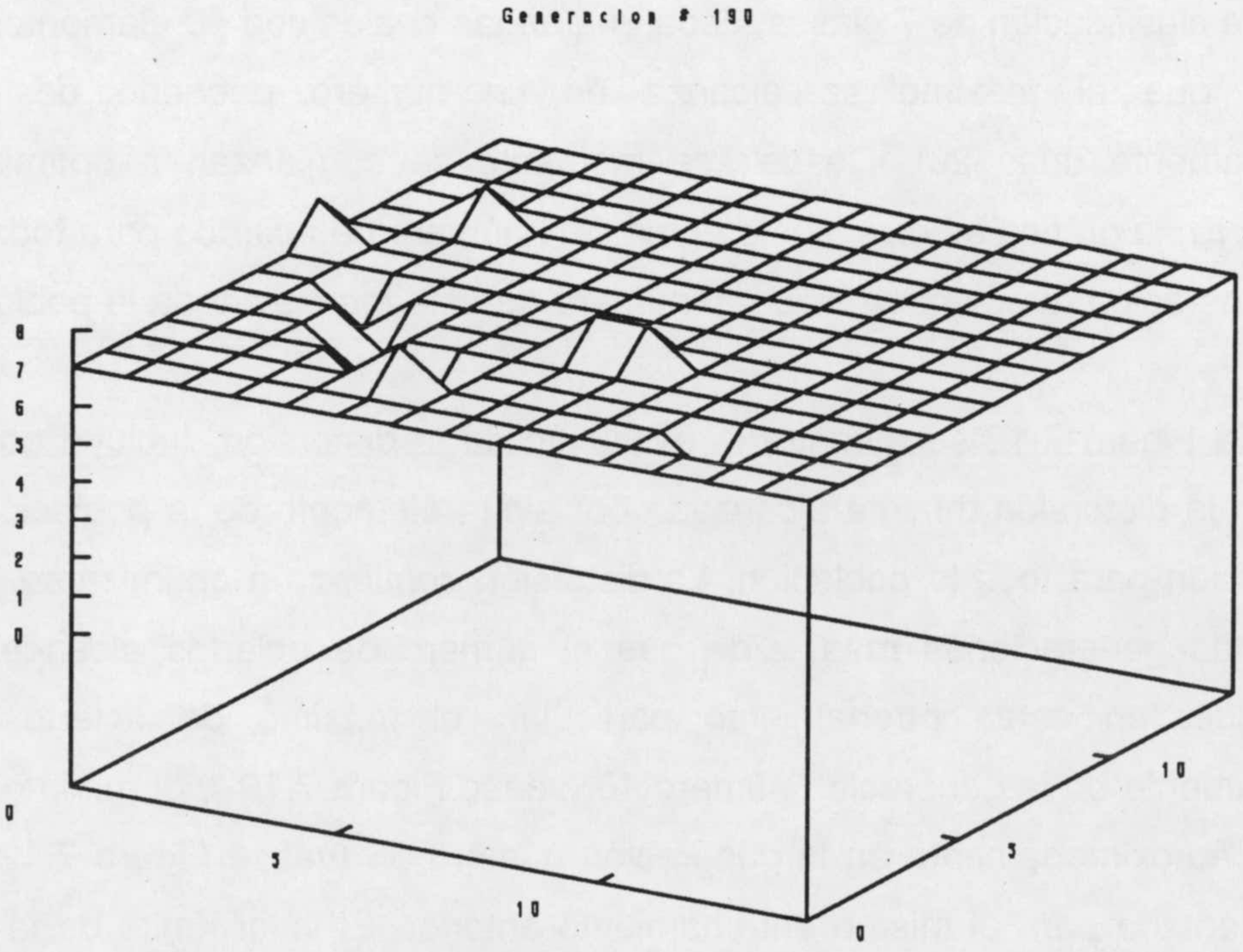



**Figura 7.9** Longitud al terminar el entrenamiento.

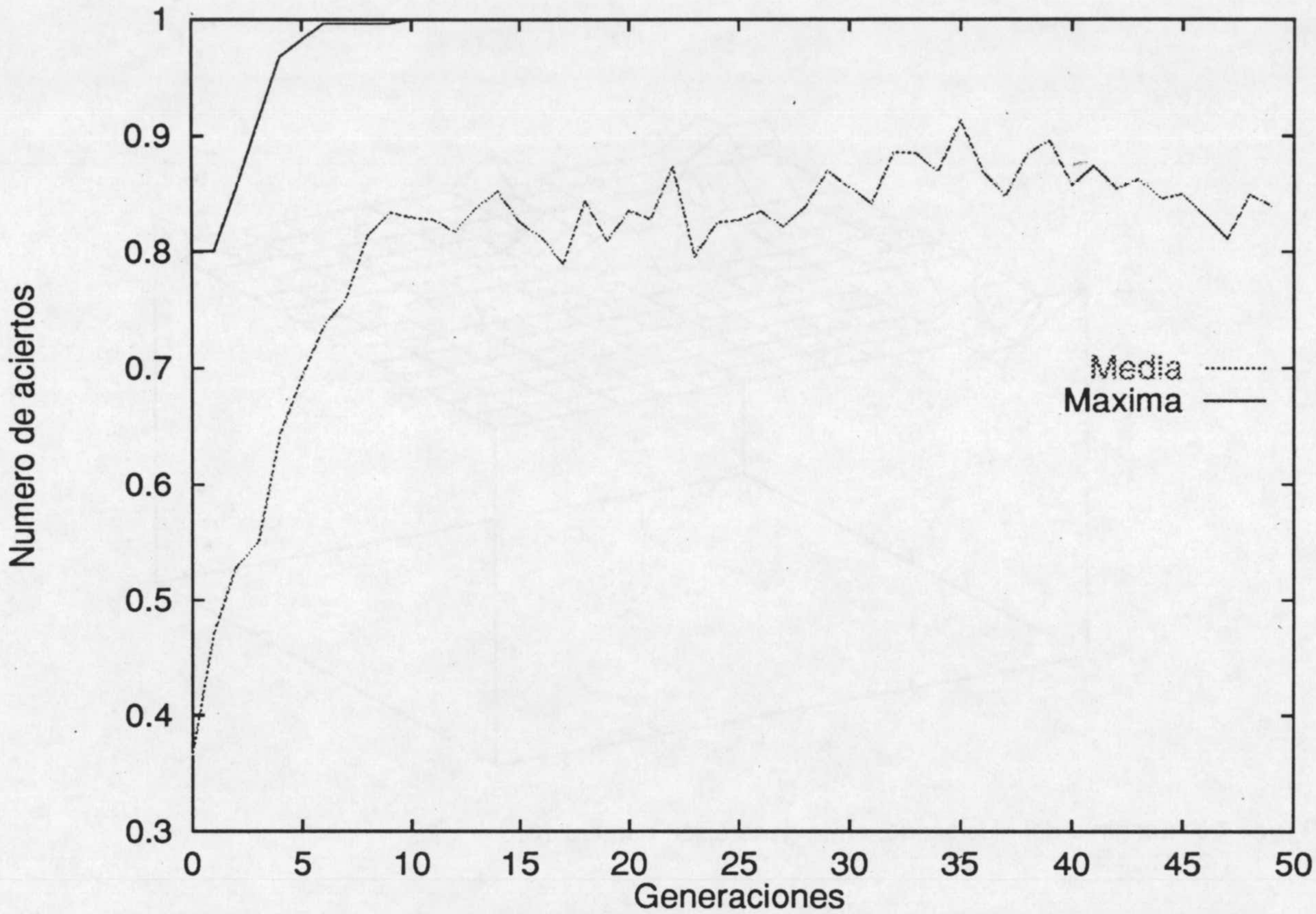



**Figura 7.10** Evolución del número de aciertos máximo y medio en un entrenamiento típico, en proporción sobre 1. Se suele escoger un genoma con número de aciertos máximo.

En la Figura 7.10 , donde se representa la evolución del número de aciertos en un problema de clasificación de 7 clases, cada una de las cuales con 40 elementos, se observa claramente que el máximo se alcanza en un número pequeño de generaciones (aproximadamente unas 20). Desde ese momento se comienzan a optimizar las otras cantidades. En la gráfica se representa el máximo número de aciertos para toda la población en cada generación, y el valor medio del número de aciertos para toda la población.

En la Figura 7.12 se muestra la evolución de la distorsión. Incluye dos líneas, que representan la distorsión mínima alcanzada por algún elemento de la población, y la media de la distorsión para toda la población. La distorsión comienza a optimizarse, y alcanza su mínimo, unas generaciones más tarde que el número de aciertos alcance su máximo; mientras que, en este entrenamiento particular, el máximo de aciertos se alcanza aproximadamente en la generación número 10 (véase Figura 7.10 ), el mínimo de distorsión se alcanza aproximadamente en la generación número 35 (véase Figura 7.12 ). La gráfica está representada para el mismo entrenamiento anterior. El valor límite de la distorsión es aproximadamente las dos terceras partes del tamaño típico de cada nube de puntos (cuyo

radio era de 0.1), para dar un valor límite de aproximadamente 0.1, al que tiende la distorsión mínima en la figura.

La evolución de la distribución de longitudes de los diccionarios que componen la población se puede ver de otra forma. Cada cierto número de generaciones, se observa cuántos diccionarios tienen una longitud determinada; la evolución temporal de este histograma, o distribución de longitudes en la población a lo largo del tiempo, es lo que se traza en la Figura 7.11 , donde se traza la representación tridimensional de la evolución de este *histograma de longitudes* a lo largo del tiempo. En abscisas se traza la longitud, la dimensión y es la dimensión temporal, mientras que en el eje *z* se representan el número de elementos de la población con una longitud determinada. Como se puede obsrvar, al principio del entrenamiento hay una distribución más o menos aleatoria de las longitudes, con un espectro amplio; este espectro de longitudes se va estrechando a lo largo del entrenamiento hasta dar un pico estrecho en el lugar de la longitud correcta en este caso, 7.

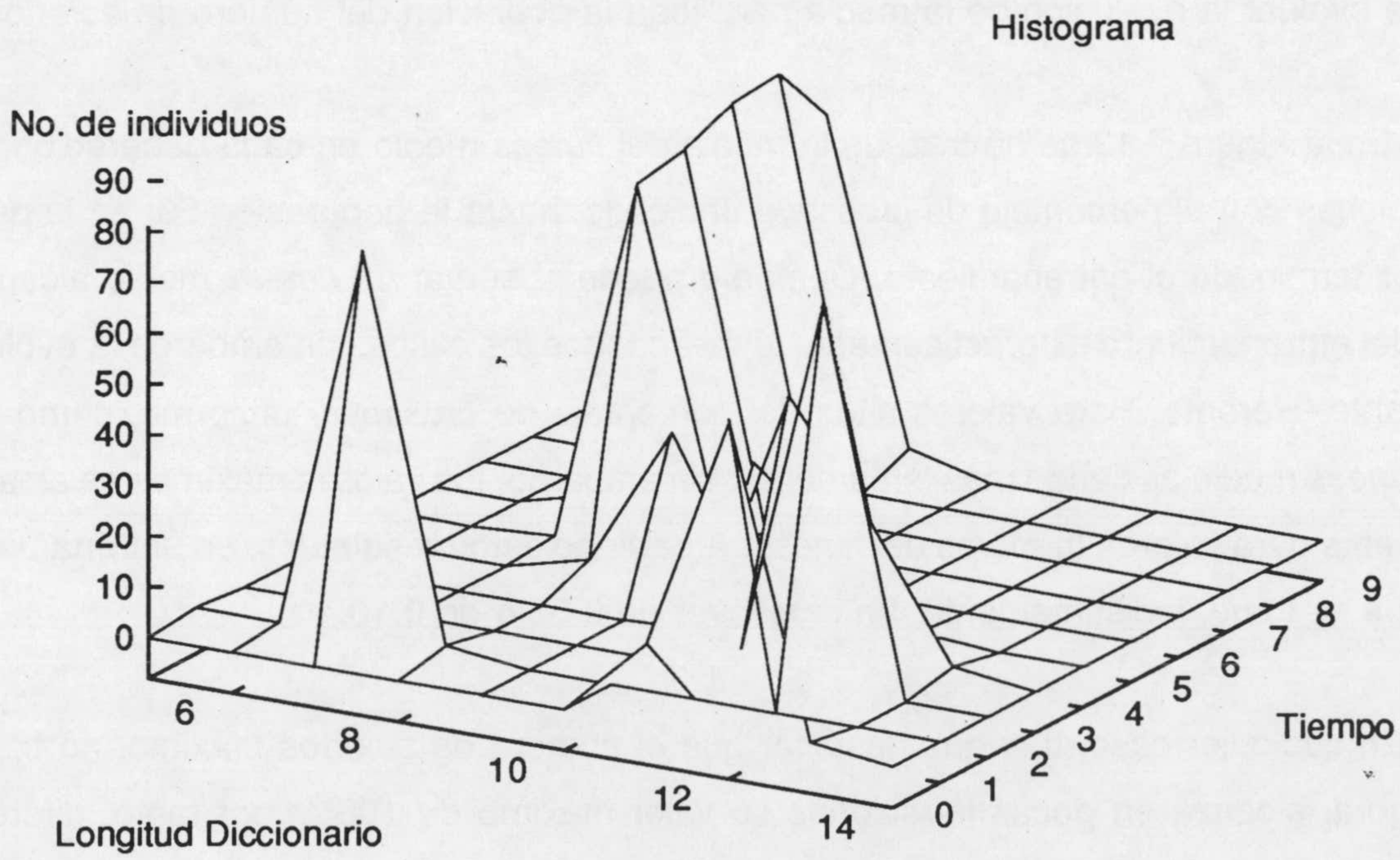



**Figura 7.11** Evolución de la distribución de longitudes de diccionarios en la población con el tiempo. La parte más cercana corresponde al principio del entrenamiento.

Por otro lado, algunos autores [Beer92] se han planteado el problema de aplicar el operador *crossover* a los algoritmos genéticos que optimizan redes neuronales, puesto que la combinación de dos configuraciones perfectamente válidas o incluso equivalentes (por ejemplo, dos configuraciones que sean idénticas salvo cambio de índices) pueden resultar una red neuronal inválida (ya que aparecerían alguna parte de la red neuronal repetida, y su eficacia se dividiría por la mitad). Este problema se presenta también en nuestro caso, con el agravante de que al utilizar genomas de longitud variable, puede suceder que se combinen dos diccionarios con buenos resultados, y den uno totalmente inválido.

Para evaluar como afecta el *crossover* a los resultados del algoritmo G-LVQ, se ha evaluado la eficacia de diferentes valores del mismo con un problema simple, de clasificación de 3 clusters. Se realizaron varios entrenamientos con diferentes valores iniciales aleatorios, para cada uno de los valores del *crossover*: 0.01, 0.1, 0.2, 0.5. Como se ha indicado en el capítulo 5, el algoritmo G-LVQ utiliza un tipo de *crossover* univorme, por lo tanto el parámetro de *crossover* indica cuantos bits se intercambiarán los dos genomas. En todos los casos, el valor máximo del número de aciertos se alcanzó en una o dos generaciones, por lo que se tuvo que evaluar la evolución de la media para toda la población del número de aciertos.

En la Figura 7.13 se ha trazado la media del *fitness* medio en cada generación, para 3 ejecuciones con el porcentaje de crossover indicado, hasta la generación 50, en la que se daba por terminado el entrenamiento. Como se puede observar, el *fitness* medio alcanzado al final del entrenamiento es prácticamente igual en todos los casos, sin embargo la evolución es bastante diferente. Para valores altos del porcentaje de *crossover* uniforme, como 0.5 y 0.7, el *fitness* medio asciende más lentamente; sin embargo, los valores máximos se alcanzan rápidamente para valores menores del fitness. A partir de estos resultados, en entrenamientos sucesivos se tomó, indistintamente, un crossover de 0.01 o de 0.10.

En cualquier caso, hay que destacar que el número de aciertos máximo, no trazado en la figura, alcanza en pocas iteraciones su valor máximo de 100%; por tanto, diferentes valores del *crossover* afectan el ritmo de crecimiento del *fitness* medio de la población, pero no a su posibilidad de éxito, puesto que en todos los casos se alcanza el máximo de aciertos, al menos para el problema de clasificación utilizado. Como se puede observar, en todos los casos se converge a unos valores similares del número medio de aciertos para toda la población.

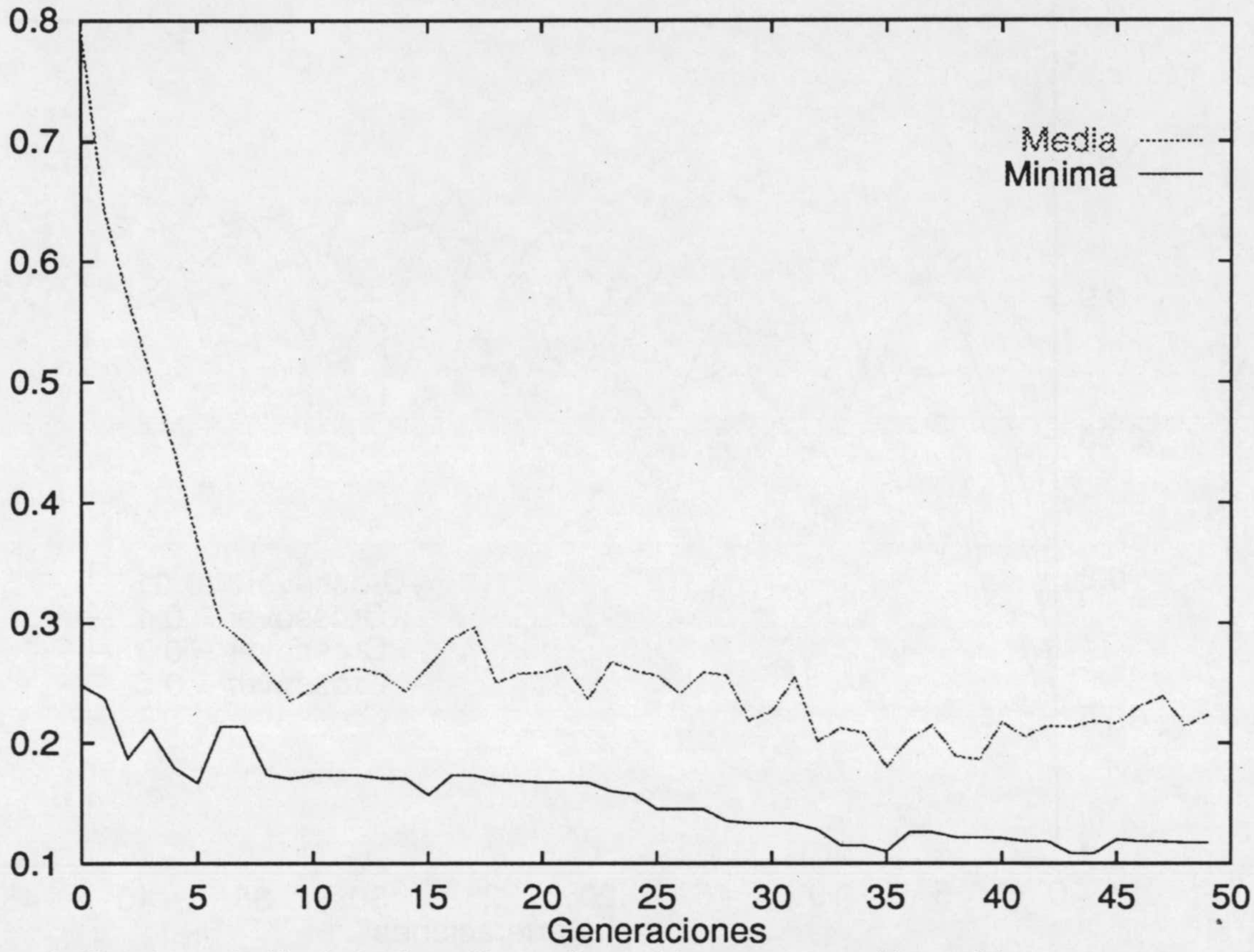



**Figura 7.12** Gráfica de evolución de la distorsión media y mínima para un problema típico.

En algunos casos [deG91][Beer92] se ha citado que el *crossover* debe de eliminarse totalmente en el caso de optimización de redes neuronales, ya que la combinación de los genomas de dos redes neuronales perfectamente válidas puede dar lugar a otra cuyo *fitness* sea prácticamente nulo. A esto se le denomina *efecto disruptor*, y no tiene porqué aparecer en cualquier tipo de red neuronal. De hecho, otros autores [Wer91] que han aplicado el *crossover* a la optimización mediante algoritmos genéticos de cierto tipo de redes neuronales, observan que no presentan ningún problema.

## 7.3 Evaluación de los operadores duplicación y eliminación

Una de las tesis que se tratan de demostrar en esta memoria es que la utilización de estos operadores permite encontrar la longitud óptima de un diccionario para el conjunto de entrenamiento dado. Los operadores genéticos duplicación y eliminación operan como un segundo nivel de algoritmo genético; en este caso la población está compuesta por los vectores código del diccionario, y el *fitness* de cada vector código es el número de veces que

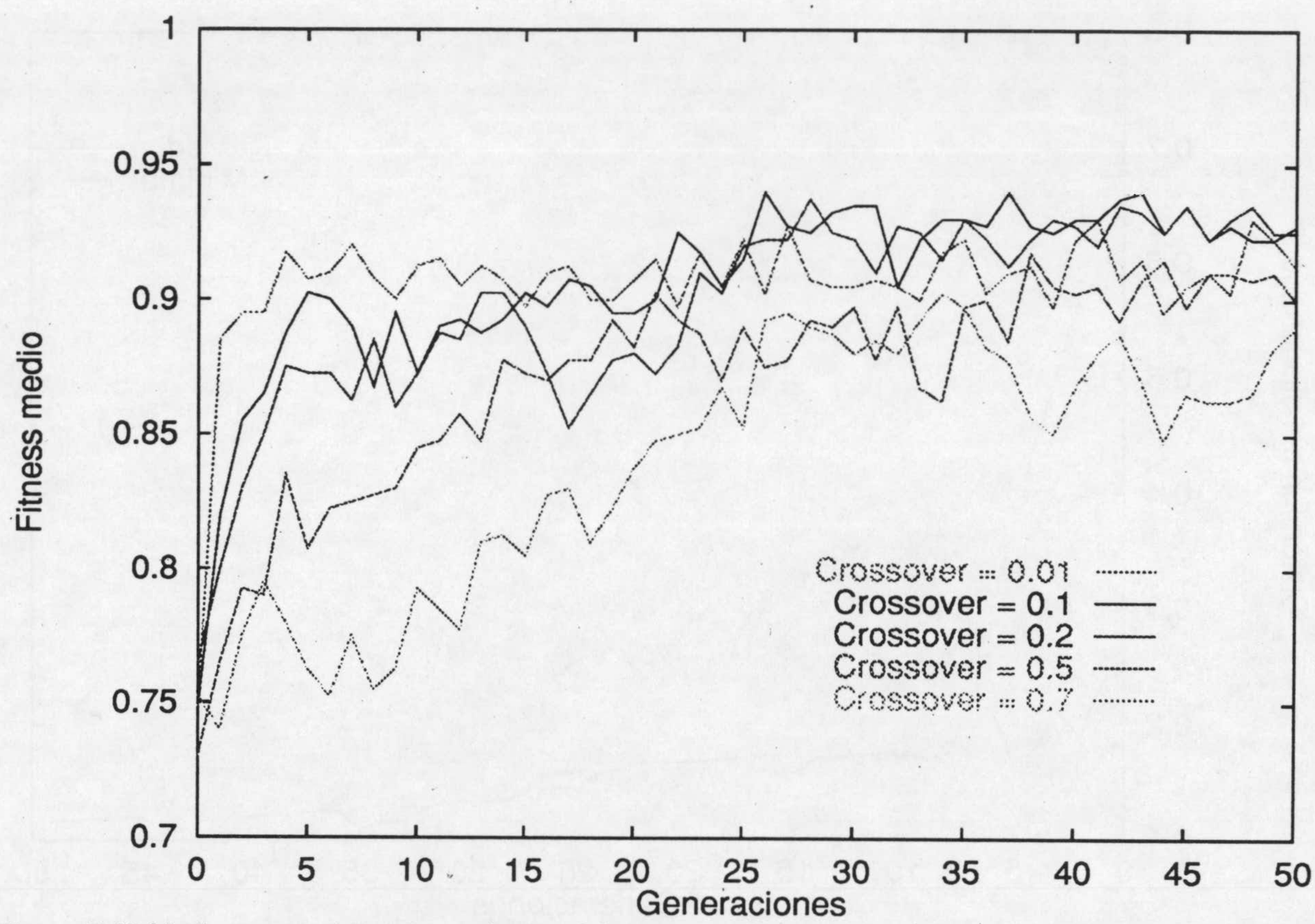



**Figura 7.13** Media para varios entrenamientos del *fitness* medio en cada generación para diversos valores del *crossover*.

el vector código ha acertado. Si ha acertado muchas veces, y por tanto está bien situado, cerca o en el centro de un grupo de vectores de su misma clase, eso significa que puede que sea mejor dividir la célula del espacio de entrada (es decir, la célula de Voronoi formada por todos los vectores del espacio de entrada a la que el vector diccionario responde) en dos partes, añadiendo otro vector. Por el contrario, si no ha acertado ninguna vez, se puede eliminar del diccionario sin afectar sus prestaciones; incluso si ha acertado pocas veces, puede ser necesario eliminarlo de la población de vectores con el objeto de que el operador duplicación pueda hallar mejores soluciones.

El paralelismo implícito del algoritmo genético implica que la aplicación de estos operadores no supone la eliminación de una buena solución; en cualquier caso, habrá probablemente otras buenas soluciones en la población. La aplicación de estos operadores genéticos supone siempre la exploración de nuevas posibilidades, sin desechar las antiguas, que ya están presentes en la población. La seleccion se encargará de eliminar las soluciones menos óptimas.

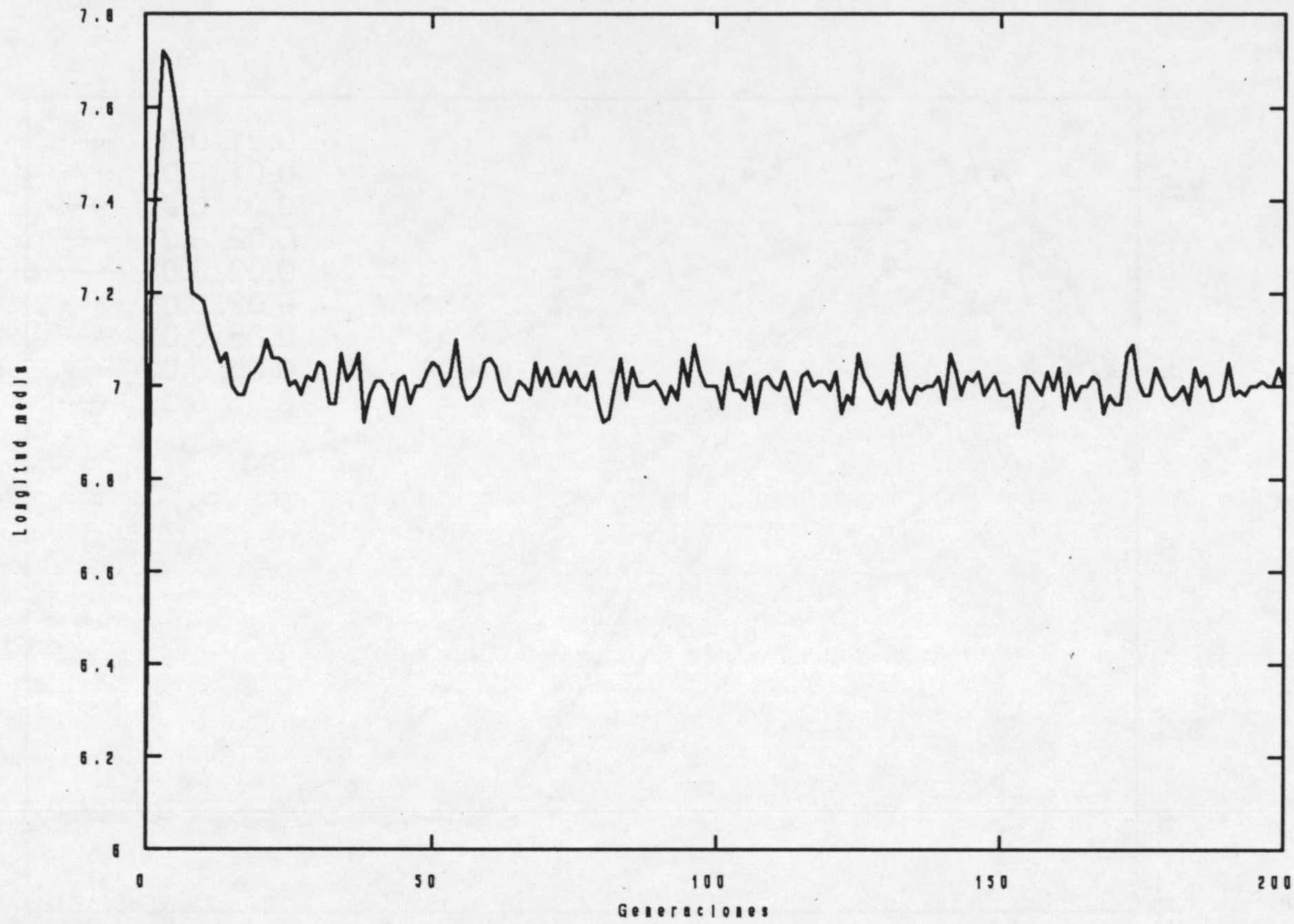



**Figura 7.14** Evolución de la longitud media del diccionario en un entrenamiento típico.

Para evaluar estos operadores, se aplicó el algoritmo G-LVQ al problema de clasificación en 7 clases, utilizando los valores implícitos de todos los parámetros. El resultado de la evolución de la longitud se puede apreciar en la Figura 7.14 , que representa la longitud media en un entrenamiento típico.

El algoritmo se inicia con una longitud inferior a la óptima. En las primeras generaciones, los genomas de longitud superior van dominando la población (obteniéndose una longitud media de aproximadamente 7.7), hasta que la población alcanza un óptimo en el número de aciertos (lo cual, evidentemente, puede no suceder durante todo el entrenamiento, en el caso de problemas difíciles). En ese momento, la longitud media empieza a disminuir, ya que empieza a optimizarse la longitud de los diccionarios, hasta llegar a la longitud óptima, a partir de lo cual la longitud media comienza a oscilar alrededor de la posición de equilibrio. En este caso se ha utilizado el mismo valor del parámetro de incremento que el de decremento, por haber observado empíricamente que este criterio proporciona resultados óptimos en problemas de este tipo, como se indica a continuación.

Este problema se analizará con más detenimiento en el capítulo siguiente.

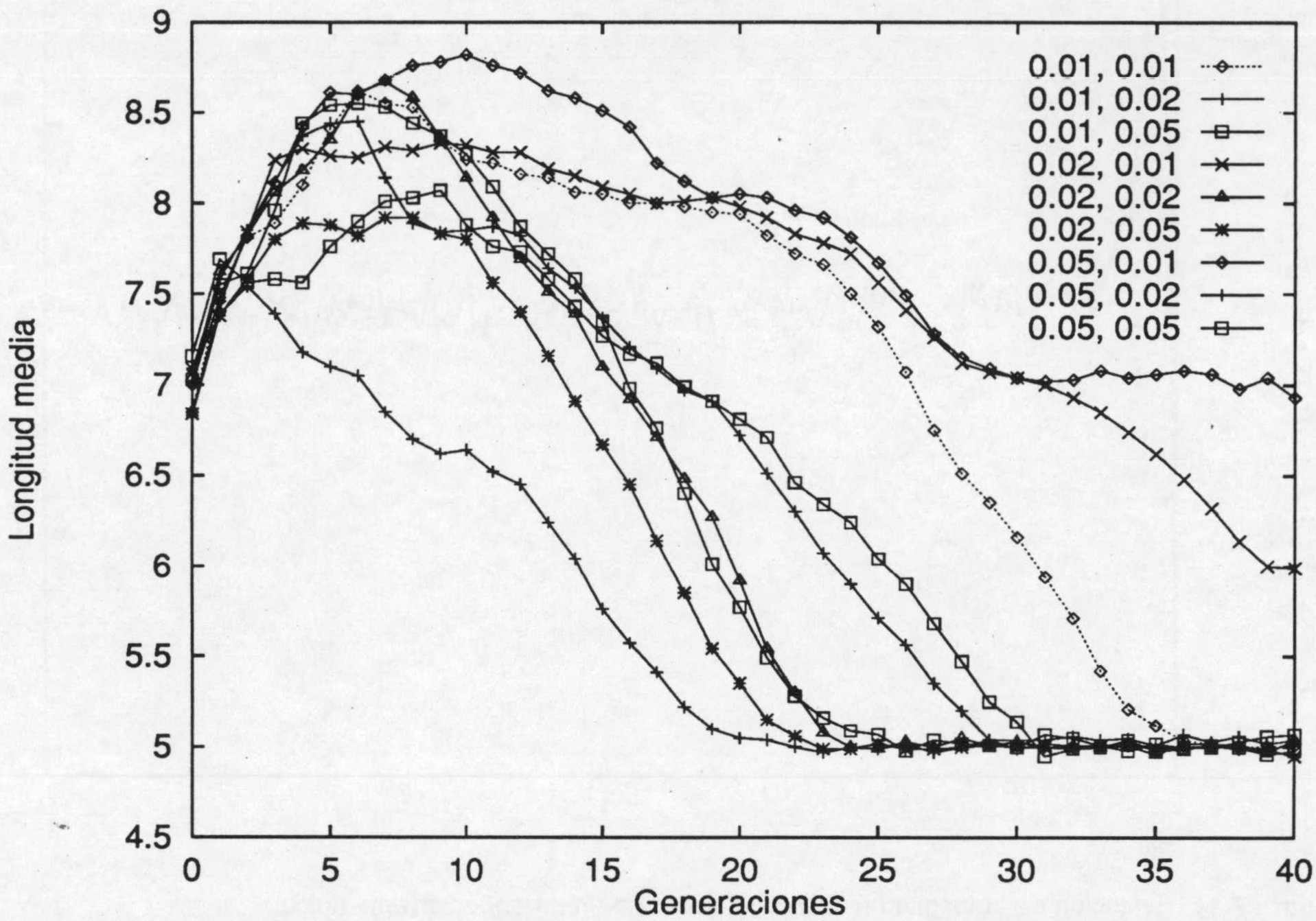



**Figura 7.15** Longitud media para diferentes valores de los parámetros de incremento (I) y decremento (D).

La aplicación de los operadores de incremento y decremento de la longitud dependen de dos parámetros que debe establecer el usuario. Para evaluar la influencia de los mismos en las prestaciones del algoritmo, se probaron diferentes combinaciones de valores sobre un problema de clasificación en 5 clases. Los valores probados para los parámetros de incremento y decremento fueron 0.01, 0.02 y 0.05. Se realizó un entrenamiento con cada combinación resultante de tomar uno de cada. La evolución de la longitud media resultante se presenta en la Figura 7.15 .

En esta figura, se puede observar que lo más adecuado es que los dos tengan valores iguales y pequeños, para que la longitud inicial media de los diccionarios, durante la fase de optimización del número de aciertos, no sea demasiado grande, enlenteciendo por tanto la operación del algoritmo. Si en cualquier caso se mantienen valores pequeños, por debajo de 0.05, los resultados serán apropiados, es decir, se encontrarán diccionarios de longitud adecuada, como se muestra en la Figura 7.15 .. Los peores resultados, con un incremento

excesivo de la longitud media de los diccionarios en la primera parte, corresponden a un parámetro de incremento de longitud 5 veces mayor que el de decremento. Sin embargo, esto no afecta en este problema a la eficacia en la búsqueda de clasificadores del algoritmo, aunque sí a su eficiencia en cuanto al tiempo que tarda en encontrar la solución.

En general, el variar los parámetros de duplicación y eliminación provocará que varíe el momento en el que se llegue a longitud óptima de equilibrio; aunque en el conjunto de parámetros por defecto, donde ambos parámetros son iguales a 0.01, suele dar buenos resultados, como se muestra en la figura anterior y se comprobará com más exactitud en el capítulo siguiente, al menos para problemas de pequeño tamaño. La conveniencia de variar los valores de estos parámetros se verá más adelante, cuando se aplique el algoritmo G-LVQ a problemas concretos.

## 7.4 Combinación de algoritmos genéticos y redes neuronales en G-LVQ

Durante la evolución de redes neuronales mediante algoritmos genéticos se produce un fenómeno, denominado *efecto Baldwin* [Ack91], mediante el cual los comportamientos aprendidos parecen incorporarse progresivamente al genoma. Esto, como defendía Darwin y posteriormente se demostró al hallar la base genética de la evolución, es imposible; sin embargo, hay ciertos comportamientos complejos que parecen ser aprendidos, y que, sin embargo, tienen una base genética.

En el caso que se trata en esta memoria, tratamos de entrenar redes neuronales mediante un algoritmo de cambio de pesos y, a la vez, mediante algoritmos genéticos. El efecto Badwin se manifestaría en una incorporación progresiva de los pesos calculados al genoma, extremo que, como se puede comprobar en el enunciado del algoritmo G-LVQ, es imposible, puesto que únicamente se heredan los pesos iniciales. Esto se puede comprobar mediante el error obtenido por los diccionarios previo al entrenamiento, es decir, directamente de los diccionarios codificados genéticamente, y de los mismos tras el entrenamiento LVQ.

Si se produce el efecto Baldwin, se observaría entonces una disminución de la diferencia entre el número de aciertos inicial máximo y el número de aciertos máximo al

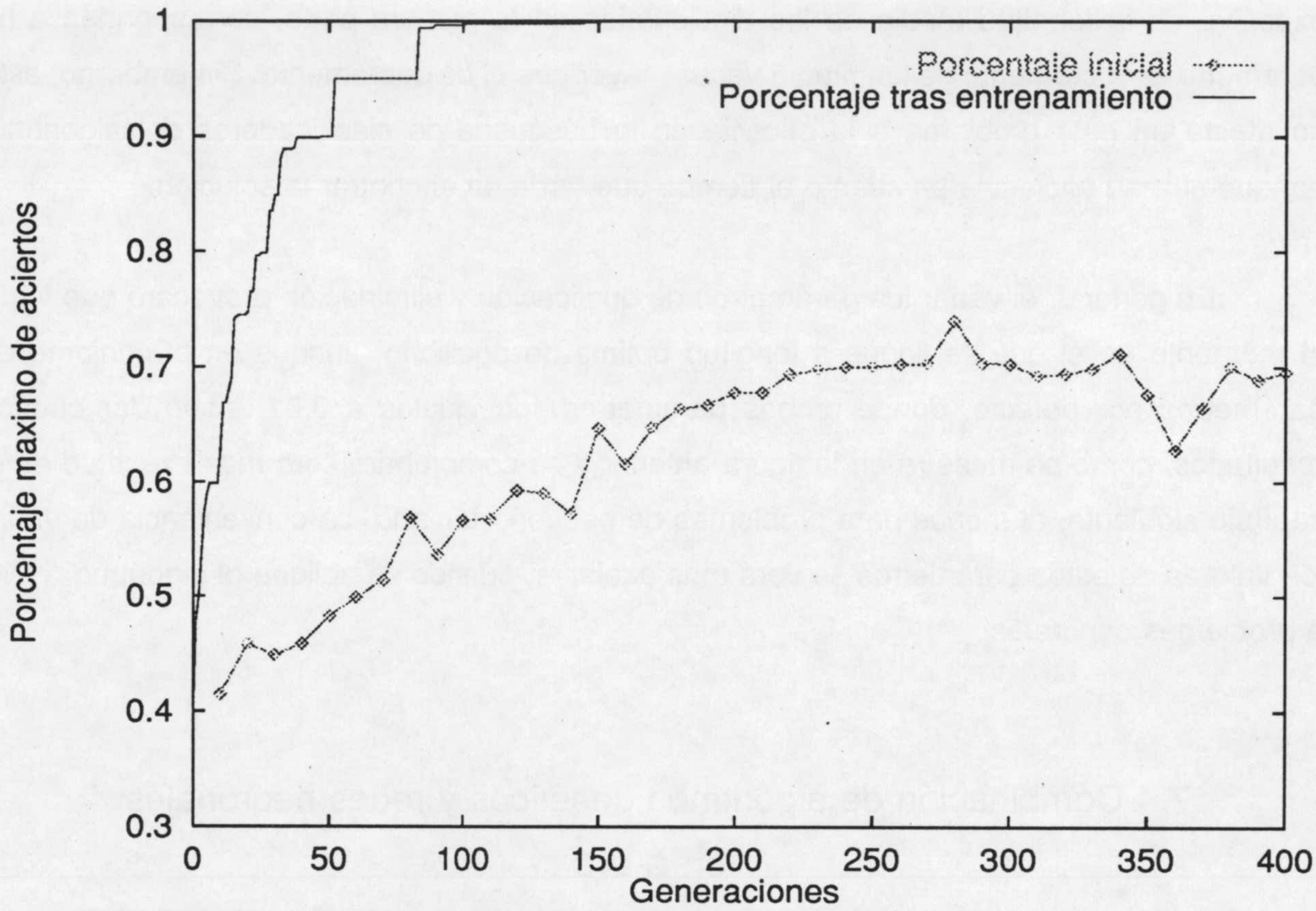



**Figura 7.16** Evolución del número de aciertos iniciales y tras el entrenamiento.

terminar el entrenamiento a lo largo de las sucesivas generaciones del algoritmo genético.

Para comprobar si el efecto Baldwin aparece en el algoritmo presentado en esta memoria, se ha utilizado un problema de clasificación en 15 clases. En la Figura 7.16 se muestra la evolución del número de máximo inicial y máximo tras el entrenamiento típico. El número de aciertos máximo inicial se calcula y graba solamente cada 10 generaciones, para no enlentecer la ejecución del algoritmo. La generación en la que se calcula y almacena el valor es lo que está indicado con un símbolo.

Se observa que, aunque al comienzo de la ejecución de G-LVQ el número de aciertos inicial, previo al entrenamiento LVQ, está bastante apartado del número de aciertos tras el entrenamiento, poco a poco ambos se van acercando, hasta ser prácticamente iguales. Hay que señalar que en ambos casos se trata del mejor resultado para un genoma, y no tiene porqué tratarse del mismo genoma el que da el mejor resultado inicial y el que da el mejor resultado tras el entrenamiento.

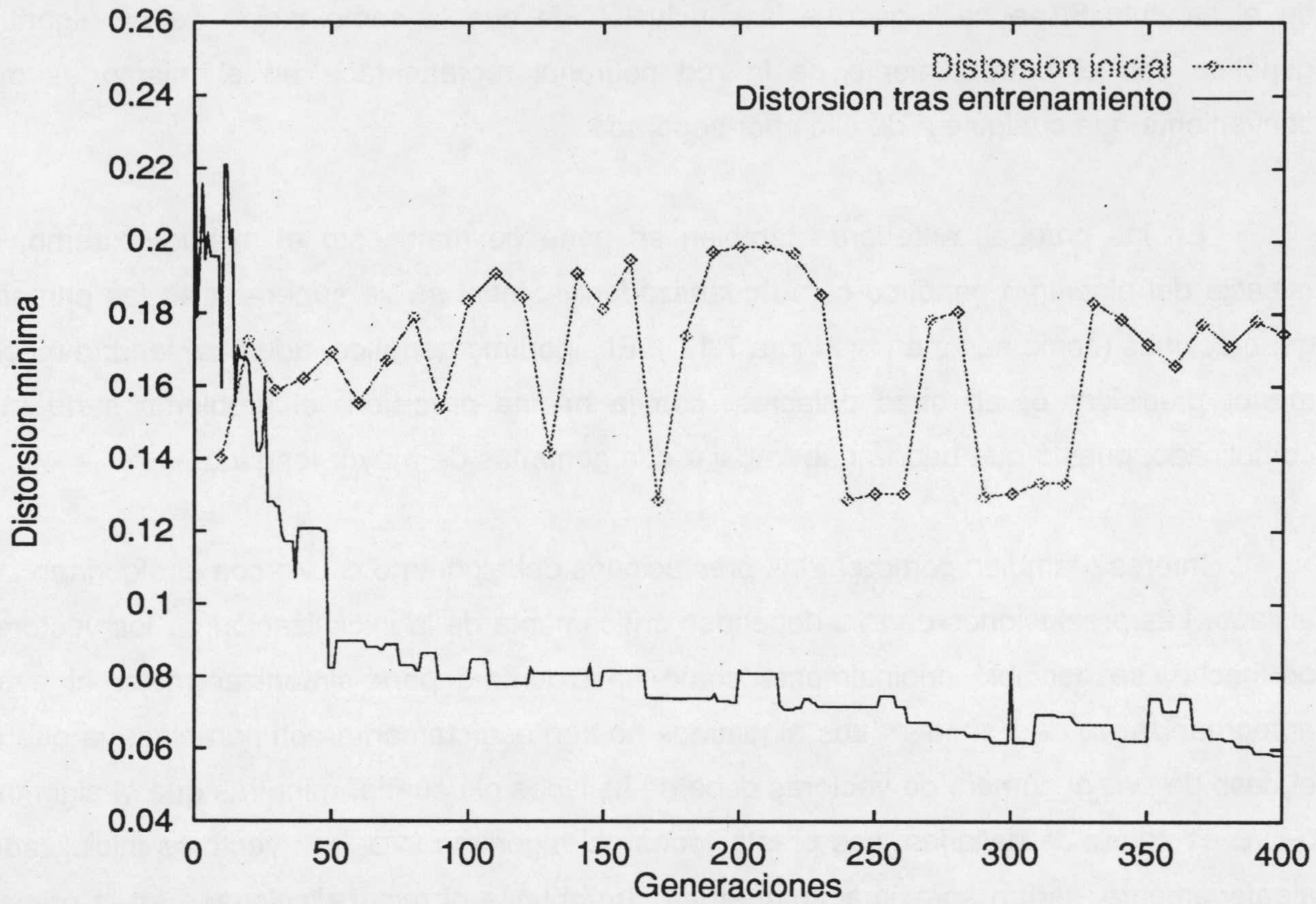



**Figura 7.17** Distorsión inicial y tras el entrenamiento, para un problema típico.

Para la distorsión se obtienen resultados bastante diferentes. En la Figura 7.17 , donde se observa la evolución de la distorsión inicial (cada 10 generaciones, con puntos unidos por una línea punteada) y de la distorsión tras el entrenamiento. Los valores aprendidos se van incorporando progresivamente al genoma, de forma que el algoritmo LVQ solamente tiene que variar ligeramente la posición de los vectores del diccionario, para hallar una buena solución. Por tanto, en el algoritmo G-LVQ se pone también de manifiesto el efecto Baldwin, como suele ser habitual en algoritmos que combinan redes neuronales y algoritmos genéticos. Sin embargo, debido a que la distorsión no es el principal parámetro a optimizar, se observan unas oscilaciones en las cuales la distorsión inicial se acerca a la distorsión tras el entrenamiento y posteriormente se aleja. En este caso no se produciría tan claramente el efecto Baldwin.

Es interesante comparar la eficacia como cuantizador vectorial del algoritmo G-LVQ frente a los dos algoritmos que lo componen. En algoritmos que comparten algunas características con G-LVQ, como el presentado por Monte [Mon93], y del que ya se ha hablado

en el capítulo 5, se ha llegado a la conclusión de que la combinación de un algoritmo genético con el entrenamiento de la red neuronal representada en el mismo es más conveniente que cualquiera de ellos por separado.

En las pruebas anteriores también se pone de manifiesto el mismo extremo. La eficacia del algoritmo genético como cuantizador vectorial se ve superada en las primeras generaciones (como se ve en la Figura 7.17 ). El algoritmo genético, además, tendría mucha menor precisión; o, en otras palabras, con la misma precisión, el problema sería más complicado, puesto que habría que trabajar con genomas de mayor longitud.

Interesa también comparar las prestaciones del algoritmo G-LVQ con el algoritmo LVQ aislado. Las prestaciones de LVQ dependen críticamente de la inicialización de los vectores; de hecho, se concibió originalmente como un algoritmo para sintonizar mejor el mapa autoorganizativo de Kohonen. Los algoritmos no son estrictamente comparables, ya que en el caso de LVQ el número de vectores debe de hallarlos el usuario, mientras que el algoritmo G-LVQ es capaz de hallarlas. Las prestaciones del algoritmo LVQ, con vectores inicializados aleatoriamente, serían aproximadamente las que obtiene el mejor diccionario en la primera generación del algoritmo G-LVQ.

Por otro lado, generalmente un diccionario que va a ser entrenado con LVQ no se inicializa con vectores aleatorios, sino que se utiliza algún criterio heurístico de inicialización, como

- inicializar los representantes de cada clase con los centroides de la misma,
- escoger el primer elemento que aparezca en el conjunto de entrenamiento de cada clase, o
- utilizar un algoritmo no supervisado de cuantización vectorial, como el *k-medias* o el SOM, para inicializar.

Este es aproximadamente el caso de las generaciones primeras en el algoritmo G-LVQ, ya que los genomas presentes en las mismas contienen ya buenas soluciones. Incluso así, en muchos casos son necesarias más generaciones para alcanzar el óptimo, con lo cual G-LVQ alcanza mejores resultados que LVQ, incluso aunque se utilicen con éste los mejores criterios de inicialización.

## 7.5 Complejidad temporal

Evidentemente, el tiempo empleado en la ejecución de todo el algoritmo dependerá fuertemente de los parámetros utilizados, sobre todo del número de generaciones. El número de generaciones debe de establecerse heurísticamente en función de los resultados que se quieran obtener.

El algoritmo G-LVQ, como ya se ha indicado en el capítulo 5, combina algoritmos genéticos y LVQ. Dado que la muestra se presenta una sola vez a cada diccionario, durante cada generación, frente a varias decenas de veces en los entrenamientos habituales, los vectores del diccionario se adaptan no sólo mediante el método de descenso de gradiente del error (descenso de la distorsión) del que les provee LVQ, sino también utilizando el muestreo estocástico del espacio de soluciones mediante *crossover*, mutación y selección proporcionado por el algoritmo genético. Esta misma característica hace que el algoritmo G-LVQ no sea excesivamente más lento que el LVQ habitual; el tiempo de entrenamiento será aproximadamente unas 200 veces superior al LVQ, dependiendo del número de generaciones (que suelen ser del orden de 100).

Como el algoritmo incluye la variación de la longitud de los diccionarios (mediante los operadores duplicación y eliminación), el tiempo empleado dependerá también de la longitud media de los diccionarios. En general, el tiempo empleado será función de N (número de generaciones), $D^2$ (número de dimensiones de la retícula al cuadrado), y $L_m$ (longitud media de los diccionarios). Por lo tanto, el tiempo empleado en el entrenamiento debería de crecer con la longitud media de los diccionarios, es decir, debería ser lineal en el tamaño del diccionario.

En realidad, como se aprecia en la Figura 7.18 , el crecimiento con el número de clases es algo superior al lineal. Esto se debe a que la exploración del espacio de longitudes posibles hace que para un número de clases superior, se exploren longitudes cada vez mayores. Para tres clases, la longitud media rara vez sube de 3.5; sin embargo, para 5 clases, es habitual durante un espacio considerable del entrenamiento una longitud media superior a 6; y para longitud igual a 10, es igualmente posible encontrar longitudes medias superiores a 12. Por lo tanto, aunque la longitud final dependa del número de clases, la

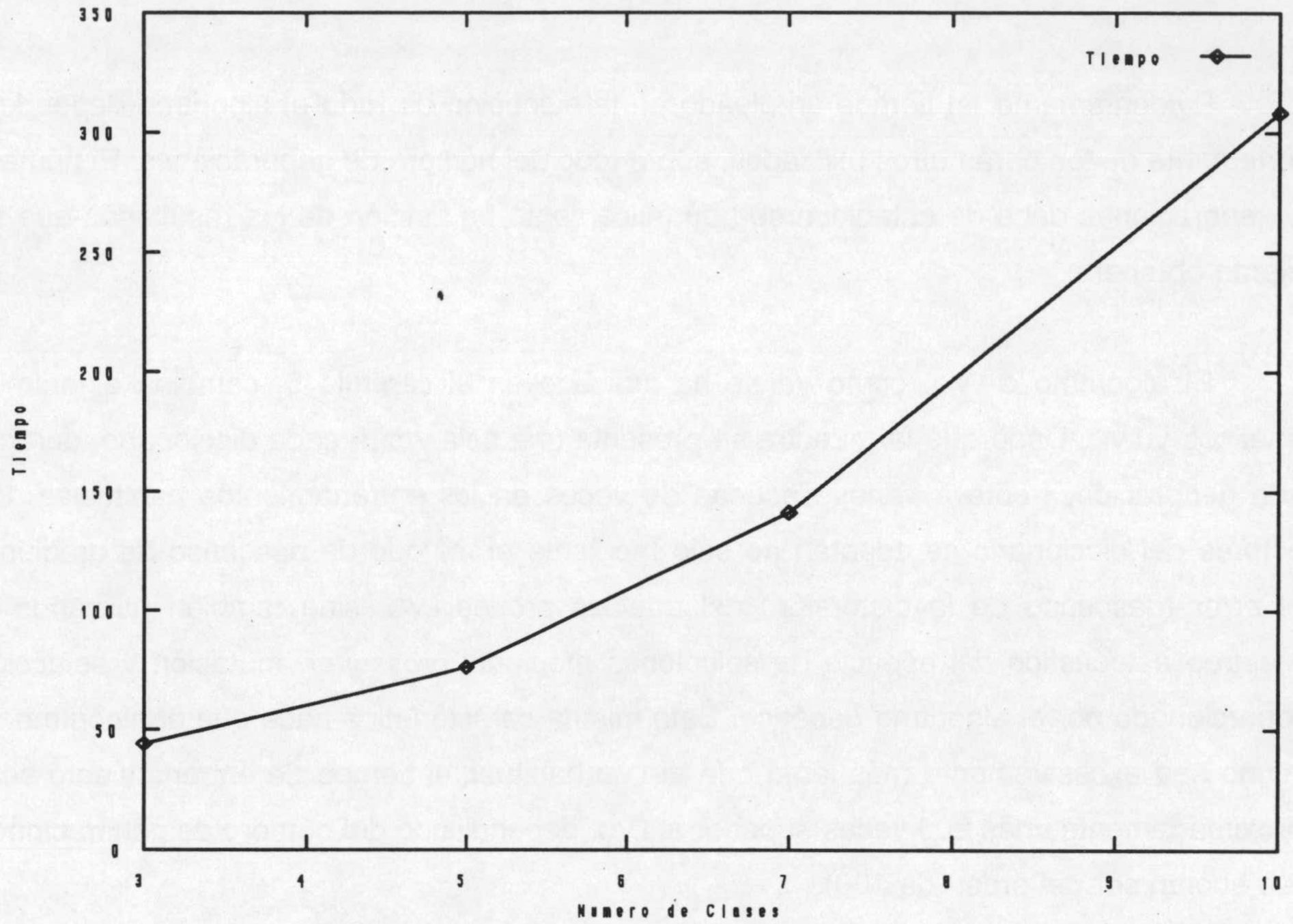



**Figura 7.18** Crecimiento del tiempo en minutos dedicado al entrenamiento con el número de clases.

longitud media crece de forma no lineal con el número de clases. A esto se debe el aumento del tiempo de forma no lineal. Es decir, que aunque es lineal en la longitud media del diccionario, no así en el tamaño óptimo del diccionario, puesto que previamente a alcanzarlo, se exploran diccionarios de mayor longitud.

En cuanto al tiempo de entrenamiento, como se ve es considerable, llegando a superar 5 horas para 10 clases; aunque todos los tiempos han sido tomados en un PC con microprocesador Intel 486DX a 50 MHz; los tiempos son sensiblemente inferiores en una SPARC a 40 MHz, o en una Silicon Graphics Iris a 100 MHz). Sin embargo, hay que tener en cuenta que una vez realizado el entrenamiento y hallado el diccionario, el tiempo de explotación se reduce al necesario para una comparación con cada uno de los vectores del dicionario, y por lo tanto depende sólo del tamaño del diccionario. Esto compensa el tiempo necesario en hallar el diccionario adecuado.

Como se ha indicado anteriormente, la complejidad del algoritmo sería lineal en la longitud media de los diccionarios y en el número de generaciones, y cuadrática en el número de dimensiones de la malla de procesadores. Sin embargo, como también se ha puesto de manifiesto, la longitud media de los diccionarios es un parámetro difícil de estimar, y las pruebas realizadas indican que puede crecer de forma exponencial. Incluso así, las constantes que multiplican a estas variables son bastante altas.

## 7.6 Conclusiones

En este capítulo se han evaluado las prestaciones del algoritmo G-LVQ aplicándolo a problemas simples. Inicialmente se ha evaluado las extensiones y variaciones que introduce el algoritmo G-LVQ sobre un algoritmo genético clásico, a saber, la restricción del emparejamiento de los genomas y la introducción de genomas de longitud variable. De esta forma, se ha podido comprobar que la restricción espacial del algoritmo genético no afecta a la eficacia del mismo, pudiéndose hallar óptimos globales, sin que se produzca el fenómeno de la consanguineidad. A continuación, se ha comprobado que los operadores creación y destrucción (duplicación y eliminación) inducen una exploración eficaz del espacio de redes neuronales con diferentes longitudes. También se ha evaluado cuál es la combinación de valores más adecuada para los mismos.

El efecto de un operador clásico en los algoritmos genéticos, el *crossover*, ha sido también evaluado, llegando a la conclusión de que lo más adecuado es que se mantenga en valores pequeños en el caso del algoritmo G-LVQ. También se ha evaluado el tiempo necesario para ejecutar el algoritmo en un ordenador personal compatible PC de altas prestaciones. Los algoritmos genéticos consumen generalmente mucho tiempo, pese a ello, el algoritmo G-LVQ consigue tiempos razonables, incluso para problemas de clasificación complicados, siendo más rápido que otros y sólo un orden de magnitud más lento que LVQ. El tiempo de ejecución crece de modo no lineal con el número de clases.

Con los ejemplos sencillos que se han utilizado, se ha puesto de manifiesto también que el algoritmo G-LVQ obtiene mejores resultados que cualquiera de los algoritmos que combina, algoritmos genéticos y LVQ, realizando la misma tarea, la cuantización vectorial. En

este extremo, el resultado aquí presentado coincide con los hallados por otros autores en algoritmos incrementales o genéticos aplicados a LVQ: la combinación de algoritmos genéticos y redes neuronales de aprendizaje competitivo resulta más eficiente que sólo un algoritmo genético o una red neuronal por separado.

# 8 Resolución de problemas de clasificación con el algoritmo G-LVQ

## 8.1 Introducción

En el capítulo anterior se ha analizado el algoritmo G-LVQ, evaluando sus prestaciones como optimizador de cuantizadores vectoriales, así como su propiedades de convergencia. A partir de esta evaluación se ha concluido que el algoritmo G-LVQ es suficientemente robusto como para esperar buenos resultados con una amplia gama de parámetros iniciales. En este capítulo se va a aplicar el algoritmo a varios problemas sintéticos clásicos dentro de la literatura de reconocimiento de patrones, así como a otros problemas reales dentro del campo de reconocimiento de patrones. Inicialmente, en la sección 8.2, el algoritmo G-LVQ se aplicará a problemas sencillos, como la clasificación de varias muestras etiquetadas divididas en diversos grupos que no se solapan entre sí. La mayor dificultad de este problema consiste en hallar el número de clases; una vez halladas, el algoritmo encuentra fácilmente los centros de estas clases que presentan una mínima distorsión.

Los problemas analizados en la sección siguiente, el de los círculos concéntricos, el problema no linealmente separable de Hart, y la simulación de la función XOR son más complicados, al menos para cuantizadores vectoriales clásicos, puesto que dependen críticamente de las condiciones de inicialización de los vectores del diccionario. Estos problemas se contemplarán en la sección 8.3. En el primer caso, el centroide de la distribución cae fuera de la distribución en sí. Además, son problemas que necesitan varios centros por clase, a diferencia de los anteriores, en los cuales se podían conseguir buenos resultados con un solo centro por clase, la posición de estos centros debe ser tal que minimice el error de clasificación y/o maximice la distorsión.

A veces, al aplicar un algoritmo con buen comportamiento en problemas sintéticos a problemas reales se hallan notables diferencias en el éxito obtenido. Por ello es conveniente

probarlo sobre problemas obtenidos de sensores reales. En esta memoria, se ha escogido un problema de clasificación de imágenes de macromoléculas obtenidas por microscopía electrónica, que previamente han sido sometidas a un preproceso para analizar su simetría circular. El segundo problema considerado es la clasificación de imágenes de sonar, un problema considerado muy adecuado para evaluar el éxito de un algoritmo de clasificación. Estos ejemplos se analizan en el apartado 8.4

## 8.2 Clasificación de muestras linealmente separables

El principal objetivo de la aplicación de G-LVQ a estos problemas es evaluar cuán eficientemente explora el algoritmo G-LVQ el espacio de diccionarios con diferente longitud. Dado que inicialmente, la longitud de los diccionarios es aleatoria, el algoritmo debe de encontrar los diccionarios con longitud óptima para obtener el número máximo de aciertos de clasificación, y de forma, además, que no se obtengan centros que no respondan a ninguna zona del espacio de entrada.

El problema que se pretende resolver es la clasificación de muestras agrupadas en varias nubes, cada una de las cuales tiene una distribución gaussiana de puntos alrededor de un centro generado aleatoriamente. Se han hecho pruebas para 3, 5, 7, 10 y 13 nubes, cada una de las cuales con 40 o 50 elementos, y una anchura de gaussiana entre 0.05 y 0.1. Las nubes están situadas en puntos aleatorios, no solapados, del intervalo x,y=[-1,1].

En la Tabla 8.i se muestran los valores de los parámetros para los cuales el algoritmo G-LVQ presente buenos resultados para cada caso. Como se puede observar, en la mayoría de los casos se utilizan los valores por defecto para los parámetros de incremento y decremento (0.01 en los dos casos), siendo necesario cambiarlos solamente para un número de *clusters* o grupos mayor o igual que diez. Como era de esperar, se observa que el número de generaciones necesarias para el entrenamiento crece con el número de grupos a clasificar, así como la generación en la cual se halla un genoma con un número de aciertos igual al 100%.

En las columnas "Generaciones" y "Tiempo total" de la tabla se indican el número de

**Tabla 8.i** Parámetros utilizados y resultados obtenidos en problemas de clasificación simples.

| Número de clases | Incremento | Decremento | Gener. | Tiempo total | |
|---|---|---|---|---|---|
| | | | | Gen. | Tiempo |
| **3** | 0.01 | 0.01 | 1 | 50 | 6-7 min. |
| **5** | 0.01 | 0.01 | 3-7 | 59 | 14-24 min. |
| **7** | 0.01 | 0.01 | 8-10 | 300 | 230 min. |
| **10** | 0.01 | 0.03 | 19-24 | 200 | 272-400 min. |
| **15** | 0.05 | 0.01 | 100 | 400 | >2600 min.* |

**Notas:** La columna **Generaciones** indica el tiempo necesario para alcanzar un número de aciertos del 100%
En la columna **Tiempo Total** se indica el número de generaciones que ha durado el entrenamiento en la columna **Gen.**, y el tiempo máximo y mínimo en minutos que han tardado diversas ejecuciones del algoritmo, en un ordenador compatible PC a 50 MHz, en la columna **Tiempo**. El número menor se refiere al entrenamiento que ha durado menos, y el superior al entrenamiento más largo.
*El tiempo para este entrenamiento ha sido tomado en una Silicon Graphics Iris.

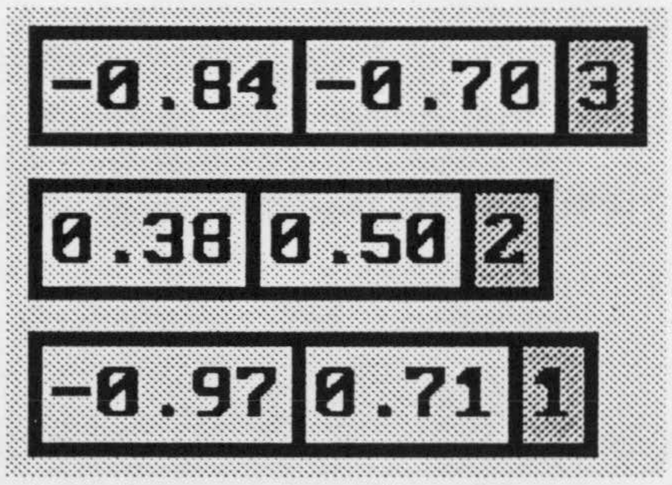



**Figura 8.1** Genoma de uno de los diccionarios ganadores. Los dos componentes de la izquierda son los iniciales del vector. La etiqueta de clase aparece sombreada.

generaciones necesarias para alcanzar un 100% de aciertos, y el tiempo que ha durado todo el entrenamiento. El tiempo indica la duración máxima y mínima observada a lo largo de varios entrenamientos, típicamente entre 3 y 5, y los tiempos necesarios para ejecutar el algoritmo en un ordenador compatible PC con microprocesador Intel 486 a 50 MHz.

En el caso de la *clasificación de 3 clases*, como se puede observar en la Tabla 8.i, el entrenamiento es sumamente rápido, aunque el problema no reviste ninguna dificultad. Por

su rapidez, este problema ha sido utilizado para evaluar diferentes valores del parámetro de crossover, como se mostró en el capítulo 7. En estos casos, la sencillez del problema hace que G-LVQ calcule varias soluciones con una distorsión similar, igual longitud e igual porcentaje de aciertos, del 100%. Este resultado es consistente, y se ha hallado en todos los entrenamientos realizados para este tamaño de problema.

Para ilustrar este resultado se han presentado de dos formas los diccionarios ganadores. Para mostrar como aparecen los valores iniciales del diccionario codificados en el genoma, se ha representado, utilizando el programa descrito en el capítulo 6, el genoma de uno de los diccionarios ganadores (Figura 8.1). En este genoma, que contiene tres vectores (tamaño ideal para un problema de clasificación de tres clases), se representa cada vector en una línea diferente. Los dos números encuadrados en fondo blanco son las dos coordenadas del vector que se decodificarán en el momento del "nacimiento" en el valor inicial de uno de los vectores del diccionario, previo al entrenamiento, mientras que la cifra encuadrada en gris es la etiqueta de clase.

Durante el entrenamiento LVQ, los valores de estos vectores se van modificando, hasta dar lugar al diccionario ya entrenado mostrado en la Figura 8.2. En esta gráfica se representa toda la información relevante al problema de clasificación de tres clases: el conjunto de entrenamiento, representado mediante una serie de puntos, el genoma, con sus diversos elementos representados mediante triángulos, que están unidos con flechas al vector en el que se convierten tras el entrenamiento, representado mediante diferentes símbolos, un símbolo para cada vector del diccionario.

Como se ve, hay un sólo vector por clase, con lo cual el diccionario es óptimo tanto en longitud como en número de aciertos. Además, los vectores se hallan bastante centrados dentro del conjunto de entrenamiento, con lo cual, aunque no se puede afirmar que sean óptimos en distorsión, sí se ha hallado un valor bastante aceptable. En cualquier caso, el resultado principal que se pretende obtener es una máxima exactitud en la clasificación con una longitud mínima, pues estos son los parámetros que más impacto tienen en las prestaciones de un clasificador; aunque también se trata de optimizar la distorsión, este parámetro se coloca en último lugar con respecto a los demás.

El caso del problema de *clasificación de 5 clusters* se considera también sencillo, y

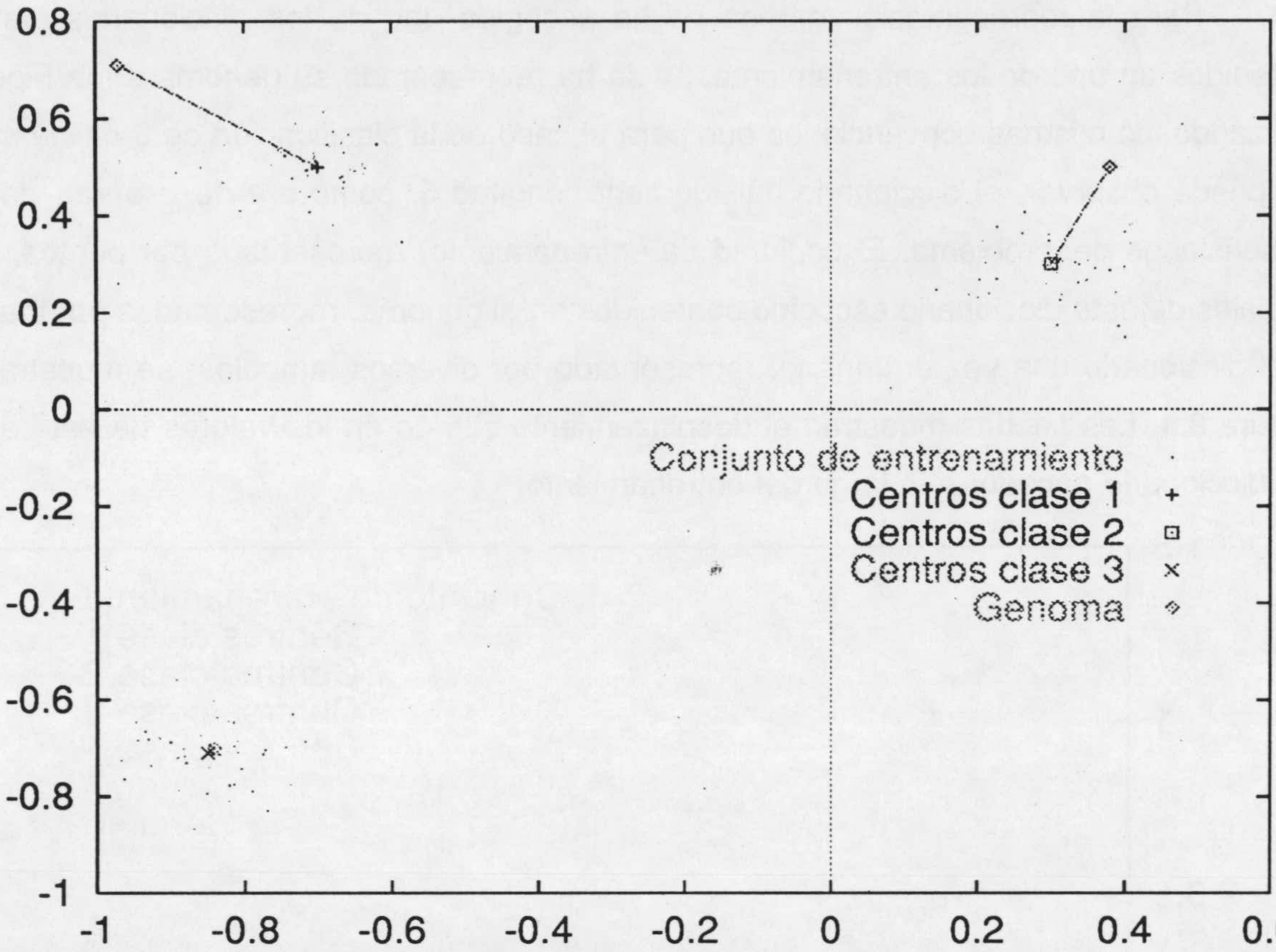



**Figura 8.2** Conjunto de entrenamiento (puntos), diccionario óptimo (puntos diversos tipos)unido por flechas a su genoma (triángulos) para el problema de clasificación de 3 clases.

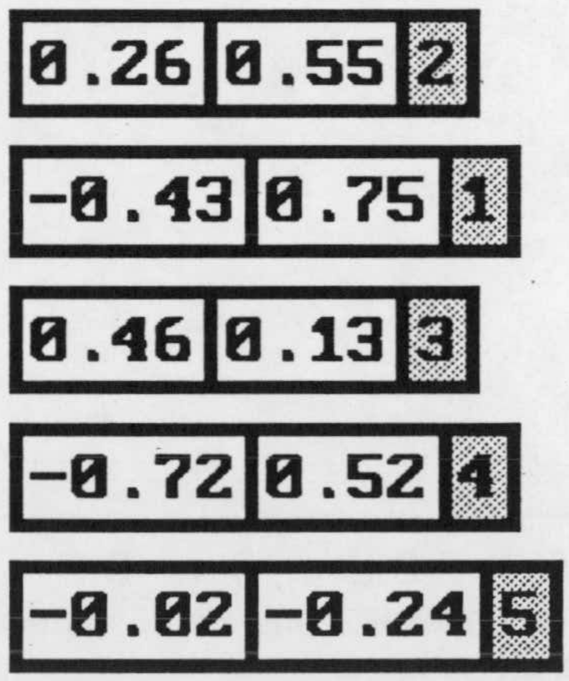



**Figura 8.3** Genoma de un diccionario ganador para el problema de clasificación de 5 clases.

de hecho, se ha utilizado en el capítulo 7 para evaluar diferentes combinaciones de los parámetros de incremento y decremento. Aunque por razones obvias el entrenamiento tarda un poco más que en el caso del problema de clasificación de 3 *clusters*, y el máximo de aciertos (igual al 100%) se alcanza unas generaciones más tarde. Igualmente se obtienen a lo largo del entrenamiento diversos diccionarios con un porcentaje de aciertos y longitud óptimos.

Para la representación gráfica se ha escogido uno de los diccionarios ganadores obtenidos en uno de los entrenamientos, y se ha representado su genoma en la Figura 8.3, utilizando las mismas convenciones que para el caso de la clasificación de 3 *clusters*. Como se puede observar, el diccionario hallado tiene longitud 5, como era de esperar, dadas las dimensiones del problema. El conjunto de entrenamiento, representado por puntos, valores iniciales de este diccionario escogido contenidos en el genoma, representados por triángulos, y el diccionario una vez entrenado, representado por diversos símbolos, se muestran en la Figura 8.4. Las flechas muestran el desplazamiento sufrido en los valores de este ejemplar de diccionario ganador a lo largo del entrenamiento.

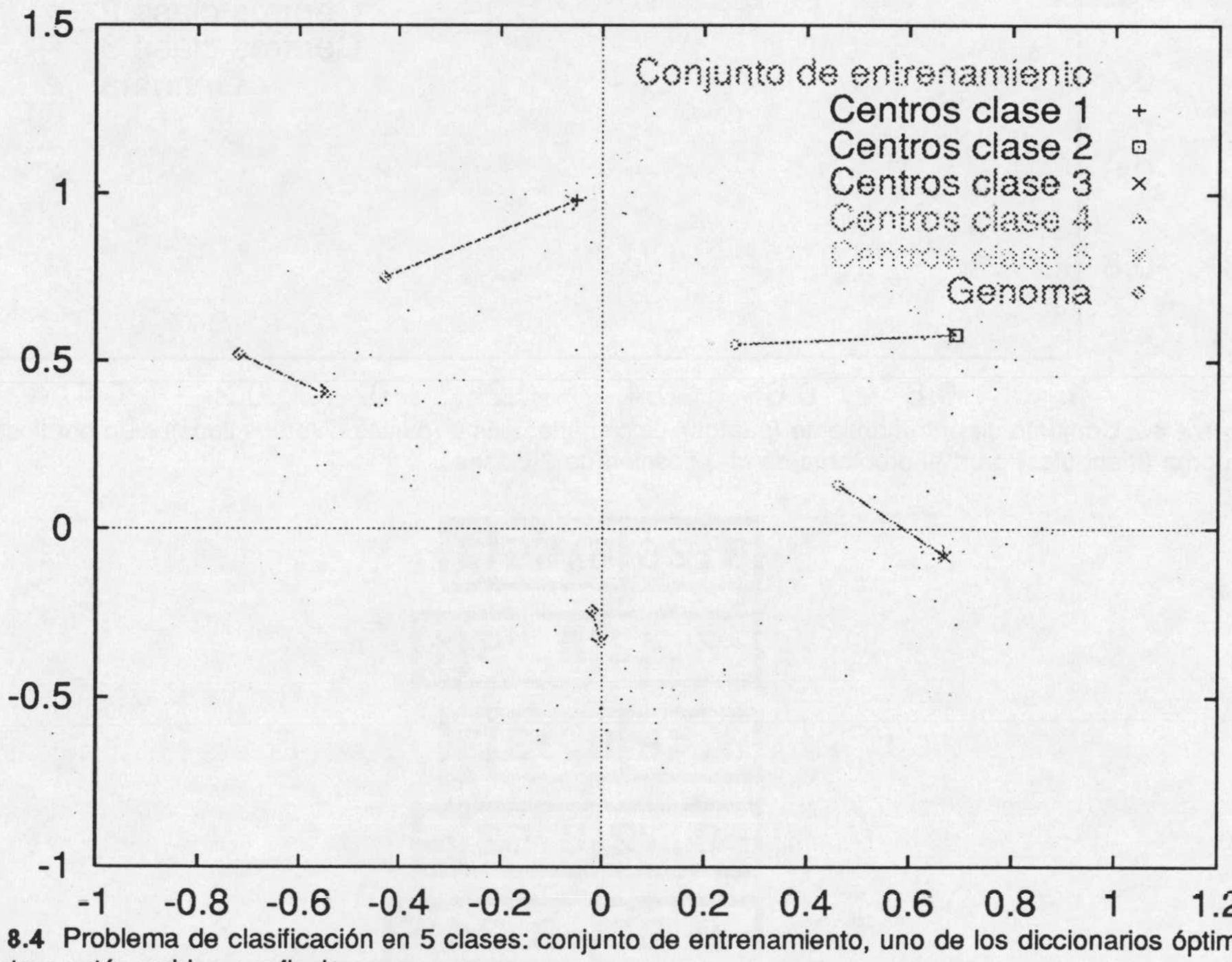



**Figura 8.4** Problema de clasificación en 5 clases: conjunto de entrenamiento, uno de los diccionarios óptimos; los centros están unidos por flechas a su genoma.

Similares buenos resultados se obtienen al calcular el diccionario para clasificación de 7 clusters. En este caso disminuye la cantidad de diccionarios óptimos calculados, lo cual se corresponde con una mayor amplitud del espacio a explorar. Uno de estos diccionarios, y el genoma correspondiente, se muestra en la Figura 8.5, junto con el conjunto de entrenamiento. En este caso no se han unido los genomas con el vector correspondiente del diccionario para no hacer más compleja la representación. Una representación del genoma

se puede observar en la Figura 8.6, donde se puede observar que el genoma ganador tiene longitud 7.

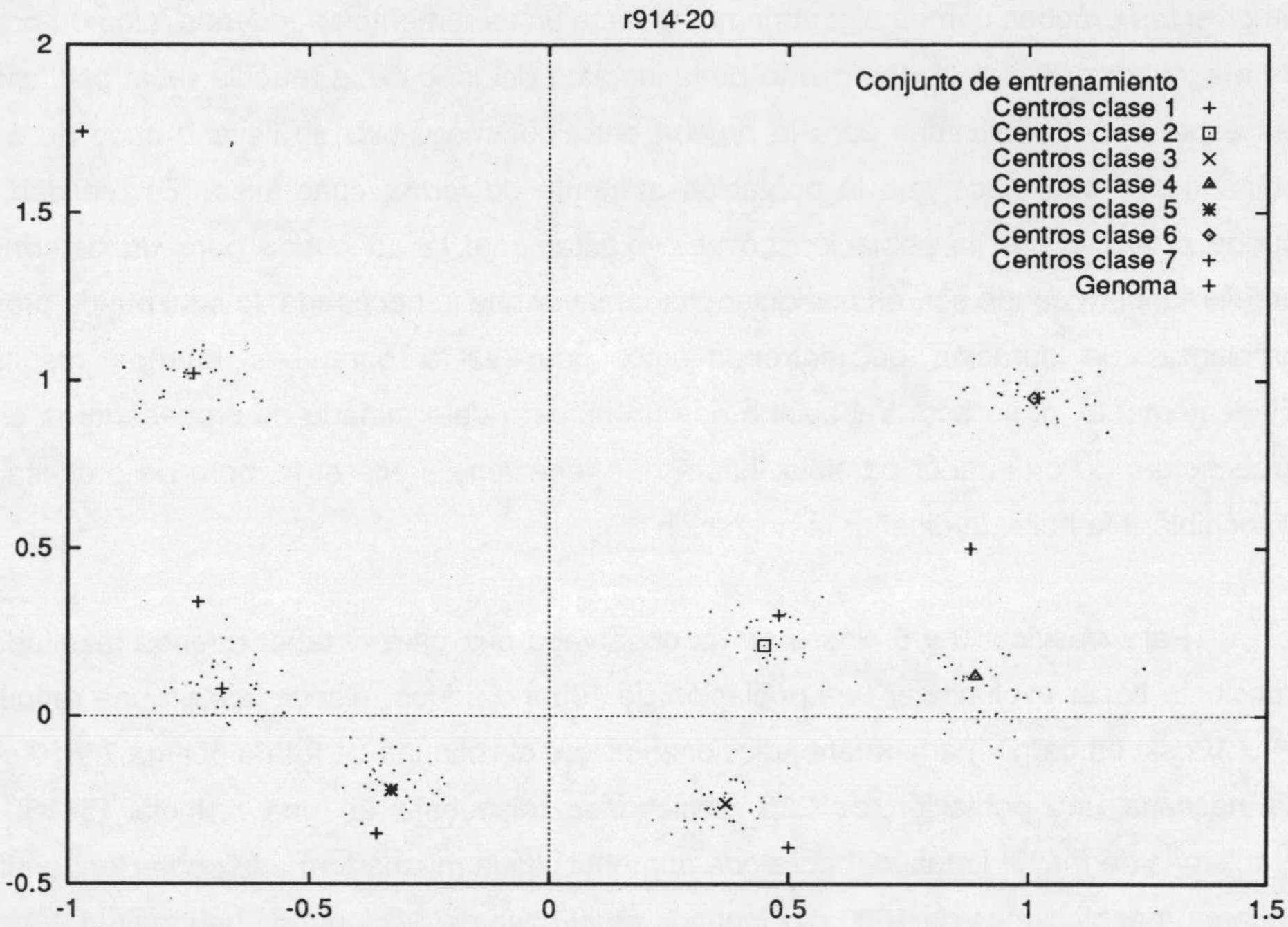



**Figura 8.5** Conjunto de entrenamiento (puntos), diccionario calculado (puntos de diversos tipos) y genoma correspondiente para el problema de clasificación en 7 clases.

Para problemas de clasificación de más de 7 clases, es necesario aumentar la población con respecto a la utilizada en los primeros casos, que era de 100, dispuesta en un retículo de 10x10, ya que, como se ha indicado en los capítulos 5, 6 y 7, la población aparece dispuesta en una retícula. El número de elementos de la población, según la teoría clásica de algoritmos genéticos [Gol85] debe ser proporcional al número de bits del genoma. Esta relación rara vez se expresa de forma explícita; únicamente, la experiencia dicta que, cuando la población queda estancada en mínimos locales (en lo que se suele denominar *consanguineidad*) se debe de aumentar la cuantía de la misma. En el presente caso el genoma tiene longitud variable, con lo cual, atendiendo a esta regla, debería de variar la población *durante* el entrenamiento. Esto introduciría una complicación adicional en el algoritmo G-LVQ, sobre todo atendiendo a la implementación en arquitecturas en las cuales el número de procesador es fijo; pero en cualquier caso la población deberá aumentar si aumenta la longitud media del genoma.

Normalmente durante la experimentación con el algoritmo descrita en esta memoria, y como también resulta común en teoría de algoritmos genéticos, el aumento en la población en diversas pruebas con un algoritmo no se hace en incrementos pequeños, sino en órdenes de magnitud; por eso, el incremento de la longitud del lado de la retícula de la población en las experiencias realizadas para la prueba del algoritmo G-LVQ se lleva a cabo de 5 en 5 unidades, lo cual hace que la población aumente de forma cuadrática. En realidad, esto puede provocar que la población no sea exactamente la adecuada para un determinado problema, pero puede ser, en cualquier caso, mayor que la necesaria, lo cual puede provocar problemas de duración del entrenamiento, pero va a garantizar buenos resultados. Evidentemente, si se está trabajando con un número determinado de procesadores en una arquitectura de ordenador paralela, habrá que tener en cuenta el número de procesadores disponible a la hora de diseñar la población.

Para clasificar 3 y 5 clases se ha observado que para obtener buenos resultados es suficiente hacer evolucionar una población de 100 individuos, dispuestos en una retícula de 10 x 10; sin embargo, para obtener diccionarios que clasifiquen de forma óptima 7 y 10 clases se necesita una población de 225 diccionarios, dispuesta en una retícula 15x15. Para problemas de mayor longitud, habrá que aumentar de la misma forma la población; pudiendo llegarse a poblaciones de 1600 diccionarios en el caso de diccionarios de longitud 20.


**Figura 8.6** Genoma de uno de los diccionarios ganadores, problema de 7 clases. A la la izquierda los componentes iniciales, sombreada la etiqueta de clase.

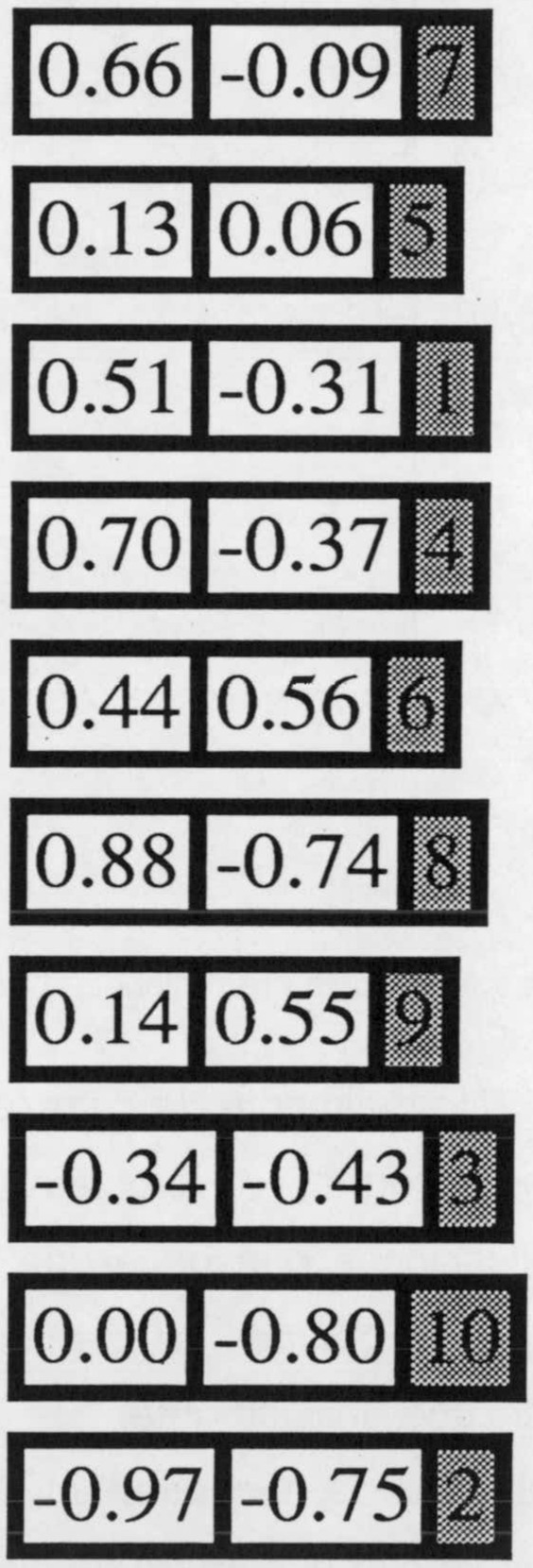



**Figura 8.7** Uno de los genomas ganadores para el problema de clasificación en 10 clases. La forma de obtenerlo se explica en en capítulo 6.

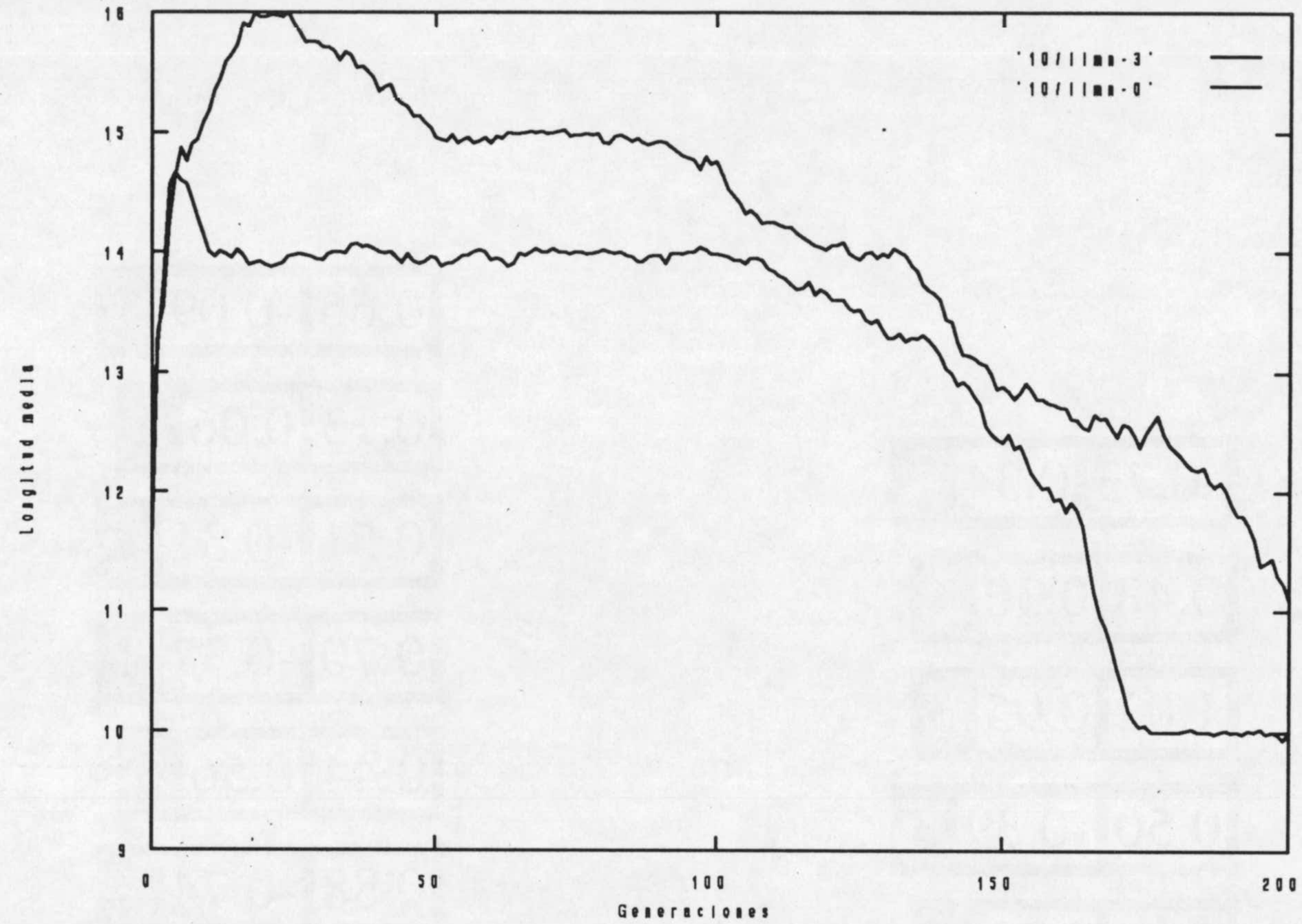



**Figura 8.8** Evolución de la longitud media del diccionario en dos entrenamientos. Dec=0.03; nº clusters = 10.

Normalmente, la complejidad del problema no crece de forma lineal con su tamaño. De esta forma, el problema de *clasificación en 10 clases* presenta una serie de complicaciones que hace que los parámetros implícitos no sean suficientes para obtener diccionarios óptimos, al menos durante el tiempo dedicado al entrenamiento. No hay que olvidar que inicialmente, los diccionarios son totalmente aleatorios, de forma que la probabilidad de encontrar un diccionario que contenga al menos un vector de cada clase disminuye con el número de clases.

El algoritmo G-LVQ, en estos casos, se comporta de la forma siguiente. Inicialmente la población se compone de diccionarios pequeños, que poco a poco van aumentando de longitud. El aumento de longitud se produce según el algoritmo genético y los operadores genéticos de longitud variable van encontrando diccionarios con mayor exactitud, a base de añadir vectores que responden a clases que no existían con anterioridad en el diccionario.

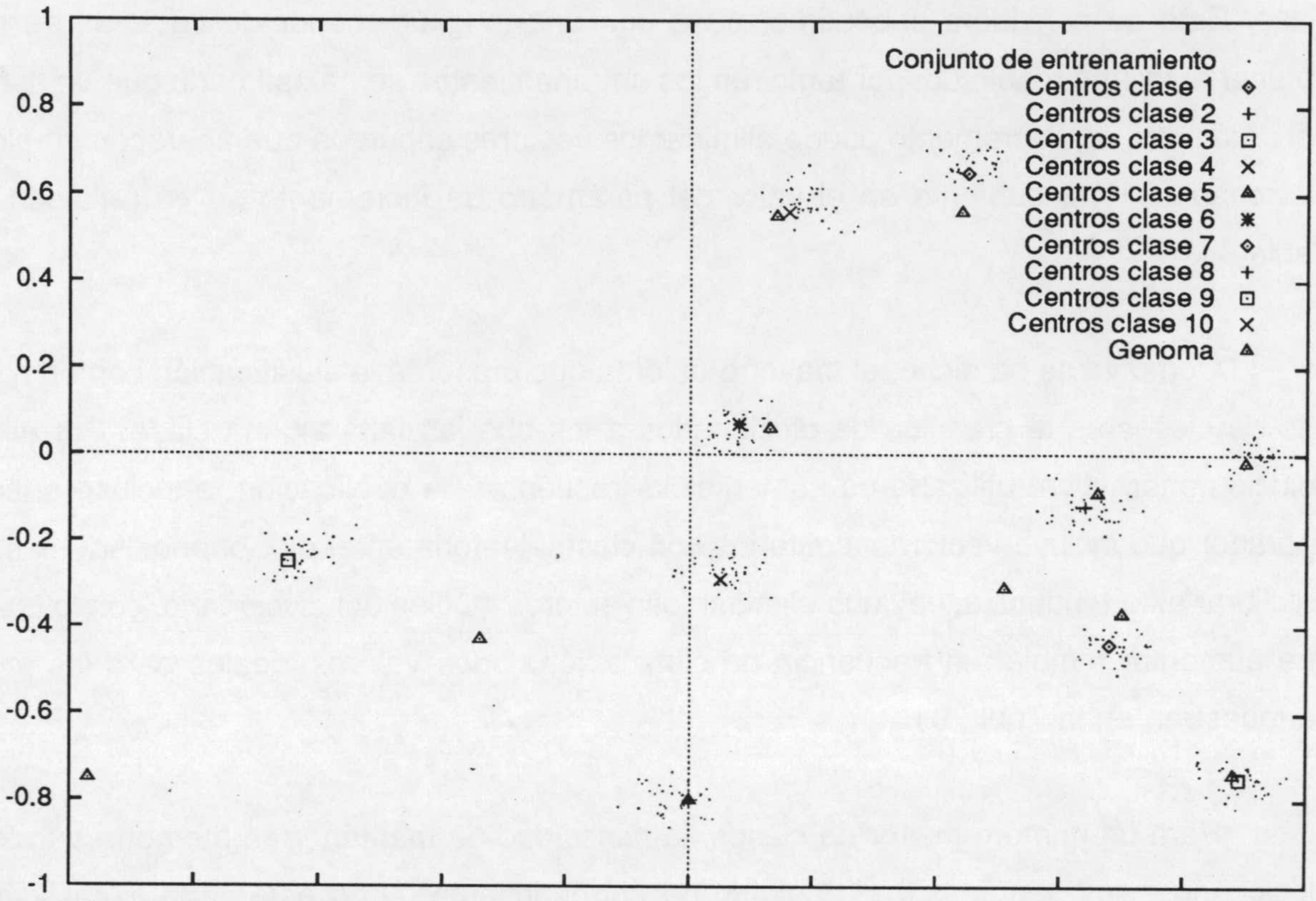



**Figura 8.9** Conjunto de entrenamiento (indicado por puntos), diccionario ganador (indicado con diversos símbolos) y genoma del mismo hallados para el problema de clasificación en 10 clases.

La longitud media, como se puede observar en la Figura 8.8, donde se representa la evolución de la longitud media durante dos entrenamientos diferentes, aumenta hasta llegar a alcanzar algunas veces una longitud de 15 o 16.

En ese momento, cuando ya se ha alcanzado un número de aciertos ideal del 100%, se comienza a optimizar la longitud del diccionario, que es el siguiente parámetro que se evalúa como componente del *fitness*. De esta forma, la longitud media de la población va disminuyendo, como se contempla en la Figura 8.8, quedándose estancada durante cierto tiempo en algún valor, para disminuir posteriormente.

En algunos casos, puede ocurrir como en uno de los entrenamientos cuya evolución de la longitud media se representa en la Figura 8.8, que al llegar el final del entrenamiento, la longitud media sea diferente al óptimo. En principio, eso no indica que la longitud del diccionario mejor sea también la misma; sin embargo, a veces ocurre también que el mejor diccionario tiene una longitud diferente del óptimo, incluyendo algún vector que no responde a ninguna zona del espacio de entrada, o incluyendo dos vectores que responden a la misma

clase. Esto se considera subóptimo, pues con una longitud menor de vectores se puede obtener el mismo resultado; por tanto, en los entrenamientos se ha estimado que un aumento del parámetro de decremento puede eliminar los vectores espúreos que aparecen en algunos diccionarios. Este aumento en el valor del parámetro de incremento se ve reflejado en la Tabla 8.i.

Como ya se ha dicho, el mayor problema que presenta la clasificación con un número alto de clases es la creación de diccionarios a los que les falta alguna clase. Por ello una solución alternativa, utilizada es aumentar la frecuencia de duplicación, e incluso añadir un operador que incluye vectores aleatorios con clase aleatoria en el diccionario. Además, para equilibrar esta tendencia, hay que eliminar los vectores inútiles del diccionario, con lo cual hay que aumentar también la frecuencia de eliminación. Unos valores ideales para las mismas se muestran en la Tabla 8.i.

Para un número mayor de clases, la necesidad de mantener en memoria a todos los diccionarios, sus genomas y el conjunto de entrenamiento (si no éste sería excesivamente lento) hace que no se pueda realizar en un ordenador PC con MS-DOS. Por ello, se han portado y compilado los programas, prácticamente sin modificación, a una Silicon Graphics Indigo R4000, utilizando el sistema operativo compatible UNIX system V Irix 4.0. El compilador ha sido el de la *Free Software Foundation*, G++. La facilidad de la recompilación de los programas pone de relieve el gran ahorro de tiempo y esfuerzo conseguido con la utilización de herramientas multiplataforma, como el lenguaje de programación C++ (capítulo 6), cuya estandarización a través de diversas plataformas es mucho mayor que la de otros lenguajes de programación.

## 8.3 Problemas de clasificación no linealmente separables

Estos casos son algo más complicados, ya que las clases no están separadas claramente por una línea recta. Cada célula de la teselación lograda por el diccionario incluirá forzosamente vectores de ambas clases.

Los tres problemas utilizados son el problema no linealmente separable de Hart

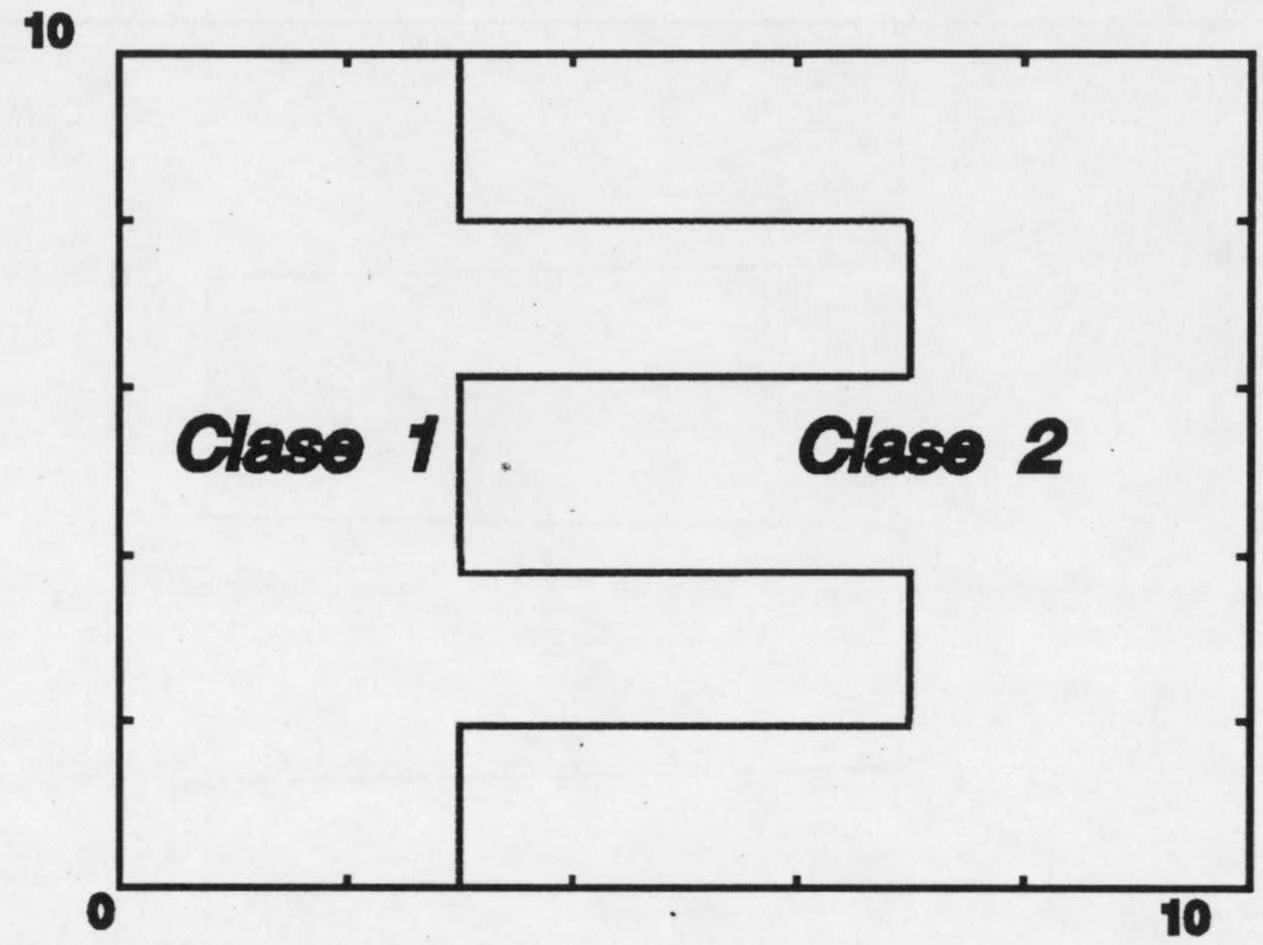



**Figura 8.10** Problema no linealmente separable de Hart bidimensional, con dos clases separadas por una línea quebrada.

[Per93] (Figura 8.10), el problema de los círculos concéntricos, adaptado a partir del problema de los círculos solapados aparecido habitualmente en la literatura, y el problema XOR, encontrado en muchas referencias de la literatura de redes neuronales.

Estos problemas son también difíciles en el sentido de que no se conoce de antemano la longitud ideal del diccionario. Evidentemente, un número mayor de niveles proporcionará una distorsión menor; pero lo que interesa en esta memoria es maximizar la probabilidad de acierto sin obtener diccionarios excesivamente largos. Con un número elevado de vectores código, puede que se obtenga una probabilidad de acierto de un 100%, pero al precio de un diccionario excesivamente grande. Tampoco se conoce el número de niveles (es decir, vectores código, como se definen en el capítulo 1) por clase, ni siquiera si cada clase tendrá el mismo número de niveles o vectores de referencia.

En el problema no linealmente separable de Hart (Figura 8.10), hay dos clases, cuyos elementos se insertan unos dentro del dominio perteneciente a la otra clase. El algoritmo G-LVQ ha hallado, entre otras, la solución indicada en la Figura 8.11, con un porcentaje de acierto cercano al 100%. Se parte en la población inicial de diccionarios con 4-6 vectores, y la longitud media va aumentando lentamente, hasta conseguir la longitud ideal, que es de 11.

En este caso, hay que cambiar el orden de evaluación del fitness implícito,

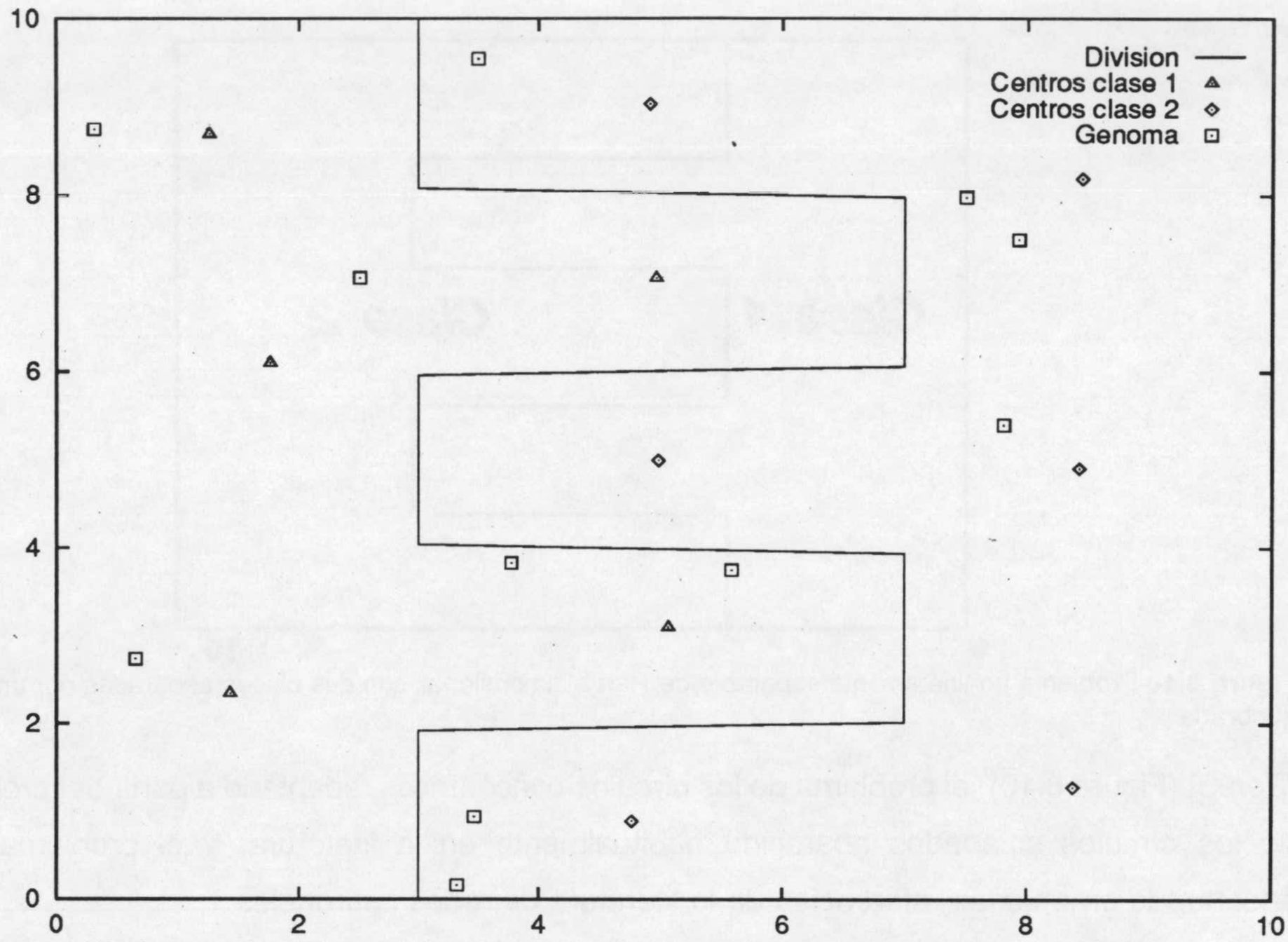



**Figura 8.11** Diccionario obtenido para el problema de Hart y genoma correspondiente.

considerando después del número de aciertos la distorsión, y por último la longitud. En casos extremos (problemas más difíciles que este) se puede poner la distorsión en primer lugar. Los resultados obtenidos se muestran en la Figura 8.11. El porcentaje de aciertos hallado es del 98% para el conjunto de entrenamiento. Los resultados obtenidos se comparan, en cuanto a error en el acierto y longitudes del diccionario, con otros métodos, en la Tabla 8.ii. El error etiquetado como *de entrenamiento* es el error obtenido por el mejor diccionario después del entrenamiento en la última generación, con el conjunto de entrenamiento. Para evaluar la capacidad de generalización, se evaluó el conjunto de los mejores vectores con otro conjunto de test generado también aleatoriamente. Debido a la naturaleza del problema, es de esperar que no varíe demasiado el número de aciertos para cualquier conjunto de test (como se puede observar también en la tabla). Por otro lado, el porcentaje de aciertos se puede aumentar mucho, hasta incluso un 100%, pero a costa de aumentar demasiado el tamaño del dicionario.

Los resultados obtenidos con el algoritmo G-LVQ se han comparado con los resultados disponibles para otro algoritmo constructivo, el descrito en [Per93], que no hace uso de

**Tabla 8.ii** Comparación de resultados obtenidos por el algoritmo G-LVQ con otro algoritmo constructivo y con LVQ1 original.

| | Número de vectores diccionario | Error test | Error entrenamiento |
|---|---|---|---|
| **G-LVQ** | 6 | 18,1% | 15,7% |
| **G-LVQ** | 10 | 2,5% | 2,1% |
| **G-LVQ** | 11 | 3,5% | 2% |
| **LVQ constructivo**[2] | 5 | 17,4% | 18% |
| **LVQ constructivo**[2] | 12 | 3,43% | 3,4% |
| **LVQ1**[1] | 4 | 17,1% | 17,1% |
| **LVQ1**[1] | 16 | 4,4% | 3,9% |
| **LVQ1**[1] | 32 | 3,3% | 3,0% |

[1] Dado que no se trata de un algoritmo constructivo, se indican los errores alcanzados para varias longitudes de diccionario. Los datos están tomados de [Per93].

[2] Tomado de [Per93]. El número de vectores indicado resulta de multiplicar el número de vectores por clase, por el número de clases.

Los datos tomados de [Per93] son una media sobre varias ejecuciones. Los datos referidos a G-LVQ son entrenamientos diferentes con parámetros iniciales diferentes.

algoritmos genéticos. Como se puede observar en la Tabla 8.ii, se obtienen diccionarios con un error menor de clasificación, y con un número inferior de niveles.

En el caso de los círculos concéntricos, el principal problema al que se enfrentan los métodos clásicos es el de la inicialización. Si se toman dos vectores diccionario, como el centroide de las clases respectivas, resultará un error del 50%, pues casi todos los elementos pertenecientes a la clase exterior se clasificarán como pertenecientes a la interior y viceversa. La longitud ideal hallada por el método G-LVQ es de 11, como se muestra en la Figura 8.12, utilizando los parámetros indicados en la Tabla 8.iv.

Otro problema relativamente sencillo dentro de esta clase es el problema XOR, equivalente a que la red neuronal sea capaz de implementar tal función lógica, cuya tabla de verdad aparece en la Tabla 8.iii. En realidad, en este caso hemos utilizado como entradas los valores -1 y +1; y como salida las clases 1 y 2, lo cual es simplemente un cambio de notación. Alrededor de estos puntos, se ha creado una nube gaussiana con una amplitud de

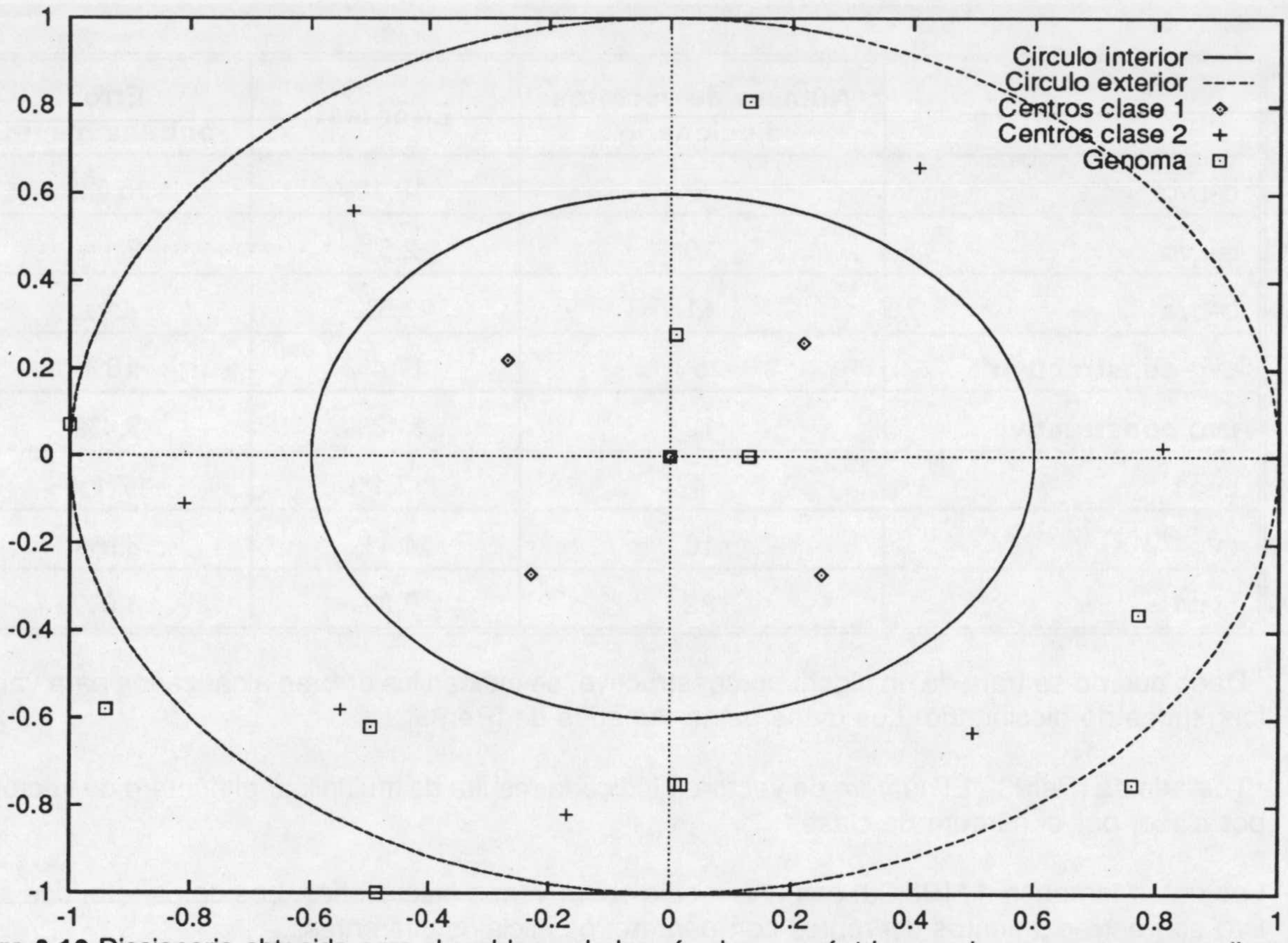



**Figura 8.12** Diccionario obtenido para el problema de los círculos concéntricos y el genoma correspondiente.

0.1

**Tabla 8.iii** Tabla de verdad de la función lógica XOR.

| x | y | x XOR y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Este problema es irresoluble en el caso del perceptrón monocapa; Minsky en su libro *Perceptrons* demostró, en los años sesenta, que tal red neuronal era incapaz de resolver este problema.

No es este el caso del algoritmo G-LVQ, al que, por sus características, no le resulta más difícil que el problema de clasificar en dos clases diferentes. El porcentaje de acierto es del 100%, y el tiempo que tarda en hallar la solución es intermedio entre el indicado en la Tabla 8.i para 3 clases y el indicado para 5 clases, es decir, unos 15 minutos de entrenamiento, alcanzando el máximo de aciertos en la segunda o tercera generación de 50.

**Tabla 8.Iv** Parámetros utilizados y resultados obtenidos, para el problema de Hart y el de los círculos concéntricos.

| | Incremento | Decremento | Generaciones | Tiempo total |
|---|---|---|---|---|
| **Hart** | 0.1 | 0.01 | 300 | (300) 425 min. |
| **Circulos** | 0.01 | 0.01 | 300 | (300) 417 min. |

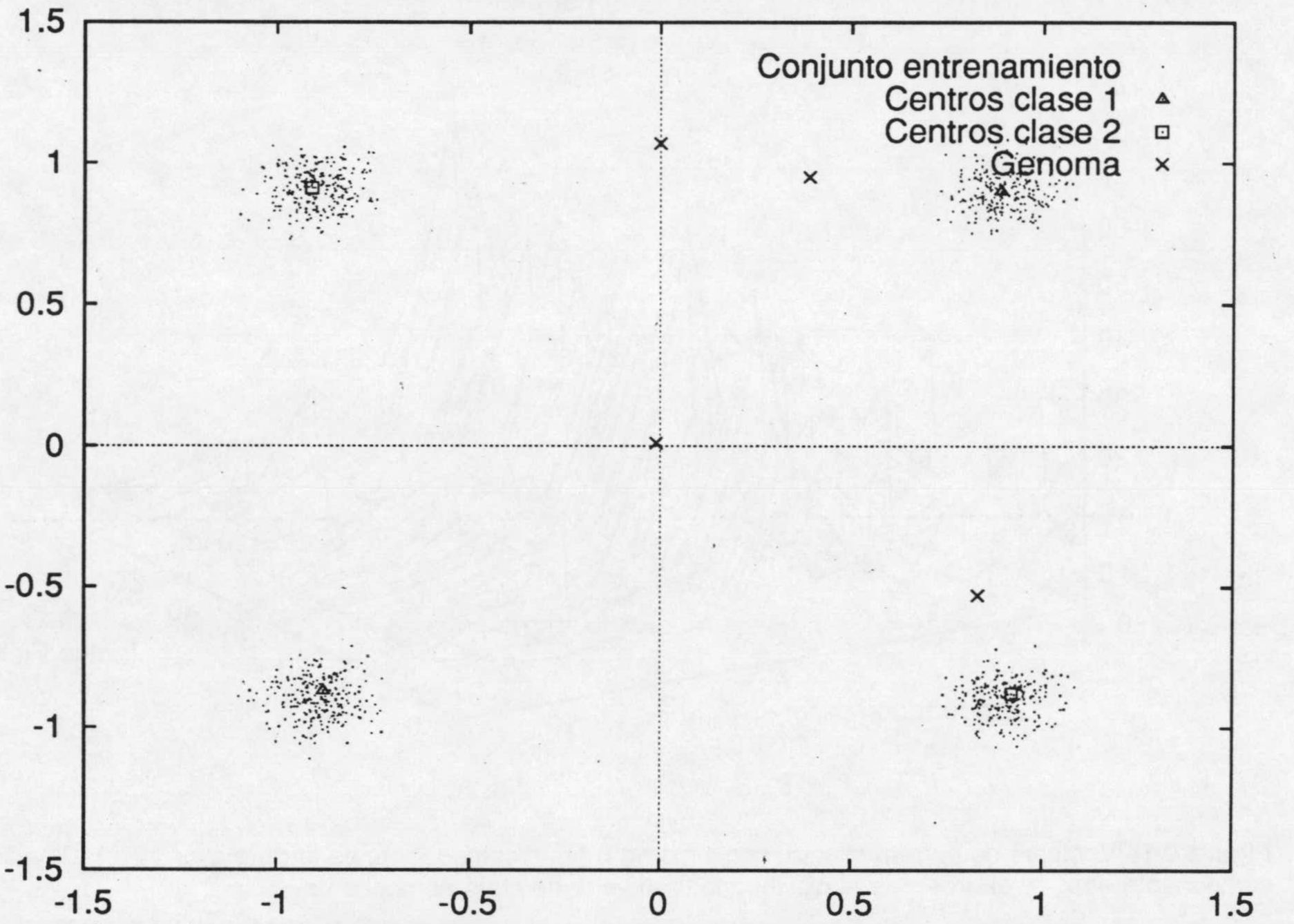



**Figura 8.13** Una de las soluciones alcanzadas por G-LVQ para el problema XOR, incluyendo el conjunto de entrenamiento.

## 8.4 Problemas de clasificación reales

Se han tomado datos de dos fuentes diferentes: unos datos procedentes de análisis de las imágenes de microscopía electrónica de macromoléculas (que se analizará en el apartado 8.4.1) otros procedentes de imágenes obtenidas por sonar [Gor88] (apartado 8.4.2).

### 8.4.1 Análisis de imágenes de macromoléculas por microscopía electrónica

En las células que forman parte de los seres vivos y de los animales unicelulares, hay

una serie de macromoléculas que sufren cambios estructurales al ser sometidos a algún factor físico o químico externo, como un cambio de temperatura. Entre estas macromoléculas están las *chaperoninas*, cuyos factores tipo GroEL tienen la función de promover el ensamblaje eficiente y correcto de oligómeros (agregaciones pequeñas de moléculas) biológicos.

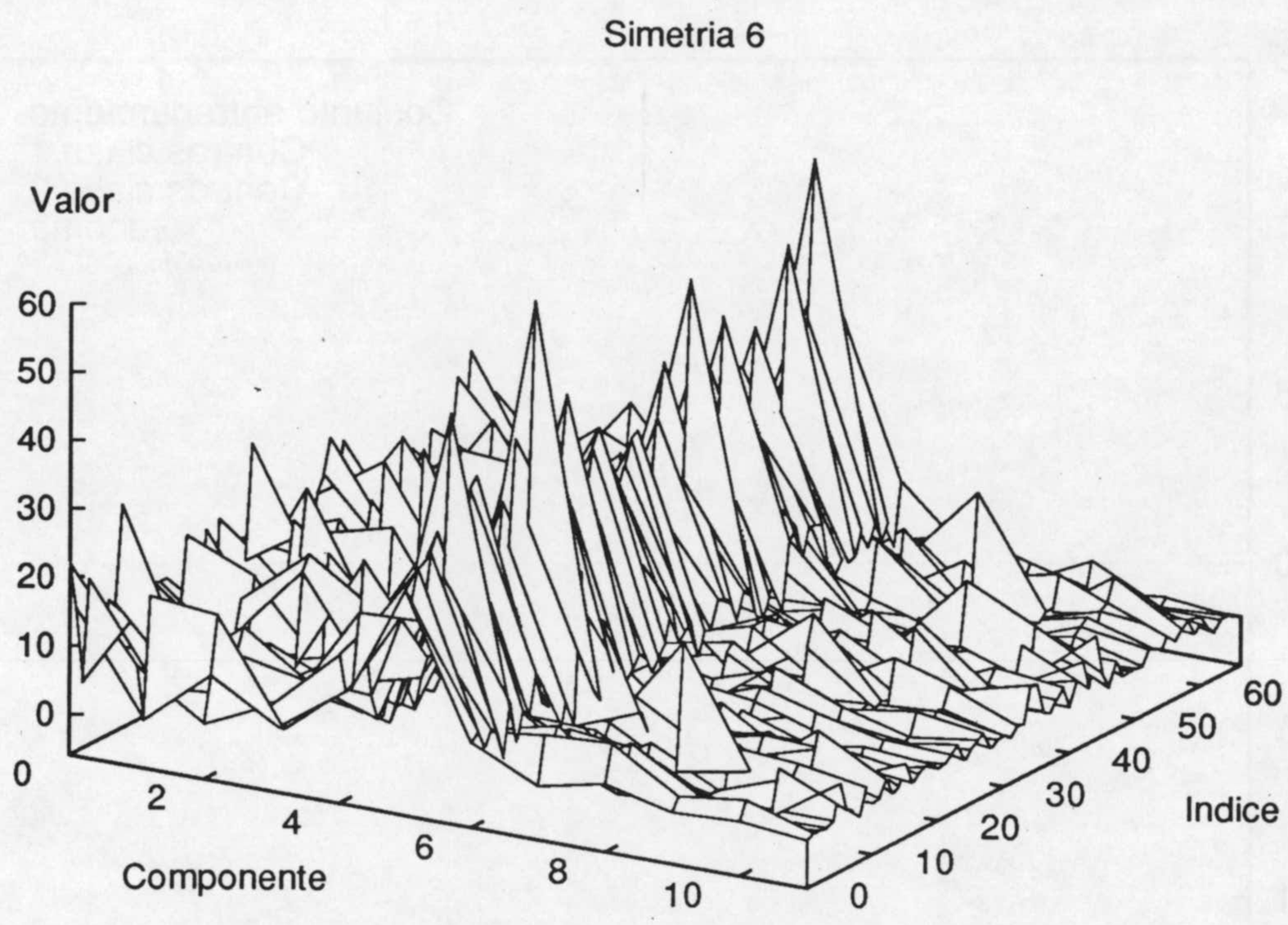



**Figura 8.14** Vectores de entrenamiento, con simetría 6 (esta componente es la de mayor valor). El índice de la componente es x, y el orden en el diccionario; z el valor de cada componente.

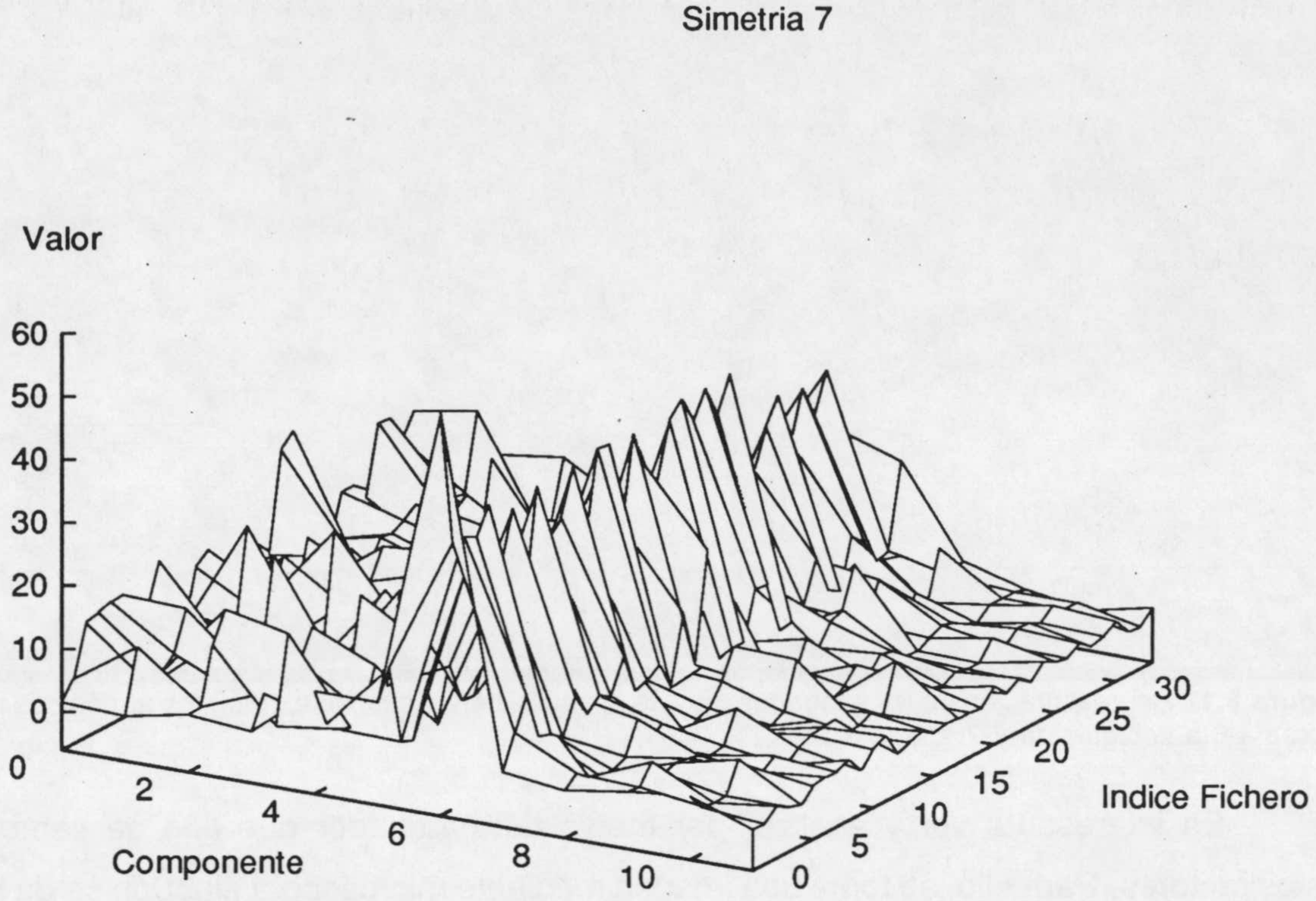



**Figura 8.15** Vectores de entrenamiento con simetría 7.

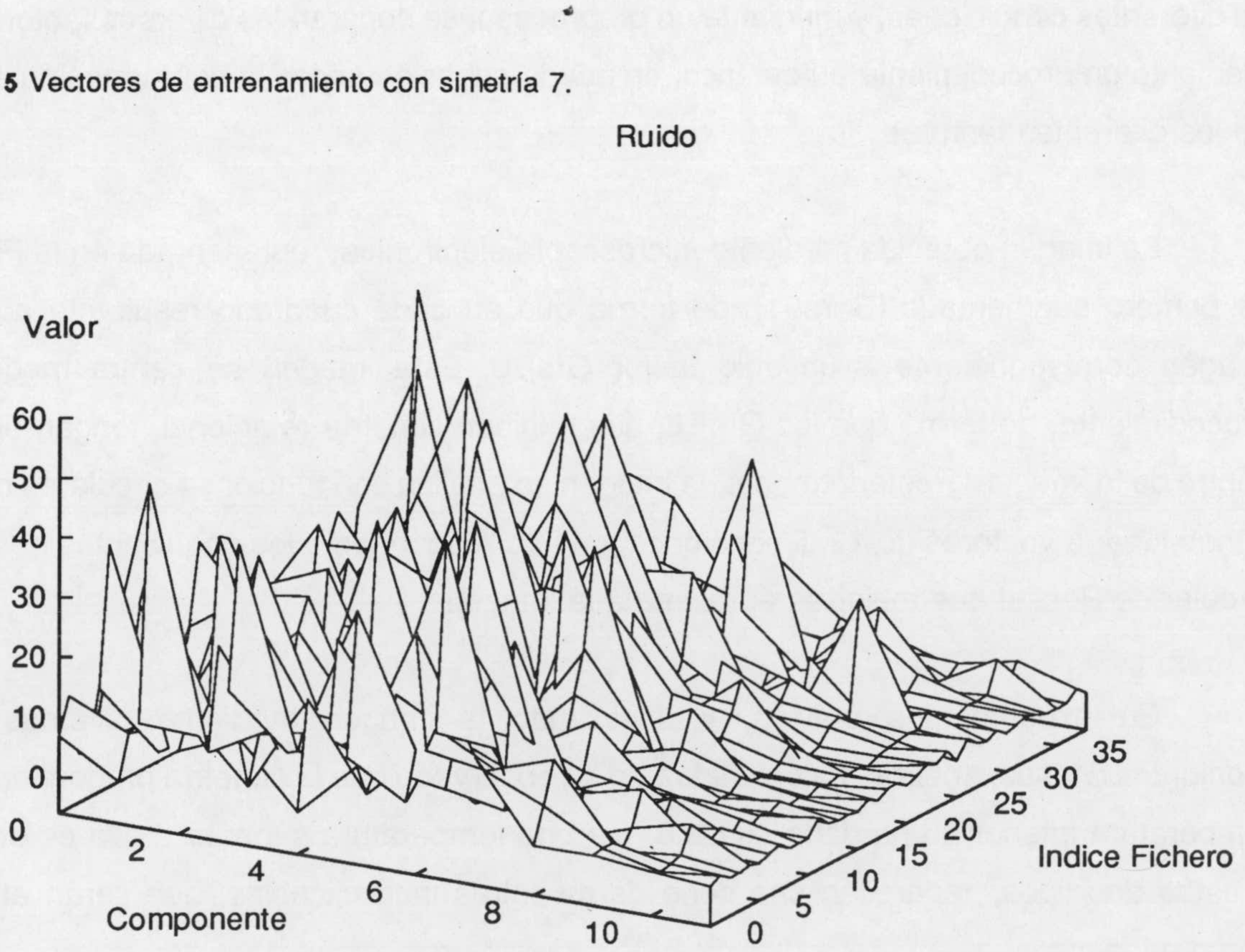



**Figura 8.16** Vectores de entrenamiento etiquetados como *ruido*.

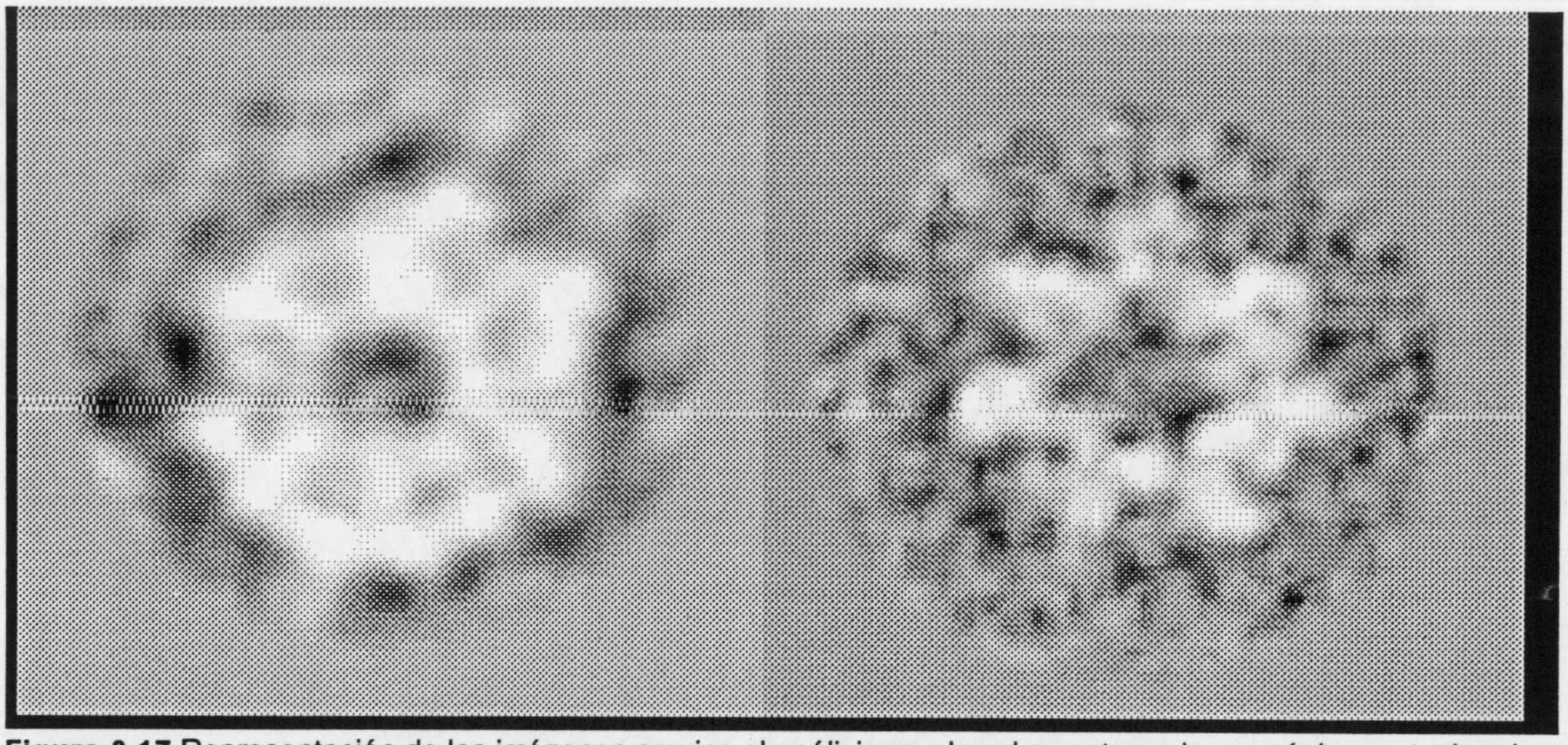


**Figura 8.17** Representación de las imágenes previas al análisis en el cual se extraen los armónicos rotacionales; simetría 6 a la izquierda y 7 a la derecha.

Es interesante ver y analizar por medio del ordenador qué tipo de cambios sufren estos factores. Para ello, se toma una imagen mediante microscopía electrónica de las células en diferentes condiciones, y mediante un preproceso, se separan los diversos factores GroEL. Mediante un procedimiento automático, se puede saber entonces la conformación estructural de los diferentes factores.

La imagen obtenida mediante microscopía electrónica, representada en la Figura 8.17 es primero segmentada [Cara91], de forma que en cada cuadrado resultante aparezca la imagen correspondiente a un solo factor GroEL. Esta imagen se centra mediante otro procedimiento, de forma que los GroEL, que exhiben simetría rotacional, tengan el eje en el centro de la imagen. Posteriormente, la imagen se analiza con funciones circulares de Bessel, dando lugar a vectores de 12 dimensiones, que corresponden a los coeficientes de la función circular de Bessel que mejor se aproxima a la imagen.

En experimentos previos, y analizando la imagen mediante diversas técnicas (conjuntos difusos, análisis multivariado), se ha observado que la simetría predominante a una temperatura inferior a una dada es de 6, y a una temperatura superior a ella es de 7. Junto a estos dos tipos, aparecen una serie de muestras inclasificables, que serán etiquetadas como ruido.

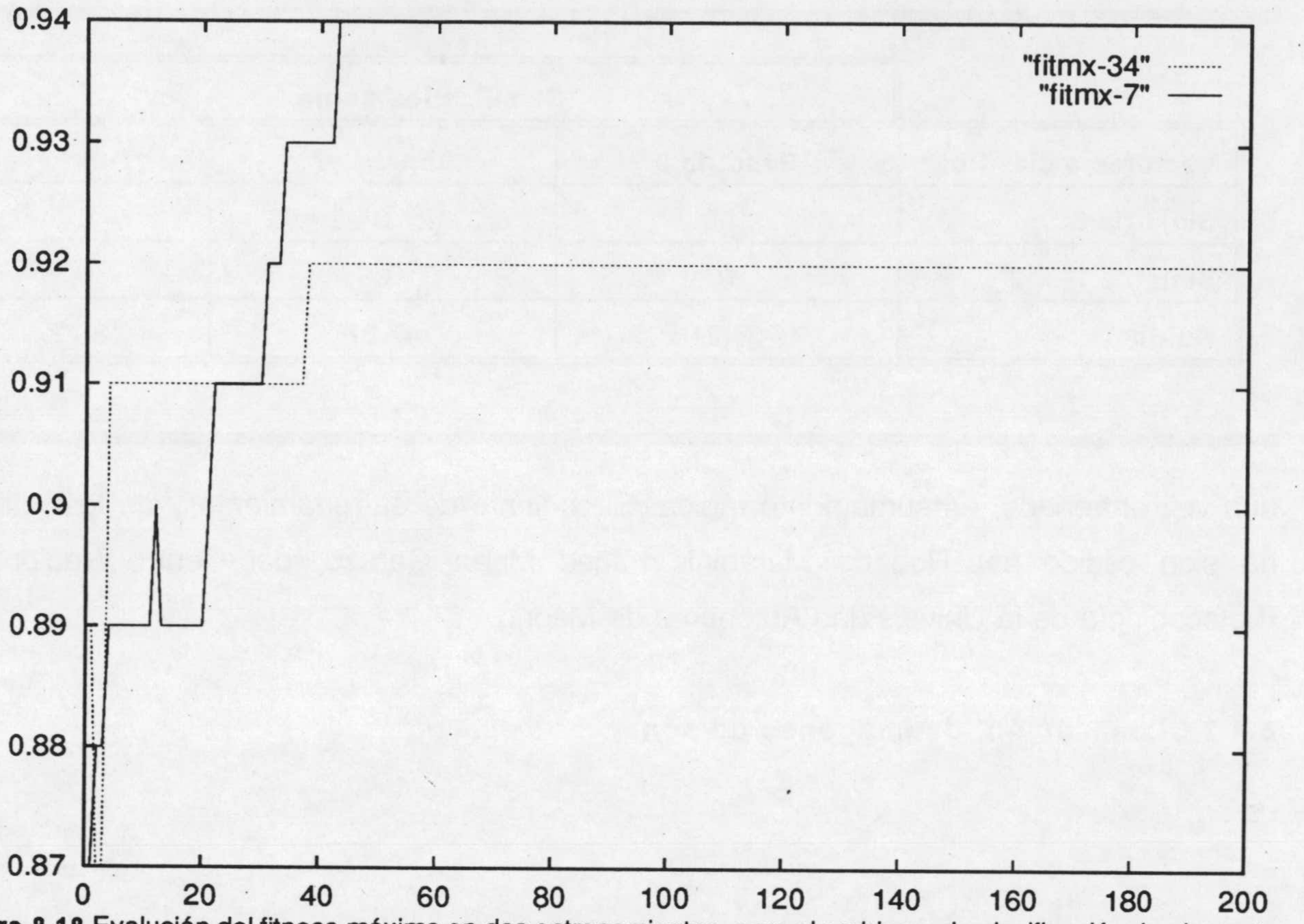



**Figura 8.18** Evolución del fitness máximo en dos entrenamientos, para el problema de clasificación de elementos estructurales de células.

El conjunto de entrenamiento está formado por 103 vectores, con un número desigual de muestras de los tres tipos (65 de simetría 6, 34 de simetría 7, y 38 muestras no clasificadas). Estos vectores se muestran en la Figura 8.14, Figura 8.15 y Figura 8.16, respectivamente.

El algoritmo G-LVQ alcanza una precisión mayor del 90% en la clasificación de los mismos. En realidad, si se descuentan las imágenes etiquetadas como *ruido*, el porcentaje de aciertos es del 100%; el problema radica en que las muestras *ruidosas* son mal calificadas por el algoritmo como pertenecientes a una de las otras dos clases. Estas muestras calificadas como ruidosas, según se indica en [Cara91], son muestras que, según otros métodos, no habían podido ser clasificadas. Por tanto, si se confía en este método, se disminuiría en bastante la cantidad de muestras sin clasificar, al quedar clasificadas como *ruido* sólo un 20% de las que lo eran anteriormente.

En la figura se muestra la evolución del número de aciertos durante el entrenamiento. El diccionario obtenido tiene entre 3 y 5 vectores, por lo cual la clasificación,

**Tabla 8.v** Resultados de clasificación con el algoritmo G-LVQde elementos estructurales en células.

| | Clasificados como | | |
|---|---|---|---|
| **Vectores a clasificar** | **Simetría 6** | **Simetría 7** | **Ruido** |
| **Simetría 6** | 100 | 0 | 0 |
| **Simetría 7** | 0 | 100 | 0 |
| **Ruido** | 34.21 | 47.37 | 18.42 |

una vez entrenado, es sumamente rápida. El conjunto de entrenamiento y de test utilizado ha sido cedido por Roberto Marabini y Jose María Carazo, del Centro Nacional de Biotecnología de la Universidad Autónoma de Madrid.

### 8.4.2 Clasificación de imágenes de sonar.

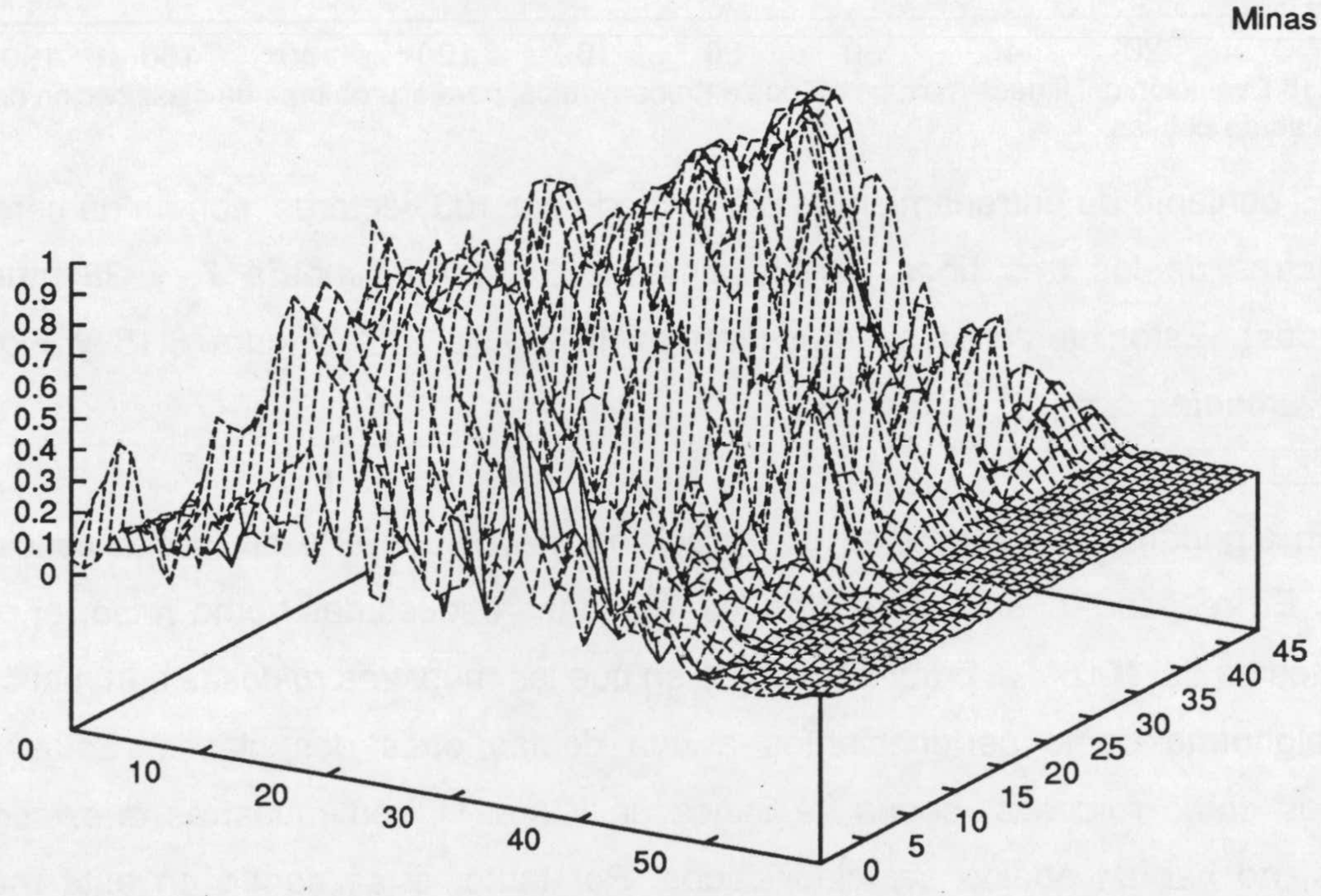



**Figura 8.19** Representación tridimensional de los vectores, para las minas; z es el valor de cada componente, y es el índice sobre el fichero de entrenamiento, x recorre las componentes.

Este problema se encuadra dentro del problema más general de indentificación de

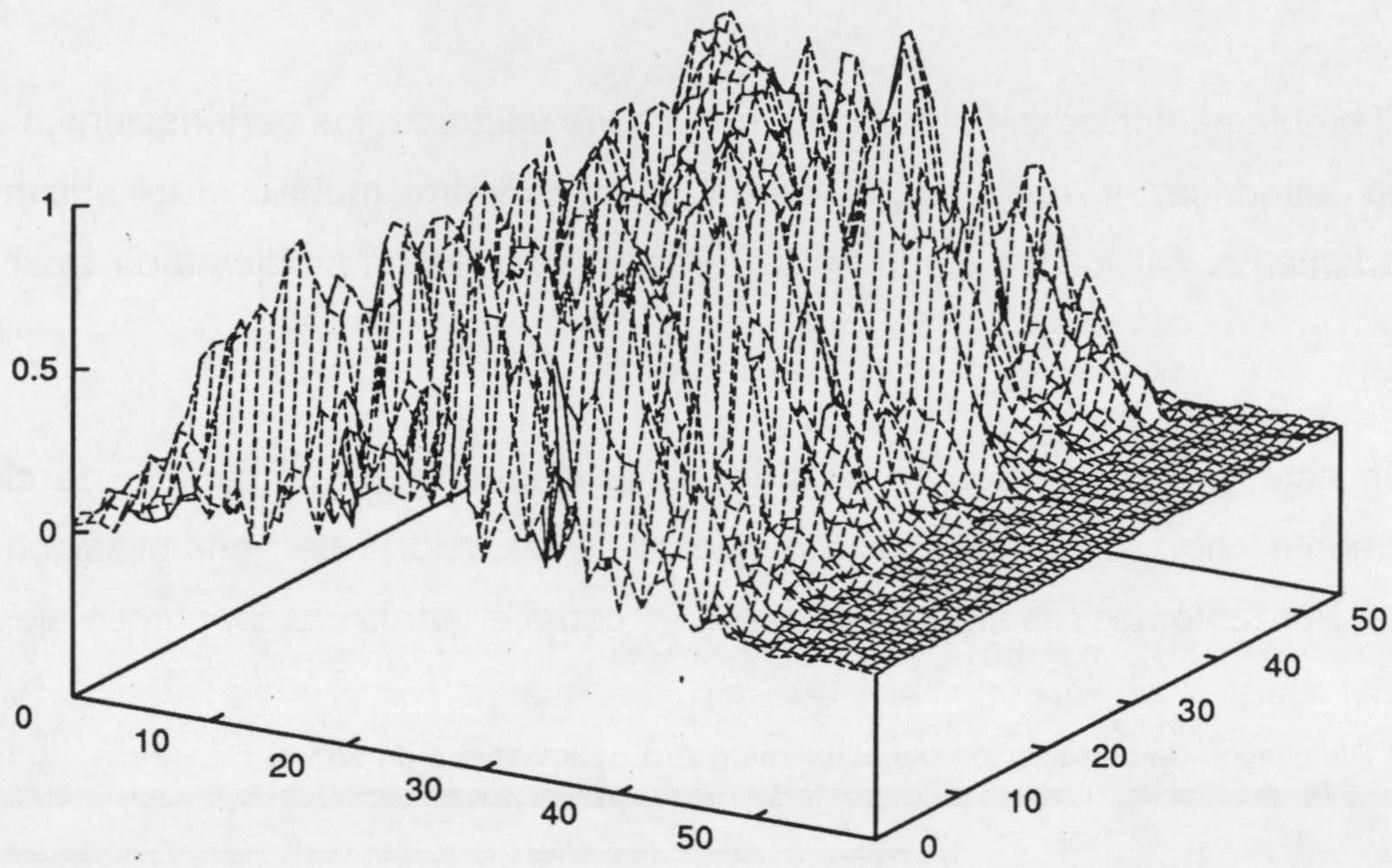



**Figura 8.20** Representación tridimensional de los vectores, para las **rocas**; la componente z es el valor de cada componente.

objetivos submarinos. Los ecos recibidos por un sistema de sonar activo deben de ser clasificados después de reflejarse en un objetivo adecuado. Los datos utilizados proceden de los ecos devueltos por un cilindro de metal y una roca de forma cilíndrica posicionada longitudinalmente en el suelo marítimo. El pulso enviado era una señal de banda ancha lineal con frecuencia modulada; los ecos se obtienen de cada objetivo a diversos ángulos con respecto al receptor [Gor88].

Hay un total de 208 ecos (111 de cilindro metálico, y 97 de la roca), seleccionados en función de su relación señal ruido. De ellos, 104 son apartados para entrenamiento y el resto para test, equilibrando la cantidad de rocas y minas presentes en la muestra. La señal reflejada por el objetivo es sometida a una transformada de Fourier; posteriormente se calcula su envolvente espectral. De cada envolvente espectral se obtienen 60 muestras, que son normalizadas a valores comprendidos entre 0.0 y 0.1. Estos vectores utilizados para entrenamiento se muestran en la Figura 8.19 (minas) y en la Figura 8.20 (rocas).

Estos datos han sido puestos a disposición de la comunidad de investigación

internacional en un nodo de la InterNet, siendo accesibles mediante FTP (programa para transferencia de ficheros desde sitios remotos) anónimo. Forma parte del *benchmark*, o campo de pruebas, de redes neuronales de la Universidad Carnegie Mellon.

El problema consiste en distinguir, entre esos vectores, los pertenecientes a una roca de forma cilíndrica, y los pertenecientes a un cilindro metálico del mismo tamaño aproximadamente, calificados como "mina". Por lo tanto, es un problema de clasificación en 2 clases.

En este tipo de problemas no siempre es deseable un porcentaje de clasificación excesivamente alto, pues esto puede perjudicar la capacidad de generalización de la red resultante. Por tanto, los resultados no alcanzan, como en otros casos, porcentajes cercanos al 100%.

**Tabla 8.vi** Resultados de clasificación del algoritmo G-LVQ de imágenes de sonar.

| | **Muestra a clasificar** | |
|---|---|---|
| Clasificada como | **Mina** | **Roca** |
| **Mina** | 72.3% | 10.1% |
| **Roca** | 27.7% | 89.9% |

En este caso ha sido necesario repetir el entrenamiento con diversos parámetros, hasta obtener resultados óptimos. Los mejores resultados obtenidos, una vez hallados los parámetros correctos, se muestran en la Tabla 8.vi. Como se puede observar, las minas se pueden clasificar mucho peor que las rocas, el porcentaje de falsos positivos en el caso de las minas es relativamente alto.

Los resultados globales son comparables con los obtenidos por Gorman et al. [Gor88], según se muestra en la Tabla 8.vii. Los resultados obtenidos por G-LVQ, que no varían para unas mismas condiciones iniciales, una vez establecidos

**Tabla 8.vii** Comparación de resultados entre el algoritmo G-LVQ y el perceptrón multicapa de Gorman.

| G-LVQ | Gorman |
|---|---|
| 80,5±1,3 | 85,7±6,3 |

los parámetros de entrenamiento, caen dentro de´una desviación estándar de los resultados obtenidos por Gorman, en el caso de un perceptrón multicapa con 2 neuronas en la capa oculta; mejoran los obtenidos en el caso de perceptrones sin capa oculta, y es peor que los obtenidos para perceptrones con más neuronas en la capa oculta.

Estos resultados se pueden mejorar, siempre que se aumente el parámetro de incremento, y se considere la distorsión como principal criterio de clasificación. En tal caso, se crean diccionarios excesivamente grandes, cuya habilidad de generalización es muy inferior. Sin embargo, hay que tener en cuenta que la forma de calcular el error en el caso de Gorman y en la memoria actual es ligeramente diferente; en el caso de Gorman se utiliza la combinación de pesos, hallada durante el entrenamiento, que da un error menor; en el caso del algoritmo G-LVQ los resultados se han obtenido automáticamente. Es también destacable que la cantidad de pesos hallados en el caso del algoritmo G-LVQ es sensiblemente inferior.

## 8.5 Conclusiones

Después de aplicar el algoritmo G-LVQ introducido en esta memoria, a diversos tipos de problemas, incluyendo problemas sencillos, problemas complejos (linealmente no separables) y problemas reales, se encuentra que el algoritmo mejora a algunos algoritmos constructivos tradicionales, tanto en la longitud de los diccionarios obtenidos, como en la exactitud de clasificación de los diccionarios resultantes. Por tanto, G-LVQ, obtiene diccionarios más compactos (y por consiguiente, con una mayor capacidad de generalización) y más precisos que métodos tradicionales, como el método constructivo de Pérez [Per93], al menos en la gama de problemas en los que han sido comparados.

En cuanto a las prestaciones obtenibles al aplicarlo a problemas reales, los resultados conseguidos por el algoritmo G-LVQ son comparables a otros algoritmos clásicos. En el caso de las macromoléculas, se pueden clasificar más imágenes que las que se podrían clasificar con los métodos previos anteriormente: análisis factorial y clasificación fuzzy supervisada. En el caso del radar, se obtienen resultados comparables a los de un perceptrón multicapa, con la diferencia de la mayor compacidad del diccionario resultante del entrenamiento G-LVQ, y su

menor complejidad.

# Conclusiones y principales aportaciones

Las principales aportaciones de este trabajo al campo del reconocimiento de patrones en general, y de las redes neuronales y algoritmos genéticos en particular, y las conclusiones obtenidas, son las siguientes:

1. Se han examinado con cierto detalle las técnicas de cuantización vectorial de mayor uso en la actualidad, evaluando sus aplicaciones y limitaciones.

2. Una de las numerosas aplicaciones de la cuantización vectorial es la clasificación de proteínas atendiendo a su estructura secundaria. Con una optimización básica de un algoritmo de cuantización vectorial, el mapa autoorganizativo de Kohonen, se han obtenido resultados satisfactorios en el cálculo de los porcentajes de estructura secundaria de proteinas distintas a las utilizadas en el entrenamiento, mejorando métodos clásicos.

3. El algoritmo aportado por esta memoria, denominado G-LVQ, se ha diseñado combinando dos de las técnicas más poderosas y generales existentes en la actualidad para clasificación y optimización: los algoritmos genéticos y las redes neuronales. El algoritmo genético permite optimizar una red neuronal, obteniendo resultados mucho mejores que los obtenibles mediante técnicas heurísticas o de prueba sistemática.

4. El algoritmo G-LVQ utiliza genomas de longitud variable para crear poblaciones de redes neuronales o diccionarios de longitud diferente. Esto permite evaluar simultáneamente diccionarios de estructura diferente, para que el método de selección escoja el más adecuado a la tarea. El modo de utilización de estos genomas de longitud variable es original, y corresponde al espíritu de los algoritmos genéticos.

5. La simple utilización de diccionarios de longitud variable provocaría que se escogieran siempre los de longitud mayor, debido a que estos son los que aportan una menor distorsión. Sin embargo, el algoritmo genético actúa a dos niveles; primero, dentro del diccionario se seleccionan y duplican aquellos vectores que han "ganado" más veces para las muestras de entrenamiento, es decir, a aquellos vectores que a lo largo de la presentación de las muestras

de entrenamiento, han estado más cerca que todos los demás; y en segundo lugar, se eliminan aquellos vectores que no han ganado nunca o que han ganado pocas veces.

6. Debido a que hay varios objetivos de optimización, no se puede utilizar una sola fórmula de la que se obtenga el *fitness* o adecuación de cada elemento de población a la tarea de clasificación. Se utiliza por contra un *fitness* vectorial, con prioridades de evaluación de unos factores (características del diccionario) frente a otros. Se consigue así optimizar simultáneamente varios objetivos: la longitud del diccionario, la distorsión y el número de aciertos en clasificación, sin sacrificar unos frente a otros.

7. Para implementar un algoritmo genético en una arquitectura paralela es necesario restringir al máximo el tráfico entre procesadores. Los algoritmos genéticos clásicos necesitan evaluar el máximo global del *fitness*, y a partir de entonces tiene lugar un esquema de reproducción que comprende a toda la población. Una de las aportaciones del presente trabajo consiste en la utilización un método de reproducción local, con el cual se reduce el *routing* o tráfico entre procesdores, a la vez que se simplifica el algoritmo genético, sin afectar a sus prestaciones.

8. La implementación de los algoritmos se ha hecho utilizando herramientas estándar de programación, como compiladores de C++ de dominio público, el lenguaje de programación PERL, y otros programas como GNUPLOT. La divulgación de estas herramientas garantiza la usabilidad por parte de la comunidad internacional de investigación de las herramientas y resultados diseñados para este trabajo.

9. La programación orientada a objetos, utilizada para implementar el algoritmo G-LVQ, es uno de los paradigmas más utilizados actualmente, y garantiza una gran facilidad de uso por parte del programador de los programas creados por otro equipo de programación, así como una sistematicidad en la creación de grandes proyectos.

10. Los resultados obtenidos por el algoritmo G-LVQ en problemas sintéticos como el problema no linealmente separable de Hart mejoran los obtenidos por otros algoritmos evolutivos y tradicionales. Asimismo, los resultados en la clasificación de imágenes de macromoléculas permiten utilizar con fiabilidad muestras que hasta entonces no se consideraban fiables. En el caso de clasificación de imágenes de sonar, los resultados obtenidos son comparables con

los obtenidos por el perceptrón multicapa, si bien la arquitectura aquí propuesta es más simple, y la desviación típica es menor.

Consideramos que este trabajo constituye una aportación al dinámico campo de reconocimiento de patrones, que permite clasificar con exactitud lo percibido por un ordenador o máquina, función que realizan con gran eficiencia los sistemas biológicos, en los cuales está, en último término, su inspiración.

# Bibliografía


[Ack91] Ackley, D. H.; Littman, M.L.; "Interaction between evolution and learning", in *Artificial Life II*, C. G. Langton et al. (Ed.), Morgan Kauffman, 1991.

[Alp91] Alpaydim, E.; "GAL: Networks That Grow When They Learn and Shrink When They Forget", TR-91-032, ICSI, Berkeley, May, 1991.

[And93] Andrade, M.A.; Chacón, P.; Merelo, J.J.; Morán, F.; "Evaluation of secondary structure of proteins from UV circular dichroism spectra using an unsupervised learning neural network", *Protein Engineering*, vol. 6, no. 4, pp. 383-390, 1993.

[Beer92] Beer, R. D.; Gallagher, J. C.; "Evolving Dynamical Neural Networks for Adaptive Behavior", *Adaptive Behavior*, vol. 1, no. 1, pp. 91-122, 1994.

[Bel90] Belew, R. K.; "Evolution, Learning and Culture: Computational Metaphors for the Adaptive Algorithms", *Complex Systems* vol. 4, pp. 11-49, 1990.

[Ben91] Bengio, Y.; Bengio, S.; "Learning a synaptic learning rule, Technical Report 751, Départment d'Informatique et de Recherche Operationelle, Université de Montréal, Canada, November 1990.

[Bohr90] Bohr, H.; Bohr, J.; Brunak, S.; Cotterill, R.M.J.; Fredholm, H.; Lautrup, B.; Petersen, S.B.; "A novel approach to prediction of the 2-dimensional structures of protein backbones by neural networks", *FEBS letters*, vol

261, no. 1, pp 43-46, 12 Feb 1990.

[Bou93] Bouton, C.; Pagès, G.; "Self-Organization and convergence of the one-dimensional Kohonen algorithm with non uniformly distributed stimuli", *Stochastic Processes and their Applications*, vol. 47, pp. 249-274, 1993.

[Bou94] Bouton, C.; Pagés, G.; "Convergence in distribution of the one dimensional Kohonen algorithm when the stimuli are ot uniform" *Advances in Applied Probability*, vol. 26, 1, 1994.

[Bru91] Brunak, S.; Engelbrecht, J.; Knudsen, S.; "Prediction of Human mRNA Donor and Acceptor Sites from the DNA Sequence", *J. Molecular Biology*, vol. 220, pp. 49-65, 1991.

[Cha90] Chalmers, D.J.; "The evoluton of learning: an experiment in genetic connectionism", en D. S. Touretzky, J. L. Elman & G. E. Hinton, (eds), *Procs. of the 1990 Connectionist Models Summer School*, 1990, pp 81-90.

[Cli93] Cliff, D.; Harvey, I.; Husbands, P.; "Explorations in Evolutionary Robotics", Adaptive Behavior, vol. 2, no. 1 1993, 32 pps.

[Com91] Compiani, M.; Fariselli, P.; Cassadio, R. "Neural Networks Extracting General Features of Protein Secondary Structure", *Procs. 4th Italian Workshop on Parallel Architectures and Neural Networks*, 1991, pp 227-237.

[Cot87] Cottrell, M., Fort, J.-C.; "Etude d'un processus d'auto-organisation", Ann. Inst. Henri Poincaré, vol 23, nº 1, p 1-20, 1987.

[Cot94] Cottrell, M., Fort, J.-C.; Pagès, G.; "Two or three things that we know about the Kohonen algorithm", Procs. ESANN94, pp 234-244.

[Das93] Daşdan, A.; Oflazer, K.; "Genetic Synthesis of Unsupervised Learning Algorithms, Technical Report, Dept. of Computer Engineering and Information Science, Bilkent University, Ankara, Turkey.

[Dav91] Davidor, Y.; "A Naturally Occurring Niche & Species Phenomenom: The Model and First Results", Procs. ICGA 91, *4th Int. Conf on Genetic Algorithms*, 1991, pp 257-263.

[deG90] De Garis, H.; "Genetic Programming: Building aritificial neural systems uting genetically programmed neural network modules". In B. W. Porter & R. J. Mooney (Eds.) *Procs. of the 7th International Conference on Machine Learning*, 1990, pp 132-139.

[Drei93] Dreiseitl, S.; Wang, D.; "Automatic Generation of C++ "Code for Neural Network Simulation", *New Trends in Neural Computation*, Mira, Cabestany, Prieto, Eds., Springer Verlag, 1993, pp. 358-363.

[Duda73] Duda, R. O., Hart, P. E., "Pattern classification and scene analysis", New York: Wiley & Sons, 1973.

[Eck91] Eckel, B.; "Aplique C++", Madrid: McGraw-Hill/Interamericana de España, 1991.

[Elli90] Ellis, M. A.; Stroustrup, B.; "The annotated C++ reference manual", Addison Wesley, 1990.

[Erw91] Erwin, E.; Obermayer, K.; Schulten, K.; "Self-Organizing Maps: Ordering, Convergence Properties, and Metastable States", Univ. of Illinois

Theoretical Biophysics Tech. Rep. TB-91-11, 1991.

[Erw92] Erwin, E.; Obermayer, K.; Schulten, K.; "Self-Organizing Maps: Ordering, Convergence Properties, and energy functions", *Biological Cybernetics*, 67, pp. 47-55, 1992.

[Fahl90] Fahlman, S.E.; Lebiére, C., "The Cascade Correlation Learning Architecture", Tech. Rep. CMU-CS-90-100, Carnegie Mellon University, 1990.

[Far93] Fariselli, P.; Compiani, M.; Cassadio, R. "Predicting secondary structure of membrane proteins with neural networks", *Eur. Biophys. J.*, vol. 22, pp. 41-51, 1993.

[Fer91] Ferrán, E.A.; Ferrara, P.; "Topological maps of protein sequences", *Biological Cybernetics*, vol. 5, pp. 451-458, 1991.

[Fer92a] Ferrán, E.A.; Ferrara, P.; "A neural network dynamics that resembles protein evolution", *Physica A*, vol. 186, pp. 395-401, 1992.

[Fer92b] Ferrán, E.A.; Pflugfelder, B.; Ferrara, P.; "Large Scale Application of Neural Networks to Protein Classification", *Artificial Neural Networks 2*, Aleksander, Taylor, Eds., 1992, p 1521-1524,.

[Frit91] Fritzke, B.; "Let it grow: self-organizing feature maps with problem dependent cell structure", *Artificial Neural Networks*, 1991, Kohonen, Mäkisara, Simula, Kangas, Eds., 1991, pp 403-408.

[Frit91] Fritzke, B.; "Growing Cell Structures: a Self-organizing Network in k Dimensions", *Artificial Neural Networks 2*, Aleksander, Taylor, Eds, 1993, pp 1051-1056.

[Fue93] Fuentes, L.; Aldana, J. F.; Troya, J. M.; "URANO: An Object-Oriented Artificial Neural Network Simulation Tool", *New Trends in Neural Computation*, Mira, Cabestany, Prieto, Eds., Springer Verlag, 1993, pp. 364-369.

[Ges90] Geszti, T.; Csabai, I.; Czakó, T.; Szakacs, R.; Serneels; Vattay, G.; "Dynamics of the Kohonen map", in *Statistical Mechanics of Neural Networks*, Springer-Verlag, 1990, pp 341-349.

[Geva91] Geva, S.; Sitte, J. "Adaptive Nearest Neighbor Pattern Classification", *IEEE Trans. on Neural Networks*, Vol. 2., No. 2., pp. 318-322, 1991.

[Gia92] Giacomini, M.; Parisini, T.; Ruggiero, C.; "Secondary structure of proteins from NMR data by neural nets", *Artificial Neural Networks 2*, Aleksander, Taylor, Eds., 1992, p 1603-1607.

[Gol89] Goldberg, D.; "Genetic Algorithms in Search, Optimization and Machine Learning", Addison Wesley, 1989.

[Gol89b] Goldberg, D.; "Zen and the art of Genetic Algorithms", *Procs ICGA89, 3rd Int. Conf. on Genetic Algorithms*, 1989, J. D. Schaffer, Ed. pp 80-85,.

[Gol91] Goldberg, D.; Deb, K.; Korb, B.; "Don't Worry, Be Messy", *Procs. ICGA91, 4th Int. Conf. on Genetic Algorithms*, R. K. Belew, L. B. Booker, (Eds.), Morgan Kauffman, 1991, pp 24-30.

[Gor88] Gorman, R.P.; Sejnowski, T. J., "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets", *Neural Networks*, vol. 1, pp 75-89, 1988.

[Gor88b] Gorman, R. P.; Sejnowski, T. J.; "Learned Classification of Sonar

Targets Using a Massively Parallel Network", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 56, no. 7, pp 1135-1140, Jul 1988.

[Gref90] Grefenstette, J. J.; "A User's Guide to GENESIS Version 5.0", Oct 1990.

[Harp91] Harp, S.A.; Samad, T.; "Genetic Optimization of Self-Organizing Feature Maps", ???; 1991

[Hir92] Hirst, J.D.; Sternberg, M.J.E.; "Prediction of Structural and Functional Features of Protein and Nucleic Acid Sequences by Artificial Neural Networks", *Biochemistry*, vol 31, no. 32, pp. 7211-7218, 1992.

[Holl75] Holland, J.; "Adaptation in Natural and Artificial Systems", Univ. Michigan Press, Ann Arbor, 1975.

[Hol89] Holley, H.; Karplus, M.; "Protein secondary structure prediction with a neural networks", *Procs. Nat. Acad. Sci. USA*, vol 86, pp 152-156, Jan 1989.

[Joc93] Jockush, S.R.; McCaskill, J.S.; "Evolutionary construction algorithms for topology conserving neural nets", *Artificial Neural Networks 2*, Aleksander, Taylor. Eds, pp. 979-982, 1993.

[Kan90] J. A. Kangas, "Variants of Self-Organizing Maps", *IEEE Transactions on Neural Networks*, vol 1, no. 1, March 1990.

[Kit90] Kitano, H.; "Designing Neural Networks using genetic algorithms with graph generation system", *Complex Systems*, vol. 4, pp. 461-476, 1990.

[Kne90] Kneller, D.G.; Cohen, F.E.; Langridge, R.; "Improvements in Protein

Secondary Structure Prediction by an Enhanced Neural Network", *J. Mol. Bio.*, vol. 214, pp. 171-182, 1990.

[Koh82a] Kohonen, T.; "Analysis of a Simple Self-Organizing Process", *Biological Cybernetics*, vol. 44, pp. 135-140, 1982.

[Koh82b] T. Kohonen, "Self organized formation of topologically correct feature maps", *Biological Cybernetics* vol. 43, pp. 59-69, 1982.

[Koh84] T. Kohonen, "Phonotopic maps - Insightful representation of phonological features for speech recognition", in *IEEE 7th Conf. on Pattern Recognition*, 1984, pp 182-185.

[Koh88a] T. Kohonen, "The 'Neural' Phonetic Typewriter", *IEEE Computer*, vol. 21, pp 11-22, March 1988.

[Koh88b] T. Kohonen; C. Barna and R. Chrisley, "Statistical Pattern Recognition with Neural Networks: Benchmarking Studies", *Procs. IEEE Int. Conf. on Neural Networks, ICNN-88*, San Diego, Cal.; 1988, pp. I-61 - I-68.

[Koh89a] Kohonen, T, "Speech Recognition Based on Topology-Preserving Neural Maps", *Neural Computing Architectures*, pp. 26-, 1989.

[Koh89b] Kohonen, T, "Self-Organization and Associative Memory", 3rd edition, Springer Series in Information Sciences, Berlin: Springer Verlag, 1989.

[Koh90a] Kohonen, T.; "The Self-Organizing Map", *Procs. IEEE*, vol. 78, pp. 1464-1480, 1990.

[Koh90b] Kohonen, T.; "Statistical Pattern Recognition Revisited", *Advanced Neural Computers*, R. Eckmiller, Ed.), pp. 137-144, Elsevier, 1990.

[Koh91] T. Kohonen, "Self organizing maps: optimization approaches", in *Artificial Neural Networks*, T. Kohonen, K. Mäkisara, O. Simula, J. Kangas. Eds., pp 981-990, 1991.

[Koh92] Kohonen, T.; Kangas, J.; Laaksonen, J.; "SOM-PAK, The Self-Organizing Map Program Package", version 1.1, Oct 15, 1992; manual de usuario del programa SOM_PAK.

[Kon91] S-G. Kong, B. Kosko, "Differential Competitive Learning for Centroid Estimation and Phoneme Recognition", *IEEE Transactions on Neural Networks*, vol. 2, no. 1, Jan 1991.

[LBG80] Y. Linde, A. Buzo, R.M. Gray, "An Algorithm for Vector Quantizer Design", *IEEE Transactions on Communications*, vol. COM-28, no. 1, 1980.

[Lin93] Linden, A.; Sudbrak, Th.; Tietz, Ch.; Weber, F.; "An Object-Oriented Framework for the Simulation of Neural Nets", *Advances in Neural Information Processing Systems 5*, Hanson, Cowan, Giles, Eds, 1993, pp 797 ss.

[Lipp87] Lippmann, R. P., "An Introduction to Computing with Neural Nets"; *IEEE ASSP Magazine*, April 1987, pp 4-22.

[Lipp89a] Lippmann, R. P., "Review of Neural Networks for Speech Recognition", *Neural Computation*, vol. 1, pp. 1-38, 1989 .

[Lipp89b] Lippmann, R. P., "Pattern Classification Using Neural Networks", *IEEE Communications Mag.* vol 27, no. 11, Nov 1989.

[Llo57] Lloyd, S. P., "Least Squares Quantization in PCM's", Bell Telephone

Laboratories Paper, Murray Hill, NJ, 1957.

[Lut89] Luttrell, S. P.; "Self-Organisation: a derivation from first principles of a class of learning algorithms", *Proc. 3rd. IEEE Int. Joint Conference on Neural Networks, vol II. Washington*, pp 495-498, IEEE NN Council (1989).

[Mak85] Makhoul, J; Roucos, S; Gish, H, "Vector Quantization in Speech Coding", *Procs. IEEE*, vol 73, no. 11, Nov 1985.

[Mara94] Marabini, R.; Carazo, J.M.; "Patter recognition and classification of images of biological macromolecules using Artificial Neural Networks", Biophysics Journal, vol. 66, pp. 1804-1814, 1994.

[Marc93] Marco, S.M.; Valpuesta, J.M.; Rivas, G.; Andrés, G., San Martín, C.; Carrascosa, L.; "A structural model for the GroEL chaperonin", *FEMS-Microbiology Letters* vol. 106, 301-308, 1993.

[Mart91] Martinetz, T.; Schulten, K.; "A "Neural Gas" Network Learns Topologies", in *Artificial Neural Networks*, T. Kohonen, K. Mäkisara, O. Simula, J. Kangas, Eds., vol I, 1991, pp 397-402.

[Mast93] Masters, T.; "Practical Neural Network Recipes in C++", Londres: Academic Press, Inc, Londres.

[McD93] McDonnell, J.R., Waagen, D.; "Evolving neural network connectivity", *ICNN 93 Procs.*, pp 863-868, 1993.

[Men88] Menéndez-Arias, L., Gómez-Gutiérrez, J., García-Fernández, M., García Tejedor, A., Morán, F. "A BASIC microcomputer program to calculate the

secondary structure of proteins from their circular dichroism spectrum", *CABIOS* vol 4., no. 4, pp 479-482, 1988.

[Mer91a] Merelo, Andrade, Ureña, Prieto, Morán, "Application of Vector Quantization Algorithms to Protein Classification and Secondary Structure Prediction", in *Artificial Neural Networks*, A. Prieto (ed.), Lecture Notes in Computer Science, 540, pp 415-421, Springer Verlag, 1991.

[Mer91b] Merelo, Andrade, Prieto, Morán, "Protein classification through a feature map", Proceedings de NeuroNimes'91, Neural Networks and their applications, Herault (ed.), pp 765-768, 1991.

[Mer92] Merelo, Andrade, Prieto, Fernández, Morán, "Using a Neural Network for protein classification", aceptado en Neural Networks for Chemistry, Lyon, junio 1992.

[Mer93] Merelo, Andrade, Prieto, Morán, "Proteinotopic Feature Maps", *Neurocomputing*, vol. 6, pp. 1-12, 1994.

[Mon93] Monte, E.; Hidalgo, D.; Mariño, J.; Hernaez, I., "A vector quantization algorithm based on genetic algorithms and LVQ", *NATO-ASI Bubión 93*, pp 231 ss, 1993.

[Mor93] Morasso, P.; Pareto, A., Sanguineti, V.; "SOC: A Self-Organizing Classifier".

[Mus92] Muskal, S.M.; Kim, Sung-Hou; "Predicting Protein Secondary Structure Content: A Tandem Neural Network Approach", *J. Mol. Bio.*, vol. 225, pp. 713-727, 1992.

[Ober92] Obermayer, K.; Blasdel, G.G.; Schulten, K.; "Statistical-Mechanical analysis of self-organization and pattern formation during the development of visual maps", *Physical Review A*, Vol. 45, No. 10, 15 May 1992.

[Ort93] Ortega, J.; Pelayo, F.J.; Prieto, A.; Pino, B.; Puntonet, C.G.; "'MapA': An Array Processor Architecture for Neural Networks", en *New Trends in Neural Computation*, J. Mira, J. Cabestany, A. Prieto, Eds., pp 432-440, 1993.

[Per93] Pérez, J.-C.; Vidal, E.; "Constructive design of LVQ and DSM Classifiers", en *New Trends in Neural Computation*, J. Mira, J. Cabestany, A. Prieto, Eds., pp 334-339, 1993.

[Perc91] Perczel, A.; Hollósi, M.; Tusnády, G.; Fasman, G.D. "Convex constraint analysis: a natural deconvolution of circular dichroism curves of proteins", *Protein Engineering*, vol 4, no. 6, pp 669-679, 1991.

[Poi91] Poirier, F. "Improving the Training and Testing Speed and the Abitility in Learning Vector Quantization: DVQ", *Procs. ICASSP-91*, Toronto, pp 649-653, 1991.

[Prie90] Prieto, A.; Martín-Smith, P.; Merelo, J.J.; Pelayo, F. J.; Ortega, J.; Fernandez, F. J.; Pino, B; "Simulation and Hardware Implementation of Competitive Learning Neural Networks", *Statistical Mechanics of Neural Networks*, Springer-Verlag, 1990, pp 415-421.

[Qian87] Qian, N.; Sejnowski, T.J.; "Learning to predict the secondary structure of proteins of globular proteins", *Evolution, Learning and Cognition*, Y.C.Lee, (Ed).

[Qian88] Qian, N.; Sejnowski, T.J.; "Predicting the Secondary Structure of Globular Proteins Using Neural Network Models", *J. Mol. Biol.* vol. 202, pp. 865-884, 1988.

[Rit89] H. Ritter, T. Kohonen; "Self-Organizing Semantic Maps", *Biological Cybernetics*, 61, 241-254 ( 1989 ).

[Rit91] Ritter, H.; "Learning with the Self-Organizing Map", in *Artificial Neural Networks*, T. Kohonen, K. Mäkisara, O. Simula, J. Kangas, Eds., pp 379-384.

[Rost93a] Rost, B.; Sander, C.; "Prediction of protein secondary structure at better than 70% accuracy", preprint submitted to J. Mol. Bio., 1993.

[Rost93b] Rost, B.; Sander, C.; "Improved prediction of protein secondary structure by use of sequence profiles and neural networks", Procs. Nat. Acad. Sciences, USA, 1993.

[Rum85] Rumelhart, D. E.; "Feature Discovery by Competitive Learning", *Cognitive Science*, 9, 75-112, 1985. También aparece en Parallel Distributed Processing.

[Scha84] Schaffer, J. D.; "Some experiments in machine learning using vector evaluated genetic algorithms", Ph. D. Thesis, Dept. of Electrical Engineering, Vanderbilt Univ. Dec. 1984.

[Scha85] Schaffer, J. D.; Grefenstette, J. J.; "Multi-Objective Learning via Genetic Algorithms", in *Procs. of the 9th international Conference on Artificial Intelligence*, pp 593-595, 1985.

[Scho91] Scholtes, J.C., "Kohonen Feature Maps in Natural Language

Processing", PhD Thesis, Dept. of Computational Linguistics, Universiteit van Amsterdam, Holanda.

[Schr92] Schraudolph, N. N.; "A User's Guide to GAUCSD 1.4", Technical Report CSE Dept CS92-249, UC San Diego, La Jolla, CA 92093-0114.

[Sie93] Siemon, H.P.; "Selection of Optimal Parameters for Kohonen Self-organizing Feature Maps", *Artificial Neural Network, 2*, Aleksander, Taylor, Eds., p 1573-1577, 1993.

[Sto92] Stolorz, p.; Lapedes, A.; Xia, Y.; "Predicting Protein Secondary Structure Using Neural Nets and Statistical Methods", *J. Mol. Biol.*, vol. 225, pp. 363-377, 1992.

[Tat89] G. Tatershall, "Neural maps applications", in *Neural Computing Architectures: the Design of Brain-like Machines,* I. Aleksander, ed., North Oxford Academic, 1989.

[Tol90] Tolat, V.V.; "An analysis of Kohonen's self-organizing maps using a system of energy functions", Biological Cybernetics, vol. 64, pp. 155-164, 1990.

[Tou74] Tou, J.T., González, R.C., "Pattern Recognition Principles", Addison Wesley, 1974.

[Tre93] Treleaven, P. C.; Rocha, P. V.; "Hybrid Programming Environments", *New Trends in Neural Computation*, Mira, Cabestany, Prieto, Eds., Springer Verlag, pp. 351-357, 1993.

[Tro91] Troya, J.M.; Aldana, J. F. , "Extending an Object Oriented Concurrent Logic Language for Neural Network simulation", *Artificial Neural*

*Networks*, A. Prieto, (Ed.), Procs. IWANN 91, Springer Verlag, 1991, pp 235-242.

[Van92] Vanhala, J.; Clementi. E.; Kaski, K.; "Protein Structure Prediction and Neural Networks", *Artificial Neural Networks 2*, Aleksander, Taylor, Eds., p 1603-1607 (1992).

[Var93] Varfis, A.; Versino, C.; "Selecting Reliable Kohonen Maps for Data Analysis", in *Artificial Neural Networks,2* , Aleksander, I., Taylor, J., Eds., pp 1583-1586, 1993.

[Visa90] Visa, S.; "Stability Study or Learning Vector Quantization", *INNC vol. 2*, pp 729-732, 1990.

[Wall90] Wall, L.; Schwartz, R.; "Programming PERL", NutShell books, O'Reilly & Associates, 1990.

[Wer91] Werner, G.; Dyer, M.G.; "Evolution of communication in artificial organisms", in *Artificial Life II*, C. G. Langton, C. Taylor, J. D. Farmer, S. Rasmussen (Eds.), Adison Wesley, 1991.

[Whit89] Whitley, D.; Hanson, T.; "Optimizing Neural Networks using faster, more accurate genetic search", in *Procs. of the 3rd. Int. Conf. on Genetic Algorithms and their applications*, J.D. Schaffer (ed.), 391-396, Morgan Kauffmann, San Mateo, CA, 1989.

[Yai92] E. Yair; K. Zeger; A. Gersho; "Competitive Learning and Soft Competition for Vector Quantizer Design", *IEEE Transactions on Signal Processing*, vol 40, no. 2, Feb. 1992.

[Yan86] Yang, J.T.; Wu, C.-S.C.; Martinez, H. M. "Calculation of protein

conformation from circular dichroism", *Methods in Enzymology*, vol. 130, pp. 208-269, 1986.

[Yao92] Yao, X.; "A Review of Evolutionary Artificial Neural Networks", Technical Report, 1992.

# Apéndice A: utilización del programa G-LVQ

## A.1 Compilación

El programa G-LVQ está escrito en c++ estándar, de forma que puede ser compilado sin problemas en los siguientes sistemas, en los que ya ha sido probado.

1. Estaciones de trabajo Sun con Solaris 4.0, y compilador de C++ G++ de la *Free Software Foundation*.
2. Estaciones de trabajo Silicon Graphics Indigo R4000 con sistema operativo Irix 4.1, y compilador de C++ G++ de la *Free Software Foundation*.
3. Ordenadores compatibles PC con compilador Borland C++ 3.1.

Para compilarlo en los dos primeros, ejecutar

```
unix% cp Makefile.{sun|sgi} Makefile
```

dependiendo de la máquina de la que se trate. Una vez copiado el Makefile, editarlo para que incluya los directorios en los que se encuentran los ficheros de cabecera en la máquina correspondiente, y escribir `make`.

En *ordenadores compatibles PC*, cargar el proyecto `g-lvq.prj`, incluido en el directorio, y compilar.

## A.2 Ejecución.

El programa G-LVQ necesita tres ficheros para funcionar: un *fichero de entrenamiento*, que contiene las muestras con las cuales se realiza el entrenamiento de toda la población de diccionarios y la prueba al final de cada generación; un *fichero de test*, que habitualmente es diferente del de entrenamiento, con el cual se realiza la prueba al terminar la última

generación sobre los mejores diccionarios, y un *fichero de configuración*, que se suministrará en la línea de comandos, que especifica los parámetros que usará G-LVQ.

El formato de los dós ficheros de entrenamiento y prueba es el siguiente

```
m_comp_1<Tab>m_comp_2<Tab>.....<Tab>m_comp_n<Tab>label<Enter>
```

y es necesario usarlo también en el fichero de prueba, aunque el programa no use esa información.

En cuanto al *fichero de configuración*, tiene el formato siguiente, aunque las órdenes pueden estar en cualquier orden.

```
Parámetro de configuracion<espacios>valor del parámetro
```

Los parámetros de configuración son los siguientes

►*Generations*: número de generaciones durante las cuales se va a llevar a cabo el entrenamiento. Ver capítulos 5 y 7 para una indicación de los valores correctos para problemas determinados; un valor por defecto adecuado es 100. Es un parámetro del algoritmo genético.

►*Dimension*: dimensión del lado de la retícula en la cual se sitúan los genomas; este número al cuadrado será la población sobre la que actuará el algoritmo genético. La dimensión depende del problema; por defecto se puede utilizar una dimensión igual a 10. También es un parámetro del algoritmo genético.

►*xOverRate*: grado de crossover utilizado en el intercambio genético entre dos genomas. El crossover usado por el algoritmo G-LVQ es uniforme, por lo tanto se refiere al número de bits que se intercambiarán entre los dos genomas. Como se ha indicado en el capítulo 7, se obtienen mejores resultados con un grado de crossover pequeño, entre 0.1 y 0.2, es decir, entre el 10 y el 20%.

►*mutationRate*: grado de mutación, o porcentaje de bits que cambiarán de 0 en 1 y viceversa. Como suele ser habitual en algoritmos genéticos, es del orden de 0.01, es decir, del 1% o inferior.

►*randIncrementRate, incrementRate, killRate*: tasas de aplicación de los operadores genéticos para genomas de longitud variable: incremento aleatorio, duplicación y eliminación. Como se ha indicado en el capítulo 7, el primero puede ser 0 y los demás del orden del 1%; en algunos casos se podrá incrementar la tasa de duplicación hasta el 10%.

▸*fileInput, resultsFile*: nombre del fichero de entrenamiento y del fichero de test, sobre el cual se evaluarán al final los resultados de los mejores diccionarios obtenidos. Ambos pueden ser el mismo. Hay que indicar el camino (*path)* completo, para que el programa pueda encontrarlos.

Al principio de la ejecución del programa, se carga y examina el fichero de configuración, y a partir de él, se carga y examina el fichero de entrenamiento, calculando su tamaño, el número de dimensiones de los vectores de entrada y el número de clases, a partir de las etiquetas de clase del fichero de entrenamiento. Se inicializan la población inicial, y comienza el entrenamiento. En pantalla aparece información sobre la ejecución del algoritmo, generación actual, aplicación de los operadores genéticos de longitud variable y número de nuevos genomas aparecidos en cada generación. Si no se desea ver esta información, puede redireccionarse a `/dev/null`.

## A.3 Resultados

Para evaluar el progreso del algoritmo y sus resultados finales, durante la ejecución del mismo sé producen una serie de ficheros, al final de cada generación o cada 10 generaciones. Estos ficheros se almacenan en el directorio `<directorio actual>/data`, que se deberá crear antes de la ejecución del algoritmo. Se aconseja que se borren los ficheros no necesarios después de cada ejecución; en versiones posteriores se incluirá la posibilidad de elegir a voluntad los ficheros que se quieren conservar. Todos los nombres, salvo algunas excepciones, están en el formato 8.3 de MS-DOS. Al principio de cada ejecución, se examinan automáticamente los ficheros existentes, y se asigna un dígito posterior al último encontrado que identifique a todos los ficheros.

Los ficheros son los siguientes:

▸`ll-*.<generación>, di-*.<generación>, fit-*.<generación>`: estos ficheros se graban cada diez generciones; almacenan la longitud, distorsión y número de aciertos de cada genoma en la generación correspondiente; el número de fila y columna en el fichero corresponden al número de fila y columna correspondiente en la rejilla de genomas.Se ha realizado también un programa en C para ver la evolución

de cada uno de los valores incluidos en estos ficheros a lo largo del entrenamiento; este programa en C usa un programa en AWK para transformar los ficheros a un formato que entienda el programa GNUPLOT.

▸`fitmx-*, dimx-*`: estos ficheros almacenan el número de aciertos máximo, en porcentaje sobre 1, y distorsión mínima en cada generación.

▸`fitmn-*, dimn-*, llmn-*`: número de aciertos medio (en porcentaje sobre 1), distorsión media y longitud media de los genomas en cada generación.

▸`iftmn-*, idimn-*`: número máximo de aciertos inicial (previo al entrenamiento), y distorsión inicial (previa al entrenamiento); útiles para comprobar el efecto Baldwin.

▸`g<número de fila><número de columna>-*`: genoma de los diccionarios con máximo número de aciertos. El formato es el siguiente

```
comp_1 comp_2 .... comp_n ## etiqueta ##
```

Para visualizar estos genomas, o bien generar gráficos que se puedan incluir en un documento, se incluye el programa PSGEN4.PL, que genera un fichero PostScript con el mismo genoma formateado.

▸`r<número de fila><número de columna>-*.<número de clase>`: vectores diccionario correspondientes a cada clase; cada línea es un vector diferente y la etiqueta de clase es la incluida en el nombre del fichero. Se ha realizado también un programa en PERL para visualizar estos diccionarios junto con el fichero de entrenamiento, PLOTGEN2.PL y con el genoma; únicamente es necesario dar como entrada el número de orden de los ficheros y el nombre del fichero de entrenamiento; se tiene que ejecutar desde el mismo directorio DATA.

▸`t<número de fila><número de columna>-*`: fichero de test, donde se incluye la clasificación de cada uno de los mejores genomas a los vectores del fichero de test, así como el grado de acierto. El grado de acierto se puede examinar posteriormente comparando las etiquetas del fichero de test con las clasificaciones hechas por los genomas.

▸`param-*`: fichero donde se almacena el nombre del fichero de parámetros utilizado en el entrenamiento. Se aconseja que este fichero de parámetros sea de sólo lectura, con el objeto de que, una vez hecho el entrenamiento, no se modifique; opcionalmente se puede copiar el fichero de parámetros a este fichero.