UNIVERSIDAD DE GRANADA

24 julio

DISEÑO Y OPTIMIZACIÓN DE REDES DE FUNCIONES DE BASE RADIAL MEDIANTE TÉCNICAS BIOINSPIRADAS

TESIS DOCTORAL

Antonio Jesús Rivera Rivas

2003

Departamento de Arquitectura y Tecnología de Computadores

A Loli

AGRADECIMIENTOS

Con estas líneas quiero expresar mi agradecimiento a mis directores en este trabajo, *Julio e Ignacio*, que siempre me han orientado en la dirección correcta, que en todo momento han estado disponibles ante mis requerimientos y que constatemente se han interesado por mis problemas.

A *Alberto Prieto*, *Carlos G. Puntonet* y *Jesús González*, por el apoyo y la ayuda prestada.

A *María José del Jesús*, por el interés que ha puesto en mi trabajo y por sus indicaciones técnicas.

A *Pedro González*, por su apoyo diario y su permanente estado de disponibilidad.

Y en general a todos mis compañeros del Departamento de Informática de la Universidad de Jaén.

La realización de esta Tesis ha sido soportada por el proyecto de Ministerio de Ciencia y Tecnología TIC2000-1348.

INDICE

0. Prólogo

La metodología que se suele seguir a la hora de solucionar un problema, consiste en realizar un análisis de todos los elementos que lo caracterizan y en función de éstos, elegir las herramientas más adecuadas para solventarlo. Tras el estudio preliminar, lo ideal es que los sistemas a modelar o controlar, estuvieran descritos de una manera precisa y completa. Para este tipo de problemas, existen una serie de métodos clásicos que los resuelven sin ambigüedades. En este tipo de modelos clásicos se podrían incluir los métodos de razonamiento lógico simbólico o los métodos de búsqueda numéricos, y a los que se les suele denominar *hard-computing*, Figura 0-1.

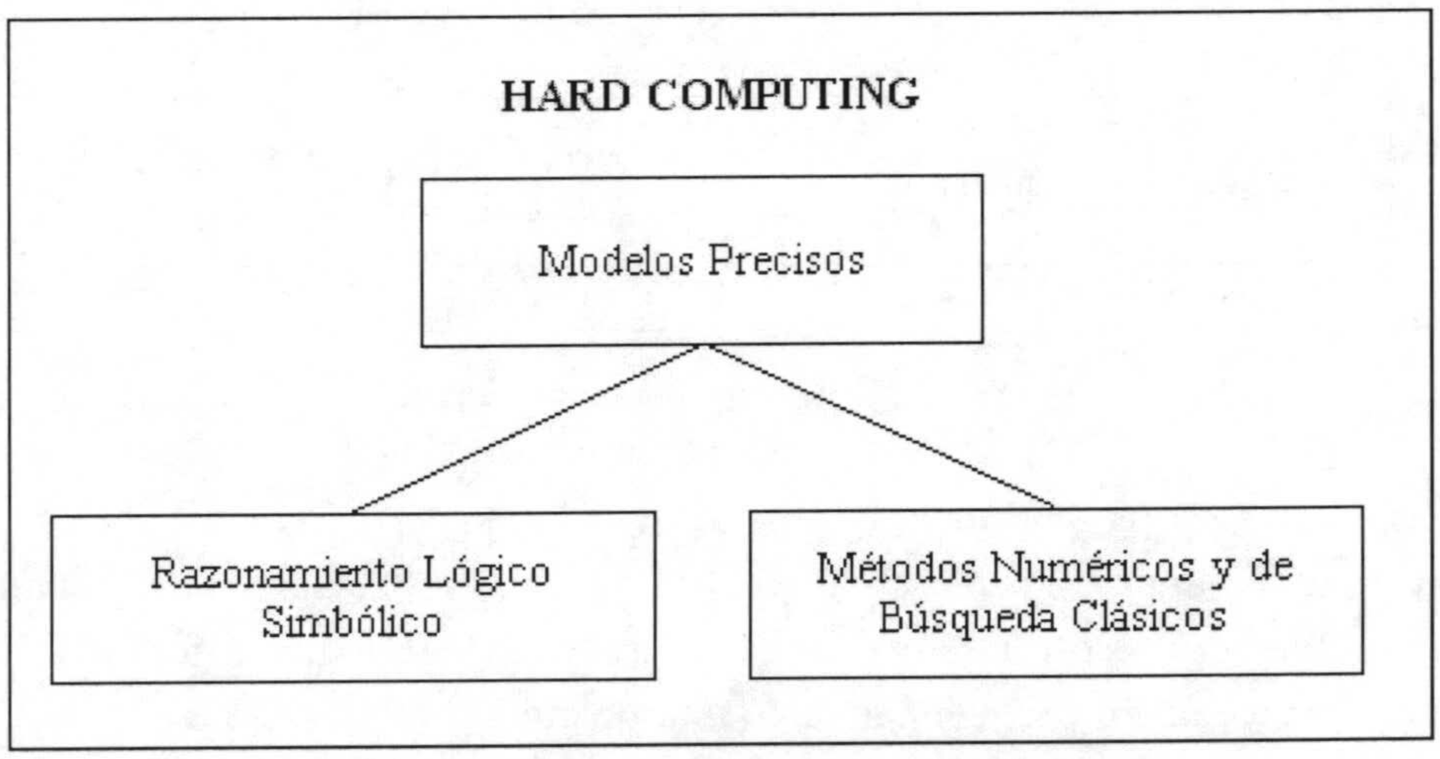

Figura 0-1: Elementos del Hard-Computing

Sin embargo, cuando se intentan resolver problemas provenientes del mundo real, nos encontramos con que suelen estar mal definidos, resultan difíciles de modelar o presentan espacios de soluciones complejos. Esto se debe principalmente a que la información disponible, y que representa la conducta del modelo suele ser parcial, fruto de la experiencia y viene dada por instancias de datos de entrada / salida. En este

caso, el uso de modelos precisos es impracticable, muy caro o simplemente, inexistentes.

Ante esta situación, y para tratar la información anterior, aparece el s*oft-computing*, Figura 0-2. Concretamente las tecnologías soft-computing proveen de un conjunto de herramientas para la computación flexible. Esta computación implementa una serie de métodos de búsqueda, de razonamientos o de optimizaciones, adaptados o especializados en la resolución de problemas provenientes del mundo real.

Según [Zadeh, 1994] el *soft-computing* engloba una serie de tecnologías cuyas características se resumen en su célebre frase "… en contraste con el *hard-computing*, el *soft-computing* es tolerante con la imprecisión, la incertidumbre y con la verdad parcial". En este contexto, se considera a la *lógica difusa*, al *razonamiento probabilístico*, a las *redes neuronales*, y a la *computación evolutiva* como los principales componentes del *soft-computing*.

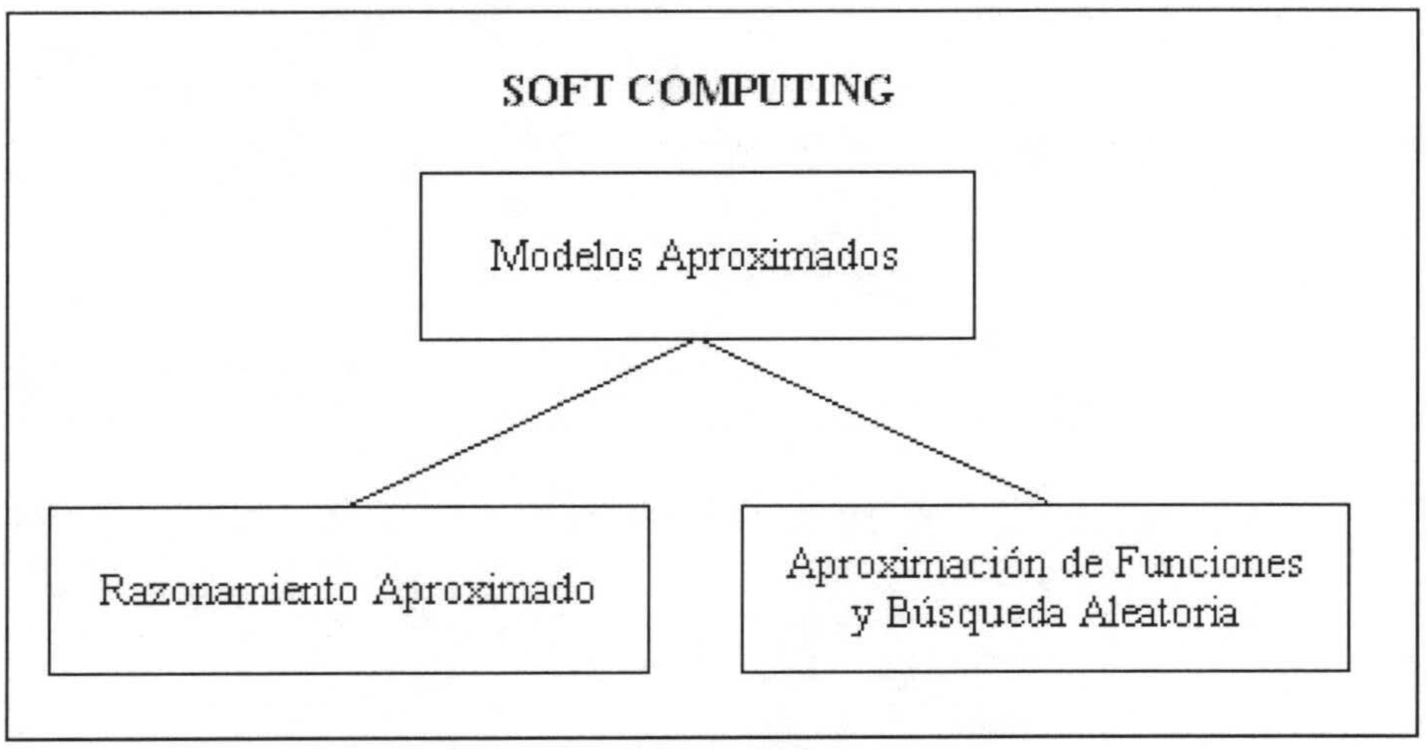

Figura 0-2: Elementos del soft-computing

La *lógica difusa*, introducida por [Zadeh, 1965], aporta un lenguaje, con su sintaxis y su semántica, al que podemos trasladar nuestro razonamiento cualitativo sobre el problema a resolver. Entre las características de la lógica difusa destacan su robustez y su mecanismo de razonamiento interpolativo.

El *razonamiento probabilístico* como las *redes de creencia bayesiana* se basan en el trabajo original de [Bayes, 1763] y en la *teoría de certeza de Dempster-Shafer*, que desarrollaron independientemente [Dempster, 1967] y [Shafer, 1976]. Éstas proveen de un mecanismo para evaluar la salida de sistemas afectados, por aleatoriedad u otros tipos de incertidumbre probabilística. La principal característica del razonamiento probabilístico, es su habilidad para actualizar estimaciones de salidas previas condicionándolas a la disponibilidad de nuevas evidencias.

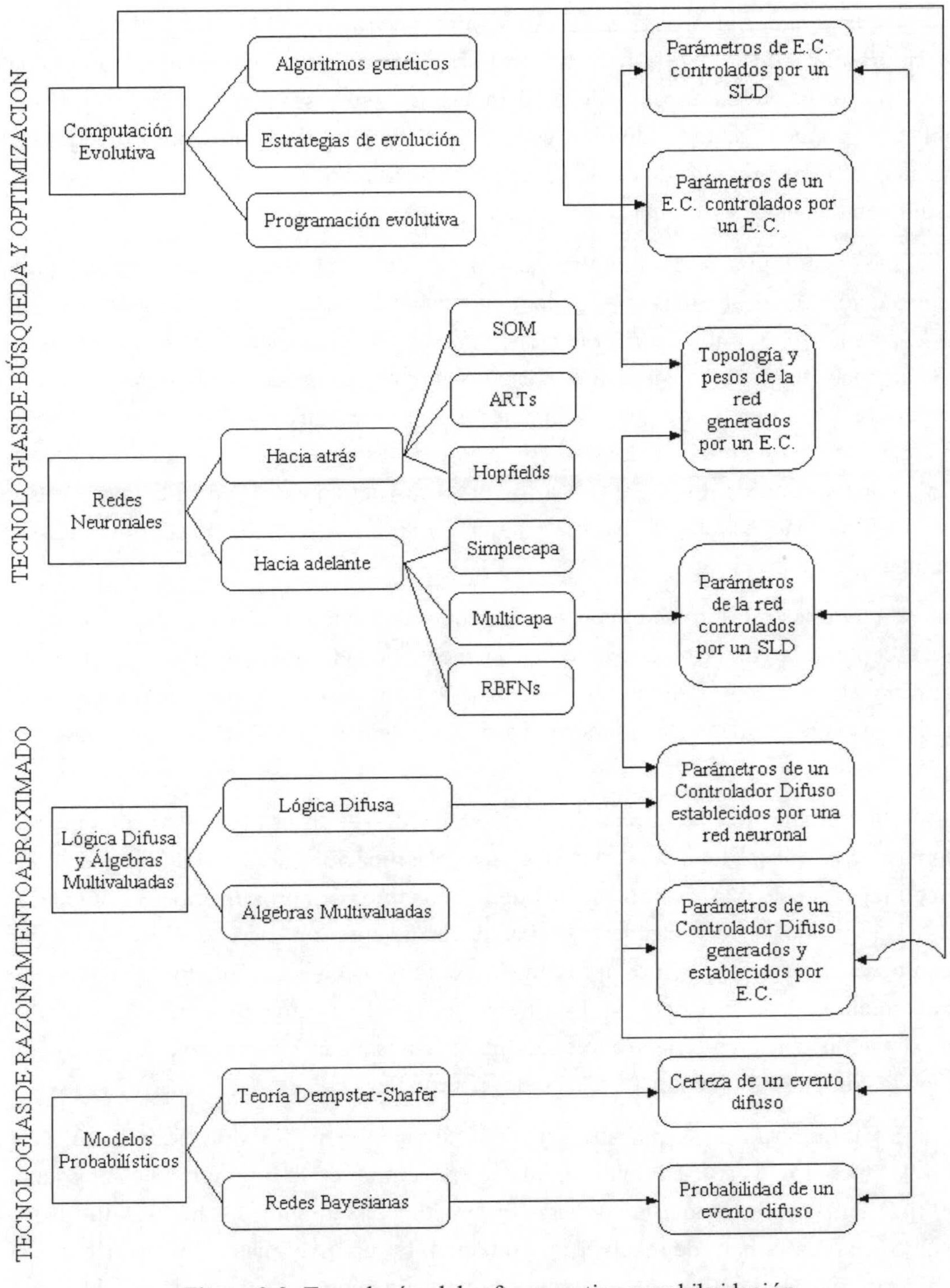

Figura 0-3: Tecnologías del soft-computing y su hibridación

Las *redes neuronales*, en las que trabajaron inicialmente [Rosenbaltt, 1959] o [Widrow, 1960], son estructuras de computación que pueden ser entrenadas para aprender patrones a partir de ejemplos. La capacidad de aprendizaje de éstas, va a venir definida, tanto por su estructura como por los algoritmos de aprendizaje que se utilicen en su entrenamiento.

La *computación evolutiva*, en sus distintas vertientes [Holland, 1975], [Fogel, 1962], [Rechenberg, 1973], [Schwefel, 1975], aporta una estrategia de búsqueda global aleatoria en un espacio de soluciones. En este espacio, se trabaja con una población, donde cada individuo representa una solución. Estos individuos se evalúan, obteniendo para cada uno de ellos, un valor de adaptación que indica la eficiencia de la solución que éste representa.

Las tecnologías soft-computing se derivan de las aproximaciones clásicas para la resolución de problemas, y por lo tanto están basadas en *lógica booleana*, en *modelos analíticos*, en *clasificaciones nítidas* y en *búsqueda determinística*. Para adaptar estas técnicas clásicas a la resolución problemas del mundo real, se han variado los fundamentos de definición de éstas, y se han introducido en ellos una serie de elementos bioinspirados. La incorporación de estos elementos, implica que las tecnologías soft-computing, emulan a los procesos naturales en su funcionamiento y por lo tanto, a sus estrategias para la resolución de problemas, como no podía ser de otra manera.

Una de las cuestiones importantes, es que las tecnologías soft-computing no se deben contemplar como excluyentes en la resolución de un problema, sino todo lo contrario. Por tanto, se debe partir de la idea de que estas tecnologías van a cooperar en la resolución de éste, dando lugar a sistemas híbridos de componentes especializados.

Un ejemplo de este trabajo conjunto se puede ver en la Figura 0-3, extraída de [Bonissone, 2000], donde se muestra una clasificación de las tecnologías soft-computing. En esta clasificación se distinguen dos tipos de aproximaciones principales para la resolución de problemas. La primera aproximación comprendería las tecnologías de razonamiento aproximado, como encontraríamos los modelos de razonamiento probabilístico y los modelos de razonamiento difuso. La otra aproximación comprendería tecnologías para la búsqueda y optimización, donde se hallan la computación evolutiva o las redes neuronales.

En la Figura 0-3 se muestran también algunos ejemplos típicos de la sinergia entre las tecnologías soft-computing, que se encuentran en la bibliografía [Bonissone, 2000]. Además y fruto de esta cooperación, podemos distinguir dos niveles diferentes de abstracción a la hora de resolver un problema. En un nivel más bajo, se encontraría el algoritmo o heurística que trabaja directamente con los datos del problema a resolver. En un nivel superior se encontrarían los métodos que configuraría de una manera dinámica al algoritmo anterior. A estos métodos del nivel superior se les denomina *meta-heurísticas*.

La estrategia clásica de resolución de problemas, consistía en utilizar un algoritmo, más o menos estático, en el nivel de interacción de datos del problema. Este

modo de trabajo tiene una serie de inconvenientes cuyo denominador común sería la falta de adaptación del algoritmo, al dominio donde se encuentran definidos los datos del tipo concreto de problema a resolver. Con la introducción de meta-heurísticas, se van a mitigar los inconvenientes anteriores, diseñándose métodos de trabajo adaptativos, y por lo tanto más robustos a la hora de solventar problemas.

El objetivo planteado en esta tesis, es el diseño y optimización de un tipo de redes neuronales, las redes de funciones de base radial, aplicadas a la aproximación funcional. Básicamente nos enfrentamos a un problema de optimización, donde se define un espacio de búsqueda, en el que cada uno de las soluciones que obtengamos representa un punto en este espacio. La labor de nuestro algoritmo será pues alcanzar el óptimo global o punto que presente un mínimo error.

Para solventar este problema vamos a aportar una solución soft-computing donde, naturalmente, se usarán varias de sus tecnologías bioinspiradas, con una estructura de varios niveles de abstracción. En un nivel más inferior tendremos nuestra red neuronal que trabaja con los datos proporcionados por el problema y para los que tendrá que proporcionar un modelo lo más general posible. Para llevar a cabo esta tarea de aprendizaje de la red, se dispondrán otras tecnologías soft-computing como la computación evolutiva o la lógica difusa, en un nivel superior o como meta-heurísticas.

Con esta arquitectura de tecnologías soft-computing, se obtendrá una solución buena y robusta en el sentido de que se encuentre cerca del óptimo global. Para finalizar, se aplicará un algoritmo de optimización clásica, con el objetivo de aprovechar su potencia en el contexto proporcionado por la fase anterior, y ofrecer una salida lo más próxima al óptimo global.

En el *capítulo 1. Introducción*, se hará una descripción de distintas técnicas de resolución de problemas, las cuales se valorarán en cuanto a sus ventajas e inconvenientes. El capítulo comienza con la descripción utilización de las técnicas clásicas empleadas en la optimización de funciones. Posteriormente se describirán con detalle las tecnologías soft-computing, que más tarde se utilizan para resolver nuestro problema. Concretamente estas tecnologías son las redes neuronales, la computación evolutiva y la lógica difusa.

El *capítulo 2. Redes de Funciones de Base Radial*, se centra en la descripción del tipo redes neuronales que va a emplear nuestro método para realizar la aproximación funcional. Primeramente se presentarán estas redes neuronales, y se enumerarán sus principales características. Después se analizarán los principales algoritmos, que en la bibliografía, se emplean en el diseño de éstas.

En el *capítulo 3. Descripción del algoritmo propuesto*, se detallará el algoritmo propuesto. Para esto, se comienza poniendo de manifiesto cuales son las directrices

que guiarán el diseño de nuestra red. Después se pasa a describir, justificar y analizar cada uno de los pasos del algoritmo propuesto.

En el *captítulo 4. Experimentación y resultados*, se pone a prueba el algoritmo propuesto. Para esto, se comienza haciendo un estudio de su robustez, con respecto a la relatividad de la configuración de sus parámetros. Después se mostrarán los resultados obtenidos por el algoritmo para un batería de funciones.

Por último y en el *capítulo 5. Conclusiones*, además poner de manifiesto las conclusiones obtenidas tras la realización de este trabajo, se enumerarán las aportaciones realizadas y se propondrán una serie de líneas de trabajo futuro.

1. Introducción

Este capítulo va a servir para describir algunas de las principales tecnologías que componen el soft computing, concretamente las *redes neuronales artificiales*, la *computación evolutiva* o la *lógica difusa*. Incluso se describirá alguna extensión de estas como son los *métodos coevolutivos cooperativos* o la utilización conjunta de algunas de estas técnicas como sería el desarrollo de *redes neuronales* mediante *algoritmos evolutivos*.

La idea es aportar una nueva solución al problema planteado de la aproximación de funciones, utilizando estas estrategias de aprendizaje, de búsqueda de soluciones y de razonamiento. Pero además, esta solución aportada se complementará con la utilización de alguna técnica más tradicional o hard computing, como es el método de optimización matemática *Levenberg-Marquardt*. Por esto se comenzará el capítulo definiendo el problema de optimización, que en esencia es el problema que queremos resolver, exponiendo las técnicas que tradicionalmente se usan para solventarlo.

1.1 El Problema de la Optimización

1.1.1 Introducción

En general un problema de optimización consiste en encontrar un vector de parámetros libres $\overline{p} \in M$ del sistema considerado, teniendo en cuenta un cierto criterio de calidad f: $M \rightarrow \Re$ (llamado normalmente función objetivo) que es máximizado o minimizado.

$$f(\overline{p}) \rightarrow \max \qquad (1\text{-}1)$$

La función objetivo puede provenir de sistemas del mundo real y es de complejidad arbitraria. La solución al problema de optimización global (1-1) requiere encontrar un vector $\overline{p}^*$ de modo que:

$$\forall\, \overline{p} \in M : f(\overline{p}) \leq f(\overline{p}^*) = f^* \qquad (1\text{-}2)$$

Sin embargo existen características que dificultan e incluso llegan a imposibilitar la tarea de optimización. Estas características son por ejemplo *multimodalidad*, es decir la existencia de varios extremos locales $\overline{p}'$:

$$\exists \varepsilon > 0 : \forall\, \overline{p} \in M : \varphi(\overline{p}, \overline{p}') < \varepsilon \Rightarrow f(\overline{p}) \leq f(\overline{p}') \qquad (1\text{-}3)$$

donde φ es una medida de distancia en M. Pueden existir también *restricciones* en el conjunto M, que vienen dadas por funciones $c_i : M \rightarrow \Re$, de modo que el conjunto de soluciones factibles $F \subseteq M$ sea sólo un subconjunto del dominio de las variables:

$$F = \left\{ \begin{array}{c} \overline{p} \in M \mid c_i(\overline{p}) = 0 \quad i = 1, 2, \ldots, m' \quad y \\ c_i(\overline{p}) \geq 0 \quad i = m'+1, \ldots, m \end{array} \right\} \qquad (1\text{-}4)$$

Cuando las ecuaciones de restricción tienen la forma $c_i(\overline{p}) = 0$, se les llama *restricciones de igualdad*, cuando tienen la forma $c_i(\overline{p}) \leq 0$ se les denomina *restricciones de desigualdad*.

Otros factores que dificultan la resolución del problema de la optimización son los conocidos como *gran dimensionalidad*, *no linearidades fuertes*, *no diferenciabilidad*, *ruido*, o *funciones objetivo variables con el tiempo*.

Los problemas de optimización suelen aparecer en diversos campos técnicos, económicos o científicos relacionados con minimización de costos, tiempos o riesgos, o maximización de la calidad o de la eficiencia [Davis, 1991]. Por esto es importante el desarrollo de estrategias que resuelvan estos problemas.

Existen muchos algoritmos de optimización, aunque un determinado algoritmo suele utilizarse para resolver un problema concreto. Es por esto que es importante reconocer las características de un problema dado para escoger la estrategia de solución que mejor se adapte a éste. Los problemas de optimización se pueden clasificar atendiendo a diferentes características como se muestra en la Tabla 1-1.

Las primeras estrategias [Dennis, 1983] desarrolladas para resolver el problema de la optimización utilizan métodos matemáticos. Estos métodos se van a basar en explotar, características matemáticas de la función a optimizar. Dependiendo ya del método elegido y para poder aplicarlo, es necesario que la función cumpla algunas

condiciones como por ejemplo continuidad, diferenciabilidad, etc. Partiendo de estas premisas se definen conceptos como los siguientes.

Característica	*Propiedad*	*Clasificación*
Número de variables de control	Una	Univariable
	Más de una	Multivariable
Tipo de las variables de control	Números reales continuos	Continua
	Enteros	Entera o discreta
	Reales continuos y enteros	Enteros mezclados
	Enteros en permutaciones	Combinacional
Funciones del problema	Funciones lineales de las variables de control	Lineal
	Funciones cuadráticas de las variables de control	Cuadrática
	Otras funciones no lineales de las variables de control	No lineal
Formulación del problema	Sujeto a restricciones	Con restricciones
	No sujeto a restricciones	Sin restricciones

Tabla 1-1: Clasificación de los problemas de optimización en base a diferentes características

Se dice que $\overline{p}^*$ es un punto estacionario de $f(\overline{p})$, si

$$g(\overline{p}^*) = 0 \qquad (1\text{-}5)$$

donde $g(\overline{p})$ es el gradiente de $f(\overline{p})$ y

$$g_i(\overline{p}) = \frac{\partial f(\overline{p})}{\partial p_i} \qquad (1\text{-}6)$$

El punto $\overline{p}^*$ es además un mínimo local de $f(\overline{p})$, si la matriz Hessiana $H(\overline{p})$, o matriz de segundas derivadas con componentes:

$$H_{ij}(\overline{p}) = \frac{\partial^2 f(\overline{p})}{\partial p_i \partial p_j} \qquad (1\text{-}7)$$

se define positiva en $\overline{x}^*$, es decir si:

$$\overline{u}^T H(\overline{p}^*)\overline{u} > 0 \quad \forall \overline{u} \neq 0 \qquad (1\text{-}8)$$

En nuestro caso el tipo de problema de optimización a resolver no tiene restricciones por lo que nos centraremos en la descripción de los algoritmos que resuelven éste. Los pasos seguidos por estos métodos se muestran en el Algoritmo 1-1.

1. Establecer un punto de partida p_0.
2. Repetir Mientras no se cumpla la condición de parada
   a. Calcular la dirección de búsqueda d_k
   b. Determinar el tamaño del paso λ_k.
   c. Establecer $\overline{p}_{k+1} = \overline{p}_k + \lambda_k \cdot \overline{d}_k$

Algoritmo 1-1: Algoritmo de optimización clásico

Tras la aplicación del algoritmo, y tal y como se observa en la Figura 1-1, a partir de un punto inicial se decide una dirección de movimiento, teniendo en cuenta algún criterio, en el que suele estar involucrado el gradiente de la función. Con esto se va determinando una línea de búsqueda y se va delimitando un intervalo que se va acotando más y más, hasta encontrar en más o menos pasos una solución con la precisión deseada. El salto que se va produciendo en la aproximación al punto deseado, va a depender de un valor prefijado.

Todo esto se va a traducir en que vamos a cambiar la configuración del vector parámetros $\overline{p}_k$, objeto de la búsqueda, a un nuevo valor $\overline{p}_{k+1}$ que va a tener menor error de aproximación. Esta nueva configuración será la que menor error de aproximación produzca cuando se desplacen los parámetros a lo largo de un vector dirección $\overline{d}_k$. Así se cumple que:

$$\overline{p}_{k+1} = \overline{p}_k + \lambda_k \cdot \overline{d}_k \qquad (1\text{-}9)$$

donde $\overline{d}_k$ va a depender del algoritmo utilizado. λ_k es el tamaño del paso utilizado en la búsqueda y también depende del algoritmo. Como condición de parada se suele establecer que la función varíe un valor pequeño durante unas pocas iteraciones.

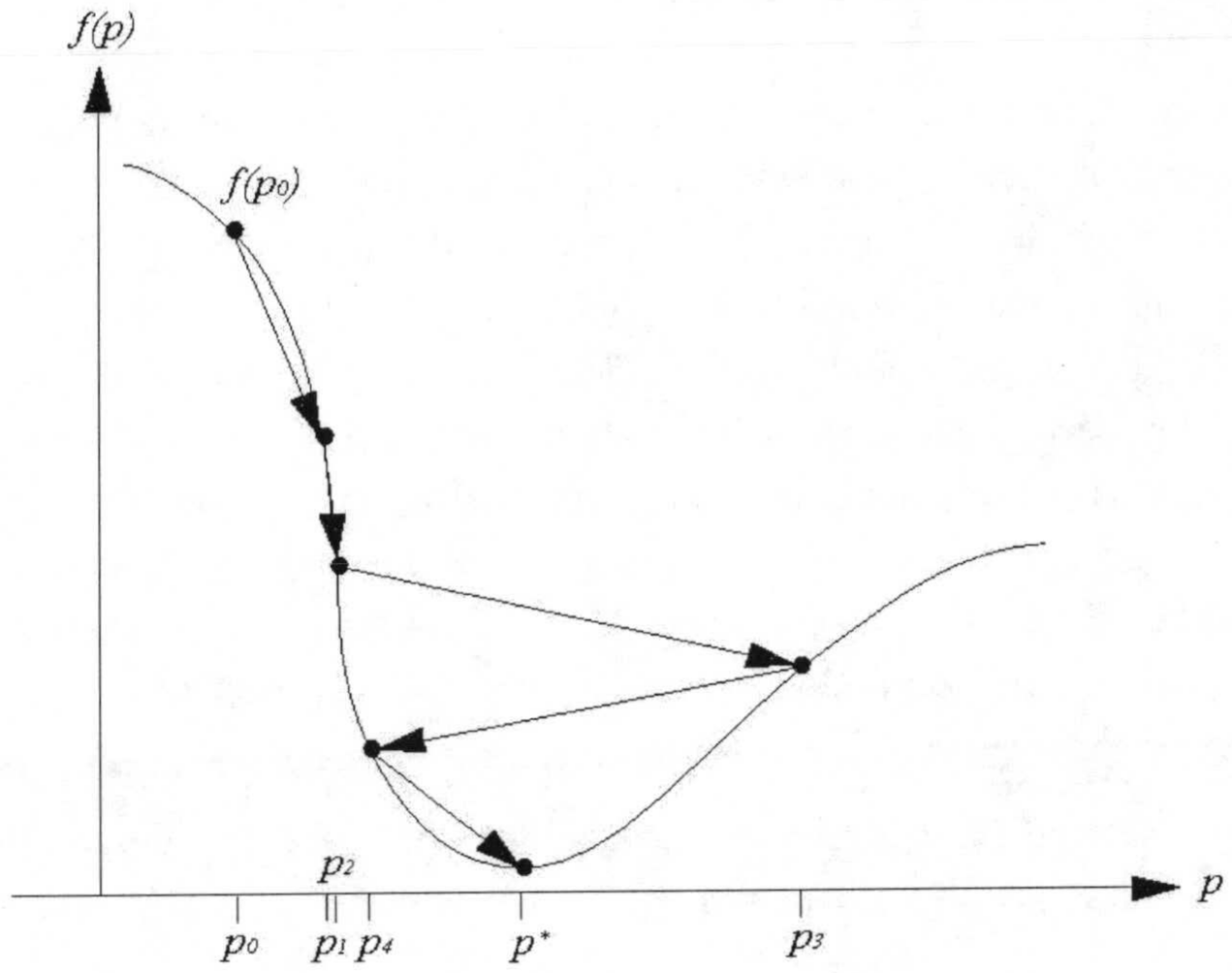

Figura 1-1: Búsqueda local realizada por un método matemático

Entre los métodos de optimización continua sin restricciones se encuentran el *método de steepest-descent*, el *método del gradiente conjugado* o el *método Levenberg-Marquardt* que se describirán a continuación.

1.1.2 Método steepest-descent

El *método steepest-descent* [Dennis, 1983], es el mas antiguo y simple de todos. Hoy en día, se suele encontrar formando parte de otros métodos, aprovechando sus características más ventajosas. También se utiliza como referencia de otras técnicas.

Su funcionamiento consiste en definir en la ecuación (1-9) el vector de descenso $\overline{d}_k$, como el opuesto al gradiente de la función *f*, es decir:

$$\overline{d}_k = -\overline{g}_k \qquad (1\text{-}10)$$

Así, cada una de las direcciones que se producen en el proceso de búsqueda es perpendicular a las anteriores.

Entre sus ventajas se tiene el hecho de que es un método bastante robusto, que es fácil de implementar y que requiere una capacidad de almacenamiento modesta. Su complejidad va a depender del coste del cálculo de la función gradiente.

En cuanto a sus inconvenientes destaca su lenta progresión al objetivo cuando se dan ciertas situaciones. Un primer ejemplo de este avance lento se produce cuando existen descensos largos y estrechos, ya que la mayoría de las veces no puede ir de forma directa al objetivo porque no encuentra una dirección perpendicular a la anterior. Al estar ligado su avance al gradiente, ocurre que cuando la configuración actual se encuentra cerca del mínimo, éste tiende a cero y el tamaño de los pasos se decrementa, de forma que cuando más cerca se esté del objetivo, mas lenta será la convergencia. Por la misma razón, habrá que tener en cuenta la condición de parada, ya que si sólo se tienen en cuenta el cambio en el error desde la iteración anterior, el algoritmo no será capaz de atravesar valles o zonas donde la función se mantenga constante. Para evitar esto una posibilidad es calcular la pendiente de regresión de las últimas n iteraciones. De este modo se podrán superar valles pequeños.

Como se ha comentado y gracias a su robustez, el método suele usarse en los primeros pasos de algoritmos de búsqueda local más potentes, ya que en estos pasos todavía la configuración se encuentra lejos del mínimo y el algoritmo avanza rápidamente.

1.1.3 Método del gradiente conjugado

El *método del gradiente conjugado* [Jacobs, 1977] fue diseñado originalmente para minimizar funciones cuadráticas convexas, pero tras ciertas variaciones se ha extendido para un funcionamiento más general.

Este método intenta solventar los problemas de avance que tenía el método steepest descent, para lo cual define el concepto de *vectores conjugados*. Así se dice que dos vectores distintos $\overline{d}_i$ y $\overline{d}_j$ son conjugados si son ortogonales con cualquier matriz (definida positiva) Hessiana Q, de modo que:

$$\overline{d}_i^T Q \overline{d}_j = 0 \qquad (1\text{-}11)$$

Esto se puede ver como una generalización de la ortogonalidad, en la que Q es la matriz unidad. Una búsqueda en la dirección $\overline{d}_i$ será perpendicular al gradiente de la función f, lo que se va manteniendo en las siguientes direcciones. La idea pues, es que ahora cada uno de los vectores de búsqueda $\overline{d}_i$, sea dependiente de todas las direcciones en las que se ha buscado para encontrar el mínimo de la función f.

El movimiento iterativo por estas direcciones nos asegura que siempre se utilizan direcciones que no se han utilizado antes por lo que la búsqueda será muy rápida. A este conjunto de direcciones de búsqueda se les llama Q ortogonales.

Los vectores gradientes conjugados son un caso especial de los vectores de dirección conjugado, donde el conjunto conjugado los generan vectores gradientes. Esta elección en el tipo de vectores tiene su base en el buen funcionamiento demostrado por los vectores gradiente en el método steepest descent, y las características de ortogonalidad demostradas por éstos.

Para una función cuadrática el primer vector se calcula igual que el método steepest descent (1-10). Los siguientes vectores conjugados se calcularían:

$$\overline{d}_{k+1} = -\overline{g}_{k+1} + \beta_k \overline{d}_k \qquad (1\text{-}12)$$

donde β_k se puede calcular como:

$$\beta_k = \frac{\overline{g}_{k+1}^T \cdot \overline{g}_{k+1}}{\overline{g}_k^T \cdot \overline{g}_k} \qquad (1\text{-}13)$$

El tamaño del paso para cada dirección vendría dado por la fórmula:

$$\lambda_k = \frac{\overline{d}_k^T \cdot \overline{g}_k}{\overline{d}_k^T \cdot Q \cdot \overline{d}_k} \qquad (1\text{-}14)$$

En teoría este conjunto de vectores puede llevar a una función cuadrática n-dimensional a su mínimo en n pasos, si no fuera por la precisión al calcular los vectores. Si la función no es cuadrática existen otras formas de calcular β_k, pero mientras menos se parezca la función, a una función cuadrática, más eficiencia se perderá, al ser más difícil encontrar vectores conjugados, y será mejor usar otro método.

1.1.4 Método de Levenberg-Marquardt

El *método de Levenberg-Marquardt* [Dennis, 1983][Marquartd, 1963] se caracteriza porque se va adaptando al contexto de la función para así conseguir acercarse rápidamente su óptimo. Para esto se parte de la utilización del método steepest descent que, como se ha comentado, basa su funcionamiento en la utilización del gradiente de la función a optimizar. Esto implica que el acercamiento al mínimo sea bastante lento, por lo que en tales circunstancias se utiliza una técnica diferente.

Para comenzar a explicar el método se parte de que se quiere aproximar un conjunto de puntos (x_i, y_i) para $i = 1, 2, \ldots, n$ mediante la función $y(x, \overline{p})$ donde $\overline{p}$ es un vector de tamaño m, que representa un conjunto de parámetros que caracterizan la

función y que son utilizados para minimizar la función de error $\aleph(\overline{p})$, que se define como:

$$\aleph(\overline{p}) = \sum_{i=1}^{n} \left(\frac{y_i - y(x_i; \overline{p})}{\sigma_i} \right)^2 \qquad (1\text{-}15)$$

donde σ_i es una medida del error (desviación estándar) en el punto i. En el caso de que no se conozca esta medida puede establecerse a 1.

Pues bien, cuando se está cerca del mínimo se espera que la función de error se aproxime a la siguiente forma cuadrática [Press, 1994], por lo que:

$$\aleph(\overline{p}) \approx \aleph(\overline{p}_i) + \overline{g}_i^T \overline{p} + \frac{1}{2} \overline{p}^T H_i \overline{p} \qquad (1\text{-}16)$$

que se obtiene a partir del desarrollo en series de Taylor de la función $\aleph$ en el punto $\overline{p}_i$ y donde $\overline{g}_i$ es el vector gradiente de $\aleph$ en el punto $\overline{p}_i$ y H_i es la matriz Hessiana de $\aleph$ en $\overline{p}_i$.

Si el error de aproximación es pequeño, se puede saltar directamente al mínimo utilizando la expresión:

$$\overline{p}_{\min} = \overline{p} - H_i^{-1} \overline{g}^i \qquad (1\text{-}17)$$

Si no se está cerca de mínimo la aproximación de (1-16) no nos serviría por lo que se utiliza el método steepest-descent para dar un paso en el sentido opuesto al gradiente utilizando la expresión (1-9).

Uno de los inconvenientes de trabajar con gradientes o con matrices Hessianas es lo incierto y lo costoso que puede llegar a ser su cálculo. En nuestro caso y puesto que se conoce exactamente la forma de la función $\aleph$, se puede calcular sin problemas tanto el gradiente como la matriz Hessiana. Concretamente esta matriz estaría formada por los siguientes elementos:

$$\frac{\partial^2 \aleph(\overline{p})}{\partial p_k \partial p_l} = 2 \sum_{i=1}^{n} \left(\frac{\partial y(x_i; \overline{p})}{\partial p_k} \frac{\partial y(x_i; \overline{p})}{\partial p_l} - (y_i - y(x_i; \overline{p})) \frac{\partial^2 y(x_i; \overline{p})}{\partial p_l \partial p_k} \right) \qquad (1\text{-}18)$$

Para simplificar la notación se definen las expresiones:

$$b_k \equiv -\frac{1}{2}\frac{\partial \aleph}{\partial p_k} \qquad a_{kl} \equiv \frac{1}{2}\frac{\partial^2 \aleph}{\partial p_k \partial p_l} \tag{1-19}$$

Con estas expresiones y haciendo $A = [a_{kl}] = 1/2\ H_i$ en (1-17) tenemos que:

$$\sum_{l=1}^{m} a_{kl}\delta p_l = b_k \tag{1-20}$$

Con esto se resuelve este conjunto de ecuaciones para calcular las variaciones δp_l que, añadidos a la aproximación actual, proporcionan la siguiente aproximación. En el contexto de mínimos cuadrados, la matriz A, que es igual a la mitad de Hessiana, se suele llamar matriz de curvatura. Así la ecuación (1-9) se rescribe como:

$$\delta p_l = \kappa \cdot b_l \tag{1-21}$$

Si se analizan los cálculos a realizar, ocurre que las segundas derivadas de la función $\aleph$ dependen de la primera y segunda derivada de la función $y(x, \overline{p})$. El problema es que el cálculo de las segundas derivadas de $y(x, \overline{p})$ es un procedimiento costoso debido a la complejidad de las ecuaciones y a su número. Además de su coste computacional, cuando el modelo se encuentra cerca del mínimo, como la segunda derivada de $y(x, \overline{p})$ va multiplicada por $(y_k - y(x_k, \overline{p}))$, que va tendiendo a cero se suelen producir cantidades aleatorias que se cancelan cuando se realiza la suma de los n ejemplos de entrenamiento, pero que debido a los errores de redondeo pueden desestabilizar el modelo, produciendo resultados no deseados. Teniendo en cuenta que la ecuación (1-20) sólo se va a usar cuando la configuración actual se encuentre cerca del mínimo para alcanzarlo en un solo paso, se puede omitir el cálculo de la segunda derivada y redefinir el cálculo de a_{jl} como:

$$a_{kl} = \sum_{i=1}^{n}\left(\frac{\partial y(x_i;\overline{p})}{\partial p_k}\frac{\partial y(x_i;\overline{p})}{\partial p_l}\right) \tag{1-22}$$

El método de Levenberg-Marquardt consiste en utilizar un método elegante para variar suavemente entre los pasos del steepest descent y el cálculo del mínimo de forma directa usando la matriz Hessiana. Marquardt siguiendo una indicación de Levenberg desarrolla este método basándose en dos ideas. La primera consiste establecer un valor para la magnitud κ en la ecuación (1-21), para lo cual analiza los componentes de la Hessiana. La cantidad calculada por la función $\aleph$ es adimensional, es sólo un número, por lo que sirve de poco para fijar el valor de κ. Por otro lado, b_j tiene dimensiones de $1/p_j$, con lo que la constante de proporcionalidad entre b_j y δp_j debe tener por tanto dimensiones de p_j^2. Revisando los componentes de la matriz A,

se puede comprobar que sólo hay una única cantidad con esas dimensiones, y que es $1/a_{jj}$, el recíproco de los elementos de su diagonal, así que esa debe ser la escala de la constante. Pero esta escala podría ser demasiado grande, así que se puede dividir por una constante adimensional λ con la posibilidad de que $\lambda >> 1$ para así poder acortar el tamaño de los pasos tanto como se desee.

Teniendo en cuenta esto la ecuación (1-21) se puede reemplazar por:

$$\delta p_l = \frac{1}{\lambda a_{ll}} b_l \qquad (1\text{-}23)$$

donde es necesario que a_{ll} sea positivo, lo que está garantizado por (1-22).

La segunda idea de Marquardt fue que las ecuaciones (1-23) y (1-20) se puedan combinar si se define una nueva matriz A' utilizando las siguientes ecuaciones:

$$a'_{jj} \equiv a_{jj}(1+\lambda) \qquad a'_{jk} \equiv a_{jk} \quad (j \neq k) \qquad (1\text{-}24)$$

y reemplazando las ecuaciones (1-23) y (1-20) por

$$\sum_{l=1}^{m} a'_{kl}\, \delta p_l = b_k \qquad (1\text{-}25)$$

Cuando λ es muy grande, la matriz A' se fuerza a ser diagonalmente dominante, así que la ecuación (1-34) tiende a ser idéntica a (1-32). Por otro lado cuando λ se aproxima a cero, la ecuación (1-34) tiende a (1-20).

1. Calcular $\aleph(\bar{p}_0)$
2. Establecer $\lambda_0 = 0.001$
3. Repetir mientras no se cumpla la condición de parada
   a. Resolver el sistema de ecuaciones (1-25)
   b. Calcular $\bar{p}_{k+1} = \bar{p}_k + \delta\bar{p}$
   c. Si $\aleph(\bar{p}_{k+1}) \geq \aleph(\bar{p}_k)$ Entonces $\lambda_{k+1} = \lambda_k \cdot 10$
      Si no $\lambda_{k+1} = \lambda_k / 10$

Algoritmo 1-2: Algoritmo de Levenberg Marquardt

En el Algoritmo 1-2 se muestran los principales pasos del algoritmo Levenberg Marquardt. Para que el algoritmo comience dando unos pasos similares a los que daría el algoritmo steepest descent se debe fijar un valor inicial de λ en torno a 0.001. Los factores de incremento y decremento, al igual que en el algoritmo anterior deben estar

cercanos al valor uno para que no se produzcan cambios bruscos en la configuración dc parada se puede pensar en parar cuando el valor de ℵ se decremente en un cantidad insignificante siempre que no se pare el algoritmo cuando λ crezca, ya que esto implicaría que todavía no se ha ajustado λ de forma óptima. Una vez parado el algoritmo se debe hacer una iteración con $\lambda = 0$ para alcanzar el mínimo definitivamente.

1.2 Redes Neuronales Artificiales

1.2.1 Introducción

Este apartado trata de describir los fundamentos y características principales de las *redes neuronales artificiales*, comenzando por una definición y conceptos preliminares sobre estas.

Una Red Neuronal Artificial (RNA) consiste en un conjunto de elementos de proceso, también conocidos como neuronas o nodos, los cuales se van a interconectar según distintos modelos.

Las RNAs suponen un nuevo modo de computación que propone un modelo de procesamiento distribuido y paralelo. En este modelo, los elementos de proceso o neuronas realizan una función muy simple, suele existir gran cantidad de éstas y están masivamente interconectadas. Estas interconexiones van a estar caracterizadas por unos pesos que regulan los envíos de señales o datos, entre neuronas.

Una RNA adquiere su habilidad o se prepara para desarrollar su cometido, aprendiendo mediante ejemplos o patrones. Este aprendizaje consiste básicamente en la adaptación de los pesos que definen las interconexiones.

Así, una RNA, va a venir definida por su arquitectura (topología y celda) y su algoritmo de aprendizaje. La computación neuronal presenta además una serie de ventajas como por ejemplo: potencial para la computación paralela masiva, robustez ante la presencia de ruido o el fallo de alguno de sus componentes, capacidad de adaptación mediante aprendizaje, etc. Todas estas ventajas implican que las redes neuronales se apliquen o se puedan aplicar a prácticamente cualquier tipo de ciencias existentes en el mundo actual.

En el siguiente aparatado se describen los fundamentos de las RNAs haciendo hincapié en las redes neuronales naturales. Después se hace un estudio del funcionamiento de las redes neuronales en cuanto a su arquitectura y algoritmos de aprendizaje. Por último se repasan los principales modelos de RNAs con la ayuda de una visión de su historia.

1.2.2 Fundamentos

1.2.2.1 Redes neuronales naturales

Los fundamentos de las RNAs se encuentran en el cerebro humano o lo que es lo mismo en las redes neuronales naturales. El cerebro humano está compuesto por un gran número de neuronas, unos 10.000.000.000, las cuales están masivamente interconectadas. Cada neurona es una celda especializada que puede propagar una señal electroquímica. Una neurona natural, Figura 1-2, consta de tres partes:

- Una estructura de entrada muy ramificada, denominándose a estas ramificaciones *dendritas*.
- Un *cuerpo*.
- Una estructura de salida, también ramificada en su parte final, denominada *axón*.

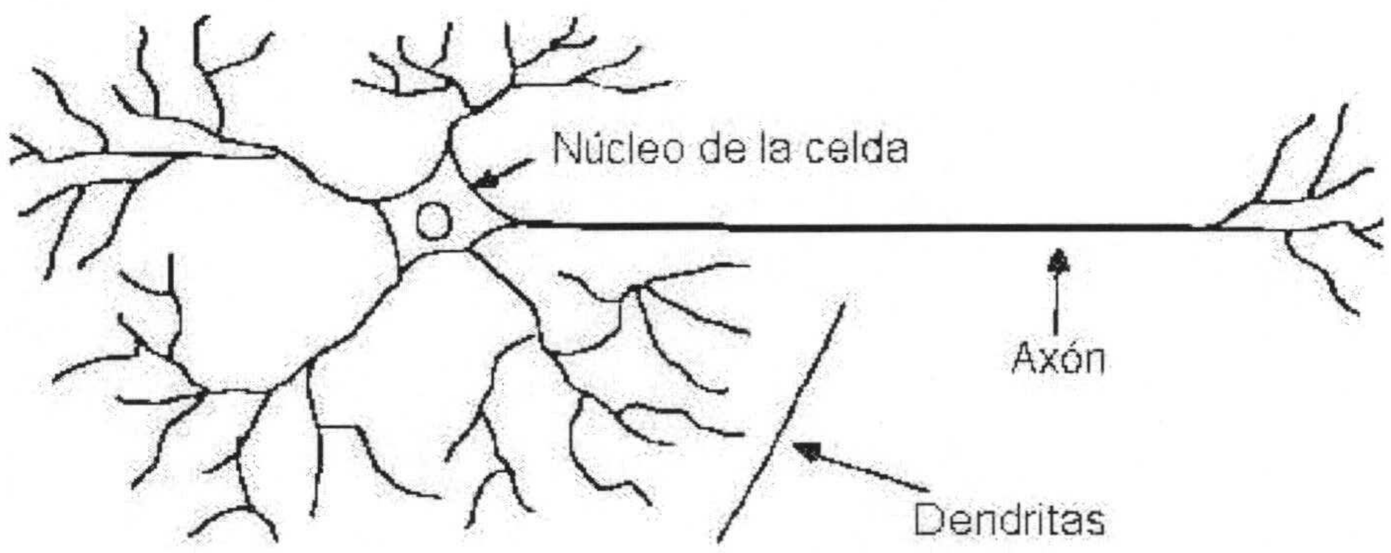

Figura 1-2: Neurona natural

El axón de una celda se conecta a las dendritas de otra mediante la sinapsis. Cuando una neurona se activa, envía una señal electro-mecánica a través del axón. Esta señal atraviesa la sinapsis de otras neuronas, y dependiendo de ciertos factores, puede a su vez activarlas. Una neurona se activa solo si el total de señal recibida en su cuerpo, por las dendritas, supera un cierto nivel, también conocido como *umbral de activación*. Por último resaltar la importancia de la sinapsis en este funcionamiento, ésta se podría describir como un espacio con neurotransmisores químicos que regularían la transmisión de la señal de una neurona a otra. De hecho, y según diversas investigaciones, durante el proceso de aprendizaje humano, se van determinando convenientemente estas sinapsis.

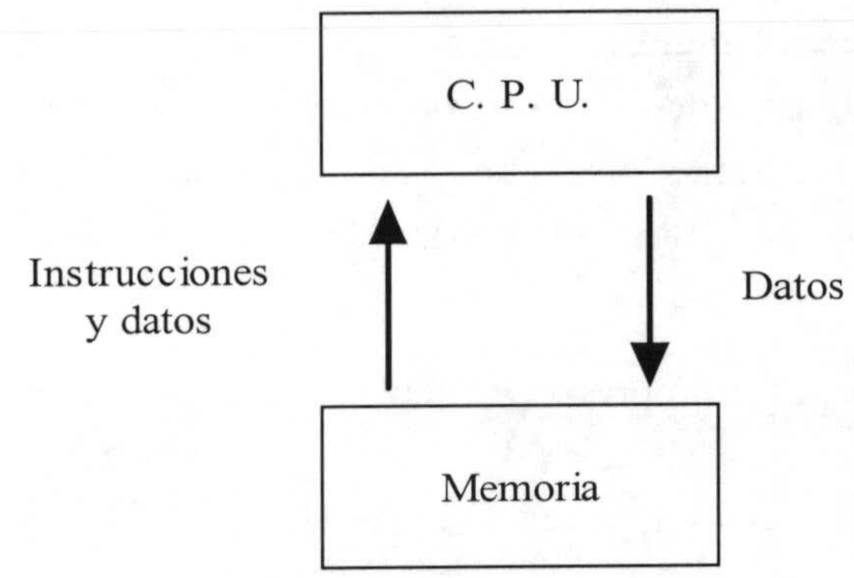

Figura 1-3: Máquina de Von Newman

1.2.2.2 Computación neuronal frente a la computación tradicional

La computación tradicional se va a identificar con aquella que realiza la conocida como máquina de Von-Newmann. Esta máquina, cuyo esquema se observa en la Figura 1-3, ejecuta repetidamente los siguientes pasos del Algoritmo 1-3.

```
1. Cargar una instrucción de memoria.
2. Cargar de memoria los datos requeridos por la
   instrucción.
3. Ejecutar la instrucción (procesar los datos).
4. Guardar los resultados en memoria.
5. Volver al paso 1.
```

Algoritmo 1-3: Funcionamiento de la máquina de Von-Newmann

Este modelo de procesamiento de información implica que las computadoras tradicionales deban ser explícitamente programadas para resolver un problema, es decir, alguien debe de haber analizado el problema en profundidad y haber definido la serie de instrucciones (programa o algoritmo) que el computador debe seguir. El problema, usualmente tiene asociados un conjunto de datos en una estructura definida, y la computación consistirá en la manipulación de los datos dirigida por el programa. Otra característica de esta computación es que hay una clara correspondencia entre las estructuras de datos tratados (números, matrices, registros,...) y el hardware de la máquina, guardándose cada uno en un bloque de memoria determinado. Además la degradación o destrucción de unas pocas localizaciones de memoria suelen provocar que el programa deje de funcionar.

La computación neuronal difiere bastante de este tipo de computación. Desde el punto vista del procesamiento, se cuenta con un gran número de elementos de proceso, masivamente interconectados que trabajan en paralelo. A la hora de resolver problemas, no se requiere una secuencia explícita de pasos a seguir, ahora el usuario

debe permitir que la red adapte, por si misma, durante un periodo de aprendizaje. Este aprendizaje está basado en patrones, de modo que éstos, se le van pasando a la red, y los pesos de las interconexiones se van ajustando. Después del suficiente entrenamiento, la red está preparada para ofrecer soluciones viables a nuevos problemas.

Este tipo de computación presenta además como característica una robustez ante ruido en las entradas o ante ciertos fallos de hardware, de modo que estos factores afectan poco o muy poco al conjunto de los resultados.

Todo esto implica también cierta diferencia en los tipos problemas que resuelven mejor un tipo de computación u otra. A groso modo, la computación tradicional está especialmente indicada en problemas donde la solución se consigue ejecutando una serie de pasos, aquellos que tienen una solución matemática clara. Por otro lado, será más idóneo usar computación neuronal en problemas que no están bien definidos, donde existen ambigüedades, y gran cantidad de información, por ejemplo en el reconocimiento de patrones.

1.2.3 Arquitectura

En este apartado se detalla el funcionamiento de la red neuronal como sistema de computación, describiendo para esto su arquitectura. Esta viene definida por su topología y la función que caracteriza a su elemento de proceso o neurona.

1.2.3.1 La neurona artificial

La neurona artificial es el elemento de proceso de este sistema de computación y en una red podemos encontrar cientos o miles de éstas. Una neurona o nodo, Figura 1-4, va a constar de una serie de entradas (las cuales suelen tener asociado un peso), una salida y viene caracterizada por una función de activación.

Una neurona recibe datos o señales por sus entradas, las cuales pueden estar conectadas a las entradas del sistema o a otras neuronas. Estos datos suelen ser multiplicados por los pesos correspondientes, para emular el efecto de la sinapsis en las neuronas naturales. Así:

$$\eta = \sum_{i=1}^{n} w_i x_i \qquad (1\text{-}26)$$

donde x_i es la entrada y w_i el peso. Partiendo de estos datos y dependiendo de la función de activación se obtendrá un valor u otro en la salida.

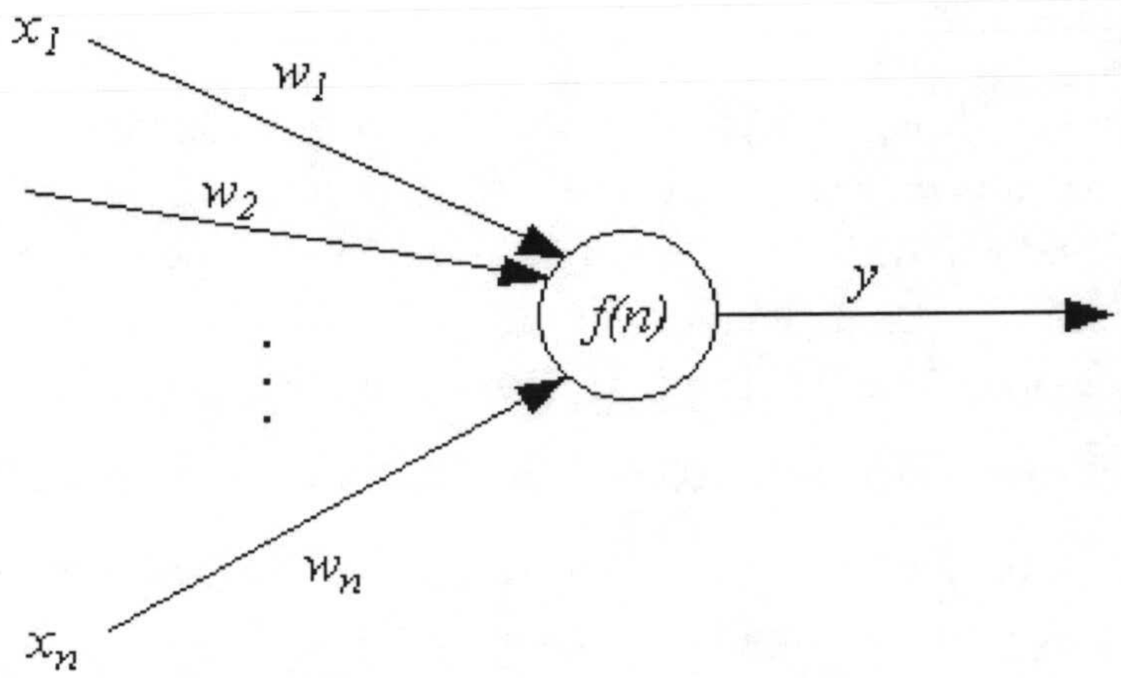

Figura 1-4: Neurona artificial

Existen funciones de activación con diferentes formas. Inicialmente se pude partir de una función básica que sólo tome dos valores en función de si se supera o no un umbral por parte de las entradas. Una de las funciones de activación más clásicas fue propuesta por (McCulloch, 1943) y su salida será igual a:

$$f(\eta) = \begin{cases} 1 & si \quad \eta > \theta \\ -1 & si \quad \eta < \theta \end{cases} \qquad (1\text{-}27)$$

Denominándose a esta función paso (*step-function* ó *hard-limiter*), siendo θ el umbral de la función de activación.

Otra de las funciones de activación más conocida es la sigmoidal, la cual emula bastante bien el comportamiento de las neuronas naturales y utiliza la siguiente fórmula:

$$f(\eta) = \frac{1}{1 + e^{-(\eta - \theta)}} \qquad (1\text{-}28)$$

Las redes utilizadas en este trabajo usan como función de activación una gausiana, cuya salida es la típica campana de Gauss. En este caso el concepto de función de activación cambia un poco, pues la salida de está depende de un centro c y un radio σ.

$$f(x) \quad = \quad e^{\{-(|x-c|/\sigma)^2\}} \qquad (1\text{-}29)$$

En este caso tanto x, como c, pueden tener varias dimensiones y $|\cdot|$ es la norma euclídea. El funcionamiento de la red completa se describe en el capítulo 2.

1.2.3.2 Topología

Las neuronas de una RNA suelen estar masivamente interconectadas y se suelen agrupar en capas que indican su situación con respecto a las entradas o salidas de la RNA. Así las neuronas de la capa de entrada o las neuronas de la capa de salida tienen una conexión directa con la entrada o salida a la red, mientras que el resto de las neuronas se engloban dentro de la que se denominan capa ocultas.

En cuanto a su modo de interconexión se distinguen dos tipos principales.

- *Redes hacia delante* (*feed-forward*): donde los datos o señales en la red viajan en un solo sentido desde las entradas, hasta la salida. Es decir, no existen ciclos y la salida de una capa no afecta a ella misma. Son ampliamente usadas en el reconocimiento de patrones.

- *Redes hacia atrás* (*feed-back*): en este caso los datos o señales viajan en ambas direcciones, existiendo bucles en la topología. Pueden llegar a ser muy poderosas, pero también son más complicadas, dinámicas y cambiantes en cuanto a su punto de equilibrio.

El número de neuronas y capas que se necesitan dado un determinado problema dependen tanto del problema como del modelo de RNA que se esté utilizando.

1.2.4 Aprendizaje

Tal y como se ha comentado, mediante el aprendizaje, la RNA adquiere su habilidad para desarrollar su labor. El aprendizaje en RNAs se realiza mediante ejemplos. Durante esta fase, también llamada entrenamiento, se ajustan los pesos de interconexión entre neuronas, para cada uno de los patrones del conjunto de entrenamiento, los cuales se dice que se van memorizando.

Básicamente el aprendizaje se pude dividir en tres tipos:

- *Aprendizaje supervisado* (*Supervised learning*): se basa en la comparación directa entre la salida actual de la red y la salida correcta (valor objetivo) para un determinado patrón. Esta diferencia se formula muchas veces en términos de minimización del error, por lo que se utilizan algoritmos de optimización basados en gradiente descendiente para ajustar iterativamente los pesos de la red. Este tipo de entrenamiento es el más usado de los tres. Un caso particular sería *Entrenamiento reforzado* (*Reinforcement learning*) donde mediante ciertas heurísticas se refuerzan ciertos comportamientos o salidas de la red.

- *Entrenamiento no supervisado* (*Unsupervised learning*): en este caso no existe un valor objetivo que sirva de referencia para comparar la salida de la red, sino que la adaptación de la red se produce solamente utilizando los patrones de entrada. A este tipo de entrenamiento también se le suele denominar *entrenamiento competitivo* (*competitive learning*), cuando se suelen considerar neuronas más importantes a aquellas que tienen más peso.

Conceptos implicados en el entrenamiento supervisado de una red son los de *generalización*, *sobre-entrenamiento* (*overfitting*) y *bajo-entrenamiento* (*underfitting*). Durante este entrenamiento las salidas de una red tienden a aproximarse a los valores objetivo de los patrones del conjunto de entrenamiento. El objetivo final de este comportamiento será conseguir la que la red generalice, es decir, que la red obtenga salidas próximas a valores objetivo para patrones que no pertenecen al conjunto de entrenamiento. Esta generalización no es siempre posible y para que se produzca se necesita tener cierto conocimiento a priori sobre la aplicación [Wolpert, 1995]. Por ejemplo, es necesario escoger un conjunto de patrones que sea relevante para que la red aprenda o que el conjunto de entrenamiento tenga cierta relación con el conjunto de patrones a generalizar.

Dos de los problemas con los que se encuentra la generalización son el *bajo-entrenamiento* y el *sobre-entrenamiento* [Geman, 1992). Así una red que no es suficientemente compleja puede fallar cuando se le pasa un conjunto de datos complicado dando lugar a bajo-entrenamiento. Una red que es muy compleja puede llegar incluso a quedarse con ruido asociado a los patrones dando lugar a sobre-entrenamiento. El sobre-entrenamiento es especialmente peligroso, en ciertas redes, ya que puede conducir a predicciones que están lejos del rango del conjunto de entrenamiento incluso aunque no haya ruido asociado a los patrones.

1.2.5 Principales modelos de RNAs

Un modelo de RNA nos va a determinar todas las características de esta red, desde su arquitectura, topología y neuronas, hasta su algoritmo de aprendizaje. En este apartado se comenzará haciendo un repaso por la historia de los principales modelos de RNAs, para después describir los más importantes.

1.2.5.1 Historia

Los primeros desarrollos en RNAs se producen a principios de los años 40 y fueron llevados a cabo por [McCulloch, 1943]. Estas simulaciones utilizaban lógicas formales y estaban basados en simples neuronas que eran consideradas como

dispositivos binarios con umbrales fijos. Los resultados eran funciones lógicas simples (OR, AND,…)

En la década de los 60 cuando nacen los principales modelos, que más tarde servirían para el desarrollo y progreso de las RNAs actuales. Así, [Rosenblatt, 1962] despertó gran interés con el diseño y funcionamiento de su *perceptron* y su aplicación para la clasificación de patrones, ya que el sistema podía aprender a conectar o asociar un entrada a una salida. Un poco después se desarrolla ADALINE (*ADAptive Linear Element*), por [Widrow, 1960]. ADALINE era un sistema con una arquitectura muy similar al perceptron, se implementó con componentes electrónicos, pero utilizaba como algoritmo de entrenamiento el conocido algoritmo LMS (*Least Mean Square*).

Sin embargo a finales de esta década se publica un libro [Minsky, 1969] en donde generalizaban las limitaciones que podía tener la expansión del perceptron de simple capa al perceptron multicapa. Esto dio lugar a que se perdiera el interés que habían despertado las RNAs.

A pesar de todo, los trabajos en el área continuaron y Grossberg desarrolló su Teoría de Resonancia Adaptativa (*Adaptative Resonante Theory*, ART), que contemplaba nuevas hipótesis sobre los principios que gobiernan los sistemas neuronales biológicos [Grossberrg, 1976]. Estas ideas sirvieron para que Carpenter y Grossberg dieran lugar a tres clases de arquitecturas de RNAs: ART 1, ART 2, y ART 3 [Carpenter 1983, 1987, 1990]. Estas eran implementaciones neuronales auto-organizativas de *algoritmos de agrupamiento de patrones* (*pattern clustering algrithms*). Otra importante teoría de fue la aportada por Kohonen y su trabajo sobre *mapas auto-organizativos* (*self-organizing maps*, SOM) [Kohonen, 1984] basados en sus propios trabajos *Learning Vector Quantization* [Kohonen, 1988].

Paul Werbos es uno de los pioneros en el desarrollo a mediados de los 70 del algoritmo de aprendizaje *back-propagation* (Werbos, 1974), sin embargo pasaron algunos años hasta que este famoso método, para el aprendizaje de redes perceptrones multicapa, se popularizara.

A principios de los 80, Hopfield y otros, basándose en trabajos anteriores de Hebb (Hebb, 1949), utilizaron en ciertas reglas de aprendizaje para un tipo de redes recurrentes hoy llamadas, *modelos de Hopfield* [Hopfield, 1982,1984]. Más tarde estos trabajos también servirían de base para la máquina de Boltzmann, que, entre otras utiliza técnicas como el *simulated annealing*.

El tipo de redes que se utilizarán en este trabajo son las *redes de funciones de base radial*. Los estudios más importantes sobre estas redes, aparecen en los 80 (Broomhead, 1988). Sin embargo, la idea básica fue desarrollada en los años 30 con el nombre de método de funciones potenciales. En el capítulo 2 se realiza una descripción en profundidad sobre estas redes.

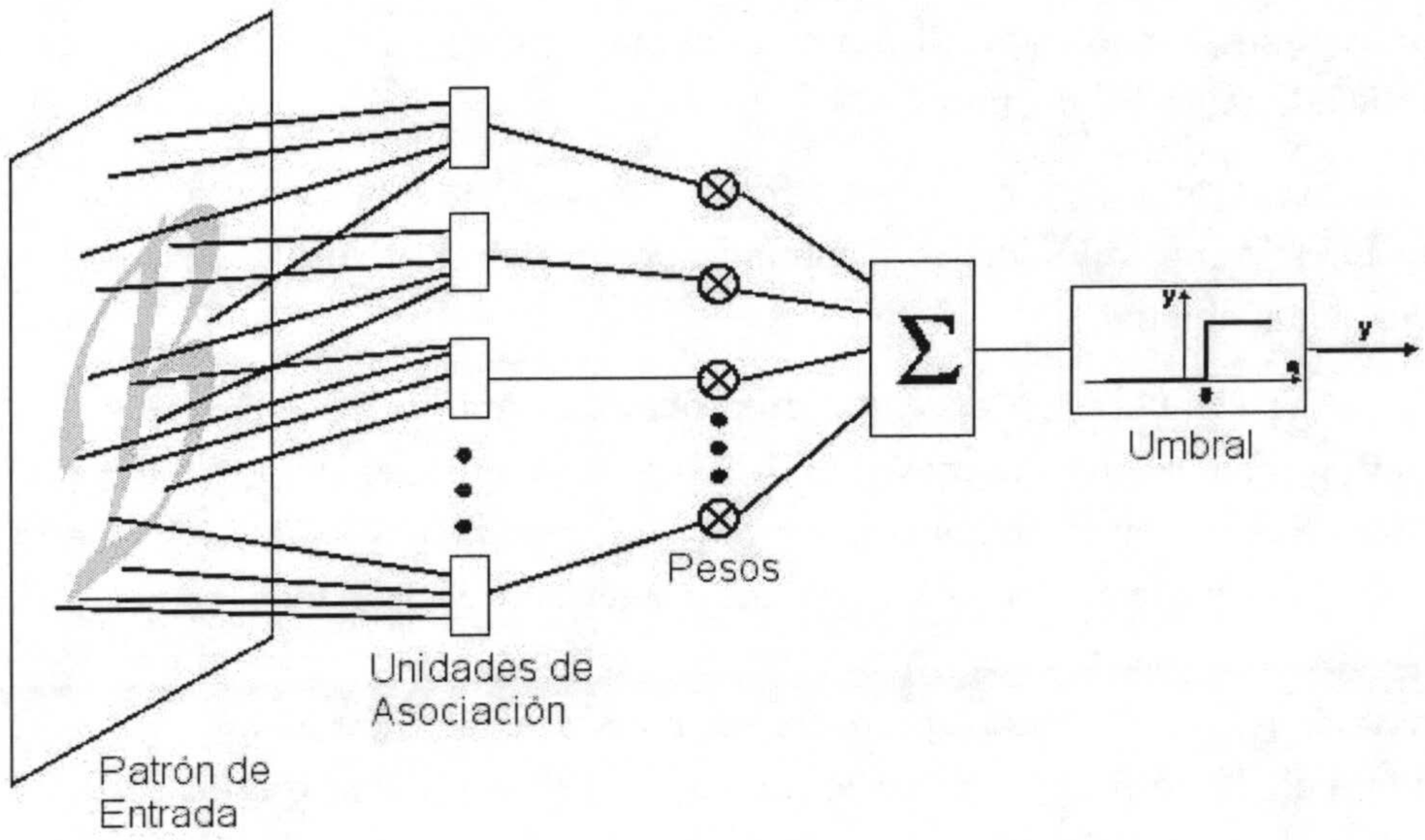

Figura 1-5: Perceptron original.

1.2.5.2 Perceptron y Adaline

Uno de los trabajos más importantes en RNAs por su repercusión posterior fue el presentado por Frank Rosenblat a principios de los años 60 [Rosenblatt, 1962]. El *perceptron*, Figura 1-5, es una red de simple capa, con tan solo una neurona. Las entradas están preprocesadas por unidades de asociación (A-units) las cuales, en principio, tienen una funcionalidad booleana arbitraria pero fija, es decir no aprenden. La neurona computa la suma ponderada por los pesos de las entradas, y su salida es de tipo umbral [+1,-1], en función de que la suma ponderada supere o no, un umbral. El perceptron es usado como clasificador, y sus propiedades han sido ampliamente estudiadas [Widrow, 1990].

1. Inicializar pesos y elegir umbral θ
2. Presentar un nuevo patrón (entrada-salida objetivo)
3. Calcular la salida actual $y(t)$
4. Adaptar pesos

$$w_i(t+1) = w_i(t) + \alpha(d(t) - y(t))x_i(t)$$

5. Volver al paso 2

Algoritmo 1-4: Algoritmo del perceptron

Los pesos y el umbral en un perceptron pueden ser adaptados utilizando diversos algoritmos de aprendizaje. La regla de entrenamiento propuesta del perceptron propuesta por Rosenblatt es muy simple y contempla los siguientes pasos reflejados en el Algoritmo 1-4.

En este algoritmo α es un factor de ganancia en el intervalo [0, 1] que regula factores como velocidad de aprendizaje y estabilidad de la red. $d(t)$ es la salida deseada u objetivo.

Una importante generalización del algoritmo de entrenamiento del perceptron fue presentada por [Widrow, 1960] como el algoritmo de aprendizaje LMS (Least Mean Square) también conocido como *algoritmo de aprendizaje de la regla delta*. La diferencia principal con la regla del perceptron es la manera en que la salida del sistema es usada en el entrenamiento. Así la regla del perceptron de una función umbral (-1 o +1) para aprender, mientras que la regla delta usa una salida más continua. La fórmula de actualización de pesos en su forma original es:

$$w_i(t+1) = w_i(t) + \frac{\alpha(d(t) - y(t))x_i(t)}{\left|x_i(t)\right|^2} \qquad (1\text{-}30)$$

donde al igual que antes t indica el ciclo de actualización; $w(t+1)$ es el próximo valor del vector de pesos, y $w(t)$ es el valor actual del vector de pesos; $x(t)$ es el vector patrón de entrada actual; $d(t)$ es la salida objetivo; $y(t)$ es la salida actual; α es un factor de adaptación cuyo valor se recomienda que pertenezca al intervalo [0, 1] y $|\cdot|$ es la operación módulo de un vector.

Por lo tanto, el error es reducido en un factor de α, adaptándose los pesos para una entrada dada. Cada vez que se presenta una nueva entrada se realiza esta adaptación. α regula la estabilidad y velocidad de convergencia [Widrow, 1985]. Para vectores de entrada independientes del tiempo, la estabilidad se asegura si $\alpha \in [0, 2]$, pero si $\alpha > 1$ puede haber sobre-correcciones del error por lo que se recomienda que $\alpha \in [0, 1]$.

Esta regla fue aplicada al *elemento lineal adaptativo*, conocido como ADALINE, desarrollado por [Widrow, 1960], cuya arquitectura de red es muy similar al perceptron. Debido a la simplicidad de la red tratada existen ciertas restricciones sobre el tipo de aplicaciones a las que se puede aplicar

1.2.5.3 Perceptron multicapa

El *perceptron multicapa* es una red hacia delante (feed-forward) con una o más capas de nodos entre la entrada y la salida. Estas capas adicionales contienen neuronas ocultas que no están directamente conectadas ni con la entrada de la red ni con la

salida. Este tipo de redes supera muchas de las limitaciones de las redes de tipo perceptron, utilizandose mucho en clasificación. Según [Lippmann, 1987} dependiendo del número de neuronas y capas de estas redes se definen las distintas regiones de clasificación, demostrándose además que no hacen falta más de tres capas para conseguir prácticamente cualquier tipo de región.

1. Inicializar pesos y umbrales a valores aleatorios pequeños.
2. Presentar patrón. Entradas $x_1,\ldots,x_n$.
   Salida objetivo $d_1, \ldots, d_m$.
3. Calcular salidas actuales $y_1, \ldots, y_m$.
4. Adaptación de pesos
   a. Ajustar los pesos con la ecuación
      $$w_{ij}(t+1) = w_{ij}(t) + \alpha\delta_j x_i$$
   b. Si la neurona j es de salida entonces
      $$\delta_j = y_j(1-y_j)(d_j - y_j)$$
   c. Si la neurona es interna
      $$\delta_j = x_j(1-x_j)\sum_k \delta_k w_{jk}$$
5. Volver a 2

Algoritmo 1-5: Algoritmo Back-Propagation simplificado

Para entrenar este tipo de redes, cuando la función de activación es de tipo sigmoidal, se utiliza el algoritmo de entrenamiento *back-propagation* [Rumelhart, 1986]. Este algoritmo es una generalización del algoritmo LMS. Básicamente es una técnica de gradiente, diseñado para minimizar el error cuadrático medio entre la salida actual de la red y la salida deseada u objetivo. De esta forma los pesos se van ajustando iterativamente para cada patrón, siendo un componente esencial del algoritmo la propagación de la información del error requerida para adaptar los pesos hacia atrás, desde las neuronas de la capa de salida, hasta la neurona de la capa de entrada. La presentación de patrones se repite hasta que los pesos convergen y la función costo es reducida hasta un valor aceptable.

1.2.5.4 Mapas auto-organizativos de Kohonen

Existe un principio de organización sensorial en el cerebro que coloca las neuronas en un cierto orden, reflejando las características físicas de los estímulos externos que están siendo percibidos. El *algoritmo de mapas auto-organizativos* de Kohonen [Kohonen, 1984] emula este comportamiento cerebral.

Así, este algoritmo, crea un vector cuantizador ajustando pesos desde nodos de entrada a nodos de salida dispuestos en una parrilla bidimensional. Los nodos de salida están ampliamente interconectados. Los vectores de entrada de valores continuos, se van presentando sin especificar salidas objetivo. Tras este proceso, los pesos especificarán centros de vectores o agrupamientos que representan el espacio de entrada y además se organizarán de manera que neuronas que están topológicamente cerca serán más sensitivas a entradas que están físicamente de igual manera.

1. Inicializar pesos a valores aleatorios pequeños
2. Presentar una nueva entrada
3. Calcular la distancia d_j entre la entrada y cada neurona
   j de salida usando
   $$d_i = \sum_i (x_i(t) - w_{ij}(t))^2$$
4. Seleccionar la neurona j más cercana
5. Actualizar los peso de la neurona j y su vecindad
   $$w_{ij}(t+1) = w_{ij}(t) + \alpha(t)(x_i(t) - w_{ij}(t))$$
6. Volver al paso 2.

Algoritmo 1-6: Algoritmo para mapas auto-organizativos.

Una de las características importantes de este algoritmo es que se define una vecindad para cada nodo. La vecindad para un determinado nodo, que indica el conjunto de nodos que están a una determinada distancia de ese nodo, va disminuyendo con el tiempo. El Algoritmo 1-6 muestra el algoritmo de aprendizaje para los mapas auto-organizativos de Kohonen. En este algoritmo $x_i(t)$ es la entrada i en el tiempo t y $\alpha(t)$ es un factor de ganancia que varía con el tiempo.

1.3 Computación evolutiva

Los principios fundamentales de los modelos de computación evolutiva se encuentran en la obra de Charles Darwin "El origen de las especies" que escribe en 1859. En este trabajo se introducen conceptos como individuo de una especie, nacimiento y desaparición de éste, lucha por la supervivencia, selección natural o herencia de un individuo a sus descendientes.

Todos estos conceptos se implementan a la hora de resolver un problema mediante computación evolutiva [Bäck, 1996, 1997]. Para esto, se mantiene una población finita de individuos que luchan por sobrevivir, donde cada individuo suele

representar una solución al problema a resolver. La supervivencia de un individuo va a depender dc cómo de buena sea la solución que este simboliza, para resolver el problema. Esto se representa asociando a cada individuo un valor de adaptación (fitness), que se calcula mediante una función de evaluación. En función de estos valores se suelen ordenar los individuos, para posteriormente seleccionar un conjunto de estos. La obtención de descendientes de la población se va a conseguir aplicando a los miembros del conjunto, anteriormente citado, una serie de operadores. Para finalizar de entre los antiguos y nuevos miembros de la población se escogerán los miembros de la siguiente generación.

El uso de modelos de computación evolutiva tiene una serie de ventajas sobre otros métodos de resolución de problemas. En principio, se pueden aplicar cuando se tiene un conocimiento limitado sobre el problema a resolver. Por ejemplo, al contrario que otros métodos y a la hora de optimizar funciones no es necesario conocer de éstas, características como derivadas, discontinuidades, etc. Lo único que tenemos que poseer es una forma de medir para cada individuo, lo que se ha definido como valor de adaptación, o como de buena es la solución al problema que éste representa.

La segunda ventaja es que la computación evolutiva es menos susceptible de quedar atrapada en mínimos locales. Esto se debe a que esta computación mantiene una solución de poblaciones alternativas y un balance entre la explotación de regiones del espacio de búsqueda, donde se han establecido ya individuos, y la exploración de nuevas regiones.

La tercera ventaja es que la computación evolutiva puede ser aplicada en el contexto del ruido o de funciones objetivo no estacionarias. Esta ventaja hace que este modelo de computación se pueda aplicar a la resolución de problemas en un amplio rango de dominios, particularmente cuando el objetivo es construir un sistema que exhiba características bioinspiradas como inteligencia o adaptación a un contexto cambiante.

El resto del apartado se estructura de la siguiente manera. A continuación se describirán los paradigmas que históricamente han definido el trabajo dentro del campo de la computación evolutiva. Después, se aborda la resolución de problemas de optimización, mediante técnicas evolutivas. Con estas premisas se pasa a describir cual es la estructura de un algoritmo evolutivo, para después analizar cuales son los elementos en el diseño de un algoritmo de este tipo.

1.3.1 Orígenes y primeros desarrollos

La mayoría de las actuales implementaciones de algoritmos evolutivos descienden de tres desarrollos independientes pero relacionados: *algoritmos genéticos*, *programación evolutiva* y *estrategias de evolución*.

Los *algoritmos genéticos* fueron introducidos por [Holland, 1975] y posteriormente estudiados por [De Jong, 1975] y [Goldberg, 1978]. Originalmente fueron propuestos como un modelo general para procesos adaptativos, aunque después, sus aplicaciones más importantes se han desarrollado en el campo de la optimización. Para lograr esto, los algoritmos genéticos modelan el proceso evolutivo, ya esbozado anteriormente, a nivel del genoma. Antes de que un algoritmo genético se use para resolver problemas, es necesario definir un código genético (representación de los individuos mediante cadenas de ceros y unos) y una relación entre el código genético y las soluciones del problema. Usando terminología biológica, al código genético de un individuo se le denomina *genotipo* y a la instanciación de este código, es decir a la solución que este representa, se le denomina *fenotipo*.

La *programación evolutiva*, introducida por [Fogel, 1962], fue originalmente concebida como un intento de crear inteligencia artificial. Esta aproximación consistía en desarrollar máquinas de estados finitos para predecir eventos a partir de observaciones iniciales. Este tipo de máquinas abstractas transforman una secuencia de símbolos de entrada en una secuencia de símbolos de salida. Esta transformación depende de un conjunto finito de estados y de un conjunto finito de reglas de transformación entre estados. La eficiencia de una máquina de estados finita con respecto a su entorno puede medirse como la capacidad de predicción alcanzada.

Las *estrategias de evolución* fueron desarrolladas por [Rechenberg, 1973] y [Schwefel, 1975]. Inicialmente fueron diseñadas con el objetivo de resolver problemas de optimización de parámetros. En este caso también se modela un proceso evolutivo, pero al contrario de los algoritmos genéticos que realizaban esta modelización a nivel del genotipo, las estrategias de evolución trabajan a nivel del fenotipo debido a los problemas para las que fueron concebidas.

1.3.2 Resolución del problema de optimización

En algunas situaciones del mundo real la función a optimizar f y las restricciones c_j no pueden ser tratadas analíticamente porque no se cumplan unas precondiciones (continuidad, diferenciabilidad, etc), por ejemplo si la definición de la función está basada en un modelo de simulación.

La aproximación tradicional en estos casos es desarrollar un modelo formal, que emule lo mejor posible las funciones originales, aplicándose más tarde métodos tradicionales como *programación lineal* o *no-lineal*. Esto conlleva, a que a menudo, se tenga que simplificar la formulación del problema original por lo que uno de los principales aspectos de la programación matemática será el diseño del modelo formal.

Esta aproximación, se ha demostrado satisfactoria en muchas aplicaciones, pero tiene ciertos problemas, sobre todo debido a las simplificaciones que se deben realizar, lo que implica que, muchas veces, las soluciones no resuelvan el problema original, llegando a considerarse muchos problemas irresolubles. Es por esto, que se buscan nuevas aproximaciones donde la computación evolutiva es una de las más prometedoras.

El diseño de un algoritmo evolutivo que resuelva un problema de optimización concreto, se puede considerar como un problema de búsqueda en un espacio, donde cada punto representa una solución. Así, dado algún criterio de calidad, como complejidad, eficiencia, etc., el nivel en que todas las soluciones, cumplan este criterio formará una superficie en el espacio. El diseño de una solución óptima es equivalente a buscar el punto más alto (bajo) en esta superficie.

Una de las principales diferencias de la computación evolutiva es que el método empleado se debe adaptar al problema en sí, tal y como se mostrará en posteriores apartados. Pero incluso en el caso de que mediante las técnicas tradicionales de optimización, ésta sea imposible, mediante computación evolutiva se puede llegar a mejorar el conjunto de mejores soluciones conocidas lo que supone un gran éxito para problemas prácticos.

Otra opción es usar algoritmos híbridos: combinaciones de búsqueda evolutiva y tradicional como en las técnicas de búsqueda basadas en el conocimiento [Davis, 1991][Renders, 1996].

1.3.3 Estructura de un algoritmo evolutivo

Los algoritmos evolutivos emulan el proceso de la evolución natural, o proceso que conduce la emergencia de estructuras orgánicas complejas y adaptadas al entorno. De forma resumida y simplificada, la evolución es el resultado de la interrelación entre la creación información genética nueva, su evaluación y selección.

Un individuo de la población se relaciona con otros individuos de la población (por ejemplo al competir por la comida, mediante depredación o cruce para reproducirse) además de con su entorno (por ejemplo con factores como la comida o el clima). Mientras más grande sea la adaptación bajo esas condiciones, mayor será la probabilidad de que sobreviva durante más tiempo y de que genere descendientes que

hereden parte de su información genética. A lo largo de la evolución esto conduce a una penetración de individuos que tengan una información genética cuyo valor de adaptación sea mayor. La naturaleza no determinística de la reproducción conduce a una permanente producción de información genética nueva y por lo tanto a la creación de descendientes diferentes.

Este modelo neo-Darwiniano de evolución orgánica se refleja en la estructura del Algoritmo 1-7. En este algoritmo $G(t)$, es una población de μ individuos en la generación t. Esta población será primeramente inicializada, siguiendo algún, método más o menos aleatorio.

```
1. t = 0.
2. Inicializar G(t)
3. Evaluar G(t)
4. Mientras no se cumpla la condición de fin
    a. P(t) = Seleccionar individuos para reproducción
                                                    {G(t)}
    b. D(t) = Aplicar operadores {P(t)}
    c. Evaluar descendientes {D(t)}
    c. G(t+1) = Reemplazar [G(t) ∪ D(t)]
    d. t = t + 1
```

Algoritmo 1-7: Algoritmo evolutivo general

Después se evalúan los individuos de esta población, calculando su valor de adaptación a partir de una función de evaluación $f(x)$, para cada una de las soluciones x, representadas por cada uno de los individuos.

Posteriormente se entra en el bucle evolutivo principal del algoritmo. Este comienza, con la creación del conjunto de individuos $P(t)$, a los cuales se les va a aplicar un operador de reproducción. La pertenencia de un individuo a este conjunto se suele determinar en función del valor de adaptación de éste.

El número de individuos de $D(t)$ es variable y puede ser igual al de $P(t)$. $D(t)$ representa la población de descendientes, su tamaño es λ, y se genera mediante operadores que actúan sobre la información genética. Algunos de los operadores más conocidos son el del *cruce* o el de *mutación* que se aplicarán sobre individuos de la población $P(t)$.

Se pasará después a evaluar a los miembros del conjunto $D(t)$. Una vez determinado el valor de adaptación de estos individuos, se aplica una estrategia de reemplazo, que consistirá en sustituir algunos miembros del conjunto $G(t)$ por algunos

de los miembros del conjunto *D(t)*. Este reemplazo se suele realizar utilizando criterios como el valor de adaptación de los individuos u otros más o menos aleatorios.

1.3.4 Diseño de un algoritmo evolutivo

Tal y como se ha mencionado existen tres tipos principales de algoritmos evolutivos que son: *algoritmos genéticos*, *programación evolutiva*, y *estrategias de evolución*. Sin embargo y a partir de estas tres clases canónicas de algoritmos evolutivos, se pueden derivar muchas variantes. Las principales diferencias estriban en:

- La representación de los individuos.
- El diseño de operadores de reproducción.
- Los mecanismos de selección y reemplazo.

Estas diferencias serán también los elementos clave a la hora de diseñar un algoritmo evolutivo. Así, a la hora de resolver un problema mediante computación evolutiva una de las tareas principales será elegir un esquema de representación para los individuos. Este esquema se diseñará en función del problema e influirá en los operadores de reproducción que más tarde se diseñarán. La importancia de estos operadores reside en el hecho de que serán los encargados de generar nuevos individuos o soluciones al problema dado. Por último resaltar también el papel de los mecanismos de selección y reemplazo dentro de la evolución de una población. De estos mecanismos, va a depender temas más importantes como la elección del subconjunto de padres o el mantenimiento de la diversidad de la población, y por lo tanto de su capacidad de búsqueda de nuevas soluciones.

Es necesario resaltar que los requerimientos para el diseño de aplicaciones evolutivas son modestos comparados con otras técnicas de búsqueda, siendo esta una de las principales ventajas de la computación evolutiva y una de las razones por la que se ha hecho tan popular.

1.3.4.1 Inicialización de la población inicial

La creación de la población inicial se puede generar de forma aleatoria o mediante heurísticas, que van a situar a los individuos iniciales "mejor colocados", dentro del espacio de búsqueda.

En el caso de los algoritmos genéticos, existen estudios [Michalewicz, 1990] que aseguran que, partiendo de cualquier inicialización, usan una estrategia casi óptima para alcanzar una solución en un tiempo finito. Sin embargo es evidentemente

que una buena inicialización, en la que se utiliza conocimiento a priori del problema, ahorrará bastantes generaciones en el proceso de búsqueda o podrá disminuir el tamaño de la población. En conclusión se obtendrá una solución final con un menor coste de computación y tiempo.

Por otro lado, la aplicación de este tipo de heurísticas para la creación de la población inicial, debe realizarse con cierta precaución. Así, será necesario vigilar que ésta sea suficientemente diversa, o de lo contrario será fácil quedar atrapado en un óptimo local.

1.3.4.2 Condición de parada

El criterio de parada establece cuando se da por concluido el proceso de búsqueda. Un criterio de parada ideal sería aquel que estudia la evolución de una población y detecta cuando se ha alcanzado una solución óptima. Una vez alcanzado este estado, no tiene sentido continuar la búsqueda, ya que no se va a mejorar la adaptación de la población. Este momento, es pues el adecuado para detener el proceso y ofrecer la solución más óptima encontrada.

Entre las condiciones de parada más usadas se encuentran:

Criterio del número máximo de generaciones:

Este criterio termina el proceso de búsqueda cuando se han completado v generaciones. Dado un algoritmo evolutivo, diseñado para resolver un problema determinado, el valor de v se puede determinar de una manera empírica, de forma que pueda usarse en posteriores ejecuciones para la búsqueda de soluciones para el mismo problema.

Dentro del paradigma de los algoritmos genéticos es donde se encuentran más estudios teóricos sobre cual debe ser el número de generaciones adecuado, para un problema determinado. Así y según [Alander, 2000], este número dependerá del tamaño de la población. De este modo, si tenemos una población de tamaño n, y suponiendo una función de evaluación bien diseñada, el umbral para el número de generaciones sería:

$$v = log_2(n) \qquad (1\text{-}31)$$

Otros autores sin embargo (Bäck, 1995) fijan este umbral en función de la longitud del cromosoma. Así para un cromosoma de tamaño l se fija un número de generaciones de:

$$v = \mathrm{O}(\sqrt{l}) \qquad (1\text{-}32)$$

En general y para cualquier paradigma de la computación evolutiva, el número de generaciones va a depender mucho del problema a tratar. Por tanto, estos se deben tener en cuenta sólo a un nivel orientativo, ya que un número insuficiente de generaciones puede que no llegue a proporcionar una buena solución. Por otro lado, si se sobrestima el valor de v, se emplean más generaciones de las necesarias que no mejoran la solución final, produciendo un algoritmo costoso en tiempo y en cálculo.

Criterio del número máximo de generaciones sin mejorar

Otra posibilidad para detener la ejecución del algoritmo consiste en hacer un estudio del estado de convergencia de la población durante la ejecución del algoritmo. De esta forma se puede terminar la búsqueda cuando se estime que la población ha convergido a una buena solución. Hay varias formas de hacer este estudio. Una de las más sencillas es contabilizar el número de generaciones en las que no se ha mejorado la aptitud del mejor cromosoma de la población y detener la búsqueda si este número de generaciones sobrepasa cierto límite v_0, propuesto a priori.

Al igual que en el criterio anterior, se debe escoger un valor adecuado de v_0, de forma que no se realicen más generaciones de las necesarias, lo que también va a depender del problema que se esté resolviendo.

1.3.4.3 La representación

En las aplicaciones del mundo real, el espacio de búsqueda se define mediante un conjunto de objetos, que poseen unos parámetros que están sujetos a optimización y constituyen lo que se denomina el *espacio fenotipo*. De otra parte los operadores evolutivos trabajan con objetos matemáticos abstractos, como cadenas binarias que representan a los objetos anteriores, y que determinarían el *espacio genotipo*. Obviamente se necesita una función de codificación entre el espacio del genotipo y el del fenotipo.

Los algoritmos genéticos clásicos usan, para un individuo, una representación binaria consistente en cadenas de longitud fija que utilizan el alfabeto {0, 1}, y a las que se les suele denominar cromosoma. La aplicación de una función de evaluación f, a una de estas cadenas nos devolverá el valor de adaptación de la solución que este individuo representa.

$$f : \{0,1\}^l \rightarrow \Re \qquad (1\text{-}33)$$

Junto a la representación binaria, a menudo es de obligado uso la utilización de funciones de codificación y decodificación h: $M \rightarrow \{0, 1\}^l$ y h': $\{0, 1\}^l \rightarrow M$ que

facilita la conversión de soluciones $x \in M$ a cadenas binarias $h(x) \in \{0, 1\}$ y a la inversa. En el caso de optimización de parámetros continuos, los algoritmos genéticos representan un vector real $\bar{u} \in \Re^n$, mediante una cadena binaria $\bar{o} \in \{0, 1\}^l$ de la siguiente manera: la cadena binaria se divide lógicamente en n segmentos de igual longitud l', por lo que $l = n \cdot l$'. Cada segmento es decodificado en su correspondiente parámetro real.

La preferencia por usar una representación binaria de las soluciones en algoritmos genéticos se deriva de la *teoría de esquemas* [Holland, 1975], donde se analizan los algoritmos genéticos en general, en función de esta teoría. El término esquema denota un patrón de similaridad que representa un subconjunto de $\{0, 1\}^l$. En esta teoría, el *teorema de esquemas de algoritmos genéticos* afirma que los algoritmos genéticos canónicos proveen de una estrategia casi óptima para alcanzar una solución en tiempo finito [Michalewicz, 1990].

Por otro lado, también existen trabajos [Eshelman, 1993][Janikow, 1991] en los que se concluye que esta codificación binaria de variables continuas, tiene ciertas desventajas. Según estos trabajos la función de codificación puede introducir cierta multimodalidad y hacer más complejo la resolución del problema en sí.

Existe otro tipo de estudio teórico sobre el funcionamiento de los algoritmos genéticos. En este estudio [Nix, 1992] se modelan los algoritmos genéticos mediante la teoría de cadenas de Markov. En esta aproximación se profundiza en las propiedades de convergencia de estos y en su conducta dinámica, desarrollando los *modelos ejecutables* que facilitan la simulación directa de los algoritmos genéticos mediante cadenas de Markov, para problemas de dimensión suficientemente pequeña.

Al contrario de los algoritmos genéticos, la representación en *estrategias de evolución* y *programación evolutiva* se basa directamente en la utilización de vectores reales que se corresponden directamente con los parámetros a optimizar.

$$f : M \subseteq \Re^n \rightarrow \Re \qquad (1\text{-}34)$$

Además y para problemas reales con complejos espacios de búsqueda, cuya representación no es tan directa, se desarrollan otras variantes de representación. Todo esto implica que los métodos de representación en estrategias evolutivas y programación evolutiva sean menos estándar y que existan menos estudios sobre estas representaciones que en algoritmos genéticos. Una ventaja que si tiene la representación que utilizan estos métodos es que su función de codificación es más simple y por lo tanto, conlleva menos problemas asociados.

Si comparamos ambas aproximaciones, para la representación mediante cadenas de bits existen trabajos, tanto empíricos como teóricos que avalan la primera aproximación. Por otro lado, la utilización de una función de codificación compleja en

esta representación, puede introducir no-linearidades adicionales y otras dificultades matemáticas que pueden dificultar el proceso de búsqueda sustancialmente. En cuanto a la representación mediante vectores de reales hay más resultados empíricos que teóricos que demuestran buenos resultados. Además su función de codificación, al ser más simple, ocasiona menos problemas de los citados anteriormente.

No hay pues, una respuesta general a la pregunta de cual de las dos aproximaciones mencionadas seguir dado un proyecto específico, pero muchas aplicaciones en la práctica han mostrado que se pueden obtener buenas soluciones después de imponer modificaciones a las representaciones tradicionales [Michalewicz, 1993].

1.3.4.4 Operadores de reproducción

Los operadores de reproducción van a servir para crear individuos nuevos en la población, a partir de un subconjunto seleccionado de entre los individuos actuales. Con este, y otros elementos, se pretende mantener una diversidad en la población que explore nuevas zonas del espacio de búsqueda. De modo que si los nuevos individuos aportan mejores soluciones, sustituirán a los ya existentes, con mucha probabilidad.

Mutación

Es uno de los operadores clásicos de variación de la información genética. En general el diseño de estos operadores debe de obedecer a las propiedades matemáticas de la representación elegida, aunque hay ciertos grados de libertad.

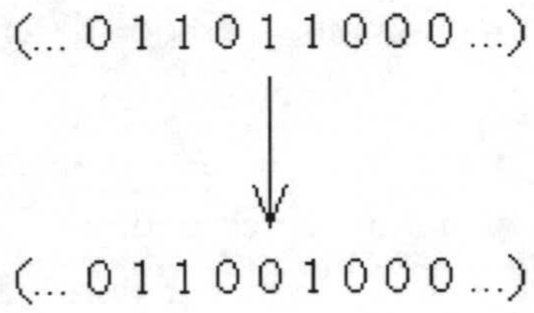

Figura 1-6: Mutación de un bit para un individuo en un algoritmo genético

La mutación en los algoritmos genéticos, inicialmente se desarrolla como un operador secundario, de pequeña importancia. En estos algoritmos este operador trabaja invirtiendo bits con probabilidad p_m, donde p_m suele ser un valor pequeño. Así es normal escoger $p_m \in [0.005, 0.01]$ o $p_m=1/l$, donde l es la longitud de los cromosomas. Este valor además puede variar a lo largo de la evolución de manera que mejore los resultados de ésta. En [Bäck, 1993] se decrementa el valor de p_m lo que ayuda en la convergencia y velocidad del algoritmo genético. En el apartado 1.3.5. Adaptación de parámetros, se estudian la adaptación de parámetros en general en un algoritmo evolutivo.

Originalmente, la mutación en la programación evolutiva fue implementada como un cambio o cambios aleatorios en función de la descripción de las máquinas de estados finitos de acuerdo a cinco diferentes modificaciones: cambio de un símbolo de salida, cambio de un estado de transición, adición de un estado, eliminación de un estado o cambio de un estado inicial. Las mutaciones fueron realizadas con probabilidad uniforme, y el número de mutaciones para un descendiente fue prefijado o elegido de acuerdo a una distribución de probabilidad. Hoy en día los esquemas de mutación frecuentemente utilizados son similares a los utilizados en las estrategias de evolución.

En estas estrategias, los individuos están formados por variables objeto. La mutación se realiza entonces independiente en cada vector elemento sumando un valor aleatorio que sigue una distribución normal con media 0 y desviación típica σ. Así la mutación del valor de una determinada variable se realizaría de la siguiente forma:

$$x'_i = x_i + N(0, \sigma) \qquad (1\text{-}35)$$

El control del tamaño de la mutación σ, también se trata en el apartado 1.3.5. Adaptación de parámetros.

Para terminar, resaltar la importancia de este operador en las estrategias de evolución y en la programación evolutiva, hasta tal punto de que muchas veces, este es el único operador considerado.

Cruce

Este operador es uno de los más importantes en los algoritmos genéticos clásicos. Estos algoritmos proponen como operador de cruce clásico, el operador de cruce en un punto, que consiste en elegir dos individuos de la población aleatoriamente, determinar una posición en la cadena de bits, también aleatoriamente, denominada punto de cruce y generar el descendiente. Para esto, se concatena, la subcadena izquierda de un padre, con la subcadena derecha del otro padre.

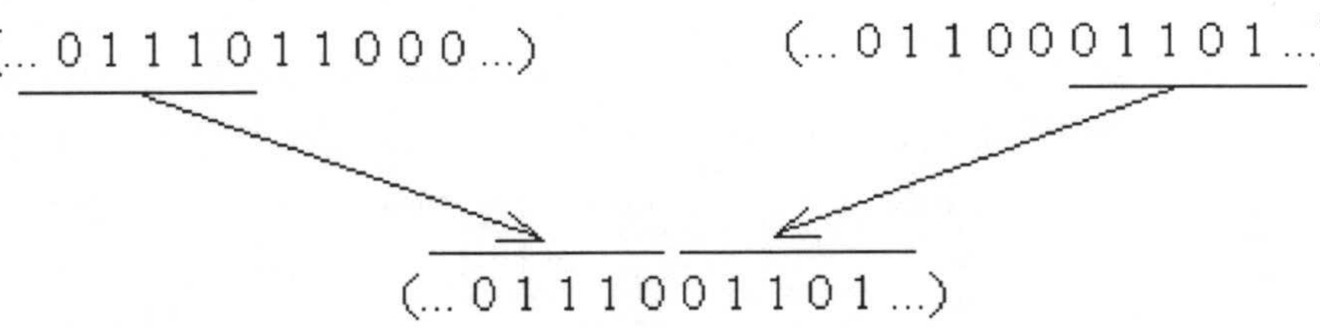

Figura 1-7: Cruce de dos individuos en un algoritmo genético

Existen además, numerosas variaciones de este operador, como por ejemplo: incremento del número de puntos de cruce [Eshelman, 1989], cruce uniforme

[Syswerda, 1989] (donde cada bit del descendiente es elegido aleatoriamente de los padres), etc. En los algoritmos genéticos el operador de cruce se aplica normalmente con una probabilidad p_c, donde $p_c \in [0.75, 0{,}95]$.

En el caso de las estrategias de evolución, la utilización de este operador no está tan clara y no existen modelos de operadores predefinidos como en el caso de los algoritmos genéticos. En estas estrategias tras aplicar este operador en el bucle principal del Algoritmo 1-7, se genera una población intermedia de λ individuos, tras aplicar λ veces el operador a la población padre y crear un descendiente por aplicación a partir de δ padres ($1 \leq \delta \leq \mu$), donde μ es el tamaño de la población. Normalmente $\delta = 2$. Existen trabajos [Beyer, 1995] también con operadores de cruce que utilizan múltiples padres, donde más de dos individuos, participan en la generación de un descendiente, representando esta generalización mejoras en la eficiencia en ciertas aplicaciones.

Como se ha comentado, los tipos de operador de cruce, para vectores reales, no son tan estándares y dependen más de la aplicación, aunque existen algunos ejemplos comunes [Bäck, 1996] como por ejemplo *cruce discreto*, que consiste en la elección aleatoria de las variables de los padres, similar al cruce uniforme en algoritmos genéticos. Otro ejemplo sería *cruce intermedio* donde se utilizan medias aritméticas de los padres. Evidentemente, también se puede utilizar otros operadores de cruce. El análisis teórico de este tipo de operador es un campo abierto de investigación.

Tal y como se ha comentado, dada una aplicación es difícil elegir a priori un operador de cruce patrón, con unos parámetros más o menos estándar con el que vayamos a tener éxito en la evolución de nuestra población, demostrándose que existen situaciones en las que puede llegar a darse divergencia de la población sino se eligen bien los parámetros de éste [Kursawe, 1995].

Con respecto a la programación evolutiva, basándose en la formación y semántica de los individuos se puede afirmar que este modelo no usa cruce. Según [Fogel, 1993], antes de centrarse en un mecanismo de cruce, se debe examinar y simular su efecto funcional e interpretar una cadena de símbolos como una reproducción de la población.

1.3.4.5 Selección / Reemplazo

El mecanismo de selección se puede considerar como un operador que en vez trabajar con la información genética de los individuos trabaja con su valor de adaptación.

En algoritmos genéticos, la selección se implementa como un operador de probabilidad, así dado un individuo a_i su probabilidad de selección es:

$$p(a_i) = f(a_i) / \sum_{j=1}^{\mu} f(a_j) \qquad (1\text{-}36)$$

denominándose a este tipo de selección, *selección proporcional* o *ruleta*. Este método requiere valores de adaptación positivos y una tarea de maximización, utilizándose funciones de escalado cuando sea necesario transformar el valor de adaptación convenientemente. Otro tipo de selección es la *selección basada en ranking*, que no usa valor de adaptación directamente sino el índice de individuos ordenados en función del valor de adaptación, para calcular la probabilidad de selección. La *selección tournament* trabaja tomando una muestra uniforme aleatoria de cierta cantidad $q > 1$ de la población, seleccionando el mejor de esos q individuos para sobrevivir en la próxima generación, repitiendo el proceso hasta que la población de descendientes se complete. Este método es muy usado porque, es fácil de implementar, computacionalmente eficiente, y permite ajustar la presión selectiva aumentando o disminuyendo el valor de q. Una descripción de los principales métodos de selección, incluyendo una caracterización de su presión selectiva en términos de medidas numéricas se encuentra en [Back, 1994].

En las estrategias de evolución se utilizan los esquemas de selección / reemplazo (μ, λ). La notación (μ, λ) indica que μ padres crean λ hijos mediante recombinación y mutación, siendo $\lambda > \mu$. Los mejores μ individuos descendientes son determinísticamente seleccionados para reemplazar a los padres. Hay que destacar que con este esquema el mejor miembro de la población en la generación $t + 1$ puede tener peor valor de adaptación que en la generación t, a este tipo de método se le denomina *no elitista*, ya que puede permitir un deterioro temporal. Esta estrategia se lleva a cabo para ayudar salir de regiones de atracción de óptimos locales y alcanzar así óptimos globales. Por otro lado las estrategias ($\mu + \lambda$) selecciona los μ supervivientes de entre la unión entre los padres y los descendientes, de esta manera el curso de la evolución es monótona

La utilización de la estrategia (μ, λ) o ($\mu + \lambda$) depende de la aplicación práctica con la que se esté trabajando, para de esta manera obtener mejores resultados. Además ambos esquemas se pueden interpretar como instancias de la estrategia general (μ, κ, λ) donde κ representa la vida máxima de un individuo en generaciones, pudiendo tomar valores entre $1 \leq \kappa \leq \infty$. Si $\kappa = 1$ entonces el método de selección sigue la (μ, λ), mientras que si $\kappa = \infty$, entonces se sigue la estrategia ($\mu + \lambda$) [Schwefel, 1995].

Una pequeña diferencia entre la programación evolutiva y las estrategias de evolución consiste en una variante probabilística de la estrategia de selección ($\mu + \lambda$) en la programación evolutiva, donde cada individuo, fuera de los individuos padres o descendientes se evalúa frente a un número q, con $1 \leq q \leq 10$, de otros individuos

pertenecientes al conjunto unión de padres e hijos. En cada una de estas comparaciones se determina un individuo ganador, que será aquel cuyo valor de adaptación sea mejor o igual que el de todos sus oponentes. Los μ individuos que hayan sido más veces ganadores serán los padres de la siguiente generación. Como se muestra en [Bäck, 1996], este método de selección es una versión probabilística de la estrategia $(\mu + \lambda)$ que es más determinística cuando el número q de competidores se incrementa. La determinación de si es mejor un esquema de selección más probabilístico que uno más determinístico es un campo de investigación abierto.

1.3.5 Adaptación de parámetros

El concepto de adaptación está implícito en la computación evolutiva. Su implicación más directa sería en el plano de encontrar la solución a un problema dado, aunque también aparece a la hora de ajustar los parámetros del algoritmo que resuelva el problema en cuestión [Eiben, 2000][Angeline, 1995][Spears, 1995].

Para resolver un problema en computación evolutiva, no solo se necesita elegir el algoritmo, la representación o los operadores a usar, sino también elegir o configurar los parámetros (probabilidad de operadores, condición de parada, etc.) que van a terminar de caracterizar nuestra estrategia general. El ajuste de la configuración o parámetros, puede determinar en gran medida la eficiencia final del método. Tradicionalmente este proceso se ha realizado de manera manual lo que puede consumir muchos recursos, por lo que se buscan soluciones donde se automatice este proceso.

La ejecución de un algoritmo evolutivo es intrínsicamente un proceso adaptativo y dinámico. El uso de parámetros fijos, parece contradecir el espíritu evolutivo general de un algoritmo de este tipo. Además existen otra serie de inconvenientes a la aproximación tradicional:

- Un error del programador al establecer los parámetros puede lugar a errores o alcanzar unos resultados no óptimos.

- El ajuste de parámetros consume grandes cantidades de tiempo.

- El conjunto de parámetros óptimo puede variar a lo largo de la ejecución.

Por lo tanto, parece natural pensar en modificar ciertos parámetros a lo largo de la ejecución de un algoritmo evolutivo. Esta tarea se realizará utilizando reglas, posiblemente heurísticas, que, o analizan lo ocurrido antes del estado actual, o emplean algún mecanismo auto-adaptativo. Estos cambios pueden afectar desde a una parte de un individuo, hasta a toda la población.

1.3.5.1 Diseño de parámetros adaptativos

A la hora de abordar un estudio sobre el diseño de un mecanismo adaptativo para los parámetros que intervienen en un algoritmo evolutivo es necesario fijar ciertos aspectos clave como:

- Elemento/s cuyos parámetro/s se van a adaptar.
- Tipo de adaptación a realizar. Estos tipos pueden ser: determinísticos, basados en el transcurso de la evolución o auto-adaptativos.
- Nivel o alcance al que se realiza la adaptación: nivel de individuo, de población, etc.

Comenzando por el primer aspecto, a la hora de diseñar un algoritmo evolutivo es necesario elegir elementos o componentes de éste como: representación de los individuos, función de evaluación, operadores, etc. Para cada uno de estos elementos, y dependiendo también del diseño, se va a poder elegir adaptar uno o varios parámetros. Así por ejemplo podemos pensar en adaptar según distintas variables de estado, el valor de la función de evaluación, la probabilidad de aplicación de los operadores, etc. Una completa revisión de este tipo de adaptaciones se encuentra en [Eiben, 2000].

En cuanto al tipo de adaptación a realizar y según [Eiben, 2000], existen tres tipos principales:

- *Determinístico*: este tipo de adaptación se produce cuando un parámetro se altera siguiendo una regla determinística. Esta regla modificará el parámetro sin usar ninguna variable de estado o realimentación del algoritmo evolutivo en sí. Este tipo de reglas suelen usar para su funcionamiento, el tiempo o generaciones transcurridas. Por ejemplo se podría actualizar la probabilidad de mutación de un algoritmo evolutivo p_m, de la siguiente manera:

$$p_m = 0.5 - 0.3 \cdot g / G \qquad (1\text{-}37)$$

  donde g es la generación actual y G es el número total de generaciones. De este modo la probabilidad de mutación variará de 0.5 a 0.2, conforme aumente el número de generaciones. Los primeros ejemplos de uso de este tipo de adaptación aparecen en [Fogarty, 1989].

- *Adaptativo*: en este caso la adaptación se produce en función de ciertas variables de estado o existe cierta realimentación a partir del algoritmo evolutivo. Esta asignación puede influir en la asignación de crédito y su

efecto se podrá propagar por toda la población. Ejemplo de este tipo de adaptación sería la *regla de éxito 1/5* de [Rechenberg, 1973] en las estrategias de evolución (1+1), cuya misión era variar el valor de la mutación. En este contexto se utilizan mutaciones gausianas que vienen caracterizadas por dos parámetros: la media que suele establecerse a cero y la desviación típica σ que puede interpretarse como el tamaño de la mutación, de este modo la mutación de un valor x_i vendría expresada cómo en (1-35), donde $N(0, \sigma)$ es un número aleatorio gausiano con media 0 y desviación típica σ. La regla de éxito 1/5 establece que el porcentaje de mutaciones con éxito p_s, o mutaciones que producen un hijo mejor que el padre, debe ser 1/5 de todas las mutaciones, incrementándose el tamaño o desviación típica de la mutación si p_s es mayor de 1/5, y disminuyendo en caso contrario, lo que se expresa como:

Si (t mod $n = 0$) Entonces

$$\sigma(t) = \begin{cases} \sigma(t-n)/c & si \quad p_s > 1/5 \\ \sigma(t-n)\cdot c & si \quad p_s < 1/5 \\ \sigma(t-n) & si \quad p_s = 1/5 \end{cases} \qquad (1\text{-}38)$$

si no

$$\sigma(t) = \sigma(t-1)$$

La adaptación de σ, ocurrirá cada n iteraciones y $c \in [0.817, 1]$ según [Bäck, 1996]. Usando este mecanismo se adapta el valor de σ, en función del estado de la ejecución del algoritmo evolutivo.

- *Auto-adaptativo*: este tipo de adaptación se caracteriza porque se va a ver afectada por un proceso de evolución. Así por ejemplo, los parámetros que se van a adaptar se van a codificar, y se van a adjuntar al genotipo del individuo. De este modo, sobre estos parámetros también actuarán los operadores que a éste se le apliquen. Estos parámetros codificados no afectan al valor de adaptación del individuo directamente, pero mejores valores de adaptación conducirán a mejores individuos y estos individuos será más probable que sobrevivan y produzcan descendientes, y por lo tanto se propaguen los mejores parámetros. Un ejemplo de este método se puede encontrar en [Schwefel, 1977] donde se adapta el tamaño de la adaptación.

Por otro lado, si se considera ahora, el nivel o alcance de la adaptación realizada, se concluye que va a depender del componente del algoritmo evolutivo

donde se realice la adaptación. Así, por ejemplo un cambio en el tamaño de la mutación, puede afectar a un gen, a un cromosoma, o a la población entera, de un algoritmo genético, dependiendo de la representación en particular. Por otro lado, una adaptación de un coeficiente de la función de evaluación, siempre afectará a toda la población. Por lo tanto el nivel o alcance de una adaptación es secundario y usualmente, depende del componente tratado y de su implementación.

Por extensión se puede considerar el hecho de controlar o adaptar varios parámetros durante la ejecución de un algoritmo evolutivo. Esto, evidentemente, tiene el inconveniente de un aumento en las dependencias de los componentes elegidos, de los parámetros a adaptar, etc., y por lo tanto implica una mayor complejidad en el estudio. Sin embargo se han realizado varios estudios satisfactorios, donde mediante la heurística se consiguen mejores resultados [Hinderting, 1996][Smith, 1996].

Como conclusión, el campo de la adaptación de parámetros es un campo que presenta ciertos inconvenientes, sobre todo relacionados con el hecho de que los resultados dependen mucho de las elecciones hechas en la fase de diseño del algoritmo evolutivo. Esto implica que los resultados obtenidos dependen de la heurística y que no existen suficientes resultados teóricos o experimentales que permitan extraer conclusiones o estrategias generales, válidas para los algoritmos evolutivos en sí.

Sin embargo, se demuestra que con la adaptación de parámetros, se pueden conseguir mejoras de resultados, aunque sea a nivel local o del algoritmo evolutivo que se esté tratando. Para conseguir esto, lo normal es utilizar reglas basadas en la experiencia.

1.4 Modelos coevolutivos cooperativos

1.4.1 Introdución

Dentro de la computación evolutiva aparece como un nuevo paradigma, los modelos coevolutivos cooperativos [Potter, 2000]. Estos modelos surgen para intentar solucionar problemas, en los que la computación evolutiva tradicional encuentra ciertas complicaciones. Existen tres clases principales de estos problemas:

- La primera clase incluiría problemas en los que se requiere, múltiples y distintas soluciones, como en el caso de la optimización de funciones multimodales.

- La segunda clase la compondrían aquellos problemas que tienen una solución compuesta por subcomponentes más pequeños y especializados,

como por ejemplo en sistemas basados en reglas o en redes neuronales artificiales. A este tipo de problemas se les suele denominar *problemas de cubrimiento* (*covering problems*).

- La tercera clase consistiría en problemas que se pueden descomponer en un determinado número de subtareas simples y que por lo tanto pueden ser resueltos usando la estrategia *divide y vencerás*. Este es el caso por ejemplo de problemas de planificación de rutas como en problemas del viajante de comercio, en el que se puede trabajar con rutas más simples. Otro ejemplo de aplicación se daría en el campo de aprendizaje de conductas que se presenta en el dominio de la robótica, donde conductas complejas pueden ser descompuestas en subconductas más simples.

Existen dos principales razones por las que los algoritmos evolutivos tradicionales tienen dificultades con este tipo de problemas.

- La primera razón es que la población de individuos alcanzada con los algoritmos evolutivos tradicionales tiene una fuerte tendencia a converger. Esto se debe a que el trabajo de estos algoritmos se localiza en ciertas regiones del espacio de búsqueda donde el valor de adaptación medio es mayor. Esto, por ejemplo, supone una desventaja cuando se trabaja con problemas de optimización multimodal donde la solución necesita de más información que una simple zona del espacio. Esta convergencia también imposibilita el mantenimiento de subcomponentes coadaptados que se requieren en la resolución de problemas de cubrimiento o en la utilización de estrategias como divide y vencerás, ya que prácticamente, cualquier individuo, menos el más fuerte será eliminado.

- La segunda razón es que un individuo de un algoritmo evolutivo tradicional representa una solución completa y se evalúa en solitario. Por tanto, partiendo de que las interacciones entre la población no se modelan, incluso aunque se mantuviera la diversidad en la población, el modelo evolutivo debería ser extendido para mantener conductas diferentes, que se puedan coadaptar.

La idea para solventar estos problemas es introducir en la computación evolutiva nociones sobre modularidad, para que de esta manera se puedan generar soluciones formadas por subcomponentes que coevolucionan, que se coadaptan y que interactúan cooperando. La dificultad está en buscar extensiones para los paradigmas evolutivos que permitan a los subcomponentes coevolucionar, sin ser diseñados a mano. Para esto habrá que proveer un entorno que permita la identificación y representación de esos subcomponentes, donde además, éstos puedan interactuar,

adaptarse, y donde la *aportación de crédito* de estos represente su contribución a la solución del problema. Todo esto se deberá realizar de la forma más independiente posible.

Para desarrollar este tema, en el siguiente apartado se hará una revisión de los principales aspectos a tener en cuenta a la hora diseñar un algoritmo coevolutivo competitivo. Por último y como en este campo no hay modelos de referencia sobre este tipo de algoritmos, se hará un repaso al trabajo previo realizado.

1.4.2 Diseño de algoritmos coevolutivos cooperativos

En este apartado se van a describir los aspectos en los que se debe incidir a la hora de desarrollar una extensión a la computación evolutiva, que permita el desarrollo de subcomponentes que coevolucionan y cooperan.

1.4.2.1 Descomposición del problema

Uno de los aspectos básicos que deben ser tratados para resolver un problema mediante la coevolución cooperativa es determinar el número apropiado de subcomponentes que debe existir, así como el papel que cada uno debe desempeñar. A esta tarea se le denomina descomposición del problema. Esta descomposición se produce en un nivel macroscópico, teniendo en cuenta estrategias divide y vencerás, donde un problema complejo se divide en subtareas, que individualmente son más fáciles de resolver, a un nivel microscópico. La solución estará pues compuesta por un conjunto de subcomponentes simples.

Para algunos problemas puede ser más o menos sencillo conocer cual es su descomposición idónea. Por ejemplo en el caso de la optimización de una función de k variables independientes, una descomposición que a priori parece lógica, sería aquella en la que determinan k elementos independientes. Sin embargo, existen muchos problemas en que disponemos de poca o nula información sobre el número subcomponentes a obtener en la descomposición. Por ejemplo en el problema de aprendizaje mediante reglas, será improbable que, a priori, se conozca el número de reglas necesario para cubrir el conjunto de ejemplos.

Así pues encontrar una descomposición adecuada no siempre es obvio y es muy importante trabajar este problema. En el caso ideal, el algoritmo debería de generar una descomposición adecuada como resultado de la evolución.

1.4.2.2 Interdependencias entre subcomponentes

Los subcomponentes en los que se descompone un problema son independientes cuando cada uno de ellos puede ser resuelto sin tener en cuenta a los

demás. Gráficamente se podría representar de manera que cada subcomponente estaría intentado alcanzar la cima de su espacio de búsqueda independiente. Desafortunadamente, muchos problemas solo pueden ser descompuestos en subcomponentes entre los que existen interdependencias.

El efecto de cambiar uno de esos subcomponentes interdependientes es a menudo descrito, como una deformación del espacio de búsqueda de los otros subcomponentes [Kauffman, 1991]. Por tanto es necesario que el diseño de nuestro algoritmo recoja esta característica.

1.4.2.3 Asignación de crédito

Cuando una tarea, es cubierta colectivamente por un conjunto de soluciones parciales, se llama *asignación de crédito* a la determinación de la contribución de cada solución parcial.

Uno de los principios de la evolución Darwiniana es que los individuos, cuyo valor de adaptación es mayor, van a pasar sus características de su genotipo a futuras generaciones. Por tanto si el objetivo es desarrollar un modelo evolutivo para resolver problemas mediante una colección subcomponentes que cooperan, debe existir un proceso que mida la aportación de cada subcomponente a la solución alcanzada.

Esta medición va a depender del número de subcomponentes del problema, de las interdependecias de estos y en general del problema en sí.

1.4.2.4 Diversidad de la población

Cuando se usa un algoritmo evolutivo clásico, en los que se busca un individuo que represente una solución para un determinado problema, la diversidad en la población se mantiene durante un determinado tiempo en la ejecución, para realizar una exploración razonable del espacio de búsqueda. Esto se observa en la estructura de funcionamiento general de la computación evolutiva mostrada en el Algoritmo 1-7. En este algoritmo, y si se obvian los efectos de los operadores usados, se favorece la creación y sustento de individuos en ciertas zonas del espacio de búsqueda donde existe un valor de adaptación superior. Este es el resultado de una continua presión selectiva que favorece la convergencia de la población.

Sin embargo, y cuando se soluciona un problema mediante una colección de subcompnentes que cooperan es necesario mantener la diversidad hasta el final. La diversidad en algoritmos evolutivos clásicos se introduce mediante los operadores usados. Como se ha comentado, esta diversidad no es suficiente para los modelos de evolución cooperativa competitiva por lo que es necesario introducir otras estrategias que la mantengan. Así por ejemplo se pueden usar técnicas como la *compartición del valor de adaptación*, (*fitness sharing*) [Goldberg, 1987] o la técnica de *nichos*

[Beasley, 1993], para poblaciones simples, u otras que mantienen poblaciones aisladas, denominadas especies, cada una trabajando en una zona del espacio de búsqueda.

La técnica de *fitness sharing* fue desarrollada para mantener la diversidad en la resolución de problemas de funciones multimodales, y se basa en modificar convenientemente el valor de adaptación (fitness). Así se define la siguiente función de compartición (sharing):

$$comp(d_{ij}) = \begin{cases} 1 \quad si \quad d_{ij} = 0 \\ 1 - \left(\frac{d_{ij}}{\sigma_s} \right) \quad si \quad d_{ij} < \sigma_s \\ 0 \quad en\ otro\ caso \end{cases} \qquad (1\text{-}39)$$

donde d_{ij} representa la distancia entre dos individuos i y j, σ_s define un radio de grupo, y α controla la intensidad con la que varia esta función dependiendo de la distancia. La distancia se puede medir tanto en el espacio del fenotipo, como en el del genotipo. El valor de adaptación se modifica de la siguiente manera con la función de compartición:

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} comp(d_{ij})} \qquad (1\text{-}40)$$

donde f sería el valor de adaptación inicial y n es el tamaño de la población. Cuando se aplica la compartición del valor de adaptación, la población se distribuye en un determinado número de agrupamientos de individuos alrededor de un máximo. A estas regiones se les denomina *nichos*. La determinación del valor de σ_s, va a depender del problema a resolver.

Posteriormente en el trabajo [Beasley, 1993] se desarrolla la *técnica secuencial de nichos*, también para resolver funciones multimodales. Esta aproximación toma prestada, de la técnica de compartición del valor de adaptación, la idea de modificar el valor de adaptación basándose en distancias. Sin embargo, en vez estar continuamente modificando el valor de adaptación, basándose en la distancia entre individuos de la actual población, la técnica secuencial de nichos reduce el valor de adaptación de un individuo en función de su distancia a máximos encontrados en ejecuciones anteriores del algoritmo genético. Esta técnica, si bien es menos costosa computacionalmente que la técnica de compartición del valor de adaptación, también es dependiente del valor de σ_s.

1.4.3 Trabajo previo

Las estrategias coevolutivas cooperativas son un paradigma nuevo dentro de la computación evolutiva, donde además las soluciones adoptadas dependen en gran medida del problema a resolver. Es por esto que no existen modelos de referencia que sirvan de base a la hora de abordar un nuevo problema. Por tanto se va a describir cuales son los trabajos más importantes realizados en este campo.

Una de las primeras extensiones al modelo evolutivo básico para soportar subcomponentes coadaptados es el *sistema de clasificación* de [Holland, 1978]. Este sistema de clasificación es un sistema basado en reglas, donde se desarrolla una población de reglas estímulo-respuesta usando un algoritmo genético. Las reglas individuales en la población trabajan juntas para formar la solución a un determinado problema. Estas reglas van a interaccionar entre ellas utilizando un modelo micro-económico y van competir por activarse. Para realizar la asignación de crédito se utiliza un algoritmo denominado *bucket brigade* que va a ir propagando un valor según una cadena de activación.

Otra aproximación al mantenimiento de una población de reglas coevolutivas se realiza en [Giordana, 1994], mediante su sistema REGAL. Este sistema aprende reglas de clasificación a partir de ejemplos preclasificados. La descomposición del problema se realiza mediante un operador de selección, que agrupa a los individuos basándose en el cubrimiento que realicen de un subconjunto de ejemplos, elegido aleatoriamente. Una solución se forma seleccionando la mejor regla de cada conjunto. Un operador semilla mantiene la diversidad del ecosistema, creando los individuos apropiados, cuando no existe nadie, con las propiedades necesarias para cubrir algún elemento del subconjunto aleatorio. El valor de adaptación (asignación de crédito) de cada individuo en cada grupo es una función de la consistencia, con respecto a un grupo de ejemplos, y a su simplicidad.

Posteriormente la eficiencia de REGAL [Giordana, 1996] se ha mejorado con el uso de islas de poblaciones semi-aisladas. A cada isla se asigna un conjunto de ejemplos preclasificados y se aplica el operador de selección y el operador semilla. Al final de cada generación ocurre una migración de individuos entre islas. La funcionamiento general se completa con la existencia de un proceso supervisor que maneja la asignación de ejemplos a las islas.

El trabajo de [Whitehead, 1996] es un ejemplo de desarrollo de redes neuronales, concretamente redes de funciones de base radial, mediante técnicas cooperativas competitivas. En este caso se utiliza un algoritmo genético básico al cual se le hacen las extensiones necesarias. Así se parte de la población de individuos, correspondiendo un individuo a una neurona de la red, por lo que existe una función de evaluación que asigna un valor de adaptación a cada individuo/neurona. A estos

individuos se les aplica la metodología evolutiva típica de un algoritmo genético. La selección, en este algoritmo, promueve de una parte, la competición entre individuos debido a la función de evaluación que se utiliza. De otra parte, también promueve la cooperación entre los mismos individuos que se van a encargar de cubrir diferentes partes del dominio de actuación. El valor de adaptación se establece en función de la disposición espacial del vector de pesos.

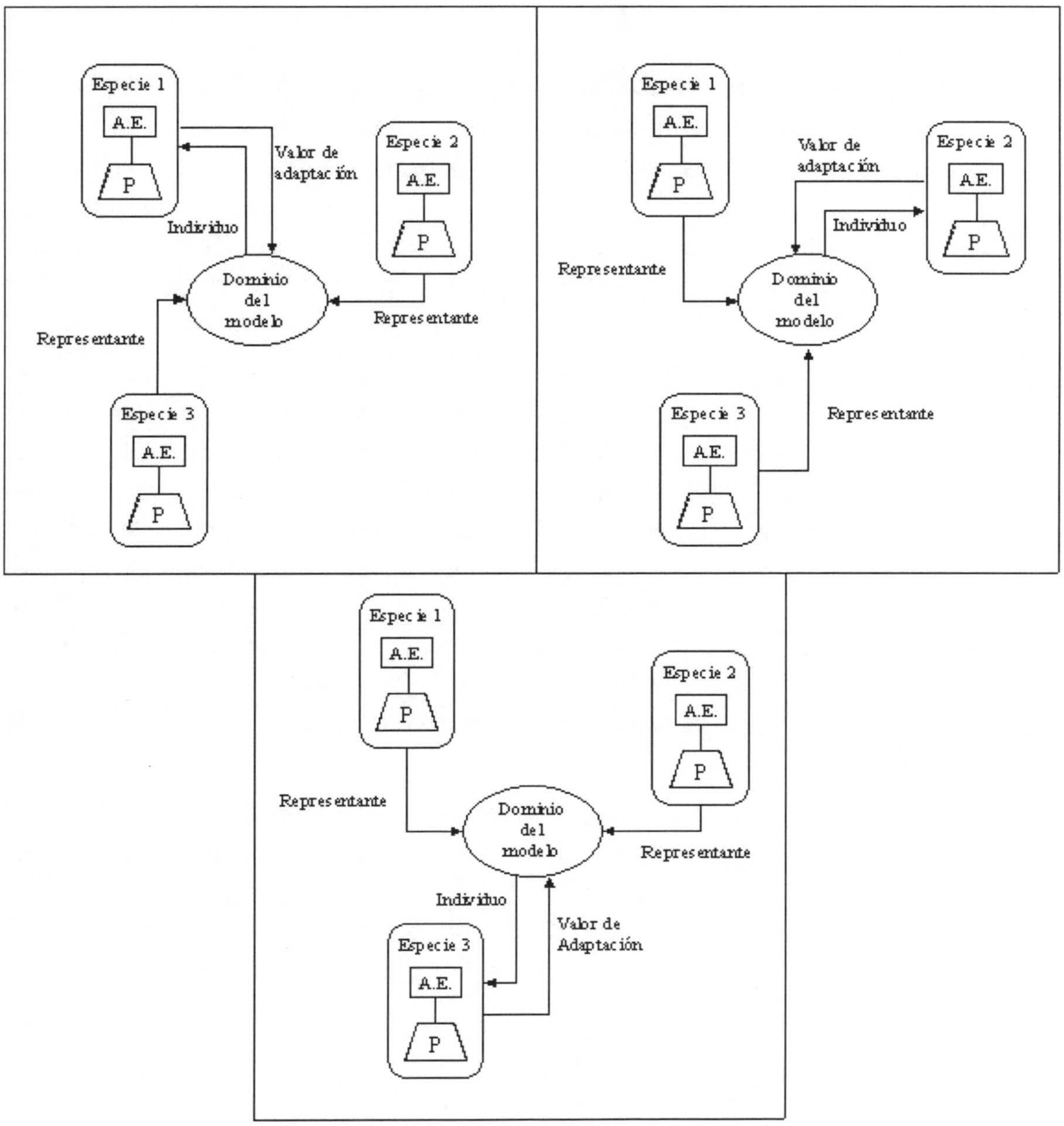

Figura 1-8: Modelo coevolutivo-cooperativo de Potter. P: Población de individuos de una especie. A.E.: Algoritmo evolutivo

Dentro del campo del diseño de redes neuronales artificiales otra de aproximación coevolutiva cooperativa es el sistema SANE [Moriarty, 1997]. En este sistema cada individuo en la población representa una neurona, especificando sus entradas, salidas y pesos asociados. Una red neuronal se constituye a partir de la selección de un conjunto de individuos. Por otro lado se mantiene una población aislada de redes (llamadas *blueprints*) constituidas por conjuntos de neuronas, que juntas, trabajan "bien". La evaluación neuronal consiste en seleccionar neuronas de estos blueprints, formar redes, evaluar estas redes y asignar el valor de adaptación obtenido al blueprint. El valor de adaptación de una neurona será la media de los valores de adaptación de las cinco mejores redes en las que participe. Como se puede observar es una medida de cómo una neurona trabaja con otras neuronas para la resolución de un problema. De esta manera se mantiene la población y se propicia formación de población de subcomponentes que cooperan.

Uno de los trabajos más importantes dentro de la coevolución cooperativa es el de modelo de [Potter, 2000]. En este trabajo propone un ecosistema donde existen dos o más especies aisladas. Cada especie funciona como una población individual donde se aplica el Algoritmo 1-7, o algoritmo evolutivo general instanciado a uno de los paradigmas clásicos. Así pues, entre los individuos de una población se producirá, reproducción, selección, etc, pero no existirá ninguna relación evolutiva entre los individuos de distintas especies. Para obtener el valor de adaptación de los individuos de una especie, se establece una cooperación con representantes de las otras especies, determinando éste en función del resultado de esta colaboración. Existen diversos métodos para elegir los representantes de otras especies con los que se colaborará. En muchos casos el representante de una especie puede ser simplemente el que tenga mejor valor de adaptación. También se puede elegir este representante de una forma más natural, escogiendo el representante de una manera aleatoria entre los mejores. En la Figura 1-8 se muestra una representación de este modelo.

En [García, 2002] se presenta un interesante y reciente trabajo que usa la coevolución cooperativa. El modelo, llamado MOBNET, se utiliza para obtener tanto la topología como los pesos de una red. En este caso los subcomponentes con los que se trabaja son subredes que se combinan para dar una red completa. La asignación de crédito a los subcomponentes se resuelve utilizando una técnica multiobjetivo donde se valoran eficiencia, complejidad, etc, de los subcomponentes. Los subcomponentes van evolucionando paralelamente en poblaciones aisladas. Una red completa se forma cogiendo un subcomponente de cada una de las poblaciones anteriores. Además existe una población de redes completas que también van evolucionando.

1.5 Diseño evolutivo de redes neuronales artificiales

Tal y como se ha descrito, las redes neuronales artificiales ofrecen un interesante paradigma para el diseño y análisis de sistemas inteligentes dentro del campo de la inteligencia artificial.

La extensa investigación realizada en el área ha conducido a lograr importantes resultados, tanto empíricos como teóricos, y al desarrollo de importantes aplicaciones prácticas. Sin embargo y normalmente, el diseño de una red neuronal para una determinada aplicación y bajo una serie de restricciones, suele ser un proceso heurístico, de prueba y error, basado en la experiencia con similares aplicaciones. Además, la eficiencia y complejidad de una red, para un problema particular, depende entre otras cosas, de la elección de las neuronas, la arquitectura de la red y el algoritmo de aprendizaje usado.

Por ejemplo se parte, de un típico algoritmo de aprendizaje, para establecer el conjunto de pesos de una red, dada su topología y un determinado conjunto de patrones de entrenamiento. Inicialmente, y para que este algoritmo tenga éxito, será necesario que este conjunto de pesos óptimo exista en el espacio de búsqueda determinado por las restricciones, fruto de las distintas elecciones realizadas (tipo de neuronas, topología de la red, etc). Por otro lado, evidentemente, el algoritmo de búsqueda debe de ser capaz, también de alcanzar este conjunto de pesos óptimo. Pero incluso aunque el algoritmo de búsqueda encuentre un conjunto de pesos adecuado, para el conjunto de patrones de entrenamiento, puede que la red no sea capaz de generalizar para patrones, que no se han usado durante el entrenamiento, o puede que la complejidad de la red obtenida no esté cerca de ser óptima. El conjunto de estos factores hacen que el diseño de una red neuronal artificial sea una tarea complicada.

Es natural, dadas estas premisas, que se investigue en técnicas para automatizar el diseño del conjunto de parámetros que define una red neuronal, para un problema determinado y bajo un conjunto de restricciones. Los algoritmos evolutivos ofrecen una aproximación atractiva y relativamente eficiente, para la resolución de este problema. Estos devolverán, por su naturaleza, una solución casi óptima en una amplia variedad de dominios de problemas.

En un nivel abstracto, el problema del diseño de una red neuronal es esencialmente una instancia de los problemas que se resuelven en otras disciplinas de la inteligencia artificial [Uhr, 1973] como el *lambda cálculo*, *máquinas de Turing*, etc. Estas ofrecen aproximaciones para buscar soluciones a problemas complejos cuando ni la estructura, ni el tamaño de las soluciones se conoce de antemano. Sin embargo los algoritmos evolutivos parecen adaptarse mejor [Balakrishnan, 1995] al diseño de estructuras de computo, masivamente paralelas, altamente interconectadas y con elementos de proceso relativamente simples, como son las redes neuronales. Incluso,

un estudio sobre el diseño de una red neuronal conduce a una exploración de las respuestas a cuestiones fundamentales de interés de las neurociencias.

La evolución en las redes neuronales [Yao, 1999][Balakrishnan, 1995][Grönroos, 1998] se suele realizar en tres niveles: evolución de los pesos, evolución de las arquitecturas, evolución de las reglas de aprendizaje.

Mediante la evolución de los pesos se pretende encontrar un conjunto de pesos casi óptimo, dada una determinada arquitectura. En este caso, como en otros, el paradigma evolutivo más utilizado son los algoritmos genéticos. Conviene también relacionar o comparar esta aproximación evolutiva con los algoritmos clásicos de entrenamiento basados en la minimización del error.

De la misma manera, con la evolución de la arquitectura de una red, se pretende conseguir que esta posea una arquitectura casi óptima. La arquitectura de una red va a determinar en gran medida la capacidad de proceso de ésta, lo que implica que este es uno de los campos de investigación más importante dentro del diseño de las redes neuronales. Existen dos aspectos importantes en la evolución de las arquitecturas: la codificación de la arquitectura y el procedimiento de búsqueda a utilizar.

Por último, decir que el campo de la evolución de las reglas de aprendizaje es el menos investigado, ya que resulta complicado codificar de manera más o menos general estas reglas. Para solventar en cierta manera este problema, se suelen establecer una serie de restricciones.

En los siguientes apartados se ampliarán cada uno de estos tres niveles en los que se puede introducir la evolución de una red.

1.5.1 Evolución de los pesos de una red

El aprendizaje de los pesos de una red, se formula usualmente como la minimización de una función de error, que suele venir dada como el error cuadrático medio total entre las salidas objetivo, dadas por los patrones de entrenamiento, y las salidas actuales ofrecidas por la red. La mayoría de estos algoritmos [Haykin, 1999] de entrenamiento, como el de backpropagation o los algoritmos de gradiente conjugado, se basan en aplicar técnicas de gradiente descendiente de una manera iterativa. Estos algoritmos se aplican con éxito en ciertas ocasiones, aunque tienen ciertos inconvenientes debidos a su naturaleza. Uno de los inconvenientes más importantes es que pueden quedar atrapados en mínimos locales. Otro inconveniente es que su aplicación depende de la diferenciabilidad de la función. Así, en general la complejidad de estos métodos será proporcional al coste asociado a conseguir las derivadas de la función. En el caso más extremo, estos métodos no se pueden aplicar si la función no es diferenciable.

Una de las alternativas a las técnicas clásicas, y a sus inconvenientes intrínsecos, es formular este proceso de entrenamiento como la evolución de los pesos de la red, en el entorno determinado por la arquitectura. Con esto se tienen una serie de ventajas. La primera es que los algoritmos evolutivos proporcionan una estrategia global de búsqueda que devolverá un conjunto de pesos casi óptimos. El adjetivo de casi óptimos viene dado por la naturaleza en sí de los algoritmos evolutivos que devuelven una solución "cercana" al óptimo global. Otra ventaja es que el valor de adaptación de una red de este tipo, puede ser definido de acuerdo a diferentes necesidades. Además, y al contrario que en los algoritmos de entrenamiento basados en gradiente descendiente, la función del valor de adaptación no tiene por que ser diferenciable, ni incluso continua. Por otro lado, el mismo algoritmo evolutivo, se puede usar para entrenar diferentes tipos de redes neuronales: hacia delante, hacia atrás, etc.

Así mismo el entrenamiento evolutivo de una red, facilita la generación de redes neuronales con características especiales. Para esto, se suele tratar convenientemente la función de evaluación, o función que devuelve el valor de adaptación. De esta manera, se puede incluir por ejemplo en ésta, un término de penalización que disminuya la complejidad de la red y que aumente su capacidad de generalización.

Como inconveniente un entrenamiento evolutivo es generalmente más lento que las variantes más rápidas de un algoritmo back-propagation [Fahlman, 1988] o de un algoritmo de gradiente conjugado [Moller, 1993], cuando la información de gradiente es fácil de obtener. A pesar de esto existen experimentos [Prados, 1992] en los que un algoritmmo genético ha demostrado ser más rápido que un algoritmo de gradiente descendiente. Este resultado se debe a la sensibilidad inicial de las condiciones iniciales

En resumen el uso de algoritmos evolutivos se adecúa a la hora de determinar los pesos de una red, ya que estos están especializados en trabajar con espacios de búsqueda grandes, complejos, que no tienen que ser diferenciables y que pueden ser multimodales, como son los espacios de búsqueda definidos por las funciones de error o las funciones definidas por el valor de adaptación.

1.5.1.1 Diseño del algoritmo evolutivo

En esta aproximación evolutiva para el entrenamiento de los pesos de una red, existen dos tareas principales. La primera consistiría en decidir la representación genotípica de los pesos de la red. En cuanto a la representación de los individuos de la población existen dos tipos:

- Representación mediante cadenas de bits: esta representación sería similar a la utilizada en los algoritmos genéticos y por lo tanto está muy ligada al uso

de estos como esquema general de evolución. Entre sus ventajas destacan la existencia de muchos estudios sobre este tipo de representación [Holland, 1975]. Su simplicidad y generalidad. La facilidad que ofrece a la hora de implementar operadores típicos de cruce o mutación. Entre sus inconvenientes se encuentran su pobre escalabilidad, por ejemplo, cuando la red a tratar posee muchas conexiones.

- Representación mediante números reales: en este caso, el peso de una interconexión es representado directamente por un número real. El uso de esta representación implica que la utilización de operadores de reproducción estándar no sea tan directa y que haya que adaptarlos al esquema de representación empleado. Así un operador que se usa típicamente es un operador de mutación donde se suma una cantidad aleatoria gausiana. En cuanto a la implementación de un operador de cruce debe ser muy específica por lo que muchas veces, sólo se utiliza un operador de mutación. Todo esto implica que exista menos trabajo de referencia sobre este tipo de representación, lo que es un inconveniente. Como ventaja no se tendrían problemas de precisión, ni los acarreados por la conversión genotipo-fenotipo.

La segunda tarea sería decidir el modelo de evolución a utilizar. Partiendo del algoritmo general evolutivo Algoritmo 1-7, habría que fijar todos sus parámetros: operadores, esquema de selección, etc. Dependiendo de estas elecciones los resultados pueden ser muy diferentes. En cualquier caso un individuo puede ser un conjunto de pesos de una red, o un peso de la red si se opta por una técnica coevolutiva cooperativa. Para evaluar estos individuos, lo normal será transformar los genotipos en fenotipos, aplicarlos a la red y medir el error que se produce. El valor de adaptación de cada individuo estará en función de este error.

1.5.1.2 Entrenamiento híbrido de los pesos de una red

Uno de los principales inconvenientes de los algoritmos evolutivos es que presentan cierta ineficiencia a la hora de afinar en una búsqueda local. Por otro lado, hay que tener en cuenta que cuando se utiliza sólo un algoritmo de matemático de minimización de error, se tiene que ejecutar varias veces en la práctica, partiendo de diferentes condiciones iniciales, para conseguir buenas soluciones.

Así pues, sería interesante mejorar la eficiencia de ambos, incorporando un procedimiento matemático de búsqueda local al final de una técnica de evolución. Con esto se conseguiría unir la habilidad de los algoritmos evolutivos para realizar búsqueda global con la característica de afinamiento de una búsqueda local. Los algoritmos evolutivos se usarían inicialmente para localizar una región del espacio

idónea, donde debería encontrarse el extremo global. Después, se aplicaría el procedimiento de búsqueda local para encontrar una solución, lo más cerca posible, a ese extremo global. Como procedimiento de búsqueda local se puede usar un algoritmo de gradiente descendiente.

En la Figura 1-9 se muestra un ejemplo de la importancia de partir de buenas condiciones iniciales, en un proceso de búsqueda del conjunto de pesos adecuado de una red. Así si se parte del punto w_{i1} un algoritmo de gradiente descendiente tenderá a acabar y a dar como solución w_{f1}, mientras que si se parte de w_{i2}, la solución final aportada por el algoritmo será w_{f2}. La misión, por lo tanto, del algoritmo evolutivo, será aportar como condiciones iniciales w_{i2}, en lugar de w_{i1}.

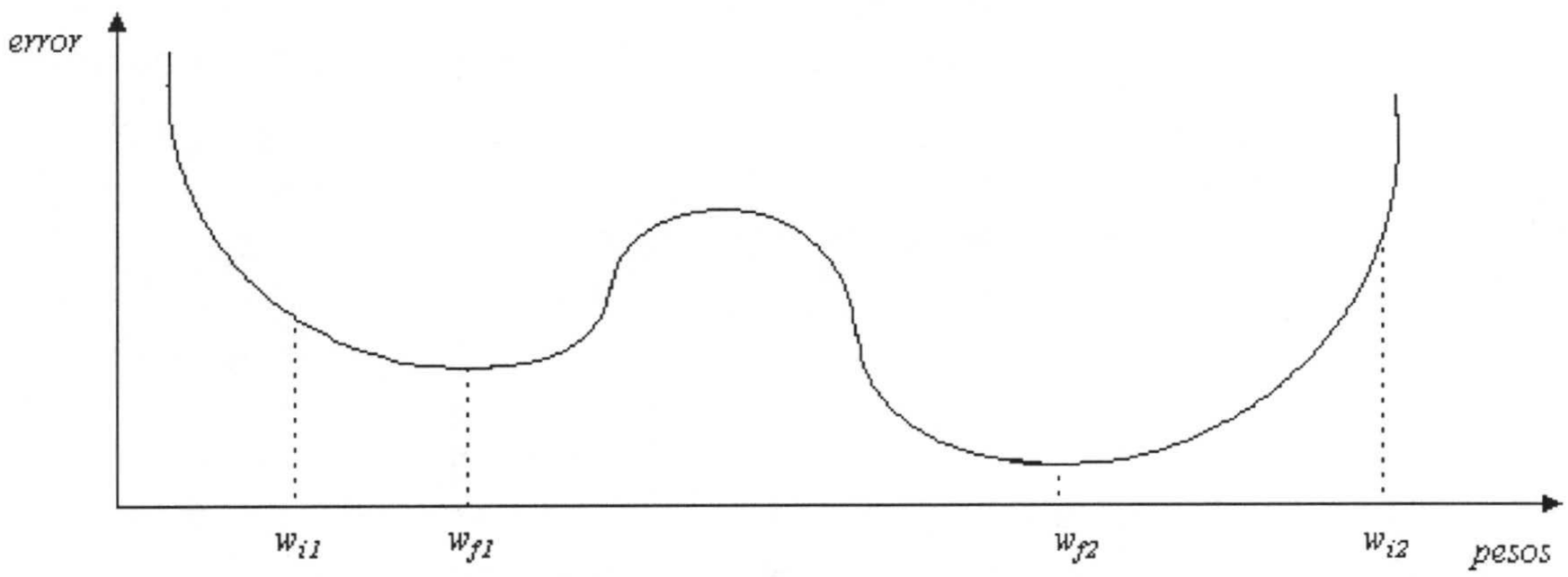

Figura 1-9: Espacio de búsqueda definido por conjunto de pesos de una red

En [Belew, 1991] se usa un algoritmo genético, que genera una búsqueda global que asigna a los pesos unos valores iniciales, y después se utiliza un algoritmo de backpropagation, que mediante una búsqueda local, afina la solución. En sus resultados se muestra que la aproximación genética/backpropagation es más eficiente que la utilización de un algoritmo genético o un backpropagation por separado.

1.5.2 Evolución de las arquitecturas

En el apartado anterior se ha descrito la técnica de evolución de los pesos de una red. En este caso se partía de que la arquitectura de esta red estaba prefijada y se mantenía fija durante toda la evolución. Ahora la idea es realizar un diseño evolutivo de la arquitectura de la red, lo que afectaría tanto a la conectividad de esta, como a las funciones de transferencia de cada nodo.

La arquitectura elegida o diseñada para una red neuronal es muy importante ya que va a definir las características de procesamiento de información de ésta y por tanto

sus posibilidades y eficiencia. Dada una tarea de aprendizaje, una red neuronal con pocas conexiones y nodos lineales no será capaz de realizarla debido a las limitaciones en la capacidad de aprendizaje. Por otro lado, si la red tiene muchas conexiones y nodos no lineales, puede que incurra en problemas de sobreentrenamiento y poca generalización.

Hoy en día, el diseño de la arquitectura de una red neuronal suele ser un trabajo heurístico, que depende tanto del conocimiento experto de la persona que lo realiza como de procesos de prueba y error. No existe un método sistemático para diseñar una arquitectura de red para una aplicación determinada. Una aproximación al diseño automático de arquitecturas son los algoritmos constructivos o destructivos [Fahlman, 1990][Frean, 1990]. En resumen, un algoritmo constructivo comienza con una red muy básica (con pocas capas ocultas, nodos y conexiones) y va añadiendo nuevas capas, nodos y conexiones conforme van haciendo falta y durante el proceso de entrenamiento. Por otra parte un algoritmo destructivo hace lo contrario, es decir, empieza con una red muy compleja y va eliminando capas, nodos o conexiones que considera innecesarios.

Al igual que en el caso de la evolución de los pesos de una red, el diseño evolutivo de la arquitectura de ésta presenta una serie de ventajas [Yao, 1999] con respecto a otras técnicas. Estas ventajas están relacionadas con una mejor adaptación a las características del espacio de búsqueda que definen las arquitecturas de las redes neuronales. La primera característica es que la superficie de este espacio es infinitamente grande ya que el número de conexiones y nodos no está limitado a priori. Por otro lado la superficie es no diferenciable ya que los cambios en el número de nodos o conexiones son discretos y pueden tener un efecto discontinuo en la eficiencia de la red. Además la superficie es compleja y existe el ruido, debido a la conversión genotipo-fenotipo, y a la dependencia con el método de evaluación usado. La superficie además da lugar a ambigüedades donde similares arquitecturas, puede tener diferentes eficiencias, y donde diferentes arquitecturas tienen una eficiencia similar.

1.5.2.1 Diseño del algoritmo evolutivo

Dentro del campo de la evolución de arquitecturas de redes neuronales [Yao, 1999], casi toda la investigación se refiere a la topología de la red, no habiéndose trabajado mucho en la evolución las funciones de transferencia de las neuronas.

En el diseño de un algoritmo evolutivo que obtenga una arquitectura de red adecuada existen dos aspectos principales, la representación del genotipo, y el proceso de evolución en sí.

En cuanto a la representación del genotipo, existen dos tipos principales [Yao, 1999]. Esta clasificación, que se describe a continuación, está basada en la cantidad de información que se va codificar.

- Esquema de codificación directa o codificación fuerte. En este tipo de esquema se codifican prácticamente todos los detalles de la arquitectura de la red. Así por ejemplo se podría utilizar una matriz C, de $n \times n$ para codificar una red con n nodos, donde c_{ij}, indicaría la presencia o ausencia de conexión entre el nodo i y el j. Con este esquema, es sencillo extraer un sistema de codificación mediante cadenas de bits. Además permite establecer fácilmente restricciones sobre el tipo de arquitecturas permitidas, mediante el establecimiento de ciertas reglas. Este esquema se adapta bien a arquitecturas pequeñas, o de pocos nodos y conexiones, pero no escala bien cuando las arquitecturas se hacen grandes. De todas formas, y partiendo de las restricciones de la red, también se puede disminuir el tamaño de las matrices.

- Esquema de codificación indirecta o codificación débil. Ahora sólo se codifican los parámetros más importantes con los que se trabaja como por ejemplo: el número de capas, número de nodos por capa, etc. El resto de los detalles los decide el algoritmo de entrenamiento, mediante reglas de desarrollo, que indican por ejemplo, qué nodos se unen y como lo hacen. Este sistema de codificación más compacto y más cercano a la naturaleza, según las neurociencias. Existen diferentes formas de realizar esta codificación como por ejemplo:

  - Representación paramétrica: Consistiría en la codificación de los principales parámetros que definen una red como por ejemplo el número de sus capas, neuronas por capas, etc.

  - Representación de reglas de desarrollo: En este caso se codificarían las reglas de desarrollo que se usan para construir arquitecturas.

  - Representaciones coevolutivas: En [Andersen, 1993] se usa una representación en la que cada individuo de una población representa una neurona de la red, en lugar de una red en sí. Para asignar el valor de adaptación, se utilizar técnicas de compartición de éste.

Con respecto al proceso de evolución, se instanciaría el Algoritmo 1-7, o algoritmo evolutivo general, definiendo todos los elementos que lo componen. Así, por ejemplo, un individuo será una red o un componente de ésta (neurona o conjunto de ellas) si se elige una estrategia coevolutiva. A continuación se formará la red correspondiente y se entrenará (si los pesos están sin determinar), para después medir

el error producido por la red. Lo normal será usar este error, para calcular el valor de adaptación de un individuo. Este esquema de funcionamiento se define en la Figura 1-10.

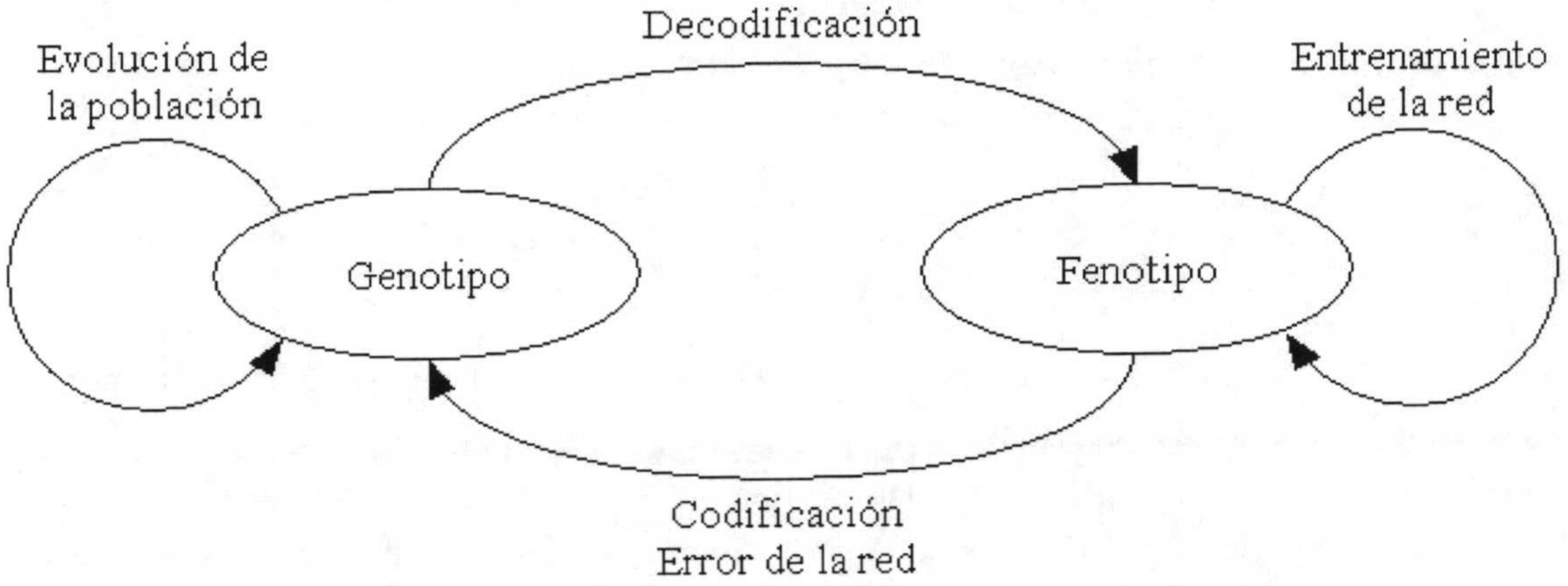

Figura 1-10: Ejemplo de un mecanismo evolutivo para el diseño de una red neuronal

En principio, la evolución de las arquitecturas de red es un proceso adaptativo donde no tiene porque existir un conocimiento a priori, pero su existencia en cualquiera de la etapas del algoritmo, facilita la eficiencia de la evolución. Con respecto a la definición de la función evaluación no existen tampoco limitaciones, en función de aspectos de continuidad, diferenciabilidad, etc. En esta función se suelen incluir sin mucha dificultad, parámetros relativos a eficiencia, complejidad, costes, etc. Todo esto va a mejorar los resultados obtenidos.

Uno de los problemas detectado en el diseño evolutivo de arquitecturas es el problema de la permutación, que ocurre cuando dos redes son funcionalmente equivalentes, pero tienen sus nodos dispuestos de diferente manera, y por lo tanto tienen dos genotipos diferentes. Esto implica se mantendrá en la población dos redes prácticamente iguales y que los descendientes de estas redes normalmente tendrán un valor de adaptación bajo. Esta es una de las razones por las que en muchos algoritmos evolutivos de este tipo, se omite un operador de cruce

1.5.3 Evolución de las reglas de aprendizaje

Un algoritmo de entrenamiento puede presentar diferentes resultados dependiendo de la arquitectura a la que se aplique. De hecho, el diseño de un algoritmo de entrenamiento, o de las reglas de aprendizaje, usadas para ajustar los pesos de una red, dependen de la arquitectura elegida para esta. El problema, es que

este diseño se suele tornar complicado cuando hay poco conocimiento a priori, sobre la arquitectura de red y la aplicación determinada.

Surge pues la necesidad de desarrollar un método automático y sistemático para adaptar las reglas de aprendizaje, a la arquitectura de la red y aplicación elegidas. En la actualidad el diseño de estas reglas de aprendizaje es realizado por expertos (humanos) en la materia realizando una serie de suposiciones que suplen la falta de conocimiento antes mencionada, pero que muchas veces son erróneas. Lo ideal sería que el sistema adaptara sus reglas de aprendizaje en función de la red con la que se trabaje y a la aplicación asignada. Así pues parece normal pensar en técnicas evolutivas para conseguir esta adaptación.

El trabajo en este campo [Yao, 1999], está en sus primeras etapas, pero pone de manifiesto que las relaciones entre evolución y aprendizaje son bastante complejas. Incluso, yendo más allá, permitiría ayudar a entender como la creatividad aparece en sistemas artificiales, lo que puede servir de antesala para modelar el proceso creativo en sistemas biológicos.

Se pueden distinguir las siguientes aproximaciones en la evolución de reglas de aprendizaje:

- Evolución de los parámetros de las reglas: En esta primera aproximación se propone un ajuste evolutivo de los parámetros que rigen las reglas y no de las reglas en su totalidad. En [Belew, 1991] se usa una técnica evolutiva para encontrar los mejores parámetros de las reglas de aprendizaje, manteniendo predefinida y fija la arquitectura de la red. Por otro lado en [Harp, 1989] se hace evolucionar tantos los parámetros de las reglas como la arquitectura de la red.

- Evolución de las reglas de aprendizaje: En este caso se produce la adaptación de las reglas en sí. La adaptación de las reglas de aprendizaje hace que estas reglas y la red en general tengan una conducta dinámica. El problema es cómo codificar esta conducta dinámica de la red en genotipos estáticos. El intentar desarrollar un esquema de representación que pueda recoger esto es impracticable, por lo que es necesario imponer una serie de restricciones. Un ejemplo de estas restricciones es que la forma de adaptación sea a partir de información local o que las reglas de aprendizaje sean iguales para todas las conexiones de una red. Una vez prefijadas estas restricciones hay que decidir que parámetros van a intervenir en las reglas, decidir la representación de los genotipos y decidir las características del algoritmo evolutivo. Ejemplos de estos algoritmo se encuentran en [Yao, 1999].

1.6 Sistemas de Lógica Difusa

1.6.1 Introducción

La Lógica Difusa (Fuzzy) nace a mediados de los años 60, cuando Lotfi Zadeh publica su trabajo "Conjuntos Difusos" [Zadeh, 1965], por el cual se le considera el padre de este campo científico.

Este paradigma propone un nuevo modelo de inferencia más cercano al lenguaje natural y al pensamiento humano. Para esto, provee de un medio efectivo para capturar la natural inexactitud y aproximación del mundo real.

En el lenguaje humano, es normal, que se manejen sentencias como: "Juan es alto", "La velocidad es baja", "El agua está caliente", etc. Estas sentencias se caracterizan porque los adjetivos empleados para distintos elementos, (alto, baja, caliente…) no tienen una cuantificación exacta y son muchas veces subjetivos al sujeto que los emite.

Es frecuente, también que las personas empleen reglas del tipo "Si la velocidad es alta Entonces presiona moderadamente el freno", "Si la temperatura es baja Entonces eleva levemente la potencia del calentador", etc. Estos ejemplos, captan el funcionamiento de la lógica difusa, y revelan que para su implementación se necesitan elementos como:

- Etiquetas lingüísticas frente a valores numéricos de una variable (muy caliente frente a 40ºC)
- Cuantificación de variables lingüísticas (u_1 puede tener número finito de etiquetas lingüísticas dentro de un rango: muy bajo, bajo, …, muy alto) mediante funciones de pertenencia difusas
- Conexiones lógicas para variables lingüísticas (Y, O, ...)
- Implicaciones (SI A ENTONCES B)
- Combinaciones de reglas, etc.

A continuación se describirán, más formalmente, todos los elementos necesarios para implementar un sistema de que nos permita realizar razonamiento utilizando lógica difusa, más conocido como Sistema de Lógica Difusa (SLD). Para esto se describen los fundamentos de la lógica difusa, las bases del razonamiento difuso y conceptos sobre defuzzificación.

1.6.2 Fundamentos de la lógica difusa

1.6.2.1 Conjuntos nítidos y conjuntos difusos

Un *conjunto nítido* es un conjunto clásico que puede ser definido mediante el listado de todos sus elementos o mediante el establecimiento de una condición que exprese los elementos que pertenecen a ese conjunto, por ejemplo:

$$A = \{ x \mid x > 5 \} \qquad (1\text{-}41)$$

Para este conjunto se puede definir una *función de pertenencia cero-uno*, también llamada función característica, denotada por $\mu_A(x)$ de modo que:

$$\mu_A(x) = \begin{cases} 0 & si \quad x \in A \\ 1 & si \quad x \notin A \end{cases} \qquad (1\text{-}42)$$

Dado un conjunto nítido, y cualquier elemento se sabe si este pertenece o no a al conjunto.

En un *conjunto difuso* la transición entre pertenecer o no pertenecer a él, es gradual, y esto es lo que caracteriza a este tipo de conjuntos. Esta transición progresiva va a venir definida por su función de pertenencia.

Un *conjunto difuso D* se puede definir en un universo de discurso *U* caracterizado por una función de pertenencia que toma valores en el intervalo [0, 1] y se denota como $\mu_F(u)$: $U \rightarrow [0,1]$. Como se observa un conjunto difuso se puede ver como una generalización de un conjunto nítido, donde la función de pertenencia sólo tomaba los valores 0 ó 1. El conjunto difuso *F* en *U* puede ser representado por un conjunto de pares ordenados compuestos por el elemento genérico *u* y su función de pertenencia.

$$F = \{ (u, \mu_F(u)) \mid u \in U \} \qquad (1\text{-}43)$$

Cuando *U* es continuo, *F* puede ser escrito concisamente como:

$$F = \int_U \mu_F(u) / u \qquad (1\text{-}44)$$

si *U* es discreto, el conjunto difuso *F* es representado como:

$$F = \sum_{i=1}^{n} \mu_F(u)/u \qquad (1\text{-}45)$$

1.6.2.2 Operaciones teóricas de conjuntos

Dados A y B dos conjuntos difusos en U con funciones de pertenencia $\mu_A(x)$ y $\mu_B(x)$ respectivamente, las operaciones de unión, intersección y complemento se definen a partir de sus funciones de pertenencia.

T-conormas	
Suma algebraica	$x \oplus y = x + y$
Suma limitada	$x \oplus y = \min(1, x + y)$
Suma drástica	$x \oplus y = \begin{cases} x & si \quad y = 0 \\ y & si \quad x = 0 \\ 1 & si \quad x, y > 0 \end{cases}$
T-normas	
Producto algebraico	$x \otimes y = x \cdot y$
Producto limitado	$x \otimes y = \max(0, x + y - 1)$
Producto drástico	$x \otimes y = \begin{cases} x & si \quad y = 1 \\ y & si \quad x = 1 \\ 0 & si \quad x, y < 1 \end{cases}$

Tabla 1-2: Distintas T-conormas y T-normas.

Unión

La unión de A y B, se nota como $A \cup B$, y se define con respecto a la función de pertenencia siguiente, para todos los $u \in U$.

$$\mu_{A \cup B}(u) = \max\{\mu_A(u), \mu_B(u)\} \qquad (1\text{-}46)$$

Intuitivamente se podría ver como el menor conjunto difuso que contenga tanto a A cómo a B. En general la unión se suele representar como:

$$\mu_{A\cup B}(u) \quad = \quad \mu_A(u) \oplus \mu_B(u) \qquad (1\text{-}47)$$

donde a $\oplus$, se le denomina operador *t-conorma* ó *s-norma*. Existen varios tipos de t-conormas, además del máximo en (1-46), estas se muestran en la Tabla 1-2.

Intersección

La intersección de A y B, $A \cap B$, se define, para todos los $u \in U$, con respecto a la función de pertenencia:

$$\mu_{A\cap B}(u) \quad = \quad \min\{\, \mu_A(u)\,,\, \mu_B(u)\,\} \qquad (1\text{-}48)$$

Esta operación se podría entender como el mayor conjunto difuso que es contenido tanto por A, como por B. También aquí se puede generalizar la intersección y expresarla como:

$$\mu_{A\cap B}(u) \quad = \quad \mu_A(u) \otimes \mu_B(u) \qquad (1\text{-}49)$$

La intersección ahora se ha expresado en función de una *t-norma*. Además de la t-norma mínimo de (1-48), existen otras t-normas reflejadas en la Tabla 1-2.

Complemento

El complemento del conjunto difuso A, $\bar{A}$, para todos lo $u \in U$, se define con respecto a la función de pertenencia:

$$\mu_{\bar{A}}(u) \quad = \quad 1 - \mu_A(u) \qquad (1\text{-}50)$$

1.6.2.3 Funciones de pertenencia

Tal y como se ha definido una función de pertenencia, $\mu_A(x)$, del elemento x en el conjunto A, devuelve un valor en el intervalo [0, 1] que refleja el grado de pertenencia de x a A. Existen diferentes tipos de funciones de pertenencia dependiendo de su forma.

Triangular

Una función triangular se define con respecto a tres parámetros {a, b, c} que determinan las coordenadas en x de sus esquinas:

$$triángulo(x;a,b,c) \quad = \quad \max\left(\min\left(\frac{x-a}{b-a},\ \frac{c-x}{c-b}\right), 0\right) \qquad (1\text{-}51)$$

Trapezoide

Se puede establecer una función trapezoide en función de sus cuatro parámetros $\{a, b, c, d\}$

$$trapezoide(x;a,b,c,d) \quad = \quad \max\left(\min\left(\frac{x-a}{b-a},\ 1,\ \frac{d-x}{d-c}\right), 0\right) \qquad (1\text{-}52)$$

Estas dos funciones son las más usadas debido a la simplicidad de sus fórmulas y a su eficiencia computacional, sobre todo en aplicaciones en tiempo real. Sin embargo, y por estar compuestas de segmentos no son de las más graduales que se pueden utilizar, para conseguir esta característica se usan las siguientes funciones:

Gausiana

Una función de pertenencia gausiana se caracteriza por un centro y un radio $\{c, \sigma\}$:

$$gausiana(x;c,\sigma) \quad = \quad e^{\{-((x-c)/\sigma)^2\}} \qquad (1\text{-}53)$$

Campana generalizada

Como caso más general de función gradual podemos definir una campana con tres parámetros $\{a, b, c\}$ que definen la amplitud de la campana $2a$, su centro c, y la pendiente lateral $-b/2a$:

$$campana(x;a,b,c) \quad = \quad \frac{1}{1+|\,(x-c)/a)\,|^{2b}} \qquad (1\text{-}54)$$

1.6.2.4 Producto cartesiano, relaciones difusas y composición de relaciones

Dados los conjuntos difusos $A_1, \ldots, A_n$ en los universos de discurso $U_1, \ldots, U_n$ respectivamente, el producto cartesiano de $A_1, \ldots, A_n$, es un conjunto difuso en el espacio $U_1 \times \ldots \times U_n$, con la función de pertenencia

$$\mu_{A_1 \times \ldots \times A_n}(u_1,\ldots,u_n) \quad = \quad \min\left\{\mu_{A_1}(u_1),\ldots,\mu_{A_n}(u_n)\right\} \qquad (1\text{-}55)$$

ó

$$\mu_{A_1\times\ldots\times A_n}(u_1,\ldots,u_n) = \mu_{A_1}(u_1)\cdot\mu_{A_2}(u_2)\ldots,\mu_{A_n}(u_n) \quad (1\text{-}56)$$

Una relación difusa representa un grado de presencia o ausencia de asociación, interacción o interconexión entre dos o más conjuntos difusos. Algunos ejemplos de relaciones difusas son: x es mucho más grande que y, y está muy cerca de x, z es mucho más verde que y, … Es tipo de relaciones juegan un papel muy importante en un SLD.

Sean U y V dos universos de discurso, una relación difusa, $R(U, V)$ es un conjunto difuso en el espacio del producto $U \times V$, y se caracteriza por una función de pertenencia $\mu_R(x,y)$ donde $x \in U$ e $y \in V$, es decir:

$$R(U,V) = \left\{\left((x,y),\mu_R(x,y)\right)\middle|\ (x,y)\in UxV\right\} \quad (1\text{-}57)$$

La generalización de una relación difusa a un espacio del producto cartesiano n-dimensional es directa.

Como las relaciones se pueden considerar como conjuntos difusos en un espacio producto, se pueden definir operaciones algebraicas y teóricas de conjunto para ellas, usando los operadores unión, intersección y complemento, definidos anteriormente. Dadas $R(x,y)$ y $S(x,y)$, dos relaciones difusas en el mismo espacio producto $U \times V$, la intersección y unión de R y S, que son composiciones de dos relaciones, se definen como:

$$\mu_{R\cap S}(x,y) = \mu_R(x,y)\otimes\mu_S(x,y) \quad (1\text{-}58)$$

$$\mu_{R\cup S}(x,y) = \mu_R(x,y)\oplus\mu_S(x,y) \quad (1\text{-}59)$$

donde $\otimes$es una t-norma y $\oplus$ es una t-conorma.

Se considera ahora la composición de relaciones de diferentes espacios productos que comparten un conjunto común, por ejemplo $R(U, V)$ y $S(V, W)$ (x es más pequeño que y e y está ceca de z). Asociado a la relación difusa R tenemos su función de pertenencia $\mu_R(x, y)$, donde $\mu_R(x, y) \in [0, 1]$ y asociada a la relación S tenemos su función de pertenencia $\mu_S(x, y)$, donde $\mu_S(x, y) \in [0, 1]$. Cuando R y S se definen en universos de discurso discretos, entonces la composición difusa de R y S, se escribe $R \circ S$. Esta relación puede se descrita por un diagrama sagital, en que cada línea es etiquetada por el valor de su función de pertenencia, o por una matriz relacional difusa, en la que cada elemento es un número real en el intervalo [0, 1]. Una forma de expresar matemáticamente la función de pertenencia de esta relación es mediante la conocida *composición sup-star* de R y S:

$$\mu_{R\circ S}(x,z) \quad = \quad \sup_{y\in V}\left[\mu_R(x,y)\otimes\mu_S(y,z)\right] \qquad (1\text{-}60)$$

cuando U, V y W son universos de discurso discretos entonces la operación sup es el máximo. Como t-normas se suelen usar el mínimo y el producto.

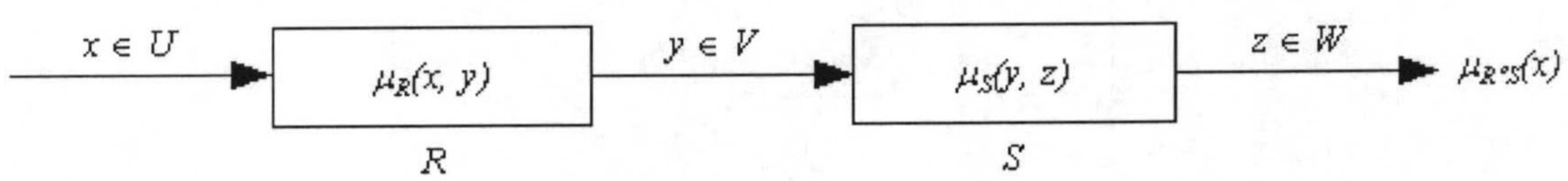

Figura 1-11: Composición sup-star

En la Figura 1-11 se muestra un diagrama interpretativo para la composición sup-star, que sugiere una manera simple para componer relaciones difusas más complicadas. Si se considera la relación difusa R tan sólo como un conjunto difuso entonces $\mu_R(x, y)$ se convierte en $\mu_R(x)$, por ejemplo "y es medianamente grande y z es menor que y". Entonces $V = U$ y la Figura 1-11 se transforma en la Figura 1-12, y representa un conjunto difuso puede activar una relación difusa. Esta una de las bases del funcionamiento del SLD.

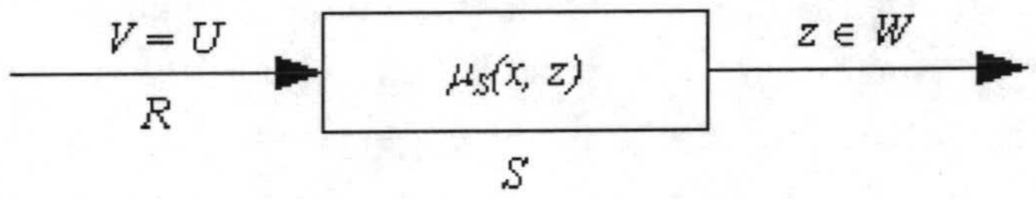

Figura 1-12: Composición sup-star, interpretando la primera relación como un conjunto difuso

Teniendo en cuenta que $V = U$:

$$\sup_{y\in V}\left[\mu_R(x,y)\otimes\mu_S(y,z)\right] \quad = \quad \sup_{x\in U}\left[\mu_R(x)\otimes\mu_S(x,z)\right] \qquad (1\text{-}61)$$

que es sólo función de la variable de salida z, por lo que podemos simplificar la notación $\mu_{R\circ S}(x, z)$ a $\mu_{R\circ S}(x)$, así que cuando R es sólo un conjunto difuso:

$$\mu_{R\circ S}(z) \quad = \quad \sup_{x\in U}\left[\mu_R(x)\otimes\mu_S(x,z)\right] \qquad (1\text{-}62)$$

1.6.2.5 Variables lingüísticas

Una variable lingüística se caracteriza por la quíntupla (x, $T(x)$, U, G, M), donde x es el nombre de la variable; $T(x)$ es el conjunto de términos o etiquetas de x; G es la

regla sintáctica para generar los valores de x; y M es una regla semántica para asociar cada valor con su significado.

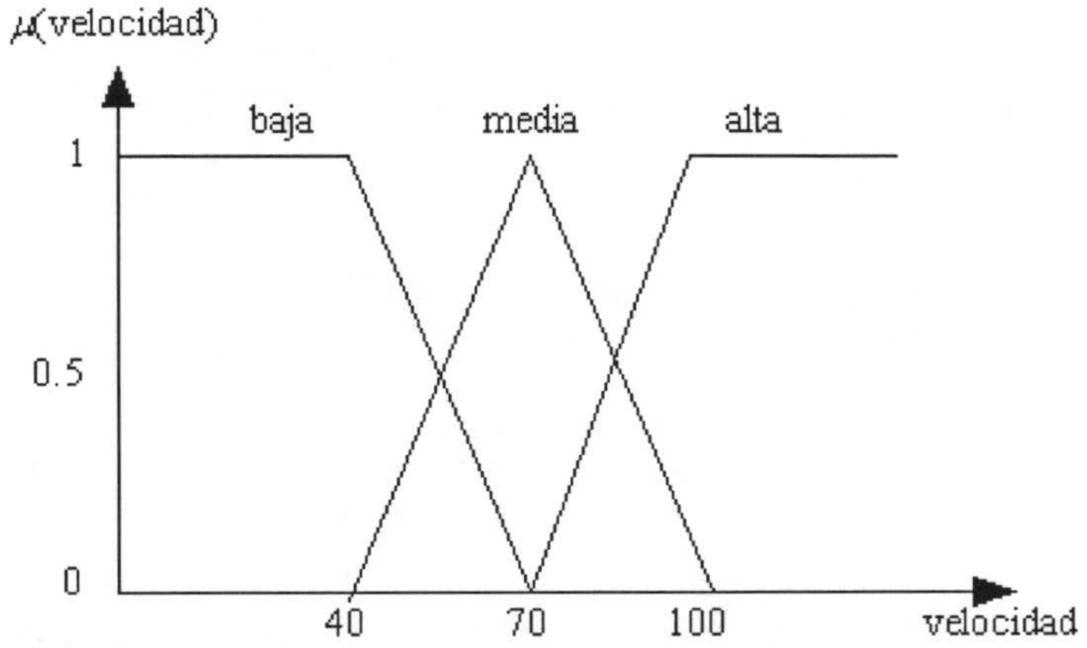

Figura 1-13: Representación de las etiquetas lingüisticas de la variable *velocidad*.

Por ejemplo, si *velocidad* es interpretada como una variable lingüística, entonces el conjunto de etiquetas T(velocidad) podría ser:

$$T(\text{velocidad}) = \{\text{baja, media, alta}\} \qquad (1\text{-}63)$$

donde cada etiqueta en T(velocidad) se caracteriza por un conjunto difuso en un universo de discurso $U = [0, 150]$. Se puede interpretar *baja* como "una velocidad por debajo 40 km / h", *media* como "una velocidad en torno a 70 km / h y alta como "una velocidad superior a 100 km / h". Estas etiquetas se pueden caracterizar como conjuntos difusos cuyas funciones de pertenencia se representan en la Figura 1-13.

1.6.3 Lógica difusa y razonamiento difuso

Como se ha descrito hasta ahora, la lógica difusa toma prestados muchos conceptos de la lógica tradicional o clásica. Uno de estos conceptos es el que se conoce como *reglas difusas*, que se notan de la siguiente manera:

$$\text{SI x es A ENTONCES y es B} \qquad (1\text{-}64)$$

donde A y B son etiquetas lingüísticas definidas en universos de discursos X e Y respectivamente. A la parte "x es A" se le llama antecedente o premisa, mientras que a "y es B" se le llama consecuente o conclusión. Las reglas difusas son ampliamente utilizadas en las expresiones lingüísticas cotidianas: "SI la carretera es sinuosa ENTONCES la conducción es peligrosa"; "SI la velocidad es media ENTONCES presionar ligeramente el freno"...

El siguiente paso es formalizar una regla difusa "SI x es A ENTONCES y es B", la cual se representa abreviadamente cómo: $A \rightarrow B$. En esencia la expresión describe una relación entre dos variables x e y; esto sugiere que una regla difusa, se puede definir en términos de una relación difusa R en el espacio producto $X \times Y$. Además una regla difusa se va a caracterizar por una función de pertenencia $\mu_{A\rightarrow B}(x, y)$ donde $\mu_{A\rightarrow B}(x, y) \in [0, 1]$. $\mu_{A\rightarrow B}$(x, y) mide el grado de verdad de la relación de implicación entre x e y. Ejemplos de interpretaciones de esta función de pertenencia son:

$$\mu_{A\rightarrow B}(x,y) = 1 - \min\left[\mu_A(x), 1 - \mu_B(y)\right] \quad (1\text{-}65)$$

$$\mu_{A\rightarrow B}(x,y) = \max\left[1 - \mu_A(x), \mu_B(y)\right] \quad (1\text{-}66)$$

$$\mu_{A\rightarrow B}(x,y) = 1 - \mu_A(x)\left(1 - \mu_B(y)\right) \quad (1\text{-}67)$$

El *razonamiento aproximado* (o *razonamiento difuso*) es un procedimiento de inferencia usado para obtener conclusiones a partir de un conjunto de reglas difusas y una o más condiciones. Para esto se usa principalmente una generalización de la regla de inferencia de la lógica clásica *Modus Ponens*, a la que se le denomina *Modus Ponens generalizado* y cuyo funcionamiento se describe a continuación.

premisa 1: x es A^*

premisa 2: SI x es A ENTONCES y es B (1-68)

consecuencia: y es B^*

Esta regla de inferencia está basada en la regla de composición de inferencia para razonamiento aproximado aportada por (Zadeh, 1973). La diferencia con la regla de inferencia clásica estriba en la inclusión de los conjuntos difusos A^* y B^*. Donde el conjunto difuso A^* no es necesariamente igual al conjunto difuso antecedente A, y B^* no es necesariamente igual al conjunto difuso consecuente B. Esto implica que en lógica difusa, una regla se "dispara" en función de la similaridad entre la primera premisa y el antecedente de la regla. Mientras que el resultado del disparo de la regla es un consecuente que tendrá cierto grado de similaridad con el consecuente de la regla.

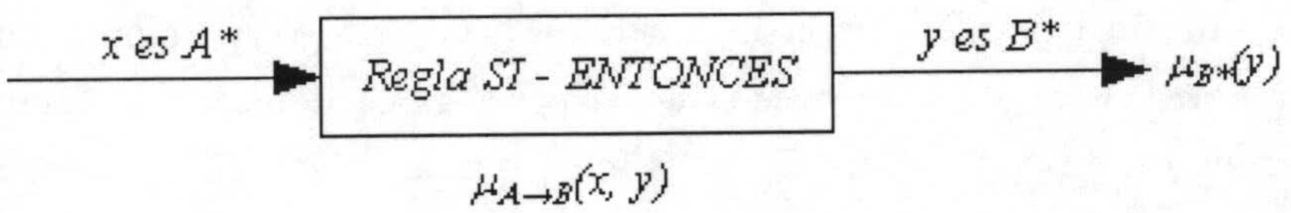

Figura 1-14: Interpretación de Modus Ponens generalizado

La Figura 1-14 representa una interpretación para el Modus Ponens generalizado que como se puede observar coincide con la Figura 1-12. De esto se puede concluir que el Modus Ponens generalizado es una composición difusa donde interviene una relación difusa.

$$B^* = A^* \circ R = A^* \circ (A \rightarrow B) \qquad (1\text{-}69)$$

De la misma manera y para obtener $\mu_{B^*}(y)$ usaremos la composición sup-star, teniendo en cuenta las figuras y haciendo las transformaciones simbólicas apropiadas en (1-62):

$$\mu_{B^*}(y) = \sup_{x \in A} \left[\mu_{A^*}(x) \otimes \mu_{A \rightarrow B}(x, y) \right] \qquad (1\text{-}70)$$

Hay diversas formas de implementar esa implicación. Entre las más comunes destaca la de (Mandani, 1974):

$$\mu_{A \rightarrow B}(x, y) = \min\left[\mu_A(x), \mu_B(y) \right] \qquad (1\text{-}71)$$

o la de (Larsen, 1980)

$$\mu_{A \rightarrow B}(x, y) = \mu_A(x) \mu_B(y) \qquad (1\text{-}72)$$

lo que se puede expresar como una t-norma:

$$\mu_{A \rightarrow B}(x, y) = \mu_A(x) \otimes \mu_B(y) \qquad (1\text{-}73)$$

Para trabajar de una forma general más general, se expresa (1-70) de la siguiente forma:

$$\mu_{B^*}(y) = \bigoplus_{x \in A} \left[\mu_{A^*}(x) \otimes \mu_{A \rightarrow B}(x, y) \right] \qquad (1\text{-}74)$$

Vamos a considerar ahora diversos casos prácticos que se nos pueden presentar:

Un solo antecedente y un solo consecuente: En este caso se puede transformar (1-74) en la siguiente ecuación:

$$\mu_{B^*}(y) = \bigoplus_{x \in A} \left[\mu_{A^*}(x) \otimes \mu_A(x) \right] \otimes \mu_B(y) = w \otimes \mu_B(y) \qquad (1\text{-}75)$$

Lo que implica que se buscará primero el grado de solapamiento máximo w entre $\mu_{A^*}(x) \otimes \mu_A(x)$ (el área sombreada en la parte antecedente de la Figura 1-15); entonces la función de pertenencia de B^* es igual que la función de pertenencia de B recortada por w, mostrada en la parte consecuente de la Figura 1-15.

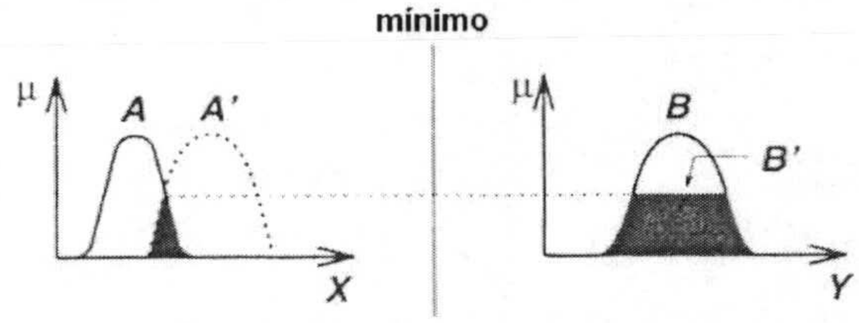

Figura 1-15: Razonamiento con un antecedente y un consecuente

Dos antecedentes y un consecuente: en este caso la regla se escribiría cómo "SI x es A Y y es B ENTONCES z es C". El problema a resolver mediante razonamiento aproximado se expresa:

premisa 1: x es A^* Y y es B^*

premisa 2: SI x es A Y y es B ENTONCES z es C (1-76)

---

consecuencia: z es C^*

La regla difusa de la premisa 2 se puede expresar como $A \times B \rightarrow C$. Intuitivamente esta relación difusa se puede expresar con la siguiente función de pertenencia:

$$\mu_{A\times B\rightarrow C}(x,y,z) \quad = \quad \mu_A(x) \otimes \mu_B(y) \otimes \mu_C(z) \tag{1-77}$$

Y C^* se puede expresar como:

$$C^* = (A^* \times B^*) \circ (A \times B \rightarrow C) \tag{1-78}$$

entonces:

$$\begin{aligned} \mu_{C^*}(y) \quad &= \quad \underset{x,y}{\oplus}\left[\mu_{A^*}(x) \otimes \mu_{B^*}(y)\right] \otimes \left[\mu_A(x) \otimes \mu_B(y) \otimes \mu_C(z)\right] \\ &= \underset{x,y}{\oplus}\left[\mu_{A^*}(x) \otimes \mu_{B^*}(y) \otimes \mu_A(x) \otimes \mu_B(y)\right] \otimes \mu_C(z) \\ &= \left\{\underset{x\in A}{\oplus}\left[\mu_{A^*}(x) \otimes \mu_A(x)\right]\right\} \otimes \left\{\underset{y\in B}{\oplus}\left[\mu_{B^*}(y) \otimes \mu_B(y)\right]\right\} \otimes \mu_C(z) \\ &= w_1 \otimes w_2 \otimes \mu_C(z) \end{aligned} \tag{1-79}$$

donde w_1 es el grado de máximo solapamiento entre A^* y A; w_2 es el grado de solapamiento entre B^* y B; y $w_1 \otimes w_2$ es el grado de disparo de la regla para el que se escoge una t-norma mínimo. En la Figura 1-16 se muestra una interpretación gráfica de este procedimiento, donde la función de pertenencia de C^* es igual la función de

pertenencia de C recortada por el grado de disparo de la regla. La generalización a más de dos antecedentes es directa.

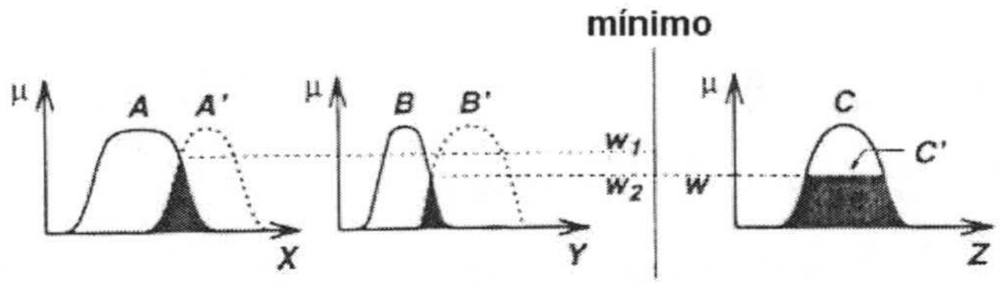

Figura 1-16: Razonamiento con dos antecedentes y un consecuente

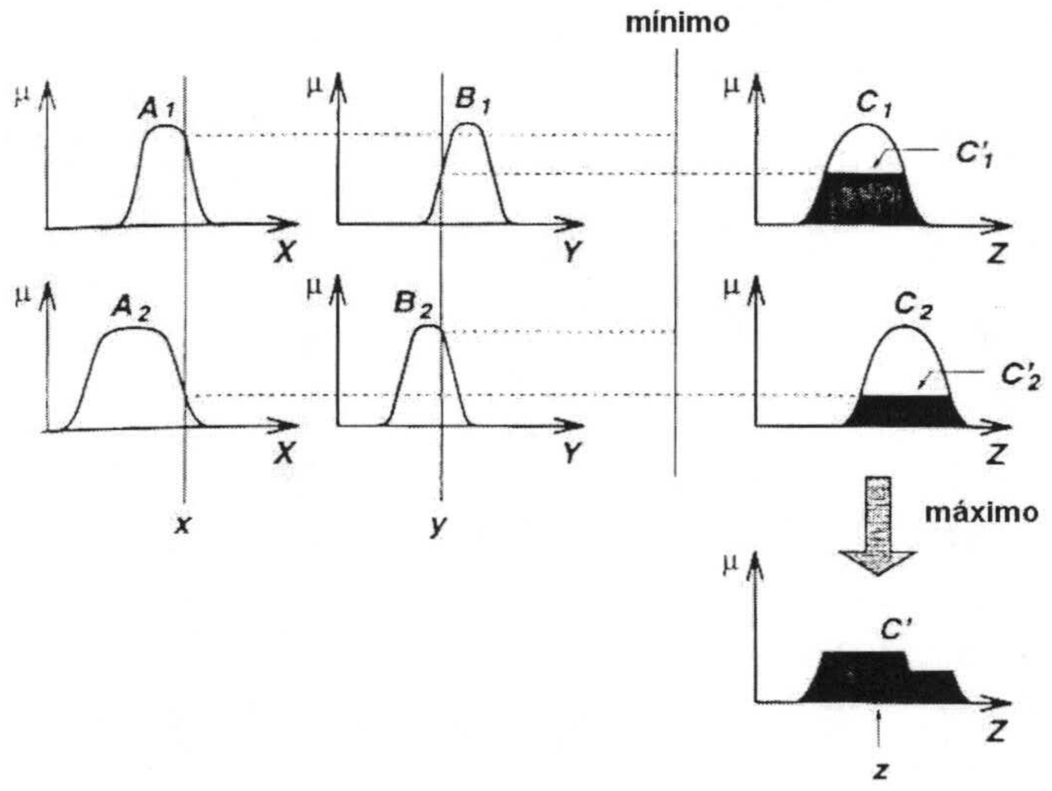

Figura 1-17: Razonamiento difuso (máximo-minimo) para múltiples reglas con múltiples antecedentes

Múltiples reglas con múltiples antecedentes: la interpretación de múltiples reglas se toma normalmente como la unión de relaciones difusas correspondientes a las reglas difusas. Por ejemplo dados los siguientes hechos y reglas:

premisa 1: x es A^* Y y es B^*

premisa 2: SI x es A_1 Y y es B_1 ENTONCES z es C_1 (1-80)

premisa 3: SI x es A_2 Y y es B_2 ENTONCES z es C_2

consecuencia: z es C^*

En este caso, y para obtener C^* se realizan las siguientes transformaciones aplicando la ley distributiva sobre el operador $\cup$.

$$
\begin{aligned}
C^* &= (A^* \times B^*) \circ \left[(A_1 \times B_1 \rightarrow C_1) \cup (A_2 \times B_2 \rightarrow C_2)\right] \\
&= \left[(A^* \times B^*) \circ (A_1 \times B_1 \rightarrow C_1)\right] \cup \left[(A^* \times B^*) \circ (A_2 \times B_2 \rightarrow C_2)\right] \qquad (1\text{-}81) \\
&= C_1^* \cup C_2^*
\end{aligned}
$$

donde C_1^* y C_2^* son los conjuntos difusos para la regla 1 y 2, respectivamente. En la Figura 1-17, se muestra gráficamente el razonamiento difuso para múltiples reglas con múltiples antecedentes, utilizando como t-conorma el máximo, y como t-norma el mínimo.

Al tipo de razonamiento explicado se le denomina de (Mandani, 1975). En este modelo de razonamiento, se puede utilizar la operación producto en la implicación en vez de mínimo. Este modelo se muestra en la Figura 1-18.

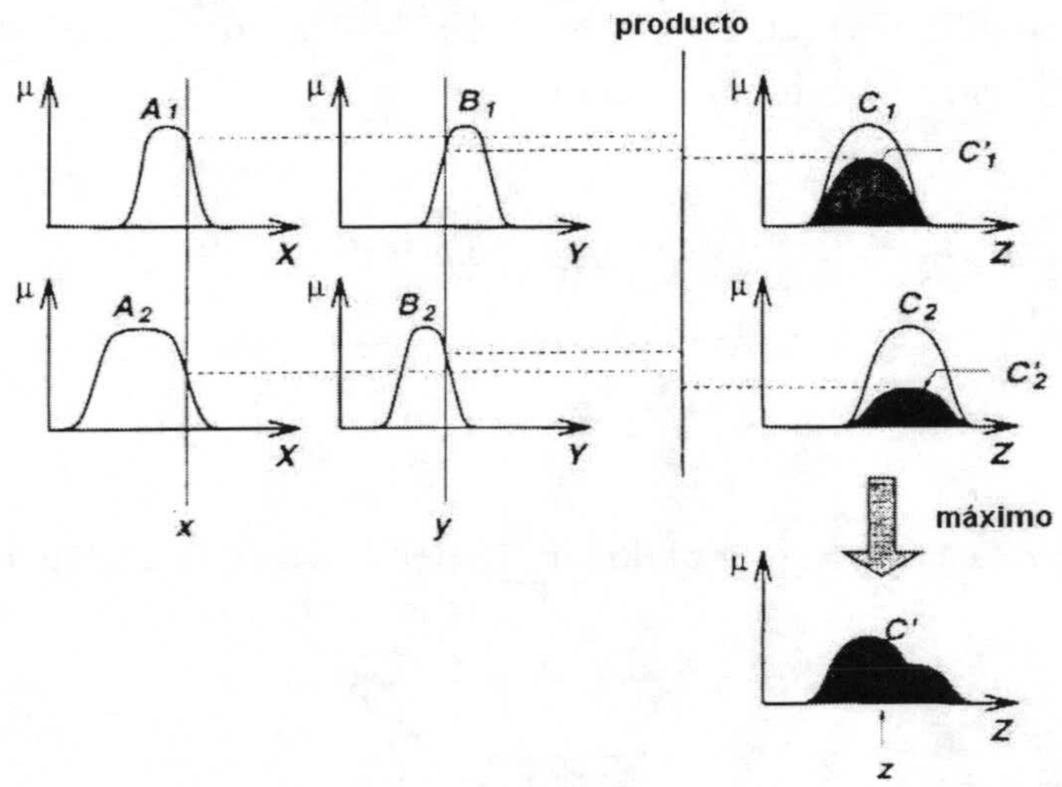

Figura 1-18: Razonamiento difuso (máximo-producto) para múltiples reglas con múltiples antecedentes

Existen otros modelos de razonamiento como el de (Sugeno, 1988), o el de (Tsukamoto, 1979) donde la salida se pone en función de la entrada.

1.6.4 Defuzzificación

Básicamente la defuzzifcación es una transformación de un espacio de decisión difuso de un universo de discurso en un espacio de decisión no difuso. La defuzzificación sólo se utiliza, cuando se necesita una salida nítida del sistema.

Sin embargo, no existe un procedimiento sistemático para elegir una estrategia de defuzzificación. Las más utilizadas son:

- *Criterio del máximo*. Este método produce el punto en que la salida del sistema alcanza su mayor valor, la primera vez (ó la última).

- *Media del máximo*. Con esta técnica se genera un punto que es la media de todos los puntos cuya función de pertenencia alcanza el máximo. En el caso de que tengamos un universo de discurso discreto, esto se representa:

$$z_0 = \sum_{i=1}^{l} w_i / l \qquad (1\text{-}82)$$

donde z_0 es la salida l es el número de puntos que alcanzan el valor máximo y w_i es cada uno de esos puntos.

- *Centro del área*. Es una de las más usadas y produce punto centro de gravedad de la salida del sistema. En el caso de un universo de discurso discreto esto se representa como:

$$z_0 = \frac{\sum_{i=1}^{n} \mu(w_i) w_i}{\sum_{i=1}^{n} \mu(w_i)} \qquad (1\text{-}83)$$

donde n es el número total de puntos del universo de discurso.

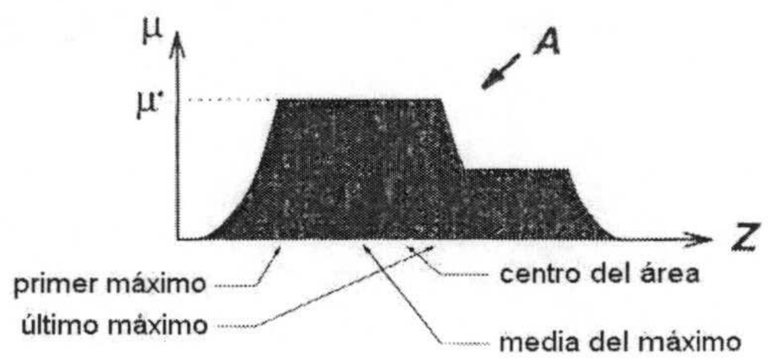

Figura 1-19: Estrategias de defuzzificación

En la Figura 1-19 se muestran distintas estrategias de defuzzificación.

1.6.5 Sistemas de lógica difusa

Un Sistema de Lógica Difusa (SLD) (Figura 1-20) va a obtener unas determinadas salidas a partir de unas entradas dadas. Esta transformación entradas-salidas puede ser expresada como $y = f(x)$.

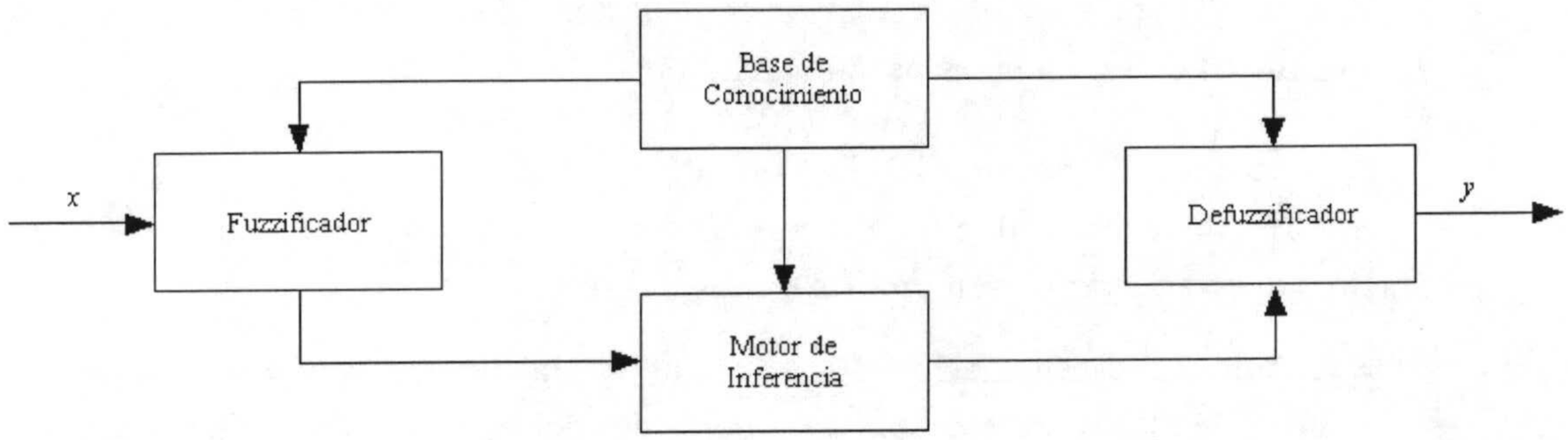

Figura 1-20: Sistema de lógica difusa

Un SLD consta de cuatro elementos: un *fuzzificador*, una *base de conocimieto*, un *motor de inferencia* y un *defuzzificador*.

- El *fuzzificador* tiene como entrada valores nítidos, los cuales evalúa, para realizar su transformación a valores del universo de discurso correspondiente. Esto implica implementar la función de difusión en sí, que consiste en convertir los valores de entrada en valores lingüísticos que pueden ser vistos como etiquetas de conjuntos difusos, y que más tarde se van utilizar para activar reglas.

- La *base de conocimiento* representa el conocimiento del dominio de la aplicación y está compuesta por la *base de datos* y la *base de reglas difusas*.

  - La *base de datos* contiene las definiciones necesarias usadas para definir tanto las reglas difusas como la manipulación de datos difusos en el sistema.

  - La *base de reglas difusas* va a caracterizar el conjunto de objetivos comprendidos por las decisiones a tomar para solventar un problema dado. El conjunto de reglas suele ser suministrado por expertos en el campo en el que esté trabajando el SLD. Estas reglas suelen tener la sintaxis de las sentencias SI – ENTONCES (SI u_1 es alta Y u_2 es muy baja ENTONCES establecer v a un valor bajo).

- El *motor de inferencia* de un SLD relaciona conjuntos difusos en conjuntos difusos, manejando la manera en que las reglas son combinadas. De igual manera que los humanos usan diferentes tipos de procedimientos de inferencia, existen diferentes procedimientos inferenciales de lógica difusa.

- En muchas aplicaciones se necesita que la salida del SLD sea un valor nítido. Esta función la realiza el *defuzzificador*, que convierte conjuntos difusos de salida en estos valores nítidos.

A la hora de diseñar un SLD hay que llevar a cabo una serie de tareas o tomar una serie de decisiones que configuran las características finales de este. Los parámetros que van a definir en un SLD son:

- Elegir la estrategia de fuzzificación y la interpretación del operador fuzzificador. Esta elección va a depender del tipo de datos a fuzzificar, del rango en el que están definidos, de si existe ruido, etc (Chien, 1990).

- En cuanto a la base de datos hay que decidir conceptos que van a caracterizar las reglas de control difusas y la manipulación de datos difusos en un SLD. Estos conceptos son definidos subjetivamente y se basan en la experiencia. Los más importantes son:

  - Discretización / normalización del universo de discurso. En general la representación depende de la naturaleza del universo de discurso. Un universo de discurso puede ser continuo o discreto. Si el universo es continuo puede ser discretizado o puede ser normalizado teniendo en cuenta necesidades sobre todo de representación de la información.

  - Partición difusa de los espacios de entrada y salida. Una variable lingüística en un antecedente de una regla constituye un espacio de entrada, mientras que una variable lingüística en un consecuente de una regla constituye un espacio de salida. En general, una variable lingüística se asocia con un conjunto de etiquetas lingüísticas, definiéndose cada una de las etiquetas en el mismo universo de discurso. La partición difusa determina cuantas etiquetas lingüísticas existen en el conjunto de etiquetas lingüísticas. Este número va a determinar la granuralidad de las reglas y de las decisiones que se toman con el SLD.

  - Completitud. Un SLD cumple esta propiedad si para cualquier estado del sistema se puede inferir una acción a tomar. Para que se cumpla esta propiedad tenemos que tener en cuenta tanto el diseño de la base de datos, por ejemplo que cubra bien los espacios de entrada y salida, como el conjunto de reglas difusas, de modo que no existan condiciones que no se contemplen.

  - Funciones de pertenencia de conjuntos difusos. Dependiendo de si el universo de discurso es discreto o continuo, la pertenencia a este se

puede definir de una manera numérica o funcional respectivamente. Si la función de pertenencia es numérica se indica para cada elemento del conjunto su grado de pertenencia. Si la función de pertenencia es funcional esta se especifica mediante una función que dará el grado de pertenencia para cada elemento del conjunto.

- El conjunto de reglas difusas de un SLD va a representar el conocimiento experto en un campo. Entre los elementos a decidir destaca la fuente y derivación de las reglas difusas. Existen diversas fuentes para derivar reglas difusas. Entre estas destacan la derivación de reglas a partir de conocimiento experto (Mandani, 1975), o el auto-aprendizaje mediante diversas técnicas de las reglas (Procyk, 1979). Aunque suele ser normal la hibridación de estas técnicas para la consecución del conjunto final de reglas.

Las estrategias de defuzzificación y la interpretación del operador de defuzzificación.

2. Redes de Funciones de Base Radial

En este capítulo se hará una descripción detallada de las Redes de Funciones de Base Radial, ya que se han elegido como el elemento computacional, que en el nivel más inferior se encarga de realizar el procesamiento de los datos de entrada, para obtener las correspondientes salidas.

Esta descripción comienza, detallando las bases estructurales y de funcionamiento de estas redes, continuando con un estudio de las propiedades que las caracterizan. Dado que nuestro objetivo es diseñar y optimizar este tipo de redes, en una segunda parte, se hará un repaso de los métodos que se han utilizado en la bibliografía para este fin. En esta exposición de métodos, no sólo se incluirán aquellos que están más relacionados con el soft computing o tengan más características bioinspiradas, sino que también se detallarán métodos más numéricos, para de esta manera, tener una visión más completa de este campo y poder desarrollar, un algoritmo de diseño y optimización más potente.

2.1 Introducción

Una Función de Base Radial (RBF), ϕ_i, se puede expresar como:

$$\phi_i(\bar{x}) = \phi(\| \bar{x} - \bar{c}_i \| / d_i) \qquad (2\text{-}1)$$

Ésta se caracteriza porque su salida es simétrica, y se incrementa (o decrementa) de forma monótona, con respecto a un centro, $\bar{c}_i$, donde de forma general $\bar{c}_i \in \Re^n$. $d_i \in \Re$ es un factor de escala para $\| \bar{x}_i - \bar{c}_i \|$ o radio y como $\| \cdot \|$ se suele escoger la norma Euclídea en $\Re^n$. Para determinar la forma de una RBF existen distintas posibilidades, las más usuales se encuentran en la Tabla 2-1.

$\phi(x)$	Nombre
X	Función lineal
x^3	Función cúbica
$\text{Exp}(-x^2)$	Función Gausiana
$x^2\ log(x)$	Función Spline
$(x^2 + 1)^{1/2}$	Función Multicuadrática
$(x^2 + 1)^{-1/2}$	Función inversa multicuadrática

Tabla 2-1: Posibles elecciones para ϕ

En [Rojas, 1997] se hace un estudio sobre distintas las distintas formas que puede adoptar una RBF a la hora de formar parte de una Red de Funciones de Base Radial. En este trabajo se concluye que una RBF con forma gausiana es una de las mejores elecciones. La Figura 2-1 muestra la forma de las funciones base gausiana y multicuadrática, con los parámetros $c = 0$ y $r = 1$.

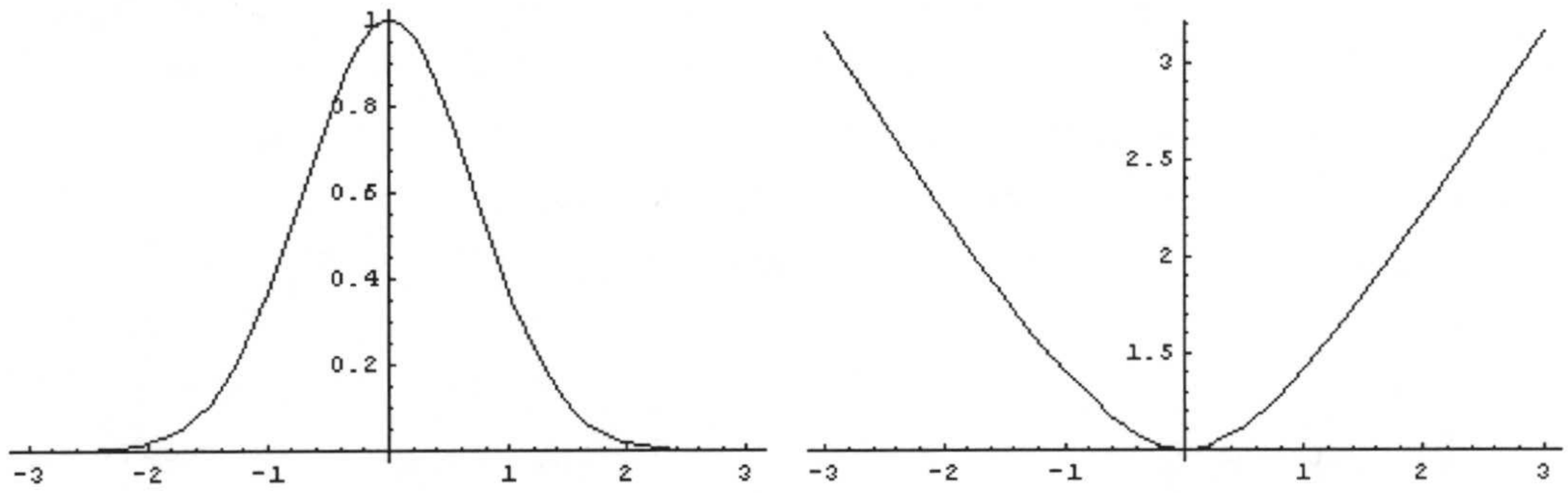

Figura 2-1: RBFs gausiana y mutlticuadrática con $c = 0$ y $r = 1$.

Una Red de Funciones de Base Radial (RBFN) es una red hacia adelante (feedforward network) que contiene tres capas: las entradas, la capa oculta, el/los nodo/s de salida. Cada neurona de la capa oculta se caracteriza por que su salida viene dada por una RBF. La activación de estas funciones es proporcional a la cercanía, medida por la norma euclídea, entre el patrón de entrada y centro c_i correspondiente. Si el patrón de entrada está cerca del centro de una RBF, la salida de está se acercará a 1, en caso contrario se acercará a 0.

A partir de [Broomhead, 1988] y cuando una RBFN se utiliza en interpolación, aproximación de funciones o predicción de series temporales, consta de un solo nodo de salida que implementa una combinación lineal de las RBFs, por las que está compuesta. Por tanto la salida de la red se expresará de la siguiente manera:

$$f(\bar{x}) = \sum_{j=1}^{m} w_j \phi_j(\bar{x}) \qquad (2\text{-}2)$$

donde w_j es el peso asociado a la neurona j. Este tipo de RBFN, se muestra en la Figura 2-2, y será la que utilizaremos en nuestro trabajo. En cuanto a la elección concreta de la forma para las RBFs, la más utilizada en estos casos es la gausiana tal y como se describirá a continuación.

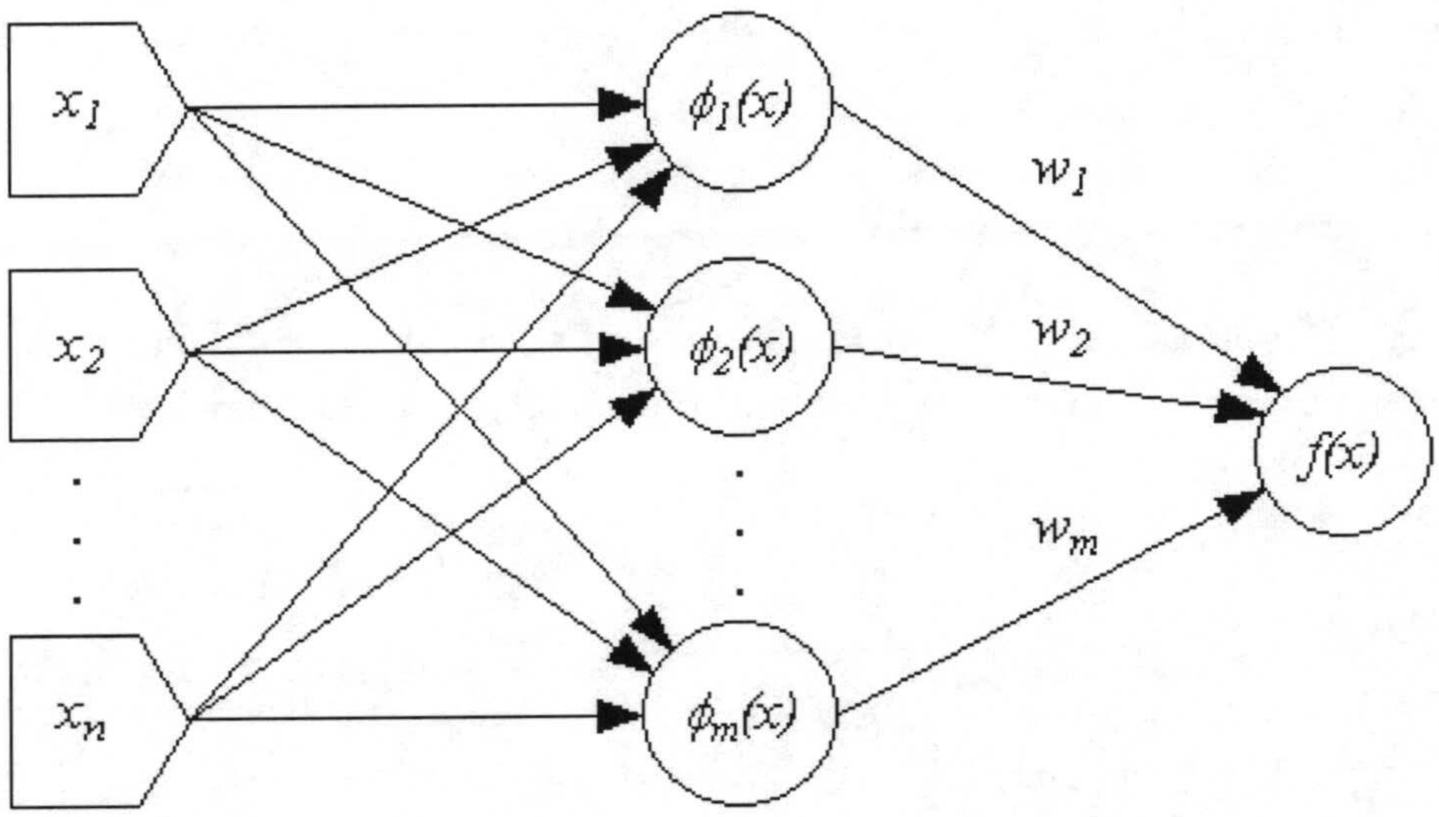

Figura 2-2: Red de Funciones de Base Radial

Haciendo un poco de historia, uno de los primeros trabajos que utiliza la superposición de funciones gausianas se realiza a principios de los sesenta por [Megdasy, 1961]. Después se demuestran las propiedades de estas a la hora de interpolar y aproximar funciones [Powell, 1985][Broomhead, 1988]. Más tarde, se renueva el interés por estas al hacerse las primeras implementaciones de redes neuronales [Lee, 1988][Moody, 1989][Poggio, 1990]. Parte de este interés se debía a las analogías con los *campos receptivos localizados* encontrados en muchas estructuras biológicas. Esto motivó el trabajo con funciones de forma de campana (gausianas), que se activaban en la vecindad de un punto.

Las propiedades de aproximador universal de la ecuación (2-2) se demuestran en [Kowalski, 1990][Park, 1991][Park, 1993]. En la actualidad las RBFNs, son uno de los modelos de red más usados en problemas de aproximación y clasificación, gracias a sus características.

2.2 Problemas clásicos a resolver con las RBFNs

Los primeros problemas a los que se aplican las Redes de Funciones de Base Radial son la interpolación exacta y la aproximación de funciones.

2.2.1 Interpolación exacta

El problema de la interpolación exacta se formula de la siguiente manera: dado un conjunto de vectores de entrada $\bar{x} \in \Re^d$, y su salida $y \in \Re$, encontrar una función continua que:

$$h(\bar{x}_i) = y_i \quad i = 1,\ldots,n \tag{2-3}$$

La solución a este problema utilizando RBFs, consistiría en elegir un conjunto de n funciones base, centradas en los n puntos de datos, y usar la fórmula:

$$h(\bar{x}) = \sum_{i=1}^{n} w_i \phi_i(\| \bar{x} - \bar{x}_i \|) \quad i = 1,\ldots,n \tag{2-4}$$

Para conseguir una solución exacta, será necesario resolver las n ecuaciones lineales del sistema siguiente:

$$\begin{bmatrix} \phi_{11} & \phi_{12} & \cdots & \phi_{1n} \\ \phi_{21} & \phi_{22} & \cdots & \phi_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ \phi_{n1} & \phi_{n2} & \cdots & \phi_{nn} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{2-5}$$

donde se determinará el valor de los pesos y

$$\phi_{ij} = \phi(\| \bar{x}_i - \bar{x}_j \|) \quad i, j = 1,\ldots,n \tag{2-6}$$

La ecuación (2-5) se expresa de forma compacta como:

$$\Phi \bar{w} = \bar{y} \tag{2-7}$$

Según los trabajos [Michelle, 1986][Light, 1992], existen muchos tipos de formas para la RBFs, como gausianas, splines o inversas multicuadráticas, para las que se puede resolver la ecuación siguiente y de esta manera conseguir una interpolación exacta. El vector de pesos se determinaría:

$$\overline{w} = \Phi^{-1}\overline{y} \qquad (2\text{-}8)$$

También estos estudios muestran que muchas características de la función interpolada son relativamente insensibles a la forma de la función de la RBF. Sin embargo una de las formas más utilizadas son las gausianas, ya que su respuesta localizada ayuda en el proceso de aprendizaje, además de demostrar otras características beneficiosas [Lee, 1988][Rojas, 1997].

2.2.2 Aproximación funcional

Hay situaciones en las que, sin embargo, no es recomendable realizar una interpolación exacta. Una de estas situaciones, sería por ejemplo cuando existe ruido, entonces la función de interpolación que pasara por los datos, podría conllevar sobre-entrenamiento y por tanto una pobre generalización. Otra razón, es que para garantizar una interpolación exacta, el número de funciones base requerido es igual al número de patrones en el conjunto de entrenamiento. Por lo tanto, si el número de patrones es muy alto, el establecer la función de interpolación puede ser muy costoso.

En los trabajos [Broomhead, 1988][Moody, 1989] se obtiene un modelo, basado en RBFs, para la aproximación y generalización funcional modificando el procedimiento de interpolación exacta. Este modelo obtiene una aproximación suave a los datos, a partir de un número reducido de funciones base. El número de funciones base depende de la complejidad de la función a aproximar antes que del tamaño del conjunto de datos. Las principales modificaciones inciden en los siguientes aspectos:

- Número de funciones base: El número de funciones base m es normalmente menor que n.

- Centros de las funciones base: Los centros de las funciones se determinan como parte de un proceso de entrenamiento, en vez de en función de un conjunto de patrones de entrada.

- Radios de las funciones base: El proceso de entrenamiento también adaptará el valor del radio de cada función base, antes que asignar el mismo radio a todas las funciones base.

- Introducción de un parámetro de adaptación (bias): De forma opcional se puede introducir un parámetro de adaptación para compensar la diferencia, si existe, entre la media de los datos de activación de las funciones base y los correspondientes valores a aproximar.

Aplicando estas modificaciones al proceso de interpolación exacta, y si se empieza introduciendo el parámetro de adaptación, la salida de la red quedaría:

$$f(x) = w_0 + \sum_{j=1}^{m} w_j \phi_j(x) \qquad (2\text{-}9)$$

El parámetro de adaptación w_0, se puede incorporar en la sumatoria de la ecuación anterior, introduciendo una función base extra ϕ_0, y estableciendo su salida al valor constante 1, con lo que quedaría:

$$f(x) = \sum_{j=0}^{m} w_j \phi_j(x) \qquad (2\text{-}10)$$

En [Kowalski, 1990] se prueba que la superposición lineal de funciones base gausianas, donde los radios se traten como parámetros ajustables, es un *aproximador universal*. En [Park, 1991][Park, 1993], se muestra que sólo con leves restricciones en la forma de las funciones base, estas propiedades en la aproximación se mantienen. Sin embargo en los trabajos, no se ofrecen datos ni sobre el número de funciones base a introducir, ni sobre procedimientos para construir la red. De todas formas, si se ofrecen fundamentos teóricos en los que apoyar las aplicaciones prácticas. Así por ejemplo, según [Poggio, 1990], una RBFN posee la propiedad de *mejor aproximador*, lo que quiere decir que existe un único conjunto de parámetros que consigue la mejor aproximación de una función determinada. Esta propiedad, por ejemplo, no la muestran los perceptrones multicapa.

2.3 Diseño de Redes de Funciones de Base Radial

2.3.1 Conceptos y técnicas básicas

Según la ecuación (2-2), para diseñar una RBFN, se tendrán que determinar varios parámetros como el tipo de funciones base ϕ, el número de estas, m, su radio, d_i, la localización de su vector de centros, $\bar{c}_i$, y los pesos, w_i, asociados a las neuronas.

En nuestro caso, como tipo de funciones base se usarán las gausianas, por sus características descritas en los apartados anteriores, viniendo también predefinido en muchas ocasiones, el número de estas funciones, que va a intervenir en la red. Esto implica que, normalmente y cuando se diseña una RBFN, se aporten estrategias para el establecimiento de los conjuntos de radios, centros y pesos de una red.

Como punto de partida, y para diseñar una RBFN, se pueden utilizar técnicas de gradiente descendiente. De la misma manera que se usan en el diseño de otras redes, éstas realizan la actualización de parámetros de forma iterativa, cada vez que se muestra un patrón del conjunto de entrenamiento. Para esto se define una función de costo a minimizar:

$$e = \sum_{i=0}^{n} (y_i - f(x_i))^2 \qquad (2\text{-}11)$$

Así, y si se usan funciones gausianas, se pueden utilizar las siguientes ecuaciones [Ghost, 1992], para actualizar los conjuntos de parámetros:

$$\Delta w_j = \alpha_1 (y_i - f(\bar{x}_i))\phi(\bar{x}_i) \qquad (2\text{-}12)$$

$$\Delta \bar{c}_j = \frac{\alpha_2 (y_i - f(\bar{x}_i))\phi(\bar{x}_i) \parallel \bar{x}_i - \bar{c}_j \parallel w_j}{d_j^2} \qquad (2\text{-}13)$$

$$\Delta d_j = \frac{\alpha_3 (y_i - f(\bar{x}_i))\phi(\bar{x}_i) \parallel \bar{x}_i - \bar{c}_j \parallel^2 w_j}{d_j^3} \qquad (2\text{-}14)$$

donde α_1, α_2 y α_3 definen la variación a realizar o la velocidad del aprendizaje. Estas ecuaciones representan una particularización del algoritmo LMS para las RBFNs.

2.3.2 Cálculo de los parámetros de una RBFN mediante métodos numéricos

En este apartado se van a presentar algunos de los métodos más usados para la determinación de los pesos de una RBFN mediante métodos matemáticos. Hay que tener en cuenta que una RBFN es una red simple en cuanto a su estructura, ya que posee una sola capa oculta, y que su salida se puede expresar de forma lineal. Esto implica, que el problema de la determinación de un conjunto de parámetros de una RBFN pueda ser enunciado muchas veces, como la resolución de un sistema de ecuaciones.

El siguiente apartado describe una serie de métodos utilizados para la determinación de los pesos de una red, en la que el resto de los parámetros ya están fijados. Por último se referenciarán los métodos de regularización.

2.3.2.1 Cálculo de los pesos de una RBFN mediante métodos numéricos

Estos métodos se usarían cuando estuvieran ya establecidos, mediante algún otro algoritmo de diseño, los parámetros que definen a las distintas RBFs. En este caso se parte de que el problema consiste encontrar los pesos en la expresión (2-2). Nuestro problema se puede expresar como:

$$\overline{y} = \Phi \overline{w} \qquad (2\text{-}15)$$

si se tienen en cuenta todas las muestras del conjunto de patrones.

Esto implica que los métodos que se utilizarán a continuación para determinar el vector $\overline{w}$, son métodos usados en la literatura para la resolución de sistemas de ecuaciones lineales. Concretamente los métodos que más se han utilizado para esta tarea en el campo del diseño de las RBFN son la *descomposición de Cholesky*, la *descomposición en valores singulares* (SVD) y el *método de los mínimos cuadrados ortogonales* (OLS).

Descomposición de Cholesky

La descomposición de Cholesky [Golub, 1996] [Press, 1994] es el más rápido de los tres métodos para resolver sistemas de ecuaciones lineales pero solo se puede aplicar a sistemas descritos por una matriz cuadrada, simétrica y definida positiva. El sistema de ecuaciones a resolver (2-15) habría que transformarlo antes ya que la matriz Φ no cumple ninguna de las restricciones anteriores.

Cuando se realizan las transformaciones, la matriz de activación debe verificar otra serie de condiciones, cuyo cumplimiento depende de las características de las RBFs que haya definidas. Así las RBFs deberían estar colocadas de forma que se cubra todo el espacio de entrada y ser linealmente independientes. Esto implica que redes con funciones base "mal colocadas", bien porque alguna no se active con ningún punto de entrenamiento o porque exista demasiado solapamiento entre las mismas, pueden producir matrices de activación singulares que el método de Cholesky sería incapaz de resolver, o bien casi singulares, para las que el método anterior proporcionaría una solución indeseable.

Descomposición en valores singulares

La descomposición en valores singulares (Singular Value Descomposition, SVD) [Golub, 1996] es un primer método para solucionar el problema de que existan RBFs mal colocacadas, algo que por otro lado es normal que ocurra durante el diseño de una RBFN. Como se ha mencionado en estos casos se producen matrices de activación singualares o casi singulares. Ocurre que en la matriz de activación cuando dos funciones son casi idénticas, se producen dos columnas prácticamente iguales,

mientras que si una función base no se activa para casi ningún punto se producirá una columna casi nula en Φ.

La descomposición en valores singulares facilita una solución para cualquier sistema de ecuaciones, con la que se obtiene una reducción del error. Además el método muestra que, si se elimina de la red alguna función base cuyo valor singular asociado tuviera una magnitud pequeña el error de aproximación no se vería prácticamente afectado, y al ser la red más pequeña se gana simplicidad y se pueden conseguir mejoras en la aproximación. Este suele ser otro de los usos del método SVD, es decir la identificación en una red, de funciones base que aportan poco al funcionamiento general de ésta. El inconveniente que plantea es que la forma en que selecciona las funciones base más importantes de la red no tiene en cuenta la salida esperada del sistema para cada vector de entrada, de forma que si en una red existieran dos funciones base muy solapadas, probablemente sacrificaría a una de las dos sin importar la influencia que dicha modificación tenga en el error de aproximación de la red.

El método de los mínimos cuadrados ortogonales (OLS)

Otra posibilidad para resolver el sistema de ecuaciones planteando es el algoritmo OLS. Este método, al igual que el anterior, es aplicable aunque las funciones base estén "mal colocadas", es decir que pueda existir alguna RBF que no se active con ningún punto de entrenamiento o que exista demasiado solapamiento entre algunas de éstas. Además no tiene tampoco el inconveniente del método anterior. Este inconveniente consistía en que al intentar detectar funciones base poco útiles para la red y encontrarse con funciones base muy solapadas, se eliminaría seguramente alguna de éstas, aunque ésta tuviera importancia en el error de aproximación obtenido por la red.

El método OLS es un algoritmo iterativo que selecciona en cada iteración la columna de la matriz de activación Φ que más contribuya a la disminución del error de aproximación al modelo. Así este va transformado las columnas de la matriz de activación Φ en un conjunto de vectores ortogonales. A estas columnas se les aplica otra serie de operaciones hasta obtener unos coeficientes que se denominan radios de reducción del error. Este radio proporciona una forma simple de escoger un conjunto de regresores importantes de forma directa. Dichos regresores se corresponden con las funciones base más importantes de la red y si se decide reducir el modelo a r funciones base ($r < m$), se deben seleccionar las r funciones base con un mayor radio de reducción del error.

Posteriormente se han presentado hibridaciones del método OLS con técnicas de regularización [Orr, 1993] produciendo el algoritmo ROLS [Chen, 1996], e hibridaciones de este último con algoritmos genéticos [Chen, 1999].

2.3.2.2 La técnica de la regularización

La *regularización* [Orr, 1993] es una técnica que favorece unas soluciones sobre otras añadiendo un término de penalización a la función de costo o en nuestro caso a la función de error a minimizar. Dependiendo de la función de costo y del término de penalización añadido, se definen diferentes técnicas para solucionar problemas. Entre estas se podría destacar por su importancia la *regresión límite* (*ridge regression*).

Una de las herramientas más importantes de las que se utilizan en la técnica de la regularización es la *matriz de proyección P*, que se define como:

$$P = I_n - \Phi A^{-1} \Phi^T \qquad (2\text{-}16)$$

donde I_n es la matriz identidad con dimensión n y $A = \Phi^T \Phi$.

P es una matriz que representa la proyección de vectores de un espacio n-dimensional, donde n era el número de patrones del conjunto de entrenamiento en un subespacio m-dimensional donde m es el conjunto de funciones base. En función de esta matriz se van a poder definir muchos conceptos importantes, como por ejemplo los relacionados con la reducción del error, con los efectos de la inclusión o eliminación de funciones base, etc.

Regresión límite

La técnica de la *regresión límite* (*ridge regression*) [Orr, 1996] utiliza el concepto de regularización consistente en incluir un término de penalización en la función de error a minimizar. Así, la función de costo a minimizar que se define es:

$$C = \sum_{i=1}^{n} (y_i - f(\bar{x}_i)) + \lambda \sum_{j=1}^{m} w_j^2 \qquad (2\text{-}17)$$

donde λ es el parámetro de regularización. Con la inclusión de este término está claro que se penalizarán redes con pesos muy altos, también dependiendo del valor de λ. El objetivo final será obtener una red con una salida suave, al tener los pesos de ésta un valor bajo o moderado. Con estas premisas y usando la matriz de proyección se define el vector de pesos óptimo como:

$$\overline{w} = A^{-1} \Phi^T \overline{y} \qquad (2\text{-}18)$$

siendo ahora A:

$$A = \Phi^T \Phi + \lambda I_n \qquad (2\text{-}19)$$

2.3.3 Algoritmos de Clustering

Los *algoritmos de clustering* se desarrollaron inicialmente como técnica de clasificación [Hartigan, 1975] aunque después se han aplicado en los campos de los sistemas difusos y redes neuronales artificiales [Bezdeck, 1981][Karayiannis, 1997].

Un algoritmo tradicional de clustering va a intentar dividir un conjunto de vectores $X = \{ \bar{x}_i : i = 1, \ldots, n\}$, en un número c, determinado a priori, de subconjuntos o grupos que van a proporcionar una partición de Voronoi $P = \{P_1, \ldots, P_c\}$, donde para cada grupo (cluster) P_j, se definirá un vector representante o prototipo $\bar{p}_j$. Los vectores se agruparán en estos grupos en función de la definición de alguna o varias características.

Dentro del campo de diseño de una RBFN, estos algoritmos se encuadrarían dentro de una fase de preproceso de datos de entrada de la red. Concretamente, el objetivo de esta fase sería intentar revelar la estructura interna de este conjunto de datos de entrada, agrupando o identificando grupos de estos datos que tengan una o varias características comunes. Una vez realizada esta fase, lo normal sería insertar una RBF en el centro geométrico de cada uno de los grupos identificados. La determinación de las RBFs insertadas termina con el establecimiento de un radio para de cada una ellas, relacionado con el espacio ocupado por el grupo en el que se encuentran ubicadas.

De todas formas es importante resaltar que los algoritmos de clustering no se suelen contemplar como estrategia única en el diseño de una RBFN. Así lo normal es que estos formen parte de una fase inicial a la hora de diseñar una red de este tipo. Tal y como se ha comentado, esta primera fase se suele corresponder con un estudio preliminar de las características del conjunto de datos, que sirve para realizar una primera determinación o inicialización de los centros, e incluso radios de las RBFs. De este modo, el proceso de diseño se complementaría con otras estrategias que refinan mucho más los parámetros finales de las RBFs en particular y de la RBFN en general.

Dentro de los algoritmos de clustering podemos establecer una clasificación en la que se distinguirían los siguientes tipos:

- *Algoritmos de clustering no supervisados*: en este tipo de algoritmos la característica común es que el mecanismo de aprendizaje (asignación de

grupos) no tiene en cuenta ninguna información aportada por la/s variable/s de salida.

- *Algoritmos de clustering supervisados*: se introducen para mejorar a los algoritmos de clustering no supervisados y se caracterizan porque van a tener en cuenta la información de que suministren la/s variable/s de salida. Esta mejora, parece razonable ya que las redes neuronales pretenden modelar las relaciones entre las entradas y salidas de los conjuntos de entrenamiento. La forma utilizar esta información que aportan la/s variable/s depende del algoritmo en sí, como se describirá a continuación.

Seguidamente se describirán algunos de los algoritmos de clustering más importantes. De estos algoritmos el *algoritmo de las c medias*, el *algoritmo de las c medias difuso* y el *algoritmo ELBG*, se clasifican como algoritmos de clustering no supervisados. Por otro lado el *algoritmo de clustering difuso condicional*, el *algoritmo de estimación de grupos alternantes* y el *algoritmo de clustering para aproximación de funciones* se clasifican como algoritmos de clustering supervisados.

2.3.3.1 Algoritmo de las *c* medias

Una primera aproximación a los algoritmos de clustering, fue el propuesto inicialmente por [Duda, 1973] y conocido como *algoritmo de las c medias* (*c means algorithm*). Esta técnica, mostrada en el Algoritmo 2-1, realiza una partición del conjunto de vectores en *c* grupos. En este caso, se establece como característica de referencia para formar los grupos la distancia entre los vectores. Así, se define como función objetivo a minimizar *J*:

$$J = \sum_{j=1}^{c} \sum_{i=1}^{n} \| \bar{x}_i - \bar{p}_j \| \qquad (2\text{-}20)$$

donde $\bar{x}_1, \ldots, \bar{x}_n$ son los vectores agrupar, y $\bar{p}_1, \ldots, \bar{p}_c$ serían los representantes o prototipos de cada partición o grupo $P_1, \ldots, P_c$, $\| \cdot \|$ sería la distancia euclídea. Además se define una función de pertenencia $\mu_{P_j}(\bar{x}_i)$ del vector de entrada $\bar{x}_i$, al grupo representado por el prototipo $\bar{p}_j$, que está definida en el rango de salida {0, 1}, como:

$$\mu_{P_j}(\bar{x}_i) = \begin{cases} 1 & \| \bar{x}_i - \bar{p}_j \|^2 < \| \bar{x}_i - \bar{p}_l \|^2 \quad \forall l \neq j \\ 0 & \textit{en otro caso} \end{cases} \qquad (2\text{-}21)$$

Esta función de pertenencia produce una partición de Voronoi P del conjunto de vectores de entrada de la forma:

$$P = \bigcup_{j=1}^{m} P_j \quad con \quad C \cap C = \emptyset \quad \forall j \neq i \qquad (2\text{-}22)$$

donde *Pj* se define como:

$$P_j = \{\overline{x}_i : \mu_{P_j}(\overline{x}_i) = 1\} \qquad (2\text{-}23)$$

Como se deduce cada uno de los vectores de entrada sólo va a ser asignado a un único grupo.

Así una partición se va a poder describir mediante la matriz de partición U de dimensión $c \times n$, cuyos elementos son las funciones de pertenencia $\mu_{P_j}(\overline{x}_i)$. De este modo se cumple que:

$$U = \left\{ \mu \in \{0,1\} \mid \sum_{i=1}^{c} \mu_{P_i}(\overline{x}_k) = 1 \ \forall k \quad y \quad 0 < \sum_{k=1}^{n} \mu_{P_i}(\overline{x}_k) < n \quad \forall i \right\} \qquad (2\text{-}24)$$

Concretamente el algoritmo empezaría con una serie de inicializaciones generales, entre las que destaca la asignación aleatoria de los prototipos $\overline{p}_j$ de las particiones. Después se entra en el ciclo principal del algoritmo donde inicialmente se determinará a que partición pertenece cada vector $\mu_{P_j}(\overline{x}_i)$ utilizando (2-21). A continuación se vuelven a recalcular los prototipos de las particiones utilizando:

$$\overline{p}_j = \frac{\sum_{i=1}^{n} \mu_{P_j}(\overline{x}_i)\overline{x}_i}{\sum_{i=1}^{n} \mu_{P_j}(\overline{x}_i)} \qquad (2\text{-}25)$$

La condición de parada del algoritmo consiste en chequear la distorsión δ de la partición actual, con la partición de la partición de la iteración anterior δ_{ant}. De modo que si el cambio es menor que un ε, prefijado, se terminará el algoritmo. El valor δ se calcula:

$$\delta = \sum_{j=1}^{c} \delta_j \qquad (2\text{-}26)$$

donde δ_j, es la distorsión producida en cada cluster y se calcula como:

$$\delta_j = \sum_{\bar{x}_i \in P_j} \| \bar{x}_i - \bar{p}_j \|^2 \qquad (2\text{-}27)$$

1. Realizar la asignación $\delta = \infty$
2. Asignar aleatoriamente el conjunto inicial de prototipos $\bar{p}_j$
3. Asignar $\delta_{ant} = \delta$
4. Calcular las funciones de pertenencia $\mu_{P_j}(\bar{x}_i)$ usando (2-21)
5. Calcular los prototipos $\bar{p}_j$ mediante (2-25)
6. Calcular δ usando (2-26)
7. Si $|\delta_{ant} - \delta| / \delta < \varepsilon$ entonces terminar, si no volver a 3.

Algoritmo 2-1: Algoritmo de las *c*-medias

Como se ha comentado este algoritmo muestra la estructura interna del conjunto de datos de entrada. De este modo datos "similares", o que estén cerca geométricamente pertenecerán al mismo grupo, mientras que será difícil que un dato lejano al prototipo de un grupo, formase parte de él.

En cualquier caso el algoritmo muestra una serie de inconvenientes como son:

- No detecta grupos vacíos o degradados.
- Puede acabar con varios prototipos idénticos.
- Como solución final encuentra el mínimo local más cercano en el espacio a la partición inicial.

Evidentemente, el más serio de los inconvenientes es el último, ya que como se deduce, la solución final, no solo va a depender de la partición inicial sino que además no asegura llegar a la partición más óptima.

2.3.3.2 Algoritmo de las *c* medias difuso

El *algoritmo de las c medias difuso* (*fuzzy c means algorithm*) fue propuesto por [Bezdek, 1981] y es una generalización del algoritmo de las *c* medias. La principal modificación que se hace es con respecto a los valores que la función de pertenencia

μ, puede alcanzar. Así, mientras en el algoritmo de las c medias la función μ estaba definida en el rango {0, 1}, ahora cada grupo se considera como un conjunto difuso, por lo que μ se define en el rango [0, 1]. Esto implica, que un vector de entrada pueda ser asignado a varios grupos, con valores de pertenencia distintos, que informarán de la cercanía del vector de entrada a sus respectivos prototipos.

Como salida se devuelve un conjunto de c prototipos, cada uno de los cuales es el prototipo de los vectores de entrada que han sido asignados al grupo que representa.

Los pasos del algoritmo de las c medias difuso son básicamente iguales a los del algoritmo de las c medias mostrados en el Algoritmo 2-1, cambiando sólo el cálculo de algunas funciones. En principio la función de pertenencia del vector de entrada $\overline{x}_i$, al grupo representado por el prototipo $\overline{p}_j$, vendría dada por la siguiente ecuación:

$$\mu_{P_i}(\overline{x}_k) = \frac{1}{\sum_{i=1}^{c}\left(\frac{\| \overline{x}_k - \overline{p}_i \|}{\| \overline{x}_k - \overline{p}_j \|}\right)^{2/(\rho-1)}} \qquad (2\text{-}28)$$

Como se observa, además de los parámetros que se manejaban en el algoritmo anterior, ahora se introduce un nuevo parámetro ρ que indica el grado de "difusión" de la partición del conjunto de entrada. Así, si $\rho \to 1$, la partición obtenida se aproxima a una partición nítida conseguida con el algoritmo anterior, mientras que si $\rho \to \infty$, el algoritmo produce una partición en la que todos los puntos pertenecen a todos los grupos.

Mientras que la matriz de partición U quedaría ahora de la siguiente forma:

$$U = \left\{ \mu \in \{0,1\} \mid \sum_{i=1}^{c} \mu_{P_i}(\overline{x}_k) = 1 \ \ \forall k \quad y \quad 0 < \sum_{k=1}^{n} \mu_{P_i}(\overline{x}_k) < n \quad \forall i \right\} \qquad (2\text{-}29)$$

La ecuación para el cálculo de los prototipos también cambia para adaptarse a la forma de la nueva función de pertenencia, quedando:

$$\overline{p}_j = \frac{\sum_{i=1}^{n} \mu_{P_j}(\overline{x}_i)^{\rho}\, \overline{x}_i}{\sum_{i=1}^{n} \mu_{P_j}(\overline{x}_i)^{\rho}} \qquad (2\text{-}30)$$

La distorsión se sigue calculando igual que en (2-26).

De todas formas el algoritmo presenta los siguientes inconvenientes:

- Pueden aparecer de nuevo grupos idénticos.

- De nuevo, el algoritmo termina en el mínimo local más cercano a la partición inicial.

Como se observa, aunque desaparezca el inconveniente de crear grupos vacíos, se siguen manteniendo, los otros dos problemas. Aunque también hay que resaltar que, con respecto al algoritmo anterior, este algoritmo es más costoso en tiempo de ejecución debido al incremento de cómputo que aparece en las nuevas fórmulas.

2.3.3.3 Algoritmo ELBG

Con este algoritmo se pretenden solucionar los problemas que presentan los algoritmos anteriores. Estos problemas son debidos principalmente a que estos algoritmos sólo producen cambios locales, hasta alcanzar el mínimo local más próximo al punto de partida.

Como ejemplo se va a partir de una situación como la que se muestra en la Figura 2-3, en la que los círculos blancos indicarían los vectores $\overline{x}_i$, mientras que los negros indicarían la disposición inicial de los prototipos $\overline{p}_j$. En este caso el prototipo aislado no se movería en todo el proceso mientras, ya que no está influenciado por ningún vector de entrada. Por otro lado si a la zona donde se encuentra el prototipo $\overline{p}_1$, se traslada algún prototipo, incluso de la zona donde se encuentran los prototipos $\overline{p}_2$ o $\overline{p}_3$, se produciría un mejor resultado ya que el conjunto de entrada, quedaría mejor cubierto. El problema es que debido al funcionamiento local de los algoritmos anteriores, esto tampoco va a ocurrir.

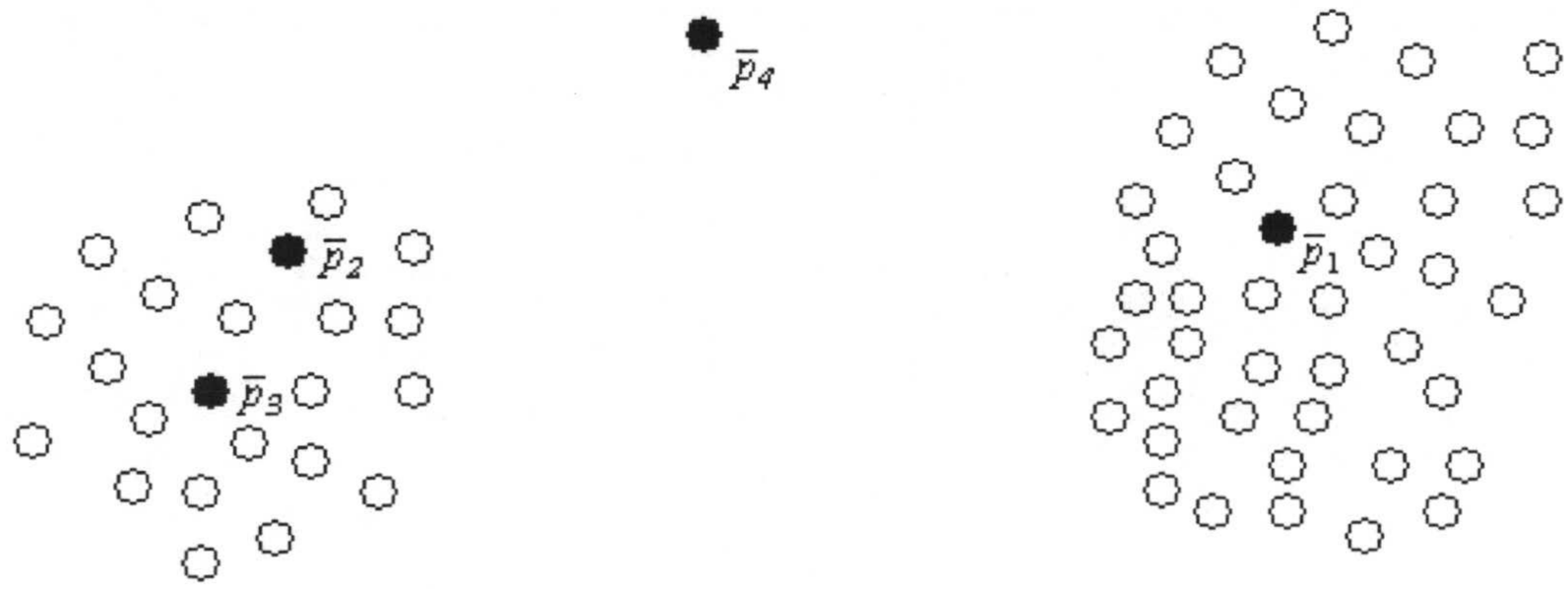

Figura 2-3: Ejemplo de prototipos mal colocados

El algoritmo ELBG propuesto por [Russo, 1999] intenta aportar una solución a estos inconvenientes. Este algoritmo es una extensión del de las *c* medias y se basa en el *teorema de distorsión total* enunciado por [Gersho, 1979] que dice que: "Cada

grupo hace una contribución igual a la distorsión total en una cuantización óptima de alta resolución".

Basándose en este teorema se define el concepto de utilidad de un prototipo $\overline{p}_j$ como:

$$u_j = \frac{\delta_j}{\overline{\delta}} \qquad (2\text{-}31)$$

dónde $\overline{\delta}$, representa la distorsión media de la partición:

$$\overline{\delta} = \frac{1}{c}\sum_{i=1}^{c}\delta_i \qquad (2\text{-}32)$$

El teorema de la distorsión total implica que en una partición óptima del conjunto de entrada, todos los clusters tendrían una utilidad igual. Así el algoritmo ELBG tratará de llegar a esta condición, mediante migraciones de prototipos con utilidad menor que 1, a grupos donde los prototipos tengan una utilidad mayor que 1.

En el ejemplo de la Figura 2-3 esto implicaría que el grupo $\overline{p}_4$ tiene una utilidad 0, que las utilidades de los grupos $\overline{p}_2$ y $\overline{p}_3$ son menores que 1 y que la utilidad del grupo $\overline{p}_1$ es mayor que 1. Por tanto el algoritmo ELBG, intentaría desplazar el grupo $\overline{p}_4$ y alguno de los grupos $\overline{p}_2$ o $\overline{p}_3$ a la zona donde está el grupo $\overline{p}_1$.

```
1. Alojar en el grupo de llegada tanto el nuevo
   prototipo como el prototipo actual de ese grupo.
2. Recalcular posiciones de los prototipos para
   generar dos grupos.
3. Recalcular posiciones de los prototipos en zonas
   de salida.
```

Algoritmo 2-2: Rutina de migración de prototipos en ELBG

En general y para realizar la migración de un prototipo que tiene baja utilidad a un grupo que tiene mayor utilidad se realizarían los pasos mostrados en el Algoritmo 2-2.

El primer paso consistiría en calcular una posición inicial dentro del grupo de llegada P_b, tanto para el prototipo que llega $\overline{p}_a$, como para el actual $\overline{p}_b$. Para esto se

considera al paralelepípedo que contiene al grupo P_b, alojando a $\overline{p}_a$ y a $\overline{p}_b$, en su diagonal principal. Para esto la diagonal se divide en tres segmentos de forma que el segmento central tenga una longitud igual al doble de la longitud de los otros dos segmentos y se colocan los prototipos en los extremos del segmento central.

Después de realizar esta primera inicialización de los prototipos $\overline{p}_a$ y $\overline{p}_b$, se ajustan aplicando el algoritmo de la c medias de forma local la grupo P_b, dividiéndolo en los grupos P'_a y P'_b.

El último paso consiste en fusionar los grupos P_a y P_c en un nuevo grupo P_c y actualizar $\overline{p}_c$ teniendo en cuenta los puntos de las dos particiones a fusionar.

Cuando acaba el proceso de migración, se compara la distorsión de los grupos afectados antes del desplazamiento:

$$\delta_{ant} = \delta_a + \delta_b + \delta_c \qquad (2\text{-}33)$$

con su distorsión después del desplazamiento:

$$\delta_{pos} = \delta'_a + \delta'_b + \delta'_c \qquad (2\text{-}34)$$

Si $\delta_{pos} \leq \delta_{ant}$ se acepta la migración, en caso contrario se rechaza.

Con este algoritmo ya se resuelven los inconvenientes de los algoritmos anteriores.

2.3.3.4 Algoritmo de clustering difuso condicional

Este es el primer algoritmo, de los descritos, que se considera como un algoritmo supervisado. Esto supone que al realizar la partición en grupos, no sólo se tienen en cuenta las variables de entrada, sino también la/s de salida.

Las bases del *algoritmo de clustering difuso condicional* (Condicional Fuzzy Clustering Algorithm)[Pedrycz, 1998] se encuentran en el algoritmo de las c medias difuso, suponiendo una extensión de éste. La parte condicional del mecanismo de agrupación, reside en las variables de salida $y_1, \ldots, y_n$ de las correspondientes variables de entrada. Es decir la variable de salida y_k del vector de entrada $\overline{x}_k$, describe el nivel en que este vector interviene en la construcción del grupo. Esto se realiza definiendo, una etiqueta lingüística en el espacio de salida, que se va a corresponder con un conjunto difuso B, B: $\Re \rightarrow [0, 1]$. De este modo $y_k = B(f_k)$, donde f_k la salida de la red para el vector $\overline{x}_k$, expresa el grado de pertenencia de y_k a B.

El problema de agrupamiento se reformula como "Agrupar los datos considerando que y es B". La forma en que y_k se asocia a las funciones de pertenencia

de la variable $\bar{x}_k$, $\mu_{P_1}(\bar{x}_k)$, ..., $\mu_{P_c}(\bar{x}_k)$ no es única. Sin embargo si se tiene que cumplir que f_k se distribuya aditivamente a lo largo de las entradas de la columna k de la matriz de partición, lo que significa que:

$$\sum_{i=1}^{c} \mu_{P_i}(\bar{x}_k) = f_k \quad k = 1,...,n \tag{2-35}$$

Esto implica que si el vector de entrada $\bar{x}_k$, es poco significativo en el contexto del conjunto difuso B, entonces $B(f_k) = 0$, lo que implica que ese patrón de entrada será excluido del proceso de clustering o agrupamiento ya que $\mu_{P_i}(\bar{x}_k)$ para cualquier valor de i. Por otro lado si $B(f_k) = 1$, el vector de entrada $\bar{x}_k$, contribuirá de forma máxima al proceso de clustering. Estos cambios implican que las matrices de partición queden:

$$U = \left\{ \mu \in \{0,1\} \mid \sum_{i=1}^{c} \mu_{P_i}(\bar{x}_k) = f_k \ \forall k \quad y \quad 0 < \sum_{k=1}^{n} \mu_{P_i}(\bar{x}_k) < n \quad \forall i \right\} \tag{2-36}$$

y que:

$$\mu_{P_i}(\bar{x}_k) = \frac{f_k}{\sum_{i=1}^{c} \left(\frac{\| \bar{x}_k - \bar{p}_i \|}{\| \bar{x}_k - \bar{p}_j \|} \right)^{2/(\rho-1)}} \tag{2-37}$$

Una importante consecuencia de este método es que se reduce el costo computacional al dividir el problema original, en una serie de problemas condicionados por el contexto.

2.3.3.5 Algoritmo de estimación de grupos alternante (ACE)

En los algoritmos de clustering presentados hasta ahora, existe una determinación inicial de los parámetros que caracterizan los métodos, como por ejemplo, la forma de las funciones de pertenencia, la actualización de los prototipos de los grupos, etc. En [Runkler, 1999] se presenta el *algoritmo estimación de grupos alternante* (Alternating Cluster Estimation ACE) o una herramienta donde los parámetros citados anteriormente se pueden directamente por el usuario.

En [Runkler, 1999] se presentan ejemplos del modelo donde no se impone como restricción la optimización de una función objetivo en particular. Así, si el usuario selecciona ecuaciones que no están relacionadas con una función objetivo, las funciones de pertenencia y los grupos se obtienen actualizando alternativamente

conjuntos de parámetros. Así el concepto de función objetivo se reemplaza por un nuevo concepto denominado estimación de grupos alternantes. Las funciones de pertenencia que aquí se proponen se inspiran en sistemas usados en sistemas neuro-difusos.

2.3.3.6 Algoritmo de clustering para aproximación de funciones

La mayoría de los métodos presentados anteriormente, han sido diseñados para la resolución de problemas de clasificación, esto hace que no se adapten bien al problema de la aproximación de funciones. Algunas de las diferencias [González, 2002] más importantes, que se resaltan entre estos problemas son:

- El espacio de salida de ambos problemas es muy diferente. Mientras en clasificación se puede considerar como discreto (conjunto de etiquetas), en aproximación de funciones es continuo.

- En la aproximación de funciones no se persigue una coincidencia máxima entre la salida del modelo y la salida objetivo, sino que se tolera cierto error. Sin embargo en problemas de clasificación si se persigue esta coincidencia.

- En los problemas de clasificación no se tienen en cuenta propiedades de interpolación que si se tienen en cuenta en la aproximación de funciones.

El algoritmo de *clustering para aproximación de funciones* (*clustering for function approximation*) [González, 2002], describe un método que va a incorporar una serie de características que van a ayudar en la tarea de la aproximación de funciones.

De todas formas su objetivo final no será un modelo de aproximación tan exacto como por ejemplo se puede conseguir utilizando métodos matemáticos de minimización tradicionales como: el del gradiente conjugado, Newton-Rapson o Levenberg-Marquardt [Dennis, 1983][Jacobs, 1977][Polak, 1971]. Estos pueden conseguir soluciones prácticamente óptimas si parten de un conjunto de parámetros inicial adecuado. Así el objetivo de este algoritmo será obtener una configuración lo más cerca posible del óptimo global.

Una de las bases de este algoritmo será partir del análisis de la variabilidad de la forma de función. Así, en zonas donde función se mantenga constante, introducirá sólo un prototipo, mientras que en zonas donde la función sea más variable será necesario introducir más prototipos. Para conseguir un compromiso entre estos parámetros se siguen los pasos mostrados en el Algoritmo 2-3.

1. Realizar la asignación $\delta = \infty$
2. Asignar aleatoriamente el conjunto inicial de prototipos $\overline{p}_j$
3. Asignar $\delta_{ant} = \delta$
4. Calcular las funciones de pertenencia $\mu_{P_j}(\overline{x}_i)$ usando (2-21)
5. Calcular los prototipos $\overline{p}_j$ mediante (2-25)
6. Calcular δ_j usando (2-38)
7. Calcular δ usando (2-26)
8. Realizar migración
9. Si $|\delta_{ant} - \delta|/\delta < \varepsilon$ entonces terminar, si no volver a 3

Algoritmo 2-3: Algoritmo de clustering para aproximación de funciones

Como se puede observar este algoritmo utiliza elementos tanto del algoritmo de las *c* medias como del algoritmo ELBG. Así y tras una inicialización aleatoria se entra en el bucle principal donde, al igual que en el algoritmo de las c medias, se realiza el cálculo de $\mu_{P_j}(\overline{x}_i)$ y de $\overline{p}_j$ como en el algoritmo de las *c* medias. Después y para calcular δ_j, un elemento del algoritmo ELBG, utilizaría la siguiente expresión:

$$\delta_j = \sum_{i:\mu_{ij}=1}^{n} \| \overline{x}_i - \overline{p}_j \|^2 \cdot (f(\overline{x}_i) - f(P_j))^2 \qquad (2\text{-}38)$$

donde $f(P_j)$ es la media de la salida de la función objetivo para todos los vectores de entrada o entrenamiento que pertenecen al grupo P_j:

$$f(P_j) = \frac{\sum_{\overline{x} \in P_j} f(\overline{x}_i)}{|P(j)|} \qquad (2\text{-}39)$$

Una vez calculados los valores δ_j, δ se calcula como en (2-26).

En el último paso del ciclo se realizaría una migración de prototipos de forma similar a como se ha descrito para el algoritmo ELBG.

2.3.4 Algoritmos para la inicialización de radios de RBFs

Para la determinación del radio inicial de una RBF existen varias técnicas clásicas. La misión de estas técnicas será encontrar unos valores iniciales para los radios de las funciones base, de forma que se cubra convenientemente el espacio de entrada. Estos métodos se caracterizan porque se usan una vez que se ha fijado el centro de la función base a tratar con respecto a los centros del resto de las RBFs. Otra característica de estos métodos es que suelen trabajar con la distancia, normalmente euclídea, entre las RBFs.

Conviene decir que estas técnicas no sólo se pueden tener en cuenta la hora de diseñar una primera fase de inicialización. Así, también se pueden usar como referencia, cuando se introducen RBFs en fases posteriores, para las que las que normalmente habrá que establecer un radio.

Si se hace una recopilación del funcionamiento de estos algoritmos, la primera solución, consistiría en el establecimiento de un radio similar para todas las funciones base. Esta técnica, a pesar de su simplicidad se ha probado suficiente para obtener un aproximador universal en [Park, 1991][Park, 1993)].

Sin embargo parece más lógico establecer un radio individual y adaptado para cada función base. En [Musavi, 1992], se demuestra que con esta variante se mejoran las prestaciones. El objetivo que persigue esta segunda opción, es conseguir cierto grado de solapamiento entre las funciones base vecinas para que interpolen de forma suave y continua las regiones del espacio de entrada que representan. Entre las heurísticas más usadas se encuentran:

2.3.4.1 Heurística de los *k* vecinos más cercanos (KNN):

Esta heurística fija el radio de cada función base a un valor igual, a la distancia media a los centros de k funciones base más cercanas [Moody, 1989]. Esta heurística pasa a ser la heurística del vecino más cercano si $k = 1$.

2.3.4.2 Heurística de la distancia media de los vectores de entrada más cercanos (CIV):

En este caso se determina el radio de la RBF j-ésima de acuerdo con la siguiente ecuación:

$$d_j = \frac{\sum_{\bar{x}_i \in C_j} \bar{x}_i}{|C_j|} \qquad (2\text{-}40)$$

donde C_j es el conjunto de vectores de entrada que están más cerca de $\bar{c}_j$ que de cualquier otro centro y $|C_j|$ representa el número de vectores que lo componen. Esta heurística produce menor solapamiento que la anterior aunque $k = 1$.

2.3.5 Algoritmos incrementales/decrementales

En este apartado se describirán los conocidos como métodos incrementales y/o decrementales. Un método se define como incremental cuando partiendo de una red prácticamente sin neuronas, va a ir agregándolas siguiendo algún criterio. Por el contrario un método se define como decremental cuando parte de una red compleja en cuanto su número de neuronas, y las va eliminando siguiendo también algunas directrices.

En este apartado se comenzará con la descripción del algoritmo RAN (Resource Allocation Network), para continuar después con sus posteriores modificaciones. El algoritmo RAN [Platt, 1991] intenta mejorar la investigación en el diseño de RBFNs que había hasta la fecha. En esta época por ejemplo los trabajos [Moody, 1989] consistían en utilizar un algoritmo de clustering, como por ejemplo el de las c medias (descrito anteriormente), para determinar los centros de las RBFs. El radio de una RBF determinada, se establecía a partir del cálculo de la RBF más cercana a ésta, mientras que para determinar los pesos se utilizaba el típico algoritmo LMS.

El método RAN es un método incremental, que por lo tanto parte de un número bajo de RBFs y que utiliza información local para ir añadiendo RBFs, y así alcanzar el diseño RBFN. La información local que se analiza cíclicamente son las muestras o patrones $(\bar{x}_k, y_k)$ del conjunto de entrenamiento. Dada una muestra, si se verifica que se cumplen dos condiciones, se introducirá una nueva RBF.

La primera condición que se tiene que verificar es que:

$$\| \bar{x}_k - \bar{c}_n \| > \delta(t) \qquad (2\text{-}41)$$

donde $\bar{x}_k$ es el vector de entrada actual, $\bar{c}_n$ es el centro de la RBF ϕ_n, más cercana según la distancia euclídea $\| \cdot \|$, al vector $\bar{x}_k$. $\delta(t)$ es un umbral de resolución variable con el número de patrones que se han presentado, donde t indica el número de patrón actual. Inicialmente $\delta(t) = \delta_{max}$, es decir el valor máximo que puede tomar y puede ser

igual al tamaño del espacio de entrada. La distancia $\delta(t)$, va disminuyendo hasta que alcanzar un valor δ_{min}. En general y para disminuir el valor de $\delta(t)$ se utiliza la siguiente expresión:

$$\delta(t) = \max[\delta_{\max} \exp(-t/\tau), \delta_{\min}] \quad (2\text{-}42)$$

donde τ es un constante.

La segunda condición que se tiene que verificar es que la diferencia entre la salida de la red $f(\bar{x}_k)$, y el valor objetivo dado por el patrón y_k, sea superior a un determinado valor, es decir:

$$| y_k - f(\bar{x}_k) | > \varepsilon \quad (2\text{-}43)$$

donde ε es un umbral que va a representar la precisión de la salida de la red.

Si no se cumplen las condiciones anteriores entonces se utilizan algoritmos tipo LMS para ajustar los pesos y centros de la red. Con esto el sistema comenzaría ofreciendo una representación de la función con mucho error con respecto a la original, para luego irla refinando mediante la inclusión de RBFs con menor radio.

Más tarde aparece el trabajo [Kadirkamanathan, 1993] que mejora la eficiencia del método RAN utilizando un método de minimización denominado Filtro de Kalman Extendido (Extended Kalman Filter, EKF) en lugar algoritmos LMS para ajustar los parámetros de la red. El método general se denomina RANEKF y demuestra obtener mejores resultados en la aproximación de funciones y la predicción de series temporales.

Otro enfoque parecido es el propuesto por el algoritmo Growing Radial Basis Networks (GRBN) [Karayiannis, 1997]. En este algoritmo se plantea un mecanismo de división de funciones base que cometan más error. Esto implica que la complejidad de la red va creciendo continuamente hasta que se cumpla una condición de parada que evalúa el compromiso entre el error de aproximación y la complejidad de la red.

Hasta ahora los métodos descritos solo tienen en cuenta la adicción de funciones base, pero nunca la eliminación de éstas, si en un momento determinado aportan poco al funcionamiento de la red. Para solventar este inconveniente aparecen los siguientes algoritmos.

El algoritmo Minimal RAN (M-RAN) [Yingwei, 1997] incorpora un mecanismo de poda de funciones contribuyan poco a la salida de la red. Para esto se determina una ventana o grupo de muestras del conjunto de entrenamiento, para las que se analiza la salida de cada función base, eliminado aquellas que no superan un umbral. Otra diferencia con respecto al método RAN es que para introducir una nueva

RBF se tiene que cumplir también que el error cuadrático medio para un grupo de patrones del conjunto entrenamiento sea superior a un valor mínimo.

El algoritmo RAN se sigue modificando en [Rosipal, 1998], donde se incluyen la descomposición QR de Givens, la cual se usa para calcular los pesos de la red (RAN-GQRD) y para realizar la de poda de funciones base (RAN-P-GQRD) reduciendo de esta manera, la complejidad de la red final.

Otra posibilidad para implementar un mecanismo de poda en el algoritmo RAN es usar la descomposición SVD de la matriz de activación de la red para determinar las funciones base menos relevantes [Salmerón, 2001].

En [Rojas, 2000a] se presenta el algoritmo PG-BF Sequential Learning Algorithm donde se emplean hasta tres criterios para identificar funciones base que no contribuyan a la salida de la red.

También existe la metodología opuesta es decir partir de una red completa e ir eliminando RBFs menos relevantes hasta alcanzar algún criterio de parada. Para determinar este tipo de funciones se puede utilizar métodos como el OLS (Chen, 1991) el SVD [Kanjilal, 1995]. Incluso han aparecido hibridaciones del método OLS con algoritmos evolutivos [Chen, 1999].

2.3.6 Métodos evolutivos

A continuación se describirán unos métodos que utilizan una estrategia evolutiva para el diseño de una RBFN. Estos métodos van a tener como característica común que van a implementar la evolución de una población, donde un individuo de la población representa una red completa.

Concretamente el primer método utilizará un algoritmo genético para el diseño de la red. El segundo método utiliza también como base un algoritmo genético, pero lo complementa con otras técnicas típicas en el desarrollo de redes neuronales como por ejemplo algoritmos de clustering, técnicas SVD, OLS, minimización de error, etc.

2.3.6.1 Una estrategia genética

En (Rivas, 2001) se presenta un típico algoritmo genético que evoluciona una población de individuos, donde cada individuo representa una RBFN. Una característica diferenciadora de este método es que un individuo no se representa mediante cadenas de bits o vectores de reales, si no que utiliza una serie de objetos predefinidos. Otras características de este método es que el tamaño de la población es fijo, se usa selección tipo tournament y un reemplazo elitista. Para calcular los pesos se utiliza SVD. El algoritmo termina cuando se alcanzan un determinado número de iteraciones. Los principales pasos de éste se muestran en el Algoritmo 2-4.

Los operadores utilizados y el cálculo de la función de evaluación se muestra en los siguientes apartados.

Operadores utilizados

Los operadores que se utilizan son:

- Operador de cruce binario: este operador trabaja cogiendo una serie consecutiva de neuronas de cada uno de los padres e intercambiándolas. No es necesario que estas secuencias de neuronas sean de igual tamaño.

- Operador de mutación de centros: este operador cambia en un porcentaje los elementos del vector centro de una neurona. Para esto se suma a cada uno de estos elementos un valor aleatorio que sigue una función de probabilidad gausiana de media cero y desviación 0.1.

- Operador de mutación radios: consiste en la suma de un valor aleatorio, que sigue una función de probabilidad igual que la definida anteriormente, una RBF de una determinada red.

- Operador de inclusión de funciones base: este operador duplica cada una de las neuronas con una probabilidad dada (la misma para todas). Cuando una neurona se duplica los valores del centro y del radio se modifican mediante una función gausiana.

- Operador de eliminación de funciones base: elimina un número aleatorio de funciones base.

```
1. Inicialización
2. Repetir mientras no se cumpla condición
    a. Seleccionar individuos
    b. Eliminar al resto de la población
    c. Generar nuevos individuos mediante el
       operador de cruce
    d. Aplicar el resto de operadores a los
       nuevos individuos
    e. Calcular el peso de los nuevo individuos
       utilizando SVD
    f. Eliminar neuronas cuyos pesos sean muy
       cercanos a cero
    g. Evaluar las redes
```

Algoritmo 2-4: Pasos principales del algoritmo

Función de evaluación

Como función de evaluación o función que calcula el valor de adaptación de un individuo/red se usa:

$$V.A. = \frac{1}{\sqrt{\frac{\sum_{i=0}^{n-1}(y_i - f(x_i)}{n}}} \qquad (2\text{-}44)$$

Los valores que se pasan para el cálculo de esta función son los relativos a un conjunto de validación con el que no se ha estado entrenando.

2.3.6.2 Un método evolutivo multiobjetivo

En [González, 2001] se describe un algoritmo evolutivo multiobjetivo para el diseño y optimización de los parámetros de una RBFN a partir de un conjunto de datos de entrenamiento. Las RBFNs diseñadas se aplican al problema de la aproximación de funciones.

La base es un algoritmo evolutivo, concretamente un genético, que evoluciona una población donde cada individuo representa una RBFN. El valor de adaptación que se establece para cada individuo es la raíz cuadrada del error cuadrático medio normalizado. Una vez que los individuos han sido evaluados se usa una técnica multiobjetivo, que trata las soluciones en función de dos objetivos: complejidad y error de una red. Con esto el algoritmo puede optimizar redes con distinto número de funciones base a la vez.

A continuación se detallarán las principales características del algoritmo.

Creación de la población inicial

Dado que el algoritmo va a tener la capacidad de evolucionar redes de distintas capacidades a la vez, se diseña un algoritmo de inicialización que cree individuos con diferentes características. Para eso este algoritmo combina diferentes algoritmos de agrupamiento o clustering para la inicialización de centros como son: el algoritmo de las c medias, el ELBG o el CFA. Para asignarle los radios a estas RBFs se utilizan algoritmos de inicialización de radios como el de los k vecinos más cercanos (KNN) con $k = \{1, 2, 3\}$ y de la distancia media de los vectores de entrada más cercanos (CIV), para un rango de tamaños de red comprendidos entre dos umbrales $[m_{min}, m_{max}]$.

Si hacen falta más redes para completar la población inicial, se obtendrán mediante la aplicación de cambios aleatorios a las redes creadas anteriormente. Si por el contrario el tamaño de la población inicial es mayor de lo necesario, se ordena a los individuos según su rango y se eliminan las peores soluciones.

Operadores evolutivos

Se utilizan una serie de operadores evolutivos para cambiar la estructura de la red añadiendo o eliminando funciones base. Para realizar estas variaciones en el número de RBFs que forman una red se estudiarán aspectos como utilidad de las RBFs o zonas del espacio de entrada que necesitan mal cubiertas por la red. Los operadores utilizados son:

- AME (Adición de una función base en el punto de mayor error). Su funcionamiento es sencillo y consiste en añadir una función base con centro en el patrón de entrenamiento en el que la red produzca el mayor error de aproximación. El radio de la RBF se fijará utilizando las técnicas de asignación de radio citadas anteriormente. Una vez fijados el centro y el radio de esta nueva función base, los pesos de la nueva red se calculan mediante el método de Cholesky.

- DIV (División de funciones base). El funcionamiento de este operador consiste en detectar la función base que produzca un mayor incremento del error de aproximación y dividirla en dos funciones base para que la red cubra mejor esta zona del espacio de entrada. Para estimar la contribución de cada función base ϕ_j al error total de la red se usa el parámetro e_j:

$$e_j = \sum_{k=1}^{n} \frac{\phi_j(\bar{x}_k)}{\sum_{i=1}^{m} \phi_j(\bar{x}_k)} | y_k - f(\bar{x}_k) | \quad j = 1,\ldots,m \qquad (2\text{-}45)$$

donde n es el número de muestras de entrenamiento y m es el número de funciones base de la red. Para elegir la función base a la que aplicar el operador, se genera una distribución de probabilidad en la que cada función base ϕ_j tiene una probabilidad de aplicación proporcional a e_j, y se escoge una función aleatoriamente. Una vez escogida la función base ϕ_j se aplica un algoritmo con sólo dos funciones base a los puntos que cubre la función ϕ_j, y una vez que la red ha sido creada, se sustituye la función ϕ_j de la red original, por las dos nuevas funciones base.

- EOLS (Eliminación de funciones base usando OLS). En este caso se utiliza el método OLS para elegir una función base a eliminar. Para esto se aplica el algoritmo OLS que calculará un vector de coeficientes de reducción de error. Al igual que antes, para elegir la RBF a eliminar, se construye una distribución de probabilidad en la que cada función base tendrá una

probabilidad de ser eliminada inversamente proporcional a su coeficiente de reducción de error.

- ESVD (Eliminación de funciones base usando SVD). Una vez que se aplica la técnica SVD a la matriz de activación de la red se obtiene el vector de valores singurlares asociados a cada una de las funciones base. Partiendo de la base de que cuanto menor sea el valor singular asociado a una función base, menos afectará su eliminación al error de aproximación de la red, la función a eliminar se escogerá con una probabilidad al valor singular asociado.

Selección

El proceso de selección/reemplazo sigue un algoritmo elitista, adaptando la implementación a una técnica multiobjetivo. El utilizar un algoritmo selección elitista indica que nunca se perderá la mejor solución obtenida hasta el momento. Para aplicar este algoritmo, se parte evidentemente de una población evaluada o una población en la que a cada individuo se le ha calculado su raíz cuadrada del error cuadrático medio normalizado. Antes de realizar este cálculo se aplican unas pocas iteraciones del algoritmo Levenberg-Marquardt que ayudan a decidir sobre los mejores individuos. Con estas premisas el algoritmo va guardando los elementos pertenecientes a la frontera del pareto.

Ajuste final de las soluciones

Una vez que finaliza el algoritmo evolutivo, este devuelve el conjunto de soluciones encontradas, éstas se mejoran bastante aplicando el algoritmo Levenberg-Marquardt de forma completa. Con esta estrategia se alcanza el objetivo de utilizar un proceso evolutivo para obtener un conjunto de soluciones de las que pueda partir el algoritmo de minimización. Con esto se consigue obtener un conjunto de soluciones muy cercano al óptimo.

Criterio de parada

El criterio de parada propuesto consiste en el cálculo de la pendiente de una recta de regresión de los mejores individuos encontrados en las últimas generaciones, para cada número i de funciones base. De esta manera cuando estas pendientes están debajo de un umbral se detendrá el algoritmo.

Evolución multiobjetivo

Para implementar la evolución multiobjetivo se introducen una asignación de pseudo-aptitudes basadas en rangos combinando características del algoritmo MOGA (Fonseca, 1993) y NSGA (Srinivas, 1995), para de esta manera distribuir apropiadamente la presión selectiva entre los individuos. Pero para que esta presión

selectiva no sea excesiva se introduce un método para la compartición de pseudo-aptitudes, basado en la distancia fenotípica entre dos redes, que va a dar lugar a la creación de nichos.

2.3.7 Algoritmos coevolutivos cooperativos

En este caso se describirá el trabajo propuesto en [Whitehead, 1996], que es un claro ejemplo de algoritmo coevolutivo cooperativo para el desarrollo de RBFNs. De hecho es un trabajo, donde el algoritmo que se propone, tiene muchos puntos en común con el que se propone en esta tesis.

Como característica propia de este tipo de algoritmos, un individuo de la población representa una neurona o RBF, en lugar de una RBFN completa. El conjunto de la población va a formar una RBFN completa. Así cuando el algoritmo termine se tendrá un conjunto de RBFs coadaptadas, que trabajan bien juntas y que, por lo tanto, dan lugar a una RBFN eficiente.

El algoritmo evolutivo utilizado es un genético y los elementos de una RBF que van a evolucionar son el centro y el radio. El problema de la aportación de crédito se resuelve teniendo en cuenta la contribución de la RBF a la eficiencia de la red. Para mantener la diversidad, se usan conceptos de la técnica de nichos, de manera que se tiene en cuenta el solapamiento entre los trabajos que las RBFNs realizan.

A continuación se describen los principales elementos que caracterizan este algoritmo:

Esquema de selección

Tal y como se ha comentado, se usa un algoritmo genético para guiar la evolución. Este algoritmo genético irá dando lugar a sucesivas generaciones G_k, de modo que cada generación, está constituida por una población de RBFs que forman una RBFN.

Para generar la población de G_{k+1}, a partir de G_k se utiliza un procedimiento de selección basado en el valor de adaptación o fitness. Concretamente, el algoritmo utiliza un esquema de selección proporcional en una población de tamaño fijo. Esto implica dos efectos. El primero es que cada individuo de la población de G_k, tiene una probabilidad de ser seleccionado igual a su valor de adaptación. El segundo efecto es que este valor de adaptación indicará el número de copias que existirán de un individuo en la siguiente generación G_{k+1}. Una vez que se haya formado esta generación G_{k+1}, se aplicarán los operadores definidos, se evaluarán los nuevos individuos, y así cíclicamente funcionará el algoritmo.

Asignación de crédito: Valor de adaptación

Para evaluar la población perteneciente a una determinada población los autores parten de dos premisas principales. La primera es que la evaluación debe dar soporte a un proceso competitivo en el que las RBFs que trabajen mejor deben desplazar a aquellas que trabajen peor. La segunda premisa es que la evaluación debe también provocar un proceso cooperativo, de modo que el trabajo de las distintas RBFs se sume para, por ejemplo, aproximar el valor de una función.

Para conseguir los objetivos anteriores normalizan las salidas de las RBFs, de modo que una RBF normalizada $\bar{\phi}_i$ se define cómo:

$$\bar{\phi}_i(x) = \frac{\phi_i(x)}{\sqrt{\sum_{j=1}^{p} \phi_i^2(x_j)}} \qquad (2\text{-}46)$$

donde p es el número de muestras. El valor de adaptación propuesto para una RBF $VA(\phi_i)$ es:

$$VA(\phi_i) = \frac{|w_i|^{\beta}}{E(|w_{i'}|^{\beta})} \qquad (2\text{-}47)$$

donde $E(x_i)$ devuelve la media de los valores x_i y β se establece al valor 3/2. Con este valor de adaptación los autores demuestran que cumplen sus objetivos.

Codificación

Los elementos que se codifican en este algoritmo son aquellos sobre los que se realiza la evolución, es decir, el centro y el radio de cada RBF. Para esto se utiliza el esquema de codificación típica utilizado en los algoritmos genéticos, es decir, las cadenas de bits. Una cadena de bits codificará pues, el centro y radio de una RBF.

Operadores

Los operadores se van aplicar a la población tras el proceso de selección explicado anteriormente y son tres: un *operador de recombinación* o cruce, un *operador de mutación* y un *operador de mutación suave*. Existe una diferencia entre los dos primeros y el último. Así, los operadores de recombinación y mutación actúan sobre los genotipos o cadenas de bits directamente, sin tener en cuenta los parámetros que estos bits codifican, ni los límites entre los parámetros. Por otro lado, el operador de mutación suave decodifica la cadena de bits, en los correspondientes valores reales, les aplica una perturbación y los vuelve a codificar otra vez.

- El *operador de recombinación* el típico operador de cruce de dos puntos, de modo que se delimitan aleatoriamente dos posiciones de los bits, y los bits que existen entre la posición de inicio y final se intercambia. La probabilidad de aplicación de este operador de cruce depende la distancia euclídea entre las dos cadenas, de modo que decrece cuando esta aumenta.

- El *operador de mutación suave* se aplica a dos cadenas que no se han recombinado con un operador de cruce, por lo que es complementario a él. Este operador aplica una pequeña perturbación aleatoria al centro y radio, dentro de un rango.

- El *operador de mutación* actúa sobre un bit alterándolo con una probabilidad $1/m$, donde m es el número de bits que conforman todas las cadenas. Este operador se aplica independientemente de los otros.

Entrenamiento de la RBFN

Para calcular los pesos de una RBFN, se utiliza el algoritmo LMS, ya que según los autores es un algoritmo cuyo coste computacional es bajo. Este algoritmo se aplica, pasándole un número n de veces pequeño, el conjunto de entrenamiento, con la idea de sólo perfilar los pesos de las distintas RBFs. Este número n, va incrementándose a lo largo del tiempo. Al final se aplica el algoritmo SVD para determinar de forma más exacta los pesos.

3. Descripción del Algoritmo Propuesto

En el capítulo anterior se han examinado las características de las Redes de Funciones de Base Radial y se han descrito distintos métodos para el diseño de éstas. Si se analizan estos métodos, se observa, que para alcanzar unos objetivos determinados, se parte de distintas bases de trabajo, que dan lugar a diferentes modos de funcionamiento. Estos elementos, que caracterizan a un método de diseño, son los que por tanto hay que detallar, a la hora de describirlo.

El objetivo principal, que persigue el método que se presenta en esta tesis, es la resolución del problema de la aproximación de funciones, mediante el uso de técnicas bioinspiradas o soft computing. Estas técnicas se van a disponer en una arquitectura en la que se puede distinguir varios niveles. En un nivel inferior tendremos una red neuronal, que se encargará del proceso de los datos en sí. En un nivel superior al anterior situaremos una técnica coevolutiva, cuya labor será la de ir configurando adecuadamente la red definida en el nivel inferior. En el nivel más alto situaríamos un sistema de lógica difusa encargado de configurar a su vez, la estrategia de aplicación de operadores usada en la técnica coevolutiva comentada.

3.1 Arquitectura bioinspirada para la resolución del problema

Realizando un proceso de abstracción, el tipo de problema al que nos enfrentamos es un problema de optimización. En este contexto tendríamos un espacio de búsqueda, donde cada uno de los puntos que lo constituyen, representa una solución (ó RBFN) a nuestro problema de aproximación. Desde el punto de vista del problema de optimización, nuestro método debería ir recorriendo el espacio de búsqueda comentado, hasta alcanzar el óptimo global, para de esta manera encontrar la mejor solución.

La estrategia que se va a usar, para alcanzar una solución óptima, tiene dos fases bien diferenciadas. En la primera fase, se obtendrá una primera aproximación, mediante el uso de técnicas bioinspiradas. Como se ha descrito anteriormente con un ejemplo en el apartado, 1.3.2, gracias al uso de este tipo de técnicas, esta solución inicial debe encontrarse en el camino adecuado para alcanzar un óptimo global, pero también y como contrapunto, esta solución no será muy precisa. Para conseguir una solución más exacta aplicaremos finalmente, un algoritmo de optimización matemática, especializado en acercarse a extremos locales. Un algoritmo de este tipo, alcanzará un óptimo global, si se parte de unas determinadas condiciones iniciales. De hecho, este tipo de algoritmos, suelen quedar atrapados en óptimos locales si no se presta atención sobre las condiciones de inicio, tal y como se describió en el apartado 1.1. Con esta estrategia de diseño se aprovechan las ventajas de ambas metodologías y se minimiza el efecto de sus inconvenientes.

Tal y como se ha comentado en un nivel inferior y encargándose del procesamiento de los datos en sí, se dispondrá de una red neuronal, concretamente una Red de Funciones de Base Radiales (RBFN). El tipo de RBFN a utilizar, es el que se propone en [Broomhead, 1988], cuya estructura se muestra en la Figura 2-2. Esta red consta de una sola salida, e implementa la función dada por la expresión (2-2). Las funciones de base radial (RBFs) que componen la red tienen una forma gausiana, y sus características se analizan en el capítulo anterior. El tipo de red elegida es ya un estándar dentro de las posibilidades que ofrecen este tipo de redes y sus propiedades, en la aproximación de funciones, se detallan también en el capítulo anterior.

Las bases del diseño de nuestra red, están en el análisis de la información que ésta nos ofrece para un problema determinado, teniendo en cuenta, que todo el proceso se encuentra dentro de un marco coevolutivo. El tipo de función base escogido, ofrece una respuesta localizada con respecto a los parámetros que la caracterizan, como son su centro y su radio, sin olvidar el peso que tiene en la salida de la red. De esta respuesta se pueden extraer una serie de datos para ajustar la función base a la zona del espacio donde se encuentre, modificando su centro o su radio. Los trabajos realizados en este campo ya muestran algunos ejemplos de estos ajustes, fruto de análisis previos. Una de las aportaciones del algoritmo presentado, es el refuerzo de esta base de funcionamiento, mediante la introducción de nuevas clases de análisis, que dan a lugar a diferentes tipos de ajustes.

Todos los elementos anteriores, son controlados por una estrategia coevolutiva, que se situaría en un nivel superior. Esta estrategia, con el paso de las generaciones, evoluciona una población de individuos (funciones base) donde cada uno de ellos representa una parte de la solución total al problema planteado. El conjunto de toda la población constituye la solución total al problema planteado. La definición de este marco coevolutivo supone otra de las aportaciones del algoritmo presentado, ya que

se basa en el paradigma de la computación evolutiva, conocido como *estrategias de evolución*. Así, el método de trabajo refuerza y complementa los elementos analíticos anteriores, de manera que en cada generación, se aplican estos elementos a los individuos para ir mejorándolos, eliminándolos en el caso de que su trabajo diste de ser el adecuado, e introduciendo nuevos individuos que complementen la solución que los anteriores aportan.

Esto da lugar a un entorno de trabajo, donde según los razonamientos anteriores los individuos cooperan en la constitución de la solución final. Sin embargo, también deben de competir por su subsistencia, ya que si no realizan un trabajo adecuado serán eliminados.

Una de las aportaciones realizadas, en la definición del método coevolutivo, es una identificación concisa de una serie de parámetros que caracterizan el trabajo de una función base y que permiten valorar de una manera más exacta la aportación de una función base a la red, o de forma más general, la aportación de un individuo a la solución final.

Para completar nuestra arquitectura híbrida bioinspirada, se ha empleado un sistema de lógica difusa como base, de la estrategia que decidirá la probabilidad de aplicación de cada operador sobre una determinada RBF. Para proporcionar esta salida el sistema tendrá como entrada, los parámetros utilizados en el cálculo de la aportación de crédito. Esta estrategia que se detallará más adelante supone otras de las aportaciones más importantes del algoritmo propuesto.

Como resultado de la utilización de todos los elementos se obtendrá una red o una solución al problema de aproximación de un conjunto de puntos, pertenecientes a una determinada función. Volviendo al problema de optimización antes comentado, con los elementos utilizados hasta ahora, se va a conseguir una solución que se encuentre relativamente cerca del óptimo global, en el espacio de soluciones en el que estamos trabajando. Por otro lado, seguramente, la solución obtenida no será muy exacta, tanto por el tipo de técnicas empleadas, cómo por la forma en que se están empleando.

Esto no supone ningún inconveniente, porque tal y como se ha mencionado lo importante es que esta primera solución que se ha obtenido, se encuentra en la dirección adecuada para alcanzar el óptimo global. Como complemento al método expuesto, se aplicará una técnica matemática especializada en alcanzar extremos u óptimos locales. Las condiciones iniciales de las que partirá esta técnica ya serán las adecuadas gracias a la metodología usada en la primera fase, por lo que el tipo de extremo al que esta técnica llegue, debe ser un extremo global.

A continuación se muestran los principales pasos de nuestro algoritmo, así como las directrices de diseño que caracterizan el funcionamiento de éstos. Los

siguientes apartados describen pormenorizadamente, cada uno de los pasos del algoritmo. La estructura de descripción de cada una de estas fases del algoritmo, va a constar de una introducción donde se hace un breve repaso, del trabajo realizado en el campo correspondiente, para después detallar nuestra propuesta concreta.

3.2 Elementos básicos del algoritmo propuesto

3.2.1 Principales pasos de nuestro algoritmo

En base a los elementos citados en el apartado anterior los principales pasos de nuestro método de diseño de Redes de Funciones de Base Radial, se muestran en el Algoritmo 3-1.

El algoritmo empieza con una inicialización de los parámetros que caracterizan una RBFN, es decir, los centros, radios y pesos de las funciones base. Esta inicialización tendrá una fuerte componente aleatoria, salvo para el caso de los radios, donde se actúa de una manera un poco más determinística, ya que estos se establecen en función de la distancia media que existe entre las funciones base que forman la red. El hecho de que esta primera inicialización sea básicamente aleatoria implica que el resto de los pasos van a estar poco condicionados por esta primera fase, y demuestra una cierta robustez del método presentado.

```
1. Inicializar la RBFN
2. Entrenar la RBFN
3. Evaluar las RBFs
5. Ordenar y seleccionar las RBFs
6. Aplicar operadores
7. Introducir RBFs nuevas
8. Entrenar RBFs
9. Si la condición de parada no se cumple ir a 3
10. Aplicar Levenberg-Marquardt
```

Algoritmo 3-1: Principales pasos de nuestro algoritmo

En el siguiente paso se hace un entrenamiento de los pesos para lo que se utiliza el algoritmo LMS. Este algoritmo es simple, y no representa un coste computacional importante, características que se acentúan por el objetivo con el que el algoritmo se aplica. Así, con este método, se pretende conseguir una configuración del vector de pesos, que simplemente nos indique de forma cuantitativa la aportación de cada RBF,

aunque la configuración en sí no sea muy precisa. Para alcanzar este objetivo, mediante el algoritmo LMS, no es necesario realizar muchas iteraciones.

Una de las aportaciones de este algoritmo es la forma de evaluar una función base. Teniendo en cuenta el entorno coevolutivo en el que se desarrolla la evolución, se propone un mecanismo de asignación de crédito que caracteriza detalladamente a una función base o más concretamente a su labor en la red. Para esta tarea se definen tres parámetros característicos por cada RBF ϕ_i. El primero, a_i, realiza una medición de la aportación cuantitativa de la función base a la red, utilizando factores como el peso de ésta en la red y el número de puntos a los que afecta. El parámetro, e_i, realiza una medición cualitativa del trabajo realizado por una determinada RBF, para lo que se trabaja con el error que se produce en la zona donde se encuentra la función base. Por último el parámetro, s_i, mide el solapamiento de la neurona con las neuronas vecinas valorando, de alguna manera, su independencia y la necesidad relativa de su trabajo en la zona.

Una vez que tenemos perfectamente analizada cada una de las funciones base, el siguiente paso consiste en la ordenación de éstas, en función de las características que las definen. Al estar manejando tres parámetros a la vez, para cada función base, no se puede utilizar una técnica clásica de ordenación. Por tanto es necesario recurrir a las técnicas multiobjetivo y a sus fundamentos, para realizar la ordenación de las funciones base. De esta manera, los tres parámetros que se calculan para cada función base, se tomarán como objetivos para realizar la ordenación anterior. En este contexto una función base ϕ_i, será mejor si realiza una mayor aportación (a_i es mayor), el error que se produce en su zona es menor, (por lo que e_i será menor) y tiene un menor solapamiento con el resto de las funciones base (lo que implica que s_i será menor).

Cuando se ha realizado la ordenación de las funciones base, ya se puede pasar a seleccionar el conjunto de éstas, sobre las que se aplicarán los operadores definidos. Así, este conjunto estará compuesto por la mitad del total de funciones base forman la red, que hayan obtenido una posición más baja en la ordenación anterior.

El siguiente paso consistiría en aplicar los operadores, sobre las funciones base que integran el conjunto anterior. Teniendo en cuenta las características del paradigma evolutivo conocido como *estrategias de evolución*, se han definido sólo dos operadores, lo que por otro lado simplifica nuestro algoritmo. El primer operador operado definido, ELIM, será un operador encargado de eliminar funciones base de la red. El segundo operador, ADAPT, adaptará la función base en función de sus parámetros actuales y del error que se produce en la zona donde se encuentra, con el objetivo de mejorar éste.

La estrategia de aplicación de los operadores es también novedosa en este campo. Ésta consiste en analizar los principales parámetros que caracterizan una

función base ϕ_i, es decir, a_i, e_i y s_i, para en función de este análisis, asignar a cada función base, una probabilidad de aplicación de cada uno de los operadores. La aportación en el proceso, se basa principalmente en la introducción de un sistema de lógica difusa (SLD) para que realice el análisis mencionado. Así, las entradas de este sistema, van a ser los parámetros que caracterizan a una función base, mientras que como salida, el sistema aportará la probabilidad de que se apliquen cada uno de los operadores, sobre la función base anterior.

Una vez que se haya aplicado el sistema de lógica difusa a cada una de las funciones base seleccionadas, ya se tendrá la probabilidad de usar un operador u otro sobre éstas. El siguiente paso consistirá en generar, para cada una de las funciones base, un número aleatorio que determine que operador se aplica realmente sobre cada RBF.

El paso posterior consiste en la introducción en la red de funciones base, tantas como se hayan borrado en el paso anterior. Esta fase también presenta novedades, ya que hasta ahora las técnicas relacionadas introducían las funciones base en puntos, más o menos aislados, donde se produjera una cierta cantidad de error. Nuestra propuesta ahora consiste en realizar el análisis anterior, más que sobre puntos individuales, sobre zonas del espacio de entrada. Este hecho aumenta la robustez del algoritmo.

Después se vuelve a entrenar la red, para ajustar sus pesos, siguiendo las mismas directrices del primer entrenamiento, y tras los cambios realizados en esta. Con este paso, termina el ciclo principal del algoritmo y se comprueba si se verifica la condición de fin de ciclo. Esta condición está basada principalmente en comprobar, si después de ciertas iteraciones, se siguen produciendo mejoras, de modo que en caso negativo, se finalizará el bucle.

El último paso consiste en la aplicación del algoritmo Levenberg-Marquardt, que realizará un ajuste fino y preciso de todos los parámetros que caracterizan la red. Este algoritmo debe devolver una configuración para la RBFN, que se encuentre muy cerca del óptimo global

3.2.2 Directrices de diseño

A continuación se van a describir una serie de directrices o líneas de diseño que caracterizan el funcionamiento de muchos de los pasos anteriores.

En principio, hay que analizar cual es la metodología y el objetivo de cada una de las dos etapas que se pueden diferenciar en el algoritmo. Así la idea, es que la fase principal del algoritmo, donde se realizan las principales aportaciones, alcance una solución dentro del espacio de búsqueda, suficientemente fiable, y que se encuentre en

el camino del óptimo global. Con una solución de este tipo, y ya en una segunda etapa, se aplica un algoritmo de optimización matemática, como el Levenberg-Marquart, del que ya se obtendrá una solución mucho más precisa o muy cercana al óptimo global.

Del análisis anterior, se pueden obtener como conclusiones que en principio la primera fase no tiene que conseguir una solución precisa, por lo que, siempre que se pueda se simplificarán los métodos usados, lo que supone un ahorro computacional. Esta pauta, evidentemente, repercute en la implementación de todos los elementos que conforman el algoritmo y se traduce por ejemplo en tener un conjunto reducido de operadores, un conjunto reducido de reglas para el sistema de lógica difusa, una implementación del algoritmo LMS que ahorra coste computacional, etc.

Por otro lado, también se parte de un tipo de red con una estructura simple, con una salida que es fruto de una relación lineal de las aportaciones de cada neurona y con una función base (neurona) que posee una respuesta localizada. Este marco de trabajo nos va a permitir extraer mucha información que caracteriza, el trabajo o la aportación que está realizando cada función base a la salida de la red. El uso de esta información también está presente en muchos de los pasos del algoritmo para llevar a cabo una serie de acciones que van mejorando el funcionamiento de la red en general.

Otra directriz, que se ha usado en el diseño de los pasos del algoritmo es que las adaptaciones y/o variaciones que realicen estos pasos, deben ser progresivas como se recomienda en el diseño evolutivo de redes neuronales [Yao, 1999], para que mantengan una conducta o una línea evolutiva a lo largo de la ejecución del algoritmo.

Además, se han tenido en cuenta las directrices que aparecen en la bibliografía, para la resolución del problema en cuestión, o las relativas al uso de las diferentes técnicas que se han empleado. Por ejemplo y según [Musavi, 1992] es conveniente permitir cierto solapamiento entre las distintas funciones base, para obtener una salida más continua y suave de la red. Otro ejemplo es que se ha tenido en cuenta en la definición de los operadores a usar [Yao, 1999], el tipo de diseño que se está realizando y las técnicas evolutivas empleadas para alcanzarlo.

El error que se utiliza tanto para medir la eficiencia de una red como para medir el estado de otros elementos dentro del algoritmo, es la raíz cuadrada normalizada del error cuadrático medio (NRMSE) que se calcula como:

$$NRMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - f(x_i))^2}{\sum_{i=1}^{n}(y_i - < y >)}} \qquad (3\text{-}1)$$

donde n es el número de puntos que constituyen el conjunto de entrenamiento, que estaría formado por parejas de patrones (x_i, y_i). $f(x_i)$ es la salida de la red para esa

determinada entrada e $<y>$ es la media de los valores y_i. Este tipo error se escoge por ser una referencia normalizada del error en la zona, y porque ofrece una medida de éste teniendo en cuenta la variación de la función.

3.3 Inicialización de la red

3.3.1 Introducción

La inicialización de la RBFN va a ser la primera fase en nuestro algoritmo de diseño. En esta etapa se hará una primera determinación de los parámetros que caracterizan una función base, es decir, para cada RBF habrá que definir su vector de coordenadas centro y su radio. Además se fijará el vector de pesos inicial.

Uno de los aspectos fundamentales a valorar en la fase inicialización de cualquier algoritmo evolutivo, es la cantidad información extraída del problema, que se va a usar. De una parte se puede realizar una inicialización donde la asignación de los primeros parámetros siga un criterio prácticamente aleatorio y donde, por lo tanto, no se use ningún tipo de información del contexto o entorno que define el problema. En el otro extremo, se tendría un proceso de inicialización que hiciera un análisis exhaustivo de los datos disponibles y relativos al problema, para realizar un primer establecimiento de los parámetros iniciales. Este análisis va a depender del problema en sí, y va a ser más determinístico.

Por ejemplo, en nuestro caso, nos enfrentamos al problema de la aproximación de funciones y nuestra herramienta para resolverlo es la utilización de una red de funciones de base radial. Esta red está formada por una capa oculta de neuronas o funciones base. Las funciones base se van a caracterizar por que tienen una salida acampanada y simétrica con respecto al centro que las caracteriza, mientras que la anchura de la campana será proporcional a su radio. Con estas premisas un primer criterio para situar los centros y establecer los radios de las funciones base, consistiría en realizar un estudio de los datos de entrada, de los de salida y sus relaciones.

A la hora de tomar la decisión sobre el uso, como método de inicialización, de un método más aleatorio o uno más determinístico es conveniente analizar sus ventajas e inconvenientes. Así, un método inicialización con más componente aleatoria, tiene las ventajas de que es más sencillo de implementar y será menos costoso en su ejecución pues ésta se basa principalmente, en la generación de números aleatorios. Otra ventaja de este método es que independiza más esta primera fase del resto del algoritmo de diseño, ya que el resto de las fases no van a quedar supeditadas a la información generada en la fase inicial. Como inconveniente de este tipo de

métodos se tendrá que guían menos el proceso general de diseño por lo que las siguientes fases seguramente deberán de realizar más trabajo.

En cuanto a un método con más componente determinístico tiene como ventajas e inconvenientes los opuestos al anterior método. Así como ventajas tiene que puede establecer una situación inicial de partida que reduzca el trabajo de fases posteriores, acotando por ejemplo el espacio de búsqueda. Por otro lado, esta ventaja se puede convertir en un inconveniente, si esta acotación coarta al algoritmo a la hora de encontrar una solución adecuada. Otros inconvenientes serían que evidentemente, este método sería más costoso de implementar y de ejecutar pues debe tratar más cantidad de información, contemplar problemas, casos especiales, etc.

Dentro del campo de diseño de funciones de base radial, se suelen utilizar diferentes métodos de inicialización, unos más aleatorios y otros más determinísticos. Entre los más usados están los algoritmos de clustering, descritos en el apartado 2.3.3. Estos algoritmos basan su funcionamiento en el análisis del conjunto de datos de entrenamiento. Como resultado de este análisis obtienen una partición en grupos (clusters) de este conjunto de datos de entrada. Cada una de las muestras va a pertenecer a una o varios de estos grupos, si estos grupos son difusos.

Una vez que un algoritmo de clustering ha establecido la partición del conjunto de entrenamiento, el algoritmo de diseño utiliza estos grupos para introducir en cada uno de ellos, una RBF. Como vector centro de esta RBF, se suele fijar el centro geométrico del grupo. El radio inicial de esta función base se suele establecer en función de la distancia al grupo más cercano.

En una clasificación de estos algoritmos se podrían definir como *algoritmos de clustering supervisados* aquellos que sólo tienen en cuenta las variables de entrada, a la hora de realizar la partición del conjunto de datos de entrenamiento. Por otro lado, los *algoritmos de clustering supervisados* son aquellos que utilizan información de la variable de salida, para realizar la partición anterior. El elemento que a nosotros nos interesa a la hora de diseñar nuestro algoritmo, es que el último tipo de algoritmos se comportan mejor, sobre todo a la hora de resolver el problema de la aproximación de funciones [González,]. Esta conclusión viene a ratificar nuestra directriz diseño, de analizar la variable de salida, como herramienta para obtener una población adecuada de funciones base.

Los algoritmos de clustering, se utilizaron inicialmente como métodos de diseño de redes, pero debido a sus inconvenientes (pueden acabar atrapados en extremos locales), suelen formar parte de la fase de inicialización de un proceso de optimización de redes más completo [González, 2001].

Si bien los algoritmos de clustering se encargan más bien de establecer el vector centro de cada una de las funciones base, también se han descrito en el apartado 2.3.4,

algoritmos para la determinación de los radios. Estos algoritmos establecen los radios de las funciones base, una vez que se han fijado los centros de éstas. El radio final de cada RBF se suele determinar en función de las distancias entre éstas.

3.3.2 Método propuesto

El método de inicialización propuesto se va a caracterizar por tener una componente principal aleatoria como ocurre en [Whitehead, 1996], [Rivas, 2001] o en muchos algoritmos evolutivos.

3.3.2.1 Inicialización de los centros

Comenzando por la inicialización de los vectores centro de las RBFs, ésta se va a definir de una manera prácticamente aleatoria. Para comenzar, se va a generar como centro $\overline{c}_i$ de cada una de las funciones base ϕ_i que conforman la red inicial, un vector aleatorio entre los extremos del rango de definición de la función. Después simplemente, y para admitir el nuevo centro generado, se va a comprobar que el nuevo centro no caiga a una distancia inferior a δ de alguno de los centros ya establecido. Para definir δ, se considera d como el radio teórico de las m funciones base que vamos a introducir, estando éstas equidistribuidas y cubriendo al máximo el espacio de entrada aunque sin solaparse. De este modo se define $\delta = d / 2$, lo que implica no se permitirá que se introduzcan centros a una distancia inferior a la mitad de estos radios teórico. Con esto no se le resta aleatoriedad al proceso de inicialización de los centros y si se le ahorra algo de trabajo a las fases posteriores, pues introducir funciones base a una distancia inferior a δ de cualquier otra RBF, no tiene sentido y muy probablemente serán eliminadas en fases posteriores.

3.3.2.2 Inicialización de los radios

A la hora de determinar el radio de cada una de las funciones base, se seguirá el criterio de que cada una de estas funciones, tenga un radio individual. De esta manera la salida de una RBF se adaptará más a su entorno, ya que su campana de salida tendrá un radio que se intentará adaptar a su entorno local. Además se pretende que exista cierto solapamiento entre las RBFs para que se interpole de forma suave. La introducción de estos elementos según [Musavi, 1992] mejora los resultados.

El establecimiento de los radios se va a caracterizar porque va a ser un poco más determinístico. Así, para fijar el radio de cada una de las funciones base se analiza

la disposición de los centros de las funciones que previamente se han establecido. Este análisis consiste en determinar para cada función base ϕ_i, la función base más cercana a esta que denominaremos ϕ_c que se encontrara a una distancia dc_i. Con estas distancias definiremos *mdc* como la media de las distancias dc_i es decir:

$$mdc = \frac{\sum_{i=1}^{m} dc_i}{m} \qquad (3\text{-}2)$$

donde m es el número de funciones base.

El valor *mdc* es el que se va a emplear como referencia para asignar los radios de las RBFs. De este modo, el valor inicial del radio d_i de una RBF ϕ_i se va establecer de manera aleatoria dentro del intervalo [$mdc \cdot 0.75$, $mdc \cdot 1.25$].

La justificación para emplear este método tiene que ver con la técnica totalmente aleatoria empleada para insertar los vectores centro y con la fase inicial en la que estamos. Está claro que con el parámetro *mdc* tenemos una medida de cual suele ser la distancia mínima que separa a dos funciones base. La idea es que no vamos a utilizar esta distancia mínima entre dos funciones cuales quiera, pues el centro para estas dos funciones se ha determinado de una manera totalmente aleatoria y no refleja ninguna característica del problema a tratar. Por esto es mejor coger una media de estas distancias mínimas, ya que este valor si va a reflejar la distribución de las RBFs, así como cuanto espacio debe de cubrir, más o menos, cada función base.

3.3.2.3 Inicialización de los pesos

Por último quedaría hablar de la inicialización de los pesos. Para los pesos se establece un valor inicial igual 0. Esta asignación o el establecimiento de los pesos a un valor bajo [Lippmann, 1987] es lo que se suele hacer en la bibliografía cuando se trabaja después con algoritmos como el LMS, como ocurre en este trabajo.

3.3.2.4 Conclusiones

Si se analiza el método se puede deducir que es bastante aleatorio, ya que para comenzar, los centros de las funciones base se determinan prácticamente al azar. En cuanto a los radios de éstas, aunque no se determinen de una manera totalmente aleatoria si tienen dos componentes aleatorios. Un primer componente sería el que reciben del cálculo de los vectores centro, mientras que el segundo componente vendría dado por la forma de establecer el radio final dentro de los extremos de un intervalo.

El método empleado, como se ha comentado anteriormente, va a tener un coste bajo no sólo en cuanto a su implementación, sino también a su ejecución, lo que es

más destacable. Por otro lado y como no define unos valores demasiado característicos o determinísticos, en principio, no va a supeditar a las fases posteriores a trabajar en espacios de búsqueda limitados o acotados. El factor anterior también implica que las fases posteriores tienen que realizar más trabajo, ya que este método de inicialización, prácticamente no ha definido ninguna dirección en la búsqueda. De todas formas y si las fases posteriores consiguen buenos resultados, se demostrará aún más, la robustez de estas últimas.

3.4 Entrenamiento de la RBFN

3.4.1 Introducción

En la fase de entrenamiento de una red se van a determinar los pesos de ésta. En los capítulos anteriores se han descrito diferentes métodos para realizar esta tarea. Estos métodos se podrían clasificar según su funcionamiento, en los siguientes dos tipos:

- Métodos que utilizan técnicas de gradiente descendiente: Estos métodos [Lippmann, 1987] son los que se utilizan de forma clásica en el campo de las redes neuronales para determinar los pesos que las caracterizan. Su funcionamiento va a consistir básicamente en ir presentando a la red iterativamente las distintas muestras del conjunto entrenamiento. Para cada una de estas muestras se contrastará la salida aportada por la red con la salida objetivo, obteniendo la diferencia entre éstas o la información de gradiente. Entre estos métodos destacan la regla delta o el algoritmo LMS (Least Mean Squares) descritos en el capítulo de introducción.

- Métodos que utilizan técnicas de resolución de matrices: Según se expuso en el capítulo anterior, gracias a la estructura de las redes de funciones de base radial y a su simplicidad, la determinación de los pesos de una red de este tipo, puede expresarse como la solución a un sistema de ecuaciones. Los diferentes métodos que se encuadrarían en esta clase aportarían técnicas, cada una con sus ventajas e inconvenientes, a la resolución a un sistema de ecuaciones. Entre estas técnicas cabe destacar, la descomposición de Choleski, el conocido método SVD (Single Value Descomposition) o el OLS (Ortogonal Least Square), las cuales se describen en el capítulo anterior.

Si se comparan ambos métodos, hay que destacar que existen métodos de determinación de los pesos de una red basados en gradiente como el LMS, que ofrecen una complejidad baja, ya que, sus cálculos suelen restringirse a dos sencillas

operaciones. Así, para cada una de las muestras que se le pasan a la red, primeramente habrá que determinar el error producido por la red, o la diferencia entre la salida aportada por la red, para esa muestra, y la salida objetivo. Una vez determinado este error, se utilizará una expresión que tampoco suele tener mucha complejidad, para determinar la variación que se deber realizar sobre los pesos. Realizando este proceso durante no muchas iteraciones, se pueden obtener unos valores aceptables para los pesos. El inconveniente principal de estos métodos sería que puede costar alcanzar una configuración para los pesos, que necesite mucha exactitud.

Si ahora se analizan los métodos especializados en la resolución de matrices hay que destacar como ventaja, que aportan soluciones de gran exactitud. Sin embargo tienen dos inconvenientes principales. El primero sería que necesitan un alto coste computacional debido a la cantidad de operaciones necesarias para resolver los grandes sistemas de ecuaciones que generan. El segundo inconveniente sería las precondiciones que necesitan para que se puedan aplicar, ya que no siempre pueden utilizarse. Estas precondiciones dependen del método a utilizar y suelen estar relacionadas con la distribución de las funciones base en el espacio de entrada, con el solapamiento entre éstas, etc.

Haciendo una revisión de los trabajos relacionados con el diseño de Redes de Funciones de Base Radial vistos en el capítulo anterior, los métodos que más se usan a la hora de determinar sus pesos son el LMS y el SVD. Incluso conviene destacar como ejemplos de su uso el trabajo [Whitehead, 1996], ya que utiliza ambas técnicas. Este trabajo realiza un diseño de una RBFN utilizando una técnica coevolutiva y es claro ejemplo, de cómo y cuando se suele utilizar cada técnica. El algoritmo va produciendo una serie de generaciones de individuos, donde cada individuo se corresponde con una función base. Para determinar los pesos de cada una de estas generaciones se utiliza el algoritmo LMS, ya que según los autores determina de una manera rápida y con una precisión aceptable la configuración del vector de pesos usado. Una vez que se ha alcanzado la última generación se aplica el algoritmo SVD para determinar de una forma más exacta la configuración de pesos definitiva. Así, aunque este algoritmo sea más costoso computacionalmente sólo se aplica una vez. Con esta estrategia de aplicación se aprovechan las ventajas de ambos métodos y se minimizan sus inconvenientes.

3.4.2 Método propuesto

Como método normal de entrenamiento o de determinación de pesos de una RBFN, se propone el algoritmo LMS (Least Mean Square). Cuando nos referimos a este método como método "normal" de determinación de pesos es porque se va a emplear una estrategia similar a la utilizada por el método [Whitehead, 1996] descrita

en el apartado anterior. Es decir, en cada iteración, el algoritmo va ir obteniendo una población de individuos o RBFs. Cada vez que se obtenga una de estas poblaciones se determinarán los pesos que configuran la red mediante el algoritmo LMS. Pero como en [Whitehead, 1996], la configuración definitiva del vector de pesos se obtendrá utilizando un algoritmo como el Levenberg-Marquardt, que aunque más lento, es mucho más preciso.

Tras esta introducción vamos a pasa a detallar el algoritmo LMS en sí, para después describir su estrategia de uso.

3.4.2.1 Algoritmo LMS

El algoritmo LMS (Least Mean Square) o algoritmo de la regla delta fue propuesto por Widrow en (Widrow, 1960), y presenta muchas similitudes con la regla del perceptron. Básicamente es una regla lineal para la corrección del error, que tras la presentación de una muestra va a realizar una corrección de éste.

```
1. Presentar un nuevo patrón (entrada-salida
    objetivo) (x̄_i, y_i)
2. Calcular la salida actual de la red f(x̄_i)
3. Adaptar pesos según ( 3-3 )
5. Volver al paso 2
```

Algoritmo 3-2: Algoritmo LMS

Sus pasos se muestran en Algoritmo 3-2. Consta de un ciclo principal donde de forma iterativa se van presentando las muestras del conjunto de entrenamiento. Una vez que se presenta una muestra, la ecuación utilizada para la actualización del vector de pesos es:

$$\overline{w}_{k+1} = \overline{w}_k + \alpha \frac{e_k \overline{x}_k}{|\overline{x}_k|^2} \qquad (3\text{-}3)$$

En esta ecuación k indica el número de iteración, $\overline{w}_{k+1}$ será el nuevo valor actualizado del vector de pesos, $\overline{w}_k$ es el valor actual del vector de pesos y $\overline{x}_k$ es el vector de entrada actual. e_k es el error lineal actual que se define como la diferencia entre la repuesta deseada y_k y la salida que ofrecida por la red $f(\overline{x}_k)$. Al valor α se le conoce como *velocidad de aprendizaje*, ya que mide el tamaño de la rectificación a realizar.

De la elección del parámetro α depende no sólo la velocidad de aprendizaje sino también la estabilidad del algoritmo (Widrow, 1990). Así para vectores de entrada independientes del tiempo se asegura la estabilidad en la mayoría de los casos prácticos si α pertenece al intervalo (0, 2). Sin embargo el asignar a α un valor superior a 1 puede dar problemas de sobrecorreción por lo que se α suele pertenecer la intervalo (0.1, 1).

3.4.2.2 Estrategia de aplicación del algoritmo LMS

Una vez descrito el algoritmo LMS, vamos a pasar a describir como lo vamos a configurar y a usar en nuestro trabajo. Para obtener una solución precisa del algoritmo LMS, hay que entender bien su funcionamiento. Este algoritmo, trabaja sobre un espacio de búsqueda determinado, y como se ha demostrado realiza una corrección del vector de pesos proporcional a α. En su funcionamiento el algoritmo LMS, va desplazándose por el espacio de búsqueda en el que trabaja y α representa, de alguna manera, el valor del desplazamiento que se produce en ese espacio de búsqueda. De forma simplificada este funcionamiento se representa en la Figura 1-1.

De este funcionamiento podemos extraer dos conclusiones. La primera es que no siempre que se aplique LMS, se obtendrá una solución mejor que la anterior pues puede ocurrir que se salte a una zona del espacio con más error. La segunda es que para obtener una solución de una forma más rápida y precisa sería conveniente que el valor de α, fuera variable. Así α, debería tener un valor mayor en las primeras iteraciones para avanzar rápidamente y no quedar atrapado en pequeñas ondulaciones del espacio de búsqueda. Por otro lado cuando nos encontramos cerca de la solución óptima, α debería de tomar un valor menor para que de esta forma los desplazamientos por el espacio de búsqueda, fueran también menores y pudiéramos alcanzar una solución más precisa.

Sin embargo, nuestra pretensión no es usar el algoritmo LMS para conseguir una solución precisa del vector de pesos de la red, si no que lo que se desea es que este algoritmo nos apunte una configuración de pesos aceptable, aunque no sea demasiado exacta. Esta estrategia que ya se emplea en [Whitehead, 1996], se sigue por tres motivos. El primero es que este algoritmo se usa en una fase intermedia para calcular la configuración de pesos, al final ya se usará el algoritmo Levenberg-Marquardt para obtener una solución una solución más exacta del vector de pesos. El segundo motivo es porque con la configuración de pesos que consigue el algoritmo LMS, basta para reconocer cual es la importancia de las funciones base en cuanto a su peso. El tercero es porque la obtención de esta solución intermedia del vector de pesos va a tener un bajo coste computacional.

Con estas premisas la forma de utilizar el algoritmo LMS es la siguiente. En principio el valor de α se va a mantener a un valor intermedio-bajo, del rango recomendado (0.1, 1), y constante, concretamente $\alpha = 0.3$. Se mantiene constante porque no se va buscando una solución demasiado precisa y se establece a un valor intermedio-bajo porque el número de cambios en la red entre generación y generación no suele ser grande, luego el espacio de búsqueda no variará mucho, y el nuevo mínimo no estará muy lejos.

En cuanto al número de iteraciones que se aplica el algoritmo LMS es el siguiente. El proceso comenzará pasando el conjunto de entrenamiento de una manera aleatoria *niLMS* veces, siendo *niLMS* un número pequeño. Después se entra en un bucle. En cada iteración de este bucle se pasa, al conjunto de entrenamiento, el algoritmo LMS, *ncLMS* veces, donde $ncLMS < niLMS$. Concretamente $niLMS = 5$ y $niLMS = 3$, habiéndose determinado estos valores de una manera empírica. También y en cada iteración del bucle, se calcula y almacena el error cometido por la red. De modo que si tras dos iteraciones consecutivas no se mejora el error, se terminaría de aplicar LMS. El esperar a que durante dos iteraciones consecutivas no se mejore el error es porque, tal y como se ha comentado, puede ocurrir que tras una iteración se haya saltado a una zona del espacio de búsqueda donde no se mejore el error. El vector de configuración de pesos que se devolverá, será el mejor que se haya conseguido durante las iteraciones.

3.5 Asignación de crédito

3.5.1 Introducción

En un algoritmo evolutivo tradicional, se mide para cada individuo un valor que representa, como ese individuo se adapta al ecosistema, o lo que es lo mismo, como de buena es la solución que éste representa. A este valor se le conoce como fitness o valor de adaptación. Cuando se trabaja con un método coevolutivo, cambian un poco los conceptos, ya que un individuo representa una parte o un componente de la solución, estando la solución completa compuesta por varios individuos. Con este entorno de trabajo, ahora se contempla un valor que mida como de bueno es el componente, de la solución total que éste representa. Al proceso de cálculo de este valor se le denomina *asignación de crédito*.

Normalmente, y si se examina la bibliografía, este valor se ha calculando relacionando en una expresión diversos parámetros que caracterizan, más o menos, el trabajo que aporta un individuo, para la consecución de la solución final. Esta tendencia está empezando a cambiar [García, 2002] y ahora la asignación de crédito

no consiste en calcular un valor que relacione estos parámetros, sino que se queda con el conjunto de parámetros como tal. Estos parámetros, que al fin y al cabo representan como contribuye un individuo en la solución final, normalmente se emplean para ordenar los individuos. En este caso, y como un individuo está caracterizado por varios parámetros, no es tan sencillo realizar esta ordenación, y es necesario utilizar una técnica multiobjetivo.

Si se comparan las dos tendencias anteriores, el poseer un único valor que represente la aportación de crédito simplifica tareas posteriores como por ejemplo la ordenación de los individuos, lo que es una ventaja. Sin embargo, el establecer una expresión que relacione los parámetros que caracterizan a un individuo acarrea una serie de problemas ya que esto supone, por ejemplo, cierta pérdida de la información que los parámetros aportan, o la búsqueda subjetiva de una ponderación adecuada, para cada uno de los parámetros, en la expresión.

Para la opción de mantener un conjunto de parámetros como resultado del proceso de la asignación de crédito se puede decir todo lo contrario. De una parte esta elección complica alguna de las tareas posteriores como la de establecer un orden en los individuos, para la cual hay utilizar alguna técnica multiobjetivo. Sin embargo, con esta técnica se pueden valorar mejor las características de un individuo, mediante un estudio más directo de los parámetros que lo caracterizan. Esto tiene muchas implicaciones, pudiendo incluso minimizar el inconveniente que puede presentar este tipo de métodos. Así, el uso de una técnica multiobjetivo va a analizar mejor a los individuos a la hora de establecer un orden para ellos, por lo que esta ordenación, normalmente, será mejor que cuando se utiliza la primera opción.

Tras valorar las ventajas e inconvenientes de cada técnica, en este trabajo se ha optado por calcular y mantener un conjunto de parámetros para cada individuo, que representen su contribución a la solución final. Esto, supone una novedad dentro del campo de los algoritmos coevolutivos para el diseño de redes de funciones de base radial.

3.5.2 Caracterización de una RBF

En este apartado se van a describir los parámetros elegidos para caracterizar un individuo o función base de nuestra población. Esta elección va a estar fundamentada en el trabajo que se ha realizado hasta ahora en el diseño de RBFNs y en los parámetros, que tradicionalmente se han utilizado, en la descripción de las funciones base. Esto elementos se mejoran y complementan con la inclusión de nuevos factores o cuantificadores.

3.5.2.1 Parámetro que mide la aportación de una función base

Una de las características que siempre se ha intentado cuantificar de una función base es su aportación a la salida de la red. Si se examina los trabajos realizados en el diseño de RBFN, descritos en el capítulo anterior, esta cuantificación se ha realizado utilizando el peso, de la función base, como parámetro de referencia. Uno de los ejemplos más claros se encuentra en [Whitehead, 1996], teniendo en cuenta que usa un método general de diseño, parecido al que aquí se propone.

Sin embargo, y para medir la aportación de una función base, un elemento más inédito y que conviene tener en cuenta es el radio de la RBF en cuestión o, mejor incluso, el número de puntos que se encuentran dentro de él. La importancia y justificación para tener en cuenta este factor, junto con el factor anterior, reside en el hecho de que puede existir una función base que tenga un peso muy alto, pero que afecte a muy pocos puntos. Un ejemplo de este caso ocurre cuando las funciones se sitúan en los "bordes" del conjunto de entrenamiento, ya que esto implica que su radio sólo afecta a la mitad de los puntos, de los que normalmente afectaría una RBF, que se encontrara en una posición más "céntrica" de este conjunto de entrenamiento. Evidentemente una función base que afecte a muy pocos puntos, aunque tenga mucho peso, no aportará a la salida de la red, tanto como otras funciones que afecten a más puntos, aunque tengan un peso menor.

Para valorar convenientemente ambos factores se define el parámetro a_i para la RBF ϕ_i, como:

$$a_i = rac_i / da \qquad (3\text{-}4)$$

donde rac_i va a guardar realmente la relación que existen entre el peso y el número de puntos a los afecta la función. *da* es un factor de adecuación que se explicará más adelante. Así, se define:

$$rac_i = |w_i| \cdot (pe_i / mpe) \qquad (3\text{-}5)$$

donde w_i es el peso actual de la función base en la red, pe_i es el número de puntos del conjunto de entrenamiento a los que afecta la función base ϕ_i, y *mpe* se define como la media de los valores pe_i es decir

$$mpe = \frac{\sum_{i=1}^{m} pe_i}{m} \qquad (3\text{-}6)$$

La introducción del factor de división *mpe* sirve para homogeneizar el resultado final.

En principio con el valor del parámetro rac_i, para una RBF ϕ_i, ya hemos conseguido nuestro objetivo de valorar más correctamente la aportación que una función base hace a la red. Sin embargo, nuestra pretensión es que un sistema de lógica difusa analice estos parámetros rac_i, para que decida qué operador, se va a aplicar con más probabilidad, sobre cada función base. El problema es que estos valores rac_i son relativos, ya que van a depender de la red, de la función que se esté aproximando, del entrenamiento, etc. Por ejemplo los pesos de una red, que esté aproximando una función con valores extremos grandes, serán superiores a los de otra red, que trabaje sobre una función que tenga unos extremos más moderados. Para esto, es necesario adaptar los parámetros rac_i, a los rangos de entrada sobre los que trabajará el SLD, para esto se van a dividir por un valor *da*.

Concretamente y para que sea efectivo el análisis de nuestro SLD, las entradas deberían de estar en torno al intervalo [0, 1]. La cuestión es decidir qué valor se establece para *da*. Los criterios que se utilizan para establecer un valor para *da* son los siguientes.

En primer lugar se parte de que los valores de entrada estarán definidos en un intervalo con un máximo y un mínimo, y que vamos a dividir todos estos valores de entrada por *da*, con la idea de que el nuevo intervalo que se defina se solape en cierta manera, con el intervalo [0, 1] que es el intervalo donde principalmente trabaja nuestro SLD.

El segundo criterio es que esta traslación de valores se tiene que llevar a cabo, con la idea y con la precaución, de que estos nuevos valores van a decidir, qué operador se aplica a una determinada función base. Así si por ejemplo, mediante la división por *da*, se decide que el nuevo intervalo de valores de entrada sea igual a [0, 1], probablemente cuando se analicen las funciones base, sus valores para este parámetro serán muy bajos, luego existirá mucha presión para ellas y se borrarán con cierta probabilidad. Lo natural, por tanto es llevar un número importante de los valores de entrada, a una zona donde en principio no haya mucha presión o no sea muy probable que se borre la RBF.

Para implementar esta idea se analizará el conjunto de valores de entrada, para cada determinada generación de RBFs, en función de su media y su desviación típica. Una vez determinados estos factores, se establecerá un valor adecuado para *da*. Para comenzar se estudiará la relación entre la media y la desviación típica. De este modo, si la desviación típica es muy grande con respecto a la media, se establecerá *da* para que la nueva media de estos valores salga fuera del intervalo [0, 1], y de esta forma hacer que disminuya la presión sobre las funciones base. Sin embargo, si esta desviación típica es pequeña con respecto a la media original de los valores de entrada, la nueva media de los valores de entrada, a_i, se establecerá dentro del

intervalo [0, 1], aunque en una parte superior de este, para que no exista mucha presión.

Para esto, se define *medrac* como la media de los valores rac_i y *desvrac* como la desviación típica de los valores rac_i. De este modo, se fija como punto base, una relación en la que la desviación típica sea igual a la mitad de la media. Para esta relación, se establecerá un valor para *da*, que traslade la media de los valores rac_i, a un valor intermedio de la mitad superior del intervalo de entrada. Proporcionalmente a como sea de superior la relación entre la desviación típica y la media, se fijará también la media de los valores rac_i.

Esto se implementa con las siguientes fórmulas. En principio en *dmrac*, se almacenará la relación entre la desviación típica y la media de la siguiente manera:

$$dmrac = \frac{desvrac}{medrac} \qquad (3\text{-}7)$$

Así se define *da* como:

$$da = \frac{medrac}{0.75(dmrac / 0.5)} \qquad (3\text{-}8)$$

Con esta definición conseguimos nuestros los objetivos antes marcados.

3.5.2.2 Parámetro que mide el error dentro del radio de la función base

Otra de las características interesantes para estudiar cada una de las función base, es el error que se produce en su radio, ya que este parámetro nos informará de cómo de bueno está siendo su trabajo en la zona, en la que está situada. Además, la idea no es manejar el error estándar directamente sino derivar un par de medidas de éste que nos sirvan para analizar mejor, el trabajo que está realizando la función base. En principio no se conocen trabajos que trabajen directamente con los parámetros que se van a definir a continuación.

La medida del error se va a realizar en función de dos factores. De una parte, se va a calcular la raíz cuadrada normalizada del error cuadrático medio (NRMSE). Este factor, aunque sea un poco más complicado de calcular nos da una información más absoluta del error, ya que nos ofrece una perspectiva de éste, con respecto a la variación de la función. Por otro lado también vamos a calcular la desviación típica del error que se produce dentro del radio. Este factor se halla ya que no sólo nos interesa saber el error en sí de cada función base, sino también la variación de éste. A partir de esta información, se puede inferir que una función base que con un error más homogéneo dentro de su zona, posiblemente tenga sólo un problema de ajuste de

parámetros. Por otro lado, si el error es más heterogéneo, el problema será mayor y quizás haga falta redistribuir las RBFs de la zona.

Con estas premisas se define el parámetro con el que se va a medir el error e_i, para una determinada ϕ_i, como:

$$e_i = rerr_i / de \qquad (3\text{-}9)$$

Este factor se define con la misma idea que a_i, así *de* será un factor para luego trasladar conveniente, estas medidas del error, hasta un intervalo adecuado para el SLD. $rerr_i$ será el factor que cuantificará el error de la siguiente manera:

$$rerr_i = nrmse_i \cdot (ndesv_i / mndesv) \qquad (3\text{-}10)$$

donde $nrmse_i$ es la raiz cuadrada del error cuadrático medio dentro del radio de la función base, $ndesv_i$ es la medida de la desviación típica del error dentro del radio, mientras que *mndesv*, es la media de las desviaciones típicas de las RBFs dentro de su radio. Este último parámetro se introduce para homogeneizar más los valores finales de los parámetros $rerr_i$.

Con los parámetros $rerr_i$ tenemos el mismo problema que con los rac_i, es decir, son valores relativos al problema a resolver, y a la red que se está utilizando para ello. Sin embargo, y para que luego estos parámetros puedan ser analizados por el SLD, necesitamos que éstos se encuentren en torno al intervalo [0, 1]. Al igual que con el parámetro anterior, vamos a utilizar la variable *de*, para realizar la traslación necesaria del intervalo donde actualmente se definen los parámetros $rerr_i$, a un nuevo intervalo que se adecue más a la entrada.

Tal y como se hizo para el parámetro anterior, *dmrerr* se define de cómo:

$$dmrerr = \frac{desvrerr}{medrerr} \qquad (3\text{-}11)$$

donde, en este caso, *medrerr* es la media de los parámetros $rerr_i$ y *desvrerr* es su desviación típica.

Para establecer el valor *de*, se establecerá:

$$de = \frac{medrerr}{0.25 / (dmrerr / 0.5)} \qquad (3\text{-}12)$$

Aplicándose la misma interpretación que en el caso anterior, es decir, si la desviación típica de los valores $rerr_i$ es grande con respecto a su media habrá desplazar los valores *rerr* que estén en torno a la media o sean inferiores a ésta a una

zona de muy baja presión. La presión sobre estos valores, o mejor dicho, sobre las funciones base que estos representan crecerá conforme los valores *desvrerr* y *medrerr*, se igualen.

3.5.2.3 Parámetro que mide el solapamiento entre RBFs

El último parámetro a estudiar de una función base es el solapamiento que tiene con otras neuronas. Esto nos informará sobre como de independiente es la aportación que ésta realiza. De este modo, si una neurona se solapa mucho con otra, posiblemente la aportación a la zona, pueda ser realizada por una de las dos, lo que supondrá poder eliminar una de estas funciones base, y disminuir la complejidad de red.

Los fundamentos de esta idea se encuentran el la técnica del *fitness sharing* [Goldberg, 1987]. Según se describió en el apartado 1.4.2.4, el objetivo de esta técnica es mantener la diversidad de la población, elemento que es muy importante cuando se trabaja en un algoritmo coevolutivo, como el nuestro, donde cada individuo representa una parte de la solución final. La técnica del fitness sharing propone básicamente que los individuos que se encuentran cerca en el espacio de búsqueda compartan su fitness o valor de adaptación, para de esta manera perjudicar este tipo de comportamientos. Esto se consigue, con la utilización de la ecuación (1-39). Esta técnica se modifica posteriormente en [Beasley, 1993] para dar lugar a la *técnica de nichos*.

Si se examinan los antecedente más claros en el diseño de RBFNs, en [Whitehead, 1996] se realiza una instanciación de esta técnica. Así de una manera más indirecta, se demuestra que mediante la función de fitness/aportación de crédito definida, se desvaforece el solapamiento de las neuronas y se propicia la formación de nichos.

En nuestro caso, para conseguir el objetivo descrito, vamos a definir un parámetro que mida el solapamiento de una función base con el resto de funciones base de la red. Después, ya se encargará el SLD de analizar este solapamiento, y generar una probabilidad de eliminación mayor, para aquellas funciones base que presenten una mayor compartición del espacio de entrada. Dada la RBF ϕ_i, se define el parámetro s_i para medir su solapamiento como:

$$s_i = \sum_{j=1}^{m} s_{ij} \qquad (3\text{-}13)$$

donde m es el número total de funciones y s_{ij} mide el solapamiento entre la RBF ϕ_i y la RBF ϕ_j, como:

$$s_{ij} = \begin{cases} (1 - (\| \phi_i, \phi_j \| / d_i)) & si \| \phi_i, \phi_j \| < d_i \\ 0 & en\ otro\ caso \end{cases} \quad (3\text{-}14)$$

donde $\| \cdot \|$ es la distancia euclídea entre dos funciones base.

Al contrario que en los parámetros anteriores con el parámetro s_i, si se obtiene una medida más absoluta del solapamiento de las neuronas, que no es relativa ni al problema, ni a la red actual. Por tanto, los parámetros s_i, son directamente analizables por el SLD sin necesidad de realizar, sobre ellos, ningún tipo de adaptación.

3.6 Ordenación y selección de las funciones base

3.6.1 Introducción

Según el apartado anterior, para cada una de las funciones base, que forman la red, se han definido tres parámetros que las caracterizan. Lo ideal, a la hora de ordenar estas funciones base, sería tener en cuenta estos parámetros, para que el resultado final refleje más fielmente las distinciones entre estas funciones base.

Para realizar esta ordenación vamos a utilizar conceptos de las técnicas de optimización multiobjetivo [Coello, 1999]. En estas técnicas se parte de problemas donde la función de evaluación devuelve vectores de un tamaño igual al número de objetivos a tener en cuenta, a la hora de realizar la optimización. Esto se puede expresar como:

$$f : E \rightarrow \Re^n \quad (3\text{-}15)$$

donde E es un espacio de representación de las posibles soluciones para el problema y n es el número total de objetivos a satisfacer. Los valores de la función de evaluación para una solución $v \in E$, serán de la forma:

$$f(v_i) = (f_1(v_i), f_2(v_i), \ldots, f_n(v_i)) \quad (3\text{-}16)$$

Como norma general de los problemas multiobjetivo, cada uno de los objetivos se opone en cierta manera a los demás, lo que hace prácticamente imposible que se pueda alcanzar el óptimo global para todos ellos de forma simultánea. En su lugar, se debe buscar cierto compromiso entre todos los objetivos para obtener una solución aceptable.

En los problemas con un único objetivo, el conjunto de posibles soluciones siempre se puede ordenar de acuerdo a una función objetivo *f*. Esto implica que para dos posibles soluciones v_1 y v_2 siempre se cumple que:

$$f(v_1) \leq f(v_2) \quad ó \quad f(v_2) \leq f(v_1) \tag{3-17}$$

Esto no es así cuando hay varios objetivos a optimizar, ya que entonces el conjunto de soluciones no está por lo general totalmente ordenado, sino parcialmente ordenado [Pareto, 1896]. Para vectores objetivo, las relaciones anteriores se pueden definir como:

$$\begin{aligned} f(v_1) = f(v_2) &\Leftrightarrow f_i(v_1) = f_i(v_2) \quad \forall i \in 1,2,...,n \\ f(v_1) \leq f(v_2) &\Leftrightarrow f_i(v_1) \leq f_i(v_2) \quad \forall i \in 1,2,...,n \\ f(v_1) < f(v_2) &\Leftrightarrow (f_i(v_1) \leq f_i(v_2)) \wedge (f_i(v_1) \neq f_i(v_2)) \end{aligned} \tag{3-18}$$

Las relaciones > y ≥ se definen de forma análoga. En función de esto se define el concepto Pareto-dominancia entre dos soluciones como:

$$\begin{aligned} v_1 \prec v_2 \quad & (v_1 \text{ domina a } v_2) && \Leftrightarrow \quad f(v_1) < f(v_2) \\ v_1 \preceq v_2 \quad & (v_1 \text{ domina debilmente a } v_2) && \Leftrightarrow \quad f(v_1) \leq f(v_2) \\ v_1 \propto v_2 \quad & (v_1 \text{ es indiferente a } v_2) && \Leftrightarrow \quad (f(v_1) \not\leq f(v_2)) \wedge \\ & && \quad (f(v_2) \not\leq f(v_1)) \end{aligned} \tag{3-19}$$

En nuestro caso, los parámetros que definen una función base van a ser los objetivos que vamos a tener en cuenta a la hora de realizar la ordenación, tal y como se describe a continuación.

3.6.2 Método de ordenación detallado

Dados los objetivos a_i, e_i y s_i, para la función base ϕ_i, se va a describir un método de ordenación para las funciones base, basado en los conceptos enunciados anteriormente. El algoritmo de ordenación es muy simple y sus principales pasos se muestran en el Algoritmo 3-3.

Su funcionamiento se basa en intentar identificar la peor función base, en cada iteración. Así, inicialmente, se buscará la peor función base ϕ_i, que será aquella que sea peor en sus tres objetivos, es decir, que tenga el menor valor en a_i, el mayor valor para e_i, y el mayor valor para s_i. Si no existe se buscarán funciones base que sean peores en dos objetivos cualesquiera. Si tampoco existen, se buscarán funciones base que sean peor en algún objetivo al azar.

Tras la aplicación del método, y dadas dos funciones base cualesquiera ϕ_i y ϕ_j, se van a poder comparar mediante la relación de orden $\preceq$, que se ha definido, cumpliéndose que:

$$\phi_i \preceq \phi_j \quad ó \quad \phi_j \preceq \phi_i \tag{3-20}$$

Una vez ordenadas el mecanismo de selección es muy sencillo. Este consiste en elegir como funciones base sobre las que se va aplicar un operador, a las $m/2$ peores funciones base, siendo m el número total de funciones base que integran la red.

```
1. Introducir en el conjunto C todas las funciones
   base
2. Mientras C ≠ ∅
   A. Si existe una RBF φp que sea peor en los tres
      objetivos sacarla de C e ir al paso E
   B. Establecer las tres parejas de dos objetivos
   C. Mientras exista una pareja de objetivos no
      escogida
      a. Escoger una pareja de objetivos al azar
      b. Si existe una RBF φp que sea peor en estos
         dos objetivos ir al paso E
   D. Escoger un objetivo al azar y determinar
      aquella RBF φp que sea peor en ese objetivo
   E. Extraer φp de C y establecerla como la
      siguiente en el orden
```

Algoritmo 3-3: Algoritmo para ordenar las funciones base

Esto quiere decir, que los objetivos/parámetros a_i, e_i y s_i de estas m / 2 funciones base serán los que analice el sistema de lógica difusa para decidir la probabilidad con que cada operador se aplicará sobre estas funciones base. El resto de las funciones base se pasan, tal y como están a la siguiente generación.

3.7 Operadores

3.7.1 Introducción

En este apartado se describen los operadores a aplicar sobre los individuos de la población. Estos operadores, junto con la introducción de nuevas funciones base, van realizar los cambios pertinentes en la red o en la población, caracterizando su evolución y la consecución de una solución.

A partir del trabajo [Yao, 1999] donde se hace un estudio del diseño evolutivo de redes neuronales en general, se puede concluir que no existen apenas, operadores estándares para este fin. Aunque, eso sí, se pone de manifiesto que no se suelen usar operadores de tipo cruce en los algoritmos clasificados como estrategias de evolución. Además, se duda de la efectividad de éstos, cuando se trabaja en el diseño de redes en general

Si ahora se examinan los operadores usados en los distintos algoritmos evolutivos, utilizados en el diseño de RBFNs, se puede concluir que existe una gran variedad de estos. Así, podemos encontrar los operadores de cruce y mutación puros en [Whitehead, 1996] formando parte de un algoritmo coevolutivo genético, aunque también destaca la utilización, en el mismo trabajo, de un operador de mutación más propio en las estrategias de evolución. En [Rivas, 2001], se implementa un algoritmo genético que evoluciona una población, donde cada individuo representa una red. En este trabajo se encuentran desde operadores que trabajan sobre individuos en sí, como es normal, hasta otros más especializados, que trabajan sobre los parámetros de una función base. En [González, 2001] se implementa un algoritmo, que como el anterior tiene una base genética, en la evolución de una población de redes. En este algoritmo, destaca la inclusión de un conjunto de operadores derivados de la utilización de diversas técnicas, típicas en el diseño de RBFNs.

La conclusión final obtenida, tras este breve estudio, es que es muy difícil establecer unos operadores estándares para el diseño de redes, y que éstos normalmente dependen del algoritmo analizado y de las líneas de actuación marcadas por este.

En este trabajo se propone el uso de tan solo dos operadores. Un primer operador de eliminación de individuos de la población, que se aplicaría en función de distintas probabilidades. Un segundo operador de adaptación, que se encargaría de adaptar la función base dependiendo de la salida ofrecida por la red, para los puntos que caen dentro del radio de la función base en sí. La justificación para utilizar estos operadores se encuentran en las directrices marcadas por el algoritmo. En primer lugar estamos ante un algoritmo que se podría tipificar dentro de la clase estrategias de

evolución, donde como se ha descrito, a veces sólo se usa un operador de mutación, lo que contribuye a simplificar el algoritmo. Por otro lado, también se quiere introducir el componente de análisis de la información que nos ofrece el problema. Estos criterios se recogen en los operadores que se detallan a continuación.

3.7.2 Descripción de los operadores

En este apartado se hace una descripción y un análisis de los operadores que se emplean en el algoritmo.

3.7.2.1 Operador de eliminación. ELIM

Tal y como se ha descrito, las funciones base pertenecientes al conjunto de candidatas, tendrá una probabilidad de eliminación, como resultado de la aplicación del sistema de lógica difusa. Así pues, el funcionamiento del operador de eliminación (ELIM), es muy simple y consiste en eliminar de la población/red, la función base que se le pase.

3.7.2.2 Operador de adaptación. ADAPT

El operador de adaptación (ADAPT), como su nombre indica va a intentar adaptar una función base a la zona del espacio de entrada en que ésta se encuentre. Para esto se calculará una variación tanto del centro, como del radio de la RBF, en función del valor que tomen los distintos parámetros. Además, se introducirá cierta componente aleatoria.

Más concretamente, el cálculo del factor de variación para el centro y radio se realiza para cada uno de los puntos que se encuentren dentro del radio de la RBF tratada. Este factor de variación dependerá tanto de los parámetros a variar como de la salida que presente la red para cada uno de los puntos citados. De esta manera se irán sumando todas estas "variaciones por punto" para obtener el sentido de la adaptación que finalmente se aplicará al parámetro que se esté tratando. La variación final sobre este parámetro será igual a un pequeño valor aleatorio, que recoja el sentido anteriormente comentado. Con este funcionamiento se pretende que el cambio que se realice en el parámetro en cuestión sea pequeño, ya que de otra forma se podrían producir cambios bruscos en la configuración de una RBF, lo que iría contra el objetivo, que persiguen este tipo de operadores, que es realizar variaciones progresivas.

Variación del centro una función base

Para calcular la variación de cada una de las coordenadas c_i, del centro $\overline{c}_i$, de la RBF ϕ_i, para un determinado punto $\overline{p}_j$, dentro del radio de ϕ_i se tiene en cuenta tanto la posición de la coordenada en el espacio, como el error producido por la red de la siguiente forma:

$$\Delta c_i = signo(c_i - p_i) \cdot e(\overline{p}_j) \cdot w_i \qquad (3\text{-}21)$$

donde la función *signo*(x), devuelve el signo de x, $e(\overline{p}_j)$ devuelve el error que presenta la red para el punto $\overline{p}_j$, y w_i es el peso de la función base ϕ_i. Como se puede observar el funcionamiento es similar al algoritmo LMS, adaptado a los centro de una función base. Las variaciones de cada componente c_i, del centro $\overline{c}_i$, se irán acumulando para cada uno de los puntos que se encuentren dentro del radio de ϕ_i.

Al final, se obtendrá una variación acumulada total por componente, de la que sólo nos interesará, por los motivos expuestos anteriormente, su signo. Así la variación real que se realizará para cada componente del centro de una función base será igual a una cantidad aleatoria, entre el 0 y el 10% del radio de la RBF, multiplicada por el signo de la variación absoluta total.

Variación del radio

El algoritmo empleado en el cálculo de la variación de radio d_i, de una función base ϕ_i, es similar al empleado para el cálculo de la variación de su centro. Así para cada punto $\overline{p}_j$ que se encuentre dentro del radio de ϕ_i, la variación del radio se define como:

$$\Delta d_i = \begin{cases} e(\overline{p}_j) \cdot w_i & si \quad \| \overline{c}_i - \overline{p}_j \| > (d_i / 3) \\ -e(\overline{p}_j) \cdot w_i & si \quad \| \overline{c}_i - \overline{p}_j \| < (d_i / 3) \end{cases} \qquad (3\text{-}22)$$

donde $e(\overline{p}_j)$ representa el error que ofrece la red para el punto $\overline{p}_j$, w_i es el peso de la función base ϕ_i, y $\| \cdot \|$ es la distancia euclídea. En este caso la variación que se calcula es similar a la que calcularía un algoritmo LMS, aplicado al radio. Como se puede observar esta variación depende en su signo de la cercanía al centro de ϕ_i.

Al igual que en el caso anterior estas variaciones se sumarán y se obtendrá una cantidad de la que sólo nos interesará su signo. De este modo, la variación absoluta que realmente se aplicará al radio será igual a una cantidad aleatoria, similar a la anterior, multiplicada por el signo de la variación inicial obtenida.

3.8 Estrategia de aplicación de los operadores

3.8.1 Introducción

Una de las novedades que aporta el método coevolutivo presentado es la estrategia de aplicación de los operadores. Normalmente en este tipo de métodos las técnicas que deciden que operador se aplica, son básicamente probabilísticas. En resumen, y tal y como se ha explicado en el primer capítulo, la estrategia clásica que se sigue en los algoritmos evolutivos en general es la siguiente. En primer lugar se determina el conjunto de candidatos sobre los que se van a aplicar los operadores. Después, se asigna a cada a uno de los individuos de este conjunto un valor, que indica la probabilidad, de que se aplique sobre este individuo, un determinado operador. Esta probabilidad depende del operador y suele ser igual para todos los individuos. Por ejemplo, es normal definir en un algoritmo genético, una probabilidad de mutación p_m para todos los miembros del conjunto de candidatos, siendo $p_m = 1/l$ donde l es la longitud de los cromosomas empleados.

Como se puede deducir de la exposición anterior, un inconveniente de este tipo de estrategias puede ser la generalidad con la que se aplican los operadores. De una parte el único criterio que se tiene para aplicar un operador es meramente probabilístico y de otra la probabilidad de aplicar un operador al conjunto de candidatos es igual para todos ellos.

En este trabajo se propone una estrategia para la aplicación de los operadores cuya principal característica sea la adaptatividad. La idea es definir una técnica que analice todos y cada uno de los individuos pertenecientes al conjunto de candidatos a los que se les va a aplicar un operador. Tras este análisis se definirán para cada uno de los individuos de este conjunto, la probabilidad de aplicación de cada uno de los operadores. Lo importante de éste método es que el diseño de las probabilidades de aplicación de los operadores, que se hace para cada candidato, es un diseño a medida o un diseño adaptado en función del individuo analizado.

Los parámetros que se van a tomar como base para realizar el análisis de un individuo van a ser los parámetros utilizados para definir la aportación de crédito de un individuo. Esto tiene sentido hacerlo desde la idea de que estos parámetros caracterizan bastante bien al individuo como tal, su labor en la red y la relación con respecto al resto de los individuos.

Tal y como se ha citado, los parámetros que caracterizan al individuo o función base ϕ_i son tres. El primer parámetro es a_i, y aporta una medida de la importancia o trabajo relativo que realiza la función base. El segundo parámetro, e_i, ofrece una medida del error que se comete dentro del radio de la función base. El tercer

parámetro, s_i, mide el solapamiento de la RBF ϕ_i, con respecto al resto de RBFs. Una descripción más detalla de estos parámetros se encuentra en el apartado donde se describe la aportación de crédito de un individuo.

El siguiente punto sería definir qué método de análisis y diseño se va a implementar para que a partir de una función base, se infieran las probabilidades de aplicación de los operadores existentes. El sistema propuesto para realizar la tarea anterior es un Sistema de Lógica Difusa (SLD). Un SLD permite realizar inferencias, o lo que es lo mismo obtener una serie de consecuencias a partir de unos antecedentes, utilizando la lógica difusa.

Las justificaciones para usar este sistema de razonamiento, se puede encontrar en las bases de la lógica difusa en particular y del soft computing en general. Para comenzar, parece lógico usar una técnica de razonamiento que se define soft computing y bioinspirada como la que aporta un SLD, y que encaja perfectamente con el resto de tecnologías soft computing y bioinspiradas, que se están utilizando. Particularizando sobre la lógica difusa, su elección, también se apoya en las ventajas que esta aporta. La lógica difusa proporciona un sistema de razonamiento muy similar al razonamiento humano, captando muy bien el tipo de sentencias, reglas, e inferencias que las personas practican cotidianamente. Por tanto, la elección de este sistema de razonamiento, nos va a permitir reflejar muy bien nuestro conocimiento sobre las redes de funciones de base radial. Este conocimiento está compuesto por ejemplo de sentencias que pueden reflejar el grado de solapamiento de una RBF, la importancia de esta, etc, o por reglas que indiquen cual es la opción más idónea ante unos determinados antecedentes.

Una vez que se han obtenido para una RBF determinada, las probabilidades de aplicación de los operadores ELIM y ADAPT, solamente restará generar un número aleatorio que decida, el operador que se aplica.

A continuación se detallan las características del sistema de lógica difusa empleado.

3.8.2 Sistema de Lógica Difusa utilizado

Tal y como se ha comentado en al apartado 1.6.5, un SLD permite realizar inferencias o razonamientos difusos. Según la Figura 1-20, un SLD va a transformar unas variables de entrada en unas variables de salida y está compuesto de un fuzzificador, una base de conocimiento, un motor de inferencia, y un defuzzificador. Por tanto para definir el SLD empleado, describiremos cada uno de estos elementos.

Siguiendo un orden lógico comenzaremos con la descripción de las entradas. Las entradas del SLD serán los elementos que, tras su proceso, se emplean para

realizar las inferencias mediante las reglas. En nuestro caso, el objetivo va a ser obtener el conjunto de probabilidades de aplicación de los operadores sobre un individuo o función base, por lo que estas serán las salidas de nuestro SLD. Las entradas serán por tanto los parámetros a analizar para cada una de las funciones base. Estos parámetros ya se han descrito anteriormente y son los que van intervenir en la definición de la aportación de crédito de una función base. La elección de estos parámetros se justifica por el simple hecho de que define la aportación de crédito de la RBF, y por lo tanto creemos que caracterizan bastante bien a una función base

3.8.2.1 La base de conocimiento

La base de conocimiento de un SLD va a ser uno los elementos que más definen y particularizan un sistema de este tipo, pues va a albergar el conocimiento experto que se utiliza en la obtención de razonamientos o inferencias. Una base de conocimiento se suele dividir en dos partes: la base de datos y la base de reglas difusas.

La base de datos

En la base datos de un SLD se almacena la información difusa, que luego se va a utilizar para realizar el razonamiento. Ejemplos de esta información serían las variables y etiquetas lingüísticas, que se usarán en el fuzzificador, el defuzzificador y en las reglas, funciones de pertenencia, etc.

En nuestro caso particular se ha definido una variable lingüística para cada una de las variables de entrada, y para cada una de las variables de salida. Así, cuando se analiza una función base, se tienen las variables de entrada, *a*, *e* y *s*, que definen las variables lingüísticas *va*, *ve* y *vs* respectivamente. Mientras que como variables de salida, para la probabilidad de aplicar el operador ELIM se ha definido la variable *velim*, mientras que para la probabilidad de aplicar el operador ADAPT, se ha definido la variable *vadapt*.

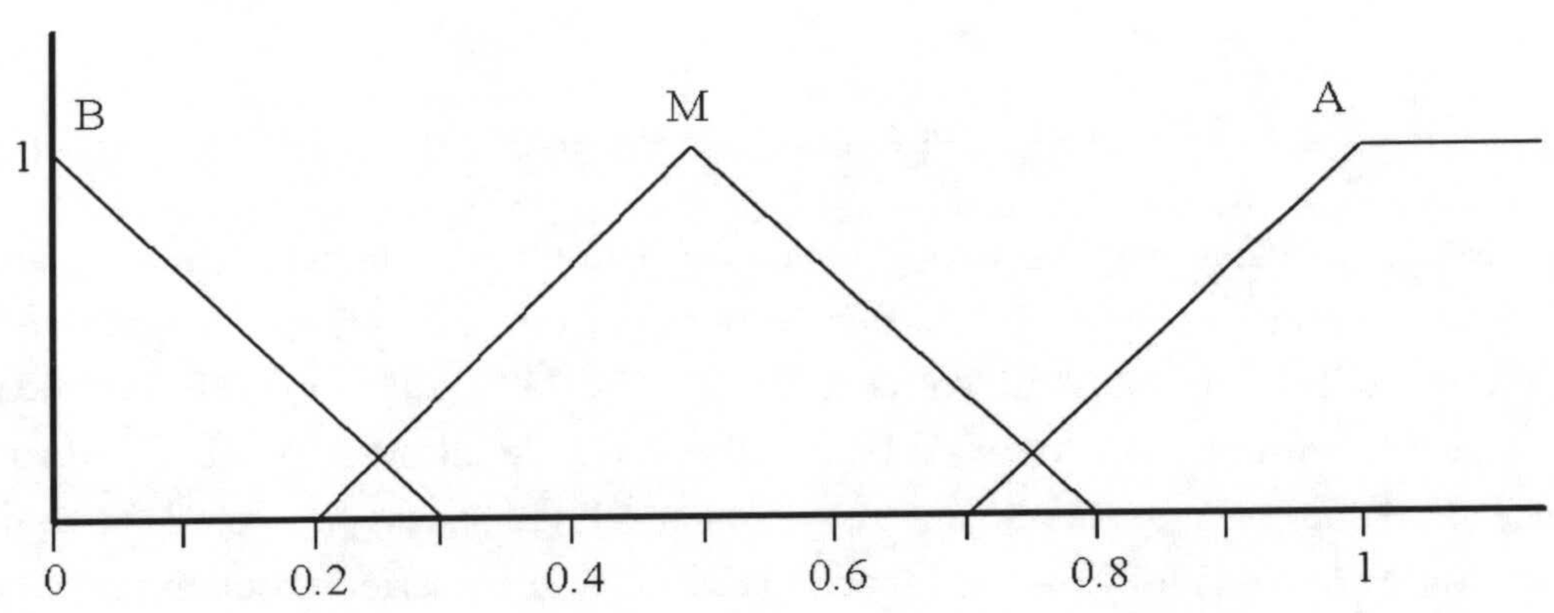

Figura 3-1: Funciones de pertenencia para las variables de entrada

Para cada una de las variables de entrada se han definido a su vez, una serie de etiquetas lingüísticas. A la determinación de las etiquetas lingüísticas de una variable lingüística se le denomina partición del espacio de entrada. Concretamente para las tres variables de entrada se define el conjunto de etiquetas lingüísticas {Baja, Media, Alta}. Estas tres etiquetas lingüísticas definen, sus correspondientes conjuntos difusos. Como función de pertenencia para estos conjuntos difusos se ha elegido una función triangular. En la Figura 3-1, se muestran las funciones de pertenencia para cada una de estas etiquetas lingüísticas. Como se puede observar cada una de las etiquetas lingüísticas tiene un centro y una base de acuerdo a la semántica que representan.

Con respecto a las variables de salida se define ahora el conjunto de etiquetas lingüísticas {Baja, Media-Baja, Media-Alta, Alta}. Esta división representa ahora lo que se conoce como división del espacio de salida. Al igual que ocurría con las variables de entrada, este conjunto de etiquetas lingüísticas es común para las variables de salida. Para las funciones de pertenencia correspondientes a los conjuntos difusos que definen estas etiquetas lingüísticas, también se ha elegido una función de pertenencia triangular. Estas funciones de pertenencia se muestran en la Figura 3-2, y tal y como ocurría anteriormente cada una de las etiquetas lingüísticas tiene un centro y una base de acuerdo a la semántica que representan.

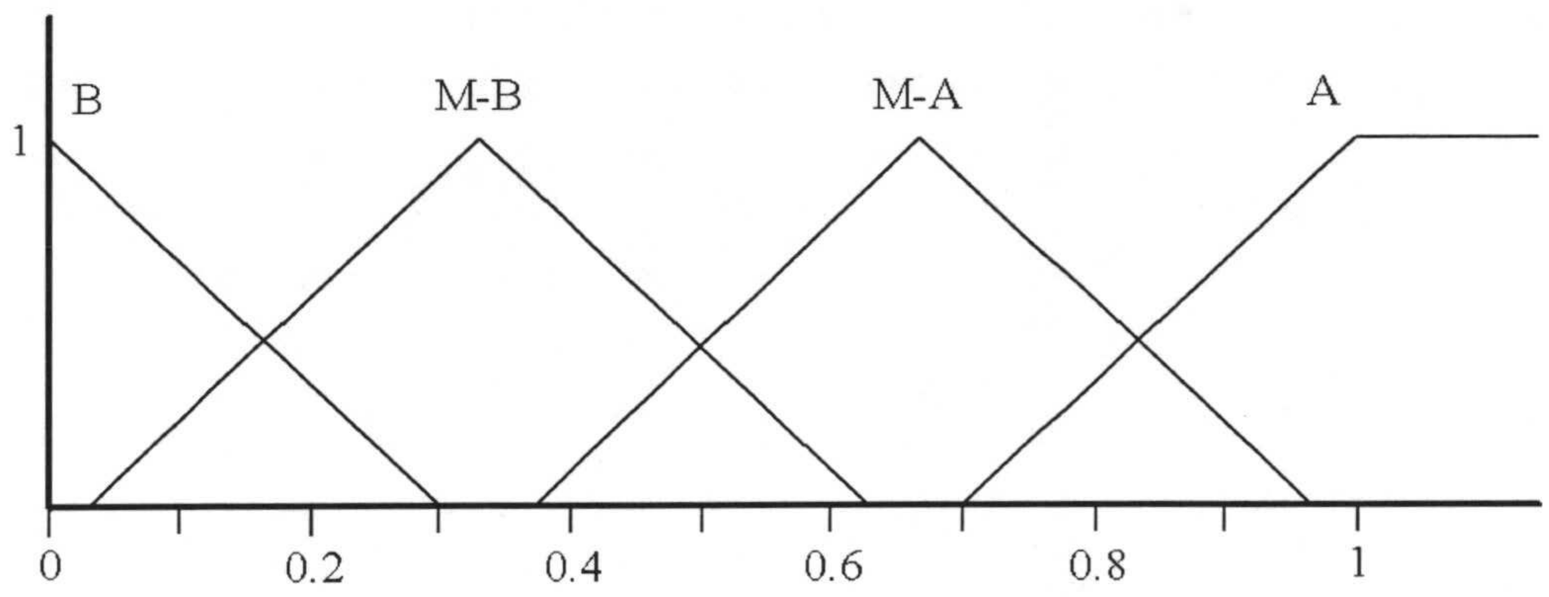

Figura 3-2: Funciones de pertenencia para las variables de salida.

Mediante estas herramientas, que nos proporciona la lógica difusa, estamos plasmando el conocimiento experto que se posee al trabajar con RBFs. Así, cuando llega una de las variables de entrada, se fuzzificará y su valor se cuantificará, estableciendo como es este valor de bajo, de medio o de alto. Las pruebas realizadas demuestran que sólo se necesitan tres etiquetas lingüísticas para obtener buenos resultados. Por otro lado, y para el espacio de salida, es necesario introducir una cuarta

etiqueta lingüística, para tener más precisión a la hora de definir los valores de las variables de salida.

La base de reglas difusas

La base de reglas difusas es el elemento principal del razonamiento difuso y por extensión de un sistema de lógica difusa. Estas reglas las definirá el conocimiento experto y almacenarán las directrices que guían el proceso razonamiento.

Una regla de un sistema difuso va a representar una relación o una implicación entre una serie de antecedentes y una serie de consecuentes. Nuestras variables de entrada, *a*, *e* y *s*, una vez fuzzificadas, *va*, *ve* y *vs* van a constituir los antecedentes de nuestras reglas. De otra parte, las variables de salida *velim* y *vadapt* se van a utilizar como las variables consecuentes de nuestras reglas. El conjunto de reglas a utilizar se muestra en la Tabla 3-1.

Si se analiza el conjunto de reglas, se observa que cumple la propiedad de completitud ya que se contemplan todos los valores que puedan tomar las variables de entrada. El hecho de usar un conjunto reducido de antecedentes y consecuentes, hace que el conjunto de reglas definido sea también sencillo, a pesar de que se cubre cualquier configuración que puedan alcanzar las variables de entrada.

Antecedentes			*Consecuentes*	
va	*ve*	*vs*	*velim*	*vadapt*
B			M-A	M-A
M			M-B	A
A			B	A
	B		B	A
	M		M-B	A
	A		M-A	M-A
		B	B	A
		M	M-B	A
		A	M-A	M-A

Tabla 3-1: Reglas utilizadas

Si ahora se estudia la semántica de las reglas o el conocimiento experto que almacenan, se descubre que está compuesto de una serie de directrices simples, que se usarán para guiar el razonamiento experto. Esta serie de directrices son fruto de la lógica y de la heurística.

Comenzaremos por la variable *velim* que representa la probabilidad de aplicar el operador ELIM u operador de borrado. Si se analiza y por lógica, existe más probabilidad de aplicar este operador a una RBF, cuando la variable *va* sea menor.

Esto implica mientras más bajo sea el peso de una RBF o ésta afecte a un menor número de puntos del conjunto de entrenamiento, mayor será la probabilidad de que se elimine esta función base. Con la variable *ve* ocurre algo similar que con *vp*, aunque en este caso la probabilidad de aplicar ELIM crece proporcionalmente al aumento de *ve*. Esto tiene su lógica, ya que cuanto más índice de error tenga la RBF tratada mayor será la probabilidad de que esta se borre. La variable *vs* tiene unas características prácticamente iguales a las anteriores, lo que implica que un mayor nivel de solapamiento de la RBF con sus vecinas, conlleva una mayor probabilidad de que se borre la RBF que se esté analizando.

Continuamos ahora analizando la probabilidad de aplicación del operador ADAPT, o los valores que va a ir tomando la variable *vadapt*. Con respecto a la variable *va,* se observa una relación relativamente proporcional entre los valores que tome *vp* y los que en consecuencia tomará *vadapt*. De esta manera, suele ocurrir que cuanto mayor sea la importancia de la RBF, o lo que es lo mismo, su peso y el número de puntos a los que afecte mayor será la probabilidad de adaptar la función base. De todas formas se observa que en general el valor de *vadapt* se va a mantener alto y constante cuando el peso supera un valor intermedio. Mediante el establecimiento de la variable *vadapt*, con unos valores altos y por encima de los de *velim*, se pretenden alcanzar varios objetivos. El primero es mantener una evolución progresiva de la red, donde entre generación y generación no ocurran un número alto de cambios que rompa una continuidad en la acercamiento a una solución final. Por otro lado cuando una RBF tiene una variable *va* baja, evidentemente no está teniendo un trabajo significativo para la red, pero eso simplemente puede ser porque no tenga sus parámetros bien ajustados y por tanto el entrenamiento haya decidido por minimizar su efecto. Por tanto es conveniente, probar a adaptar esta función base antes que borrarla. Para las variables *ve* y *vs* las reglas son las mismas y por lo tanto se pueden hacer los mismos comentarios que para la *va*. Así para la *ve*, que mide el error que se produce en la zona donde se encuentra la función base, el hecho de que exista cierto error puede ser porque la RBF esté mal ajustada, por lo que se prefiere aumentar la probabilidad de que se adapte la función base antes de que se borre. Con la variable *vs* pasa algo similar y es que uno de los criterios que se ha predefinido a la hora de diseñar una RBFN, es que vamos a permitir cierto solapamiento entre las funciones base, por lo que igualmente se prefiere tener más probabilidad de adaptar la RBF que de borrarla.

De todas formas siempre hay que tener en cuenta que el valor final de *velim* y de *vadapt*, va a depender de los valores de los tres antecedentes. Así cada antecedente va a realizar una cierta aportación a los valores de *velim* y *vadapt*. Esta aportación la ponderará conveniente el motor de inferencia, y el defuzzificador extraerá de esta ponderación valores finales para las probabilidades de aplicar los operadores, ELIM y ADAPT.

3.8.2.2 Motor de inferencia

El motor de inferencia en un SLD, se encarga de extraer unas consecuencias, tras aplicar las reglas a unos determinados antecedentes. Con esta premisa, los valores que van tomar las variables consecuentes, van a depender tanto de los antecedentes que el fuzzificador haya extraído de las variables de entrada que se presentan en un momento dado, como de la base de reglas difusas que haya establecida. Pero también estos valores de las variables consecuentes van a depender del motor de inferencia que se haya elegido, ya que existen diferentes métodos para extraer consecuentes en función de unas determinados reglas y de unos determinados antecedentes.

El motor de inferencia elegido en nuestro trabajo es uno de los más utilizados y fue propuesto por Mandani en [Mandani, 1975]. Este mecanismo de razonamiento se termina de configurar estableciendo como t-conorma el máximo, y como t-norma el mínimo. El funcionamiento concreto de este motor de inferencia se muestra en la Figura 1-17.

3.8.2.3 El fuzzificador

La función de un fuzzificador en un sistema de lógica difusa es transformar los valores nítidos que se reciben por la entrada en valores difusos. Estos valores difusos se pasarán al motor de inferencia, donde se convertirán en los antecedentes de las reglas.

En nuestro sistema, el fuzzificador recibirá los valores de las tres variables de entrada *a*, *e* y *s*. Para cada una de las variables se fuzzificará su valor y se obtendrá una cuantificación difusa de él, para cada una de las tres etiquetas o conjuntos difusos de que está compuesta su variable lingüística correspondiente.

Para realizar la fuzzifcación se usa el método tradicional, que consiste en situar el valor de la variable de entrada en el eje *x* de la Figura 3-1 y obtener el valor del eje *y* para cada una de las tres funciones de pertenencia o conjuntos difusos correspondientes.

3.8.2.4 El defuzzificador

Tras aplicar nuestro mecanismo de razonamiento se obtienen unos valores que se van a corresponder con etiquetas lingüisticas de variables lingüisticas, o conjuntos difusos. Para un determinado consecuente esto se va a poder representar gráficamente con una función. El defuzzificador se va a encargar de obtener un valor nítido a partir de los valores almacenados en las etiquetas lingüísticas. Existen diferentes tipos de defuzzificadores tal y como se muestra en el apartado 1.6.4, mostrándose en la Figura 1-19, su funcionamiento.

En nuestro caso se ha optado por elegir una de las estrategias de defuzzificación más comunes como es la estrategia centro del área. Gráficamente, su funcionamiento consistiría en obtener el centro del área de la forma de la función que delimitarían las etiquetas lingüisticas, para una variable lingüística determinada. Para esto se utiliza la fórmula de la expresión (1-83).

La salida del defuzzificador es la salida de nuestro sistema de lógica difusa. Esto implica que este módulo ya nos devolverá las probabilidades que hay de aplicar el operador ELIM o el ADAPT.

3.9 Introducción de nuevas RBFs

3.9.1 Introducción

En esta parte del algoritmo, se procederá a la introducción de un número determinado de RBFs. El número de RBFs a introducir, será igual al número de RBFs que se hayan eliminado en el paso anterior, como consecuencia de la aplicación del operador de eliminación, ELIM.

Para comenzar realizaremos un estudio de los trabajos, descritos en los capítulos anteriores, que utilizan métodos para introducir neuronas en una red en general o para la introducción de funciones base en RBFNs, en particular. Las conclusiones de este estudio serían que la aparición de nuevas RBFs en una red suele tener dos orígenes principalmente. El primer origen suele ser la existencia en el algoritmo evolutivo, de un operador de cruce determinado. Así, dependiendo de este operador, se cogerá la información de los padres que intervengan en el cruce, para formar el nuevo individuo de la población. El segundo origen está relacionado con el error que presenta la red cuando se le pasan determinados puntos del conjunto de entrenamiento. Este hecho normalmente se aprovecha para introducir, teniendo en cuenta algún que otro criterio, una RBF en los puntos que más error presenten.

Si se analizan ambos métodos, en cuanto a la utilización de un operador de cruce, se puede decir que está ligado a algoritmos evolutivos, y que no es muy estándar, ya que depende mucho del algoritmo en el que se encuentre. Incluso en [Yao, 1999] se duda de la eficacia de un operador de este tipo en algoritmos evolutivos para el diseño de redes neuronales, sobre todo si el tipo de algoritmo evolutivo utilizado, no es un genético. Ejemplos de este tipo de operador se encuentran en [Whitehead, 1996] o [Rivas, 2001], teniendo ambos una base genética.

Tal y como se ha comentado, la otra forma clásica de introducir RBFs en una RBFN, consiste en hallar los puntos, para los que la red funciona peor y genera un

mayor error. En algunos de estos puntos se añadirán las nuevas RBFs. Esta forma de agregar RBFs se utiliza, por ejemplo en los algoritmos de la familia RAN y en prácticamente todos los algoritmos incrementales/decrementales, tal y como se ha descrito en el capítulo anterior. También se usan variaciones de este método en algunos de los algoritmos de clustering supervisados como el CFA (González, 2001b). Pero incluso esta técnica se introduce en algunos algoritmos evolutivos que trabajan en el diseño de RBFNs, como se muestra en [Gonzalez, 2001].

En nuestro caso se ha optado por agregar RBFs a nuestra red, utilizando una técnica basada en el análisis del error de la red para una determinada configuración. Una de las razones para hacerlo así, es el tipo de algoritmo evolutivo que se está manejando. Este algoritmo se podría clasificar como un tipo de *estrategia de evolución*, donde no se tiene tan claro [Yao, 1999] el uso de un operador de cruce, entre otras cosas porque el tipo de codificación empleada en la representación de los individuos, no se presta demasiado a ello. Otra razón para usar este tipo de método, son las directrices generales de funcionamiento del algoritmo que se está implementado. Estas directrices, proponen un funcionamiento donde se analiza a fondo, toda la información que el problema nos suministra para tomar decisiones que nos lleven a la solución de éste. Así, por ejemplo, para decidir que operador aplicar, sobre una determinada función base, se hace un estudio de tres parámetros representativos de ésta. El uso de un operador de cruce está ligado más algoritmos de tipo genético, donde los criterios de funcionamiento no analizan tanto la información, que en un momento determinado, puede suministrar el problema.

El método que se propone para la introducción de RBFs aunque tenga unos fundamentos clásicos aporta ciertas novedades. El funcionamiento del método clásico consiste normalmente en delimitar un primer conjunto de puntos donde la red genere un mayor error. Partiendo de ese conjunto se identificarán aquellos puntos, que además estén a una distancia mínima de una RBF o de su radio. Normalmente, en estos puntos se introducirá una función base. Nuestro método propone que en lugar de determinar puntos aislados donde se produzca un mayor error, se delimiten zonas de error, con un determinado radio. Con esto se pretende que el algoritmo sea más robusto a la hora de introducir RBFs, ya que las funciones base se añaden en zonas de mayor error y no en puntos de mayor. Además trabajando con zonas y no con puntos, nos aseguramos que nuestro algoritmo funcionará mejor en los casos en el conjunto de entrenamiento contenga ruido. En este tipo de conjuntos de entrenamientos es muy probable, que puntos de mayor error no se correspondan con zonas de mayor error, y está claro que en esa situación conviene introducir la nueva RBF en una zona de mayor error.

3.9.2 Descripción del método

El método para la introducción de RBFs en la red va a tener dos partes principales. La primera consiste en la identificación de un determinado número de zonas donde se produzca mayor error. La segunda parte situara las RBFs en esas zonas.

3.9.2.1 Delimitación de las zonas

El algoritmo usado para la determinación de estas zonas se muestra en el Algoritmo 3-4. Este método cuenta a su vez, con dos partes. En la primera, hasta el paso 3, se delimitarán los puntos que se encuentran fuera del radio de acción de cualquier RBF, introduciéndose éstos en un conjunto C. Estos puntos son los que se tratarán para formar las zonas de inclusión de RBFs. El problema es que si el espacio de entrada está muy cubierto por las distintas RBFs, ocurrirá que no existirán puntos donde establecer nuestras zonas. Por esto se establece que el conjunto deba tener un mínimo de puntos, ε, para poder continuar, ya que en caso contrario no tendrá mucho sentido delimitar zonas. En caso de que C no tenga este mínimo número de puntos, se vuelve a delimitar un conjunto de puntos C, pero considerando que las funciones base tienen tan sólo un tanto por ciento de su radio original. Esto nos proporcionará más puntos para el conjunto aunque las nuevas RBFs se solapen un poco con las antiguas. Esta disminución del radio se llevará a cabo mientras C no tenga un conjunto suficiente de puntos. El solapamiento que se pueda producir al introducir las nuevas RBFs, es un elemento que nuestras directrices contemplan. Además en posteriores generaciones, este solapamiento será analizado por los operadores y si es muy grande, se borrará la RBF con mucha probabilidad.

La segunda parte de nuestro algoritmo, puntos 4 y 5, es la que propiamente identifica las zonas. Para esto se usa un bucle donde en cada iteración se calcula el punto p_m de C para el que la red genera un mayor error. Con este punto, se delimitará una zona, donde p_m será el centro de esta zona. Como en principio, no se permite que las zonas compartan puntos, se eliminarán de C, todos los puntos que estén dentro de la zona que delimita p_m. Para esto se establece un radio *medrad* para la zona en cuestión, calculándose este radio como la media de los radio d_i de las RBFs ϕ_i, que actualmente se encuentran en la red. En resumen se eliminan de C todos los puntos que estén a una distancia inferior a *medrad* de p_m. Este proceso continua hasta que C se queda sin elementos.

1. *muld* = 1;
   Establecer *medrad* como la media de los radios d_i de las RBFs actuales;
   Establecer ε al 10% del número de puntos del conjunto del entrenamiento.
2. Delimitar los puntos del conjunto de entrenamiento que están a una distancia superior a $d_i \cdot muld$ de cualquier RBF e introducirlos en el conjunto C.
3. Si Cardinal(C) > ε entonces continuar
   si no $muld = muld \cdot 0.75$ y volver a 2.
4. Mientras $C \neq \varnothing$
   a. Hallar el punto de mayor error p_m de C.
   b. Eliminar los puntos de C que estén a una distancia de p_m inferior a *medrad*.
   c. Establecer zona con centro p_m.
5. Si el número de zonas es suficiente acabar
   si no $medrad = medrad \cdot 0.75$ y volver a 4.

Algoritmo 3-4: Algoritmo de detección de zonas de mayor error

Ahora la siguiente cuestión es si se han delimitado suficientes zonas, teniendo en cuenta el número de funciones base que hay que introducir. La respuesta será afirmativa si simplemente se ha identificado, al menos tantas zonas como RBFs hay que introducir. En caso contrario lo que se hace es volver a repetir los pasos anteriores estableciendo unas zonas más pequeñas, o lo que es lo mismo, un radio más pequeño para estas. Esta disminución de radio se realizará hasta que se delimiten suficientes zonas, en cuyo caso terminará nuestro algoritmo.

El siguiente paso consiste en hallar el error que se produce dentro una zona. El error que se va a calcular para cada zona es el NRMSE. Se ha escogido este error porque, aunque sea un poco más costoso del calcular, nos da una mejor información del error medio que existe en la zona con respecto a la variación de la función. Hay que tener en cuenta que nuestro objetivo es minimizar este error, a nivel global del conjunto de entrenamiento, luego será interesante trabajar las zonas que presenten un valor mayor de este error.

3.9.2.2 Introducción de las RBFs

Una vez delimitadas las zonas de error, el siguiente paso consistirá en introducir en cada una de las zonas, una función base. Si se dispone de más zonas de error, que

número de RBFs hay que introducir, las zonas donde se introducirán estas RBFs, serán las de mayor error.

Para la introducción de una RBF en una zona, habrá que establecer el centro y el radio de la RBF con respecto a esa zona. La delimitación del centro es una tarea sencilla, ya que básicamente se le va a hacer coincidir con el centro de la zona. Concretamente el centro de la RBF será igual al centro de la zona en la que se introduce, más una pequeña perturbación aleatoria. Esta perturbación se introduce porque, aunque se intuye cierta relación directa entre una zona y una función base, tampoco se puede asegurar que haya una correspondencia total.

La delimitación del radio de la RBF es un proceso más elaborado que va a depender tanto de la media de los radios de las actuales funciones base como de la cercanía de otras RBFs.

La justificación para hacerlo así es la siguiente. En principio si solo se tiene en cuenta la media de los radios de las funciones base que actualmente componen la red, *medrad*, la función base que se introduce no se adapta a su entorno, sobre todo para intentar lograr un cierto grado de solapamiento con sus RBFs vecinas. Existen dos ejemplos claros donde el funcionamiento no sería el deseado. El primero es cuando la nueva RBF ϕ_n se introduce cerca de otro RBF ϕ_i, que ya existe en la red. Si se establece un radio d_n para la nueva RBF, simplemente en función de la media, probablemente estaremos formando una red, con dos RBFs demasiado solapadas. Un segundo ejemplo sería cuando se introduce un nueva RBF ϕ_n, en una zona aislada, donde el resto de las RBFs están muy lejanas. Está claro que al igual que antes si sólo se atiende a la media de los radios del resto de las RBFs, para fijar el radio d_n, estaremos dejando cierta cantidad del espacio de entrada sin cubrir.

Por otro lado si para fijar el radio de la nueva función base solo se atiende a la distancia de la RBF más cercana como en los métodos KNN y CIV explicados anteriormente, puede ocurrir que se introduzcan RBFs con radios demasiado pequeños o demasiado grandes, que por experiencia entorpecen el funcionamiento del algoritmo y enlentecen la consecución de una solución final.

Para solventar los inconvenientes de usar estos parámetros de una forma aislada, se propone una solución de compromiso entre ambos. Sus principales pasos se muestran en el Algoritmo 3-5. Así, dada una RBF ϕ_n para la que se va a establecer un nuevo radio d_n, se calcula la distancia a la RBF ϕ_c, más cercana $dist_{nc}$. En principio esta será la distancia *distref* sobre la que se calculará el radio d_n, salvo que la diferencia entre $dist_{nc}$, esté fuera del intervalo definido por la media de los radios de las RBFs actuales. En este caso se establecerá *distref*, al extremo correspondiente que haya superado la diferencia anterior. Al final d_n se establecerá como un valor aleatorio entorno a *distref*.

```
1. Calcular dist_nc como la distancia de la RBF φ_n
   a su RBF φ_c más cercana.
2. distref = dist_nc.
3. Si (distref > medrad · 1.25)
                entonces distref = medrad · 1.25
4. Si (distref < medrad · 0.75)
                entonces distref = medrad · 0.75
5. Establecer d_n a un valor aleatorio entre
   [distref, distref · 1.25]
```

Algoritmo 3-5: Algoritmo para el cálculo del radio de una nueva función base

3.10 Condición de parada

La condición de parada de nuestro bucle principal, define cuando éste termina de ciclar. Como condición de parada clásica de este tipo de bucles, normalmente se utiliza el cumplimiento de un número determinado de iteraciones, como se muestra en el apartado 1.3.4.2. Es no siempre es así, ya que en [Gonzalez, 2001] se propone una condición de parada, basada en la pendiente de la recta de regresión de las últimas soluciones obtenidas.

En nuestro algoritmo proponemos una condición de parada adaptativa en la línea de las adaptaciones que se proponen en el apartado 1.3.5. Así la idea es que se cumplan un mínimo de iteraciones iniciales para a partir, seguir iterando sólo si se producen mejoras.

Como número mínimo de iteraciones iniciales, *lim*, se proponen 300. Si embargo este límite puede variar a partir de la iteración 250. De este modo, *i*, es la iteración actual, perteneciendo *i*, al intervalo [250, *lim*], y en esta iteración se produce una mejora de la solución, se incrementa *lim*, de la siguiente manera:

$$lim = lim + 50 \qquad (3\text{-}23)$$

Este criterio sin ser costoso de calcular, refuerza la característica adaptativa del algoritmo en general.

3.11 Aplicación del algoritmo Levenberg-Marquardt

3.11.1 Introducción

Como resultado de la primera etapa del algoritmo, se obtendrá una red, que se encuentre en la dirección adecuada del óptimo global, pero todavía a una distancia relativa. El objetivo de esta fase será pues aplicar un algoritmo de optimización matemática especializado en, acercarse bastante, a un óptimo.

El problema de la optimización matemática y algunos de los principales métodos que se utilizan en este campo, se describen en el apartado 1.1. En este apartado también se muestra que estos algoritmos pueden acabar atrapados en un extremo local, dependiendo de la configuración de la que partan. Es por esto que primero se utiliza un algoritmo evolutivo, que son menos susceptibles de quedar atrapados en óptimos, para generar una primera solución. Esta solución no suele ser muy precisa, también como consecuencia de usar para su obtención la computación evolutiva. Sin embargo, si será un punto de partida idóneo para un algoritmo de optimización matemática que conseguirá, refinar esta solución mucho más, hasta prácticamente alcanzar el óptimo global.

De entre los algoritmo de optimización, uno de los más potentes por su funcionamiento es el Levenberg-Marquardt, tal y como se discute en el apartado 1.1. Este funcionamiento se caracteriza por ser adaptativo, de modo que en función de su distancia al óptimo, se aplica una metodología u otra. Usando estos elementos como justificación, se decide usar este método en el último paso de nuestro método.

3.11.2 Parámetros de uso del algoritmo Levenberg-Marquartdt

El algoritmo Levenberg-Marquardt va a realizar un ajuste final de todos los parámetros que caracterizan, la red, devolviendo por lo tanto una solución más precisa. Los principales pasos para este método se muestran en el Algoritmo 1-2.

Por otro, lado como parámetros de ejecución del algoritmo Levenberg-Marquardt, se han utilizado los que se definen de una manera estándar para él y que ya se comentaron en el apartado 1.1.4. Entre éstos destacan los que definen los tamaños en los desplazamientos en el espacio de búsqueda. Como condición de parada se establece que el algoritmo no obtenga una mejora de al menos una milésima, del error que él define, durante tres iteraciones consecutivas.

4. Experimentación y resultados

En este capítulo se va a poner a prueba el algoritmo, para cual que se le pondrá a trabajar en la aproximación de una serie de funciones con distintas características. Este conjunto comprende funciones unidimensionales y bidimensiones, y ya han sido usadas por otros autores para probar sus algoritmos. Por último también se ha testeado el algoritmo en la aproximación de series temporales.

4.1 Resultados experimentales del algoritmo

El conjunto de funciones con las que se va a trabajar, se ha concretado teniendo en cuenta dos directrices. La primera es que sea un conjunto de funciones que normalmente se utilicen en el problema de aproximación y la segunda que exista algún otro algoritmo de diseño de Redes de Funciones de Base Radial con el que podamos comparar los resultados.

Para realizar las comparaciones con otros algoritmos tendremos en cuenta dos factores: el primero será la complejidad del algoritmo de diseña que genera los modelos y el segundo, la complejidad de los modelos en sí. Para medir esta última se tienen en cuenta dos parámetros. El primer parámetro m, es una medida general de la complejidad del modelo. Así en el caso de redes neuronales, como es el nuestro, indicaría el número de neuronas que forman la red. En el caso de tratarse de un sistema difuso, m haría referencia al número de reglas utilizadas por el sistema. Esto implica que cada una de las funciones base de la red, junto con su peso asociado, puede verse como una regla neurodifusa.

El parámetro n_p, indica el número de parámetros libres que hay que establecer para definir el modelo. Así en el caso de que se trabaje con una Red Neuronal de Base Radial, n_p, será igual al número de neuronas por el número de parámetros libres que determinan cada neurona, es decir, su peso, su radio, y el número de coordenadas de

su vector centro. De todas formas estos parámetros se mostrarán siempre que tengan sentido, teniendo en cuenta el tipo de modelo que se genere, y sean conocidos.

Además de realizar estas comparativas con otros autores existen dos trabajos que son denominador a la hora de contrastar nuestro algoritmo. El primer trabajo es el de [Pomares, 2000] donde se presenta un algoritmo sistemático de identificación de sistemas difusos, el cual se valida con el problema aproximación de funciones. Éste se basa trabajar con una tabla de reglas definidas usando una partición triangular que se obtiene mediante un algoritmo iterativo en el que se añade complejidad a la tabla de reglas hasta que se alcanza un criterio de parada.

El segundo trabajo es el [González 2001], donde también se propone un método para el diseño de RBFNs, el cual se ha descrito ampliamente en el apartado 2.3.6.2. Básicamente, lo que el autor presenta en un algoritmo genético donde se mantiene una población, en la que cada individuo codifica una RBFN completa. Esto implica que en cada generación se pueden estar tratando a la vez varias decenas de redes a la vez, lo que a priori dice mucho de la potencia del algoritmo. Sin embargo esta misma característica, conlleva una gran complejidad y coste computacional asociado, si además se tienen en cuenta las técnicas numéricas que en él se manejan, ya que en todas las generaciones se aplican algoritmos como el SVD, OLS o el de Levenberg-Marquardt. Para terminar este algoritmo también lleva a cabo una fase de refinamiento del error con otra aplicación final del algoritmo Levenberg-Marquartd.

Las comparativas a realizar, no se restringen sólo a estos dos algoritmos, sino que para cada función también se tienen en cuenta otras técnicas que las han usado en su capítulo de resultados y que se explicarán brevemente en el apartado correspondiente.

Las tablas de resultados se van a confeccionar tras ejecutar el algoritmo propuesto 30 veces, que es un valor suficientemente grande para poder obtener conclusiones estadísticas. El valor obtenido para cada ejecución es el error NRMSE, ya comentado anteriormente. Así, para el conjunto de entrenamiento y para el de test, se obtienen cuatro resultados: el de la peor ejecución, el de la mejor ejecución, el valor medio de los resultados obtenidos y la desviación típica de estas 30 ejecuciones. Algunas veces para poder comparar con otros métodos, también se mostrarán los valores de otros tipos de error.

4.1.1 Funciones de una dimensión

A continuación se va a utilizar el algoritmo propuesto para aproximar una serie de funciones de una dimensión. Para estas funciones, los conjuntos de entrenamiento están formados por 100 puntos equidistribuidos en el intervalo de definición de la

función correspondiente. En cuanto a los conjuntos de test, están formados por 1000 puntos distribuidos uniformemente en dicho intervalo. Estos conjuntos de entrenamiento son iguales a los usados en [González, 2001] y similares a los de otros trabajos.

De forma general y si se analizan los resultados obtenidos por nuestro algoritmo para el conjunto de funciones propuesto, se puede observar como el error va disminuyendo de una manera coherente con la introducción de nuevas funciones base, hasta un determinado punto. A partir de este punto la inclusión de nuevas RBFs, no contribuye en nada a la mejora del error y puede empeorarlo, al empezar a darse problemas de sobreentrenamiento.

Si ahora se estudia la desviación típica, ésta tiene unos valores suficientes para afirmar que el algoritmo es robusto en la consecución de resultados. Por otra parte y si se compara los resultados del conjunto de entrenamiento y el de test se observa que son bastante similares y que el algoritmo generaliza bien.

4.1.1.1 Funciones wm_1 y wm_2

Uno de los primeros trabajos diseñados para la construcción de un sistema aproximador de funciones a partir de un conjunto de ejemplos es el algoritmo propuesto por Wang y Mendel en [Wang, 1992]. Dicho algoritmo generaba una tabla de reglas difusas, esencialmente una regla por cada ejemplo de entrenamiento, y posteriormente seleccionaba las que más se activaban.

Posteriormente Sudkamp y Hammell [Sudkamp, 1994] mejoraron este algoritmo reduciendo la posibilidad de que una regla se viera afectado por datos ruidosos. Así mismo, desarrollaron una teoría para completar la base de reglas basada en el concepto de que la similitud en la entrada debe producir similitud en la salida. Para ello presentaron dos métodos relativamente equivalentes: el *método de crecimiento por regiones* (*region growing*) y el *método de la media ponderada* (*weighted average*). Como batería de funciones para probar el funcionamiento de sus algoritmos escogieron, entre otras, las siguientes funciones:

$$wm_1(x) = x^3 \quad x \in [-1, 1] \tag{4-1}$$

$$wm_2(x) = sen(2\pi x) \quad x \in [-1, 1] \tag{4-2}$$

Si se analizan los resultados para wm_1, se observa que no merece la pena meter más de 4 funciones a la hora de formar una red, ya que a partir de aquí el error no mejora y si aumenta la complejidad de la red. Esta cota de RBFs parece razonable teniendo en cuenta la forma de la función. Si se comparan los resultados obtenidos con

los de otros algoritmos, se demuestra que el algoritmo los mejora a todos, salvo al de [González, 2001], y sólo en el caso de que el número de RBFs, se incremente. La causa puede estar en los niveles de error que se están manejando, que ya dependen mucho de la aplicación del Levenberg-Marquartd, por lo que esta diferencia puede atribuirse a ciertas diferencias en su configuración.

Si ahora se estudian los resultados de wm2, parece que no tiene sentido introducir más 7 RBFs, ya que a partir de aquí el error no mejora. Comparando los resultados con otros algoritmos, se observa que nuestro algoritmo consigue mejores valores que prácticamente cualquier otro método salvo el de [González, 2001], para el se obtienen peores resultados con un número bajo de RBFs, pero al que se supera cuando este número empieza a crecer.

En resumen el algoritmo ha demostrado un comportamiento bueno para estas funciones y ha superado en resultados a los métodos basados en sistemas difusos, incluso cuando se usa un número inferior de parámetros. La calidad de las soluciones aportadas se debe en cierta manera a las herramientas elegidas para solventar el problema de la aproximación de funciones, ya que el uso de funciones gausianas es un elemento muy recomendado para resolver este problema tal y como se ha mencionado anteriormente. Por otro lado los sistemas difusos con los que se ha comparado usan una partición triangular, con lo que no pueden llegar a aproximar de una manera tan suave como lo hace nuestro algoritmo.

Si se comparan estos métodos de diseño basados en sistemas difusos en cuanto a un índice de complejidad y coste computacional, hay que resaltar que estos métodos tendrían, en general un valor similar en este índice, al algoritmo propuesto.

Para el método de diseño de RBFNs [González, 2001], si existe cierta igualdad en el campo de los resultados. Aunque, por otro lado, si se compara la complejidad y coste computacional de ambos, el algoritmo propuesto superaría en estos términos al de [González, 2001], es decir, es menos complejo y tiene un coste computacional menor.

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.0101	0.0252	0.0115	0.0029	0.0106	0.0257	0.0117	0.0030
4	0.0028	0.0084	0.0040	0.0016	0.0030	0.0088	0.0042	0.0016
5	0.0022	0.0129	0.0064	0.0037	0.0023	0.0136	0.0067	0.0039
6	0.0015	0.0164	0.0080	0.0047	0.0016	0.0173	0.0085	0.0047
7	0.0012	0.0107	0.0052	0.0033	0.0014	0.0123	0.0059	0.0037
8	0.0014	0.0115	0.0053	0.0031	0.0019	0.0136	0.0063	0.0035
9	0.0009	0.0081	0.0040	0.0019	0.0013	0.0096	0.0050	0.0022
10	0.0012	0.0088	0.0043	0.0022	0.0018	0.0115	0.0057	0.0027

Tabla 4-1: Resultados del algoritmo propuesto para la función wm_1

Algoritmo		*m*	n_p	*Error Medio*	*NRMSE*
[Wang, 1992]		15	-	0.029	-
		25	-	0.018	-
Mejora de [Wang, 1992] en [Sudkamp, 1994]		15	-	0.079	-
		25	-	0.088	-
[Sudkamp, 1994]	RG	15	-	0.044	-
		25	-	0.021	-
	WA	15	-	0.044	-
		25	-	0.021	-
[Pomares, 2000]		4	6	0.026	0.080
		6	10	0.011	0.032
		8	14	0.006	0.017
[González, 2001]		3	9	0.0042 ± 0.0005	0.0134 ± 0.0011
		4	12	0.0001 ± 5.4E-5	0.0004 ± 0.0002
		5	15	2.1E-5 ± 1.8E-5	6.8E-5 ± 5.7E-5
Algoritmo propuesto		3	9	0.0032 ± 0.0009	0.0115 ± 0.0029
		4	12	0.0012 ± 0.0004	0.0040 ± 0.0016
		5	15	0.0018 ± 0.0009	0.0064 ± 0.0037

Tabla 4-2: Comparativa de resultados obtenidos por distintos algoritmos para la función wm_1

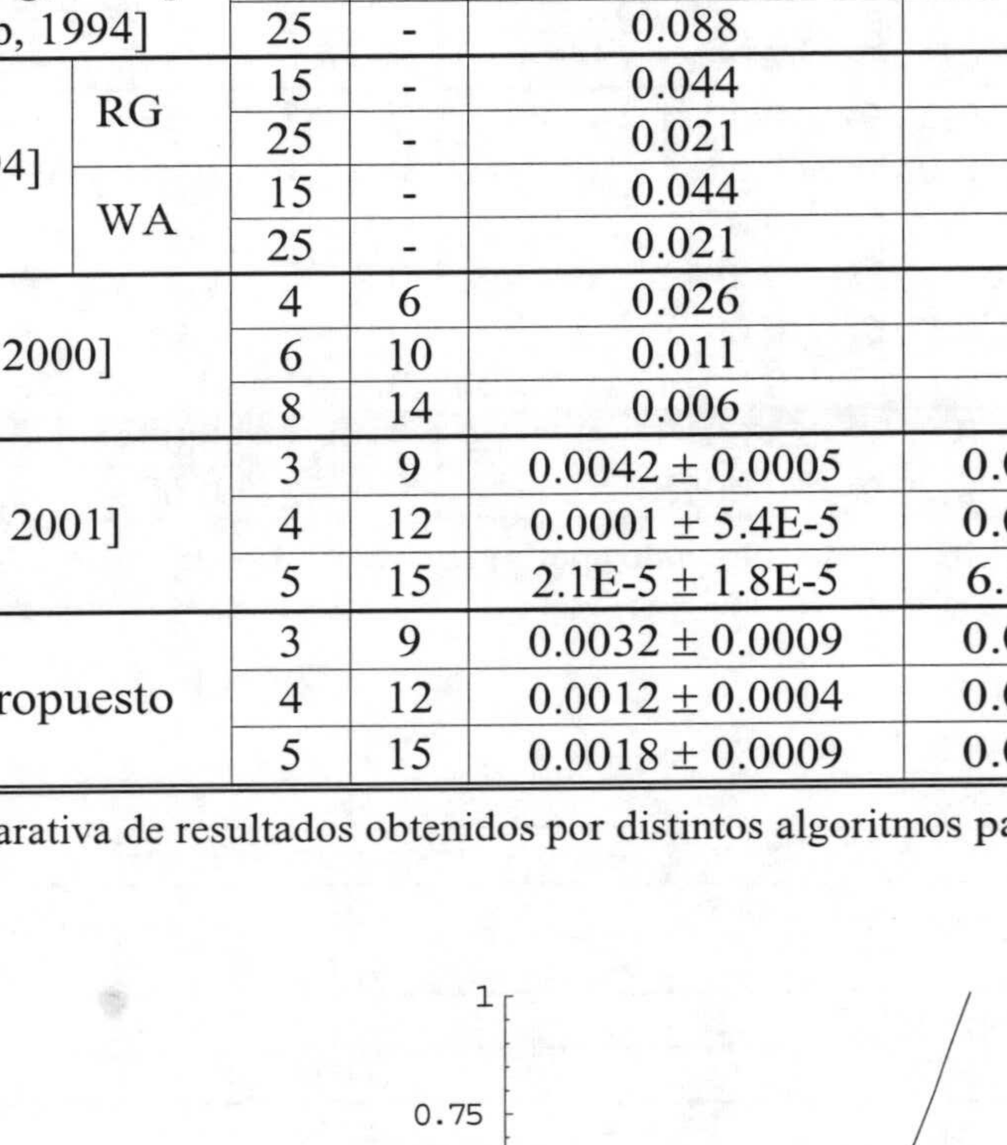

Figura 4-1: Función wm_1

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.5069	0.5076	0.5071	0.0003	0.4968	0.5376	0.5267	0.0180
4	0.0344	0.0568	0.0481	0.0063	0.0404	0.0583	0.0489	0.0056
5	0.0288	0.0345	0.0310	0.0015	0.0348	0.0400	0.0371	0.0015
6	0.0000	0.0025	0.0011	0.0008	0.0000	0.0026	0.0011	0.0009
7	0.0000	0.0019	0.0008	0.0006	0.0000	0.0020	0.0009	0.0006
8	0.0001	0.0025	0.0012	0.0008	0.0001	0.0027	0.0013	0.0009
9	0.0001	0.0039	0.0018	0.0012	0.0001	0.0045	0.0021	0.0014
10	0.0001	0.0034	0.0015	0.0011	0.0002	0.0038	0.0017	0.0012

Tabla 4-3: Resultados obtenidos por nuestro algoritmo para la función wm_2

Algoritmo		*m*	n_p	*Error Medio*	*NRMSE*
[Wang, 1992]		15	-	0.060	-
		25	-	0.026	-
Mejora de [Wang, 1992] en [Sudkamp, 1994]		15	-	0.071	-
		25	-	0.031	-
[Sudkamp, 1994]	RG	15	-	0.131	-
		25	-	0.052	-
	WA	15	-	0.180	-
		25	-	0.082	-
[Pomares, 2000]		6	10	0.068	0.112
		8	14	0.0473	0.0823
		10	18	0.0262	0.0237
[González, 2001]		3	9	0.0582 ± 0.003	0.1169 ± 4.0E-5
		4	12	0.0107 ± 0.0111	0.0200 ± 0.0237
		5	15	0.0068 ± 0.0068	0.0143 ± 0.0173
		6	18	0.0028 ± 0.0013	0.0049 ± 0.0023
Algoritmo propuesto		3	9	0.1898 ± 0.0013	0.5071 ± 0.0003
		4	12	0.0227 ± 0.0033	0.0481 ± 0.0063
		5	15	0.0127 ± 0.0011	0.0310 ± 0.0015
		6	18	0.0006 ± 0.0004	0.0011 ± 0.0008

Tabla 4-4: Comparativa de resultados de distintos algoritmos para wm_2

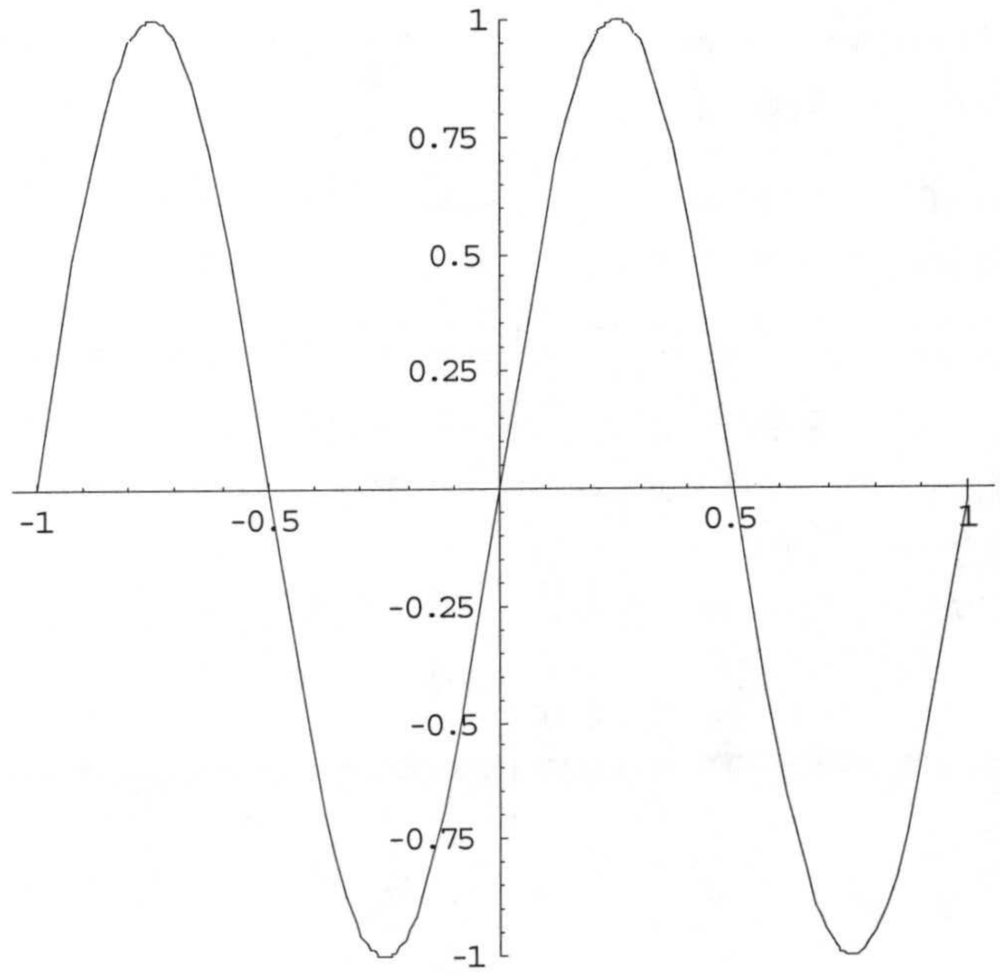

Figura 4-2: Función wm_2

4.1.1.2 Función *dick*

Esta función de una dimensión fue originalmente propuesta por [Dickerson, 1996]. En este trabajo se utilizaba un sistema híbrido neurodifuso con reglas elipsoidales y combinaron distintos valores para los pesos de las reglas con varios métodos de aprendizaje.

Comparando los modelos obtenidos con los basados en rejilla de reglas como los usados en [Pomares, 2000], es importante destacar que al optimizar cada una de las reglas neurodifusas de forma independiente el número de parámetros libres del sistema es mayor que en un sistema difuso basado en una rejilla para el mismo número de reglas. Por otra parte, la independencia de cada una de las reglas con respecto a las demás permite cubrir el espacio de entrada con un número de reglas que se colocarán en el espacio de entrada según la variabilidad de la salida de la función objetivo.

La función *dick* se define como:

$$dick(x) = 3x(x-1)(x-1.9)(x+0.7)(x+1.8) \quad x \in [-2.1, 2.1] \qquad (4\text{-}3)$$

Si se estudian los resultados obtenidos, se puede observar como influye de una forma proporcional, la inclusión de nuevas RBFs, en la disminución del error. Así esta inclusión de funciones base es recomendable hasta que se alcanzan 7 RBFs, ya que a partir de aquí, aunque el error disminuye, lo hace de una forma muy lenta.

Si se comparan los resultados con los de los otros algoritmos, se observa que claramente se superan los resultados de [Dickerson, 1996], teniendo en cuenta que ambos tienen una complejidad similar.

Con respecto al algoritmo de [Pomares, 2000] existe cierta paridad en los resultados, siendo además estos algoritmos también similares en complejidad.

Lo mismo ocurre en el campo de los resultados entre [González, 2001] y el algoritmo propuesto, ya que no existen diferencias importantes entre ambos. De todas formas si conviene recordar que nuestro algoritmo es más simple y tiene menos coste computacional que el de [González, 2001].

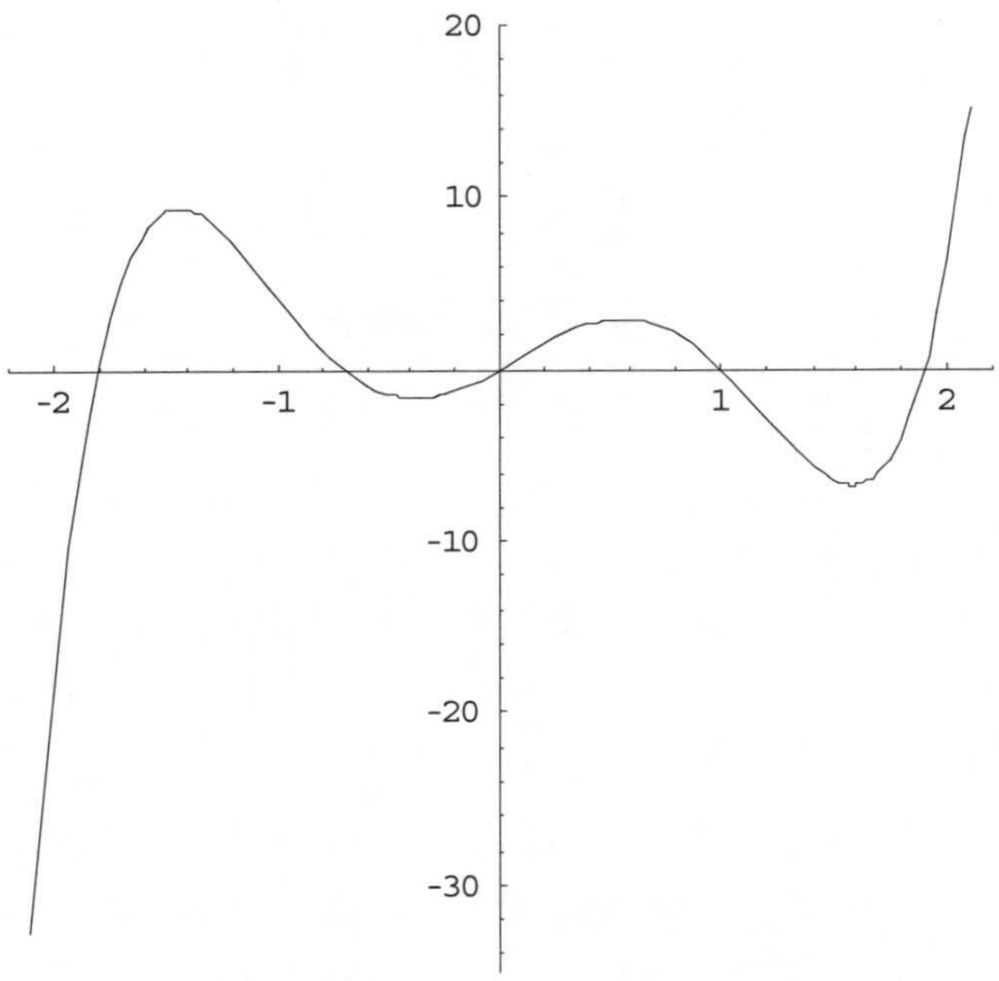

Figura 4-3: Función *dick*

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.2839	0.2839	0.2839	0.0000	0.3077	0.3077	0.3077	0.0000
4	0.1008	0.2403	0.1671	0.0285	0.1106	0.2336	0.1824	0.0291
5	0.0015	0.0819	0.0739	0.0239	0.0016	0.0891	0.0804	0.0260
6	0.0010	0.0192	0.0036	0.0052	0.0011	0.0209	0.0039	0.0057
7	0.0008	0.0024	0.0016	0.0003	0.0009	0.0027	0.0018	0.0003
8	0.0006	0.0026	0.0013	0.0004	0.0008	0.0029	0.0017	0.0004
9	0.0010	0.0024	0.0012	0.0003	0.0013	0.0034	0.0017	0.0005
10	0.0003	0.0027	0.0011	0.0004	0.0006	0.0040	0.0017	0.0006

Tabla 4-5: Resultados obtenidos por nuestro algoritmo para la función *dick*

Algoritmo			*M*	n_p	*MSE*	*NRMSE*
[Dickerson, 1996]	Distintos pesos en reglas	$w_k = 1$	6	-	94.65	-
		$w_k = 1/v_k$			28.25	-
		$w_k = 1/v_k^2$			10.53	-
	Aprend. no superv.				7.927	-
	Aprend. superv.				3.069	-
[Pomares, 2000]			5	8	5.01	0.33
			6	10	1.35	0.17
			7	12	0.46	0.10
[González, 2001]			3	9	5.57 ± 0.00	0.3455 ± 0.0000
			4	12	0.99 ± 0.49	0.1415 ± 0.0390
			5	15	0.30 ± 0.02	0.0797 ± 0.0023
Algoritmo propuesto			3	9	$3.57 + 0.00$	0.2839 ± 0.0000
			4	12	1.27 ± 0.38	0.1671 ± 0.0285
			5	15	0.26 ± 0.09	0.0739 ± 0.0239

Tabla 4-6: Comparativa de resultados de distintos algoritmos para la función *dick*

4.1.1.3 Función *nie*

En [Nie, 1996], Nie y Lee se presentaron tres algoritmos distintos para resolver problemas de aproximación funcional, a los que llamaron P1, P2 y P3. Para demostrar su funcionamiento aproximaron la función unidimensional:

$$nie(x) = 3e^{-x^2} sen(\pi x) \quad x \in [-3, 3] \tag{4-4}$$

En general se utilizaba un sistema difuso compuesto por 30 reglas y usaron el error cuadrático medio (MSE), para evaluar la eficiencia de los modelos.

Posteriormente esta función también la usa Pomares para probar su algoritmo de construcción de sistemas difusos. En este caso se utilizaron sistemas difusos compuestos de 6, 7 y 10 reglas definidas mediante una partición triangular, obteniendo mejores resultados de aproximación tanto en error como en complejidad.

Estudiando los resultados obtenidos por el algoritmo propuesto, se observa que el algoritmo se comporta bien conforme se van introduciendo nuevas RBFs. Como en casos anteriores, parece que no tiene sentido introducir más de 7 RBFs pues el error no mejora, a partir de este valor.

Si se comparan los resultados de nuestro algoritmo con los del algoritmo [Nie, 1994], se observa que éstos se superan claramente, teniendo en cuenta además que la complejidad de los modelos generados por este algoritmo es mucho mayor que la complejidad que nosotros estamos manejando.

Con respecto al algoritmo de [Pomares, 2001], también se superan más o menos, con la misma magnitud los resultados que éste obtiene. Con respecto a la complejidad de los modelos generados, aunque ésta en [Pomares, 2000] es mucho menor que en [Nie, 1994], nuestros modelos presentan un valor inferior aún.

Los mismos comentarios que se apuntaron con las funciones anteriores se pueden realizar en la comparación con [González, 2001] para esta función. Básicamente los resultados son similares y dependiendo del número de funciones base un algoritmo supera a otro, aunque como siempre nuestro algoritmo presenta en su implementación y ejecución una complejidad menor.

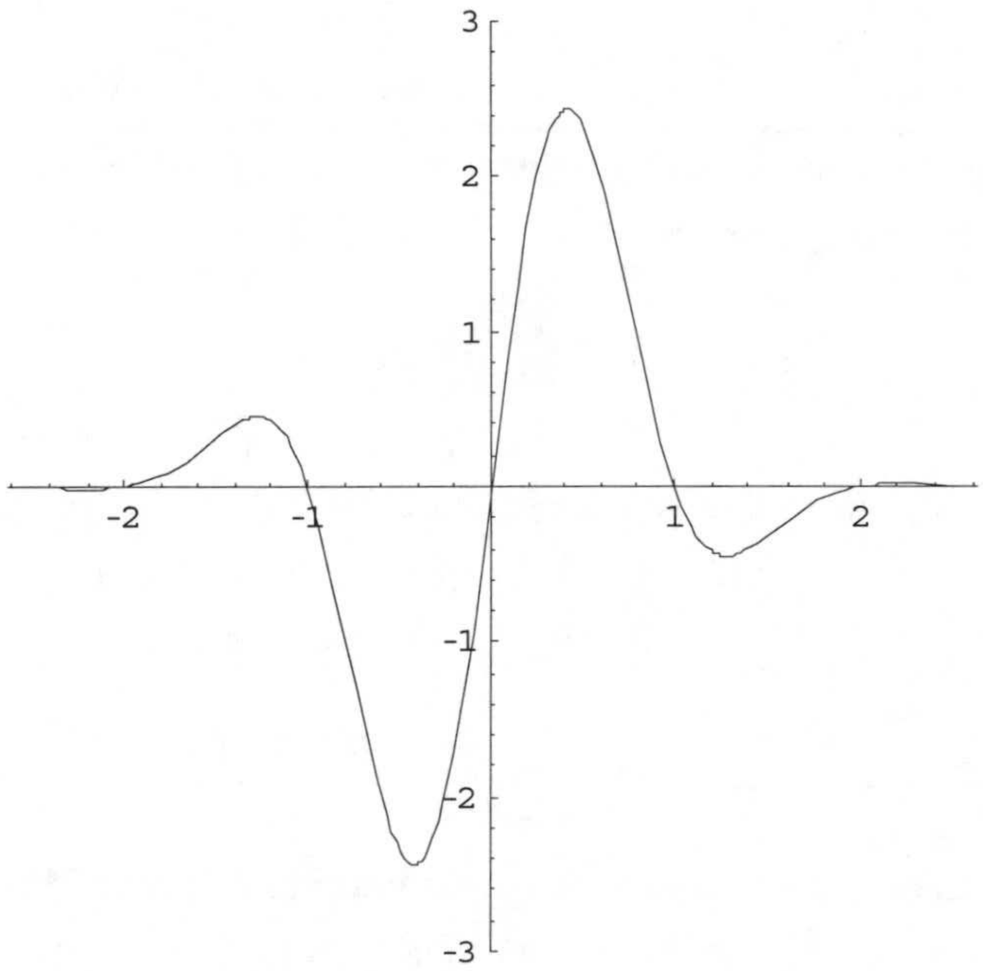

Figura 4-4: Función *nie*

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.0200	0.1310	0.1199	0.0333	0.0200	0.1412	0.1264	0.0355
4	0.0071	0.0104	0.0083	0.0006	0.0071	0.0104	0.0083	0.0006
5	0.0009	0.0074	0.0057	0.0011	0.0008	0.0074	0.0057	0.0011
6	0.0001	0.0022	0.0010	0.0007	0.0001	0.0023	0.0010	0.0007
7	0.0001	0.0016	0.0008	0.0004	0.0001	0.0016	0.0008	0.0004
8	0.0002	0.0019	0.0009	0.0005	0.0002	0.0019	0.0009	0.0005
9	0.0002	0.0023	0.0011	0.0007	0.0002	0.0024	0.0011	0.0007
10	0.0001	0.0015	0.0007	0.0004	0.0001	0.0015	0.0007	0.0004

Tabla 4-7: Resultados obtenidos por nuestro algoritmo para la función *nie*

Algoritmo		*m*	n_p	*MSE*	*NRMSE*
[Nie, 1994]	P1/γ1	30	-	0.0114	-
	P1/γ1			0.0078	-
	P2/γ2			0.0071	-
	P2/γ2			0.0068	-
	P3/γ3			0.0081	-
	P3/γ3			0.0082	-
[Pomares, 2000]		6	10	0.0113	0.111
		7	12	0.0092	0.099
		10	18	0.0024	0.051
[González, 2001]		3	9	0.0066 ± 0.0085	0.0647 ± 0.0601
		4	12	5.7E-5 ± 1.03E-5	0.0078 ± 0.0007
		5	15	2.9E-5 ± 3.1E-5	0.0048 ± 0.0032
		6	18	1.7E-5 ± 1.6E-5	0.0037 ± 0.0024
Algoritmo propuesto		3	9	0.0145 ± 0.0046	0.1199 ± 0.0333
		4	12	6.5E-5 ± 1.0E-5	0.0083 ± 0.0006
		5	15	3.1E-5 ± 0.8E-5	0.0057 ± 0.0011
		6	18	0.1E-5 ± 0.1E-5	0.0010 ± 0.0007

Tabla 4-8: Comparativa de resultados para la función nie

4.1.1.4 Función *pom*

La última función unidimensional con la que realizar pruebas, fue introducida en la tesis doctoral [Pomares, 2000], para manifestar la influencia de una buena inicialización en el posterior resultado del algoritmo de modelado. La función se expresa como:

$$pom(x) = e^{-3x^2} sen(10\pi x) \quad x \in [0, 1] \tag{4-5}$$

Esta función objetivo presenta una gran variabilidad en la salida para valores cercanos a 0 y va estabilizándose conforme x crece. Un buen algoritmo de entrenamiento debería detectar esta característica y colocar más funciones base en el intervalo de entrada en el que la salida es más variable.

Nuestro algoritmo, sigue presentado un buen comportamiento para la función *pom*, de modo que va reduciendo el error conforme se van introduciendo nuevas funciones base, hasta que se satura la red para un valor de 12 – 13 RBFs y ya no merece aumentar la complejidad del modelo, pues el error no disminuye.

Si comparamos nuestros resultados con los obtenidos por [Pomares, 2000], vemos que son ligeramente inferiores tanto error, como en complejidad de los modelos manejados.

Con respecto a [González, 2001], se observa que si bien su algoritmo empieza obteniendo errores ligeramente mejores, cuando el número de RBFs no es muy alto, las cosas cambian, cuando aumenta este valor. Aunque, siempre conviene recordar que este algoritmo es mas complejo que el nuestro.

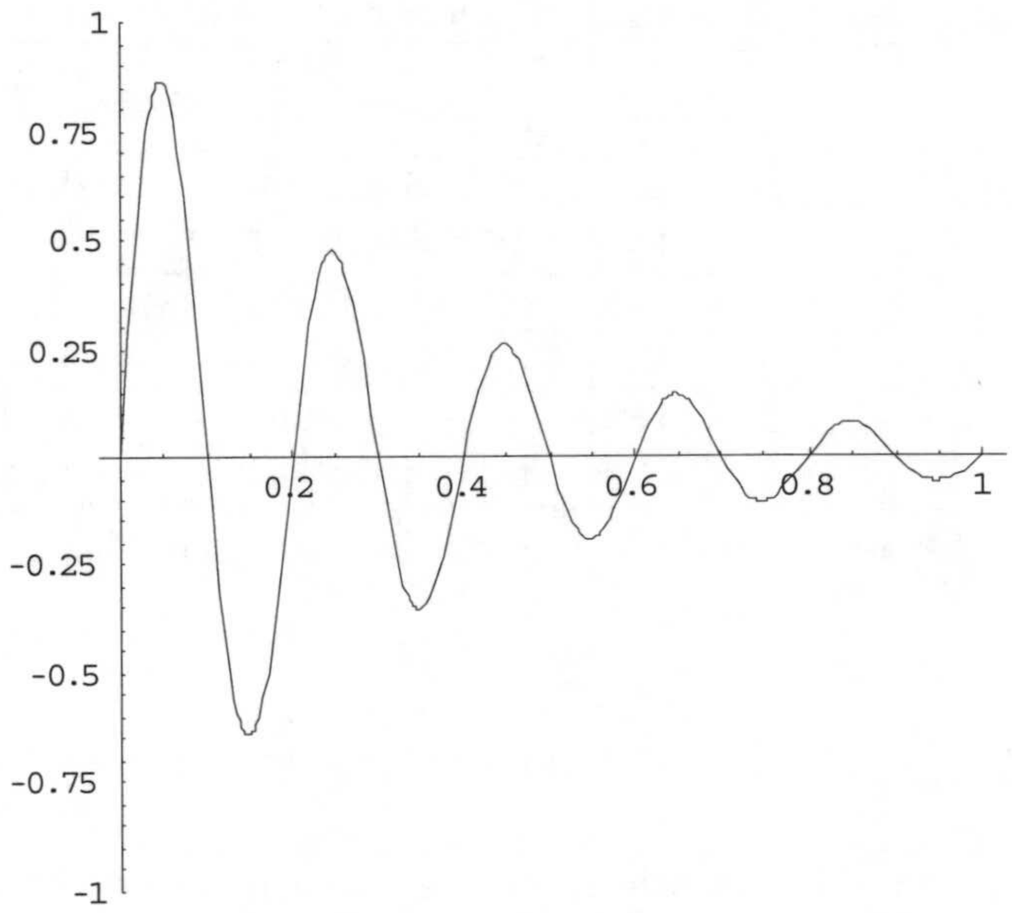

Figura 4-5: Función *pom*

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.3569	0.4782	0.4154	0.0157	0.3559	0.4692	0.4128	0.0146
4	0.2494	0.3121	0.3075	0.0109	0.2450	0.3084	0.3042	0.0110
5	0.1880	0.2349	0.2286	0.0131	0.1822	0.2301	0.2238	0.0134
6	0.1416	0.1780	0.1690	0.0128	0.1334	0.1714	0.1624	0.0134
7	0.0857	0.1368	0.1263	0.0124	0.0852	0.1282	0.1183	0.0114
8	0.0578	0.1144	0.1023	0.0129	0.0571	0.1144	0.0982	0.0135
9	0.0473	0.0784	0.0718	0.0093	0.0472	0.0782	0.0718	0.0093
10	0.0090	0.0477	0.0420	0.0124	0.0094	0.0475	0.0421	0.0120
11	0.0025	0.0097	0.0046	0.0020	0.0032	0.0110	0.0053	0.0019
12	0.0024	0.0096	0.0047	0.0021	0.0024	0.0105	0.0052	0.0018
13	0.0009	0.0078	0.0038	0.0015	0.0008	0.0090	0.0045	0.0016
14	0.0022	0.0079	0.0039	0.0016	0.0021	0.0084	0.0044	0.0015
15	0.0014	0.0083	0.0040	0.0019	0.0012	0.0091	0.0045	0.0021

Tabla 4-9: Resultados obtenidos por nuestro algoritmo para la función *pom*

Algoritmo	*m*	n_p	*NRMSE*
[Pomares, 2000]	7	12	0.304
	8	14	0.237
	9	16	0.182
	10	18	0.153
	11	20	0.126
	12	22	0.114
[González, 2001]	5	15	0.1817 ± 0.0549
	7	21	0.0896 ± 0.0354
	9	27	0.0453 ± 0.0319
	11	33	0.0169 ± 0.0068
	13	39	0.0113 ± 0.0073
Algoritmo propuesto	5	15	0.2286 ± 0.0131
	7	21	0.1263 ± 0.0124
	9	27	0.0718 ± 0.0093
	11	33	0.0046 ± 0.0020
	13	39	0.0039 ± 0.0016

Tabla 4-10: Comparativa de resultados para la función *pom*

4.1.2 Funciones de dos dimensiones

En este apartado se probará el algoritmo propuesto para una batería de funciones de una dimensión ampliamente usadas en la literatura. Además de mostrar los resultados que nuestro algoritmo obtiene, al igual que en el apartado anterior, se compararán con los resultados de otros algoritmos. Como en las funciones unidimensionales, esta comparación se realizará en base a dos factores. El primero es la complejidad del algoritmo de diseño del modelo, y el segundo será la complejidad en sí, del modelo generado. Para este último factor se reflejarán en las tablas comparativas, los parámetros *m*, que indicaban de alguna manera, el número de elementos de proceso para un modelo determinado, y el parámetro n_p, que indicaba el número de parámetros libres de éste.

También y a la hora de comparar, hay dos trabajos que hacen de referencia común como son los trabajos de [Pomares, 2000] y [González, 2001], cuyo funcionamiento se ha descrito anteriormente.

Los conjuntos de entrenamiento que se han utilizado para las funciones bidimensionales están formados por 400 puntos distribuidos en un rejilla de 20×20 celdas en el dominio de entrada, escogiendo un punto de cada cuadrícula de forma aleatoria. Como conjuntos de test se han escogido 961 puntos distribuidos en una

rejilla de 31 × 31, donde en cada cuadrícula también se escoge un punto aleatoria. Estos conjuntos de entrenamiento y test son similares a los utilizados en [González, 2001], [Pomares, 2000] y similares a los utilizados en otros trabajos como el de [Cherkassky, 1996], que utiliza un número de puntos igual pero distribuidos en forma de espiral.

Como en el caso de las funciones de una dimensión, y realizando un análisis más formal, el algoritmo se comporta de una manera coherente ya que de forma general el error disminuye razonablemente, conforme se van añadiendo funciones base. Por otro lado, presenta desviaciones típicas bajas lo que indica que es más o menos robusto en la consecución de resultados. Por último, otra característica a destacar es que el algoritmo generaliza bien obteniendo unas diferencias normales entre el error de entrenamiento y el error test.

4.1.2.1 Funciones y_3 e y_5

Estas dos funciones fueron propuestas por Rovatti y Guerrieri [Rovatti, 1996] para comprobar el funcionamiento de un algoritmo de construcción de sistemas difusos en el que tanto las funciones de pertenencia en la entrada, como en la salida estaban fijas a priori, con lo que simplemente se realizaba un proceso de minimización simbólica. Matemáticamente estas funciones se describen como:

$$y_3(x_1,x_2)=\frac{1}{2}e^{-20\left[(x_1-0.5)^2+(x_2-0.5)^2\right]}+e^{-10(x_1^2+x_2^2)} \quad x_1,x_2\in[0,1] \qquad (4\text{-}6)$$

$$y_5(x_1,x_2)=\frac{1}{2}\left[sen(2\pi x_1)cos(2\pi x_2)\right] \quad x_1,x_2\in[0,1] \qquad (4\text{-}7)$$

En el trabajo anterior el número de reglas difusas, así como las funciones de pertenencia, venían fijas de antemano antes de ejecutar el algoritmo. Sin embargo en [Rojas, 2000b] se presentó el método SOFGR, un algoritmo de identificación de sistemas difusos a partir de un conjunto de ejemplos que también optimizaba el número de reglas mediante el uso de un índice que determinaba la controversia entre las mismas.

Posteriormente, [Pomares, 2000], también utilizó la función y_3, para comparar los resultados obtenidos con su algoritmo de identificación de sistemas difusos.

De estas dos funciones, destaca la función y_3, ya que su ecuación de definición contiene la suma de dos términos exponenciales. Lo importante es que si se analiza cada uno de estos términos, se observa que se corresponden con la ecuación que define a una función base, por lo que la salida de la función y_3, sería igual a la de un

RBFN compuesta por dos funciones base con la siguiente configuración. La primera función base tendría un peso de 0.5, un centro con coordenadas (0.5, 0.5) y un radio igual a $(1/20)^{0.5}$. La segunda función base tendría un peso de 1, un centro situado en (0, 0) y un radio igual $(1/10)^{0.5}$.

Este ejemplo va a servir para testear de una forma muy fiable como trabaja nuestro algoritmo, ya que podremos medir hasta que punto es capaz de lograr una red con la configuración anterior. Tras aplicar nuestro algoritmo los resultados son inmejorables, ya que ha obtenido exactamente una red con la configuración anterior, por lo que el error final es 0. Además hay que destacar que la desviación típica es también 0, lo que quiere decir que en las 30 iteraciones de nuestro algoritmo siempre se ha llegado a la configuración óptima. Este resultado demuestra el gran funcionamiento que también ofrece la primera parte y principal de nuestro algoritmo, ya que realiza adecuadamente su trabajo, disponiendo las funciones base en un lugar apropiado, cerca del óptimo global, para que después el algoritmo de minimización Levenberg-Marquartd lo alcance.

Si se comparan los resultados para el resto de los algoritmos, evidentemente en cuanto al error, se parte de que son inmejorables. Concretamente se superarán bastante los resultados obtenidos por [Rovatti, 1996]. Con respecto a [González, 2001], y aunque consigue un error bastante ajustado, si hay que resaltar, que no es capaz de llegar a la configuración óptima, a la que si ha llegado el algoritmo propuesto. Además y para conseguir esta configuración llega a emplear hasta 70 individuos (RBFNs) por generación, mientras que nosotros sólo estamos trabajando con una, aunque de una forma más intensiva.

Si ahora se analizan los resultados para la función y_5, se observa que el algoritmo sigue comportándose bastante bien disminuyendo el error de aproximación siempre que se incluye una función base más.

Para los resultados aportados por [Rovatti, 1996], [Rojas, 2000b] y [Pomares, 2000], se observa que nuestro algoritmo los supera en general, incluso con modelos de menor complejidad.

Por otro lado los resultados del trabajo [Gonzalez, 2001], con el que se puede hacer una comparación más directa, también se suelen superar en general, aumentando la brecha entre ambos algoritmo cuando el número de funciones base se incrementa.

Algoritmo	m	n_p	*RMSE*	*NRMSE*
[Rovatti, 1996]	5×5	25	0.055	-
[González, 2001]	2	8	$0.0002 \pm 4.1E\text{-}5$	0.0008 ± 0.0002
Algoritmo propuesto	2	8	0 ± 0	0 ± 0

Tabla 4-11: Comparativa de distintos algoritmos para la función y_3

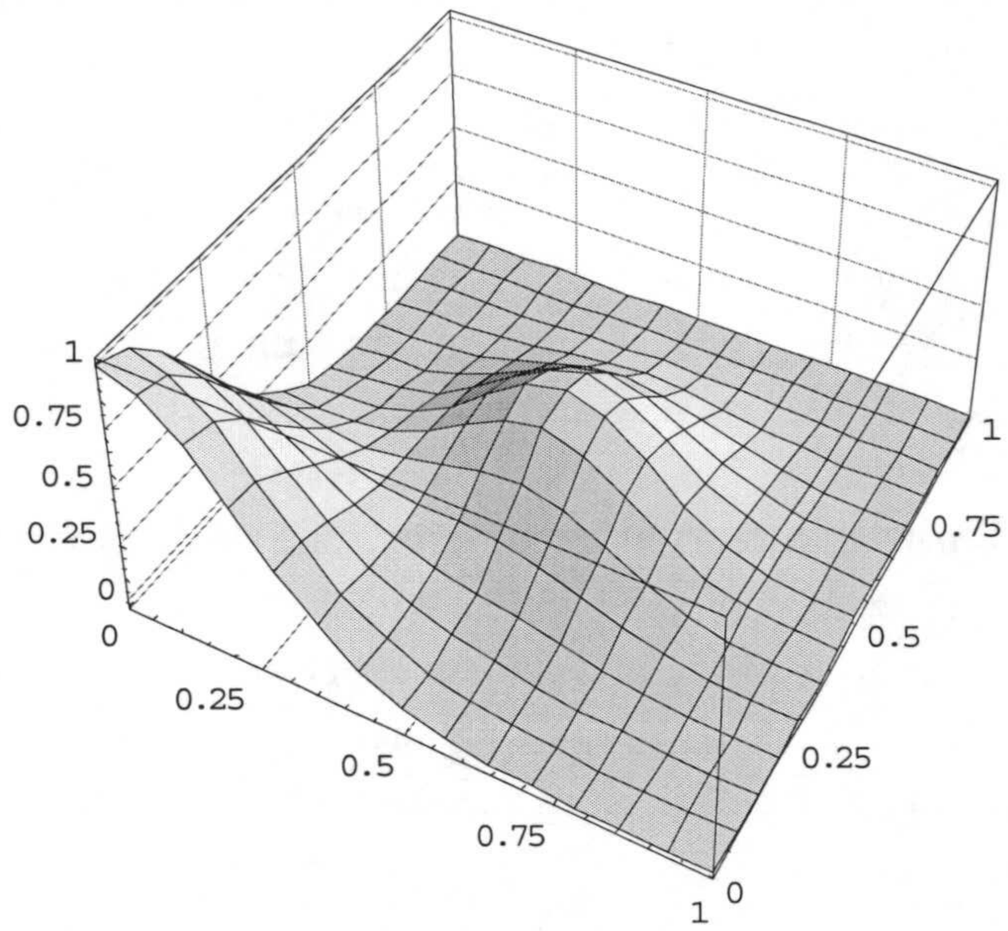

Figura 4-6: Función y_3

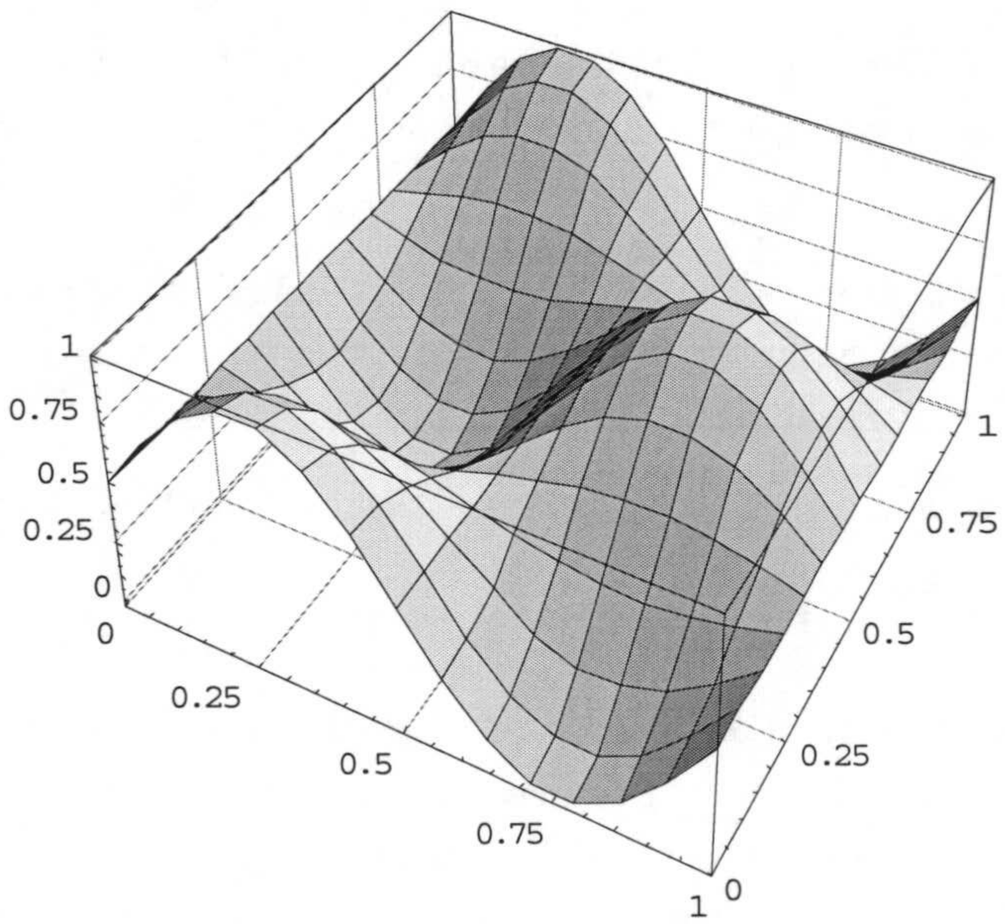

Figura 4-7: Función y_5

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.3836	0.3836	0.3836	0.0000	0.3800	0.3800	0.3800	0.0000
4	0.2992	0.3686	0.3342	0.0181	0.2862	0.3655	0.3311	0.0231
5	0.2109	0.3329	0.2687	0.0289	0.2065	0.3347	0.2676	0.0331
6	0.1018	0.2279	0.1914	0.0441	0.1026	0.2307	0.1905	0.0440
7	0.0372	0.1568	0.1226	0.0203	0.0370	0.1594	0.1226	0.0208
8	0.0593	0.1042	0.0920	0.0079	0.0570	0.1071	0.0915	0.0079
9	0.0234	0.0367	0.0271	0.0044	0.0236	0.0362	0.0271	0.0042
10	0.0165	0.0352	0.0262	0.0047	0.0157	0.0352	0.0264	0.0045
11	0.0133	0.0321	0.0224	0.0045	0.0127	0.0317	0.0222	0.0045
12	0.0146	0.0295	0.0194	0.0037	0.0141	0.0297	0.0195	0.0039
13	0.0102	0.0238	0.0170	0.0028	0.0103	0.0239	0.0171	0.0028
14	0.0116	0.0207	0.0161	0.0022	0.0125	0.0211	0.0164	0.0022
15	0.0080	0.0225	0.0146	0.0028	0.0087	0.0230	0.0148	0.0028

Tabla 4-12: Resultados del algoritmo propuesto para la función y_5

Algoritmo	*m*	n_p	*RMSE*	*NRMSE*
[Rovatti, 1996]	5×5	25	0.130	-
[Rojas, 2000b]	4×5	25	0.142	-
	6×7	51	0.068	-
	7×8	67	0.052	-
[Pomares, 2000]	4×4	20	0.080	0.158
	5×5	31	0.030	0.122
	6×6	44	0.014	0.056
[González, 2001]	5	20	0.0643 ± 0.0114	0.2556 ± 0.0454
	6	24	0.0484 ± 0.0094	0.1925 ± 0.0372
	8	32	0.0252 ± 0.0061	0.1003 ± 0.0242
	10	40	0.0131 ± 0.0036	0.0522 ± 0.0143
	11	44	0.0109 ± 0.0058	0.0433 ± 0.0232
Algoritmo propuesto	5	20	0.0673 ± 0.0072	0.2676 ± 0.0331
	6	24	0.0479 ± 0.0111	0.1905 ± 0.0440
	8	32	0.0231 ± 0.0020	0.0915 ± 0.0079
	10	40	0.0066 ± 0.0012	0.0264 ± 0.0045
	11	44	0.0056 ± 0.0011	0.0222 ± 0.0045

Tabla 4-13: Comparativa de algoritmos para la función y_5

4.1.2.2 Funciones f_1, f_4, f_5, f_6, f_7 y f_8

En este caso se van a estudiar las funciones f_1, f_4, f_5, f_6, f_7 y f_8 como conjunto de funciones relacionadas y que se usan en distintos trabajos.

En principio las funciones f_5, f_6 y f_7, fueron utilizadas en [Cherkasky, 1991] para probar diferentes paradigmas utilizados en la aproximación funciona. Entre los métodos comparados se encuentran el *projection pursuit* (PP) [Friedman, 1981], un método adaptativo que utiliza sumas de funciones que dependen linealmente de las entradas, el *multivariate adaptative regression splines* (MARS) [Friedman, 1991], que adaptativamente divide el espacio de entrada en regiones inconexas y modela cada región con un valor constante, el *constrained topological mapping* (CTM), método introducido en [Cherkassky, 1991] y que es muy parecido a MARS donde se usan mapas autoorganizativos para determinar la partición inicial, y un perceptrón multicapa (MLP) con 15 neuronas en la capa oculta. Dichas funciones se definen de la siguiente manera:

$$f_5(x_1, x_2) = 42.659(0.1 + x_1(0.05 + x_1^4 - 10x_1^2x_2^2 + 5x_2^4)) \qquad x_1, x_2 \in [-0.5, 0.5] \qquad (4\text{-}8)$$

$$f_6(x_1, x_2) = 1.3356\begin{bmatrix} 1.5(1-x) + e^{2x-1}sen(3\pi(x_1 - 0.6)^2) + \\ + e^{3(x_2-0.5)}sen(4\pi(x_2 - 0.9)^2) \end{bmatrix} \qquad x_1, x_2 \in [0,1] \qquad (4\text{-}9)$$

$$f_7(x_1, x_2) = 1.9\left[1.35 + e^{x_1}sen(13(x_1 - 0.6)^2)e^{-x_2}sen(7x_2)\right] \qquad x_1, x_2 \in [0,1] \qquad (4\text{-}10)$$

Posteriormente, se añaden a la batería de funciones anteriores, las funciones f_1, f_4 y f_8:

$$f_1(x_1, x_2) = sen(x_1, x_2) \quad x_1, x_2 \in [-2, 2] \qquad (4\text{-}11)$$

$$f_4(x_1, x_2) = \frac{1 + \sin(2x_1 + 3x_2)}{3.5 + \sin(x_1 - x_2)} \quad x_1, x_2 \in [-2, 2] \qquad (4\text{-}12)$$

$$f_8(x_1, x_2) = sen\left(2\pi\sqrt{x_1^2 + x_2^2}\right) \quad x_1, x_2 \in [0,1] \qquad (4\text{-}13)$$

Estas funciones se introducen en [Cherkassky, 1996], donde se realiza un trabajo muy completo sobre la comparación de paradigmas para la aproximación de funciones. En dicho trabajo, además de los paradigmas anteriores, se introducen

métodos bastante mejorados para la optimización de perceptrones multicapa, logrando resultados bastante buenos. En cuanto al número de parámetros que utilizaban estos paradigmas, los métodos basados en redes neuronales presentaban 40 neuronas en su capa oculta. Esto implica un número total de 161 parámetros (40×2+40 pesos entre las neuronas ocultas y la de salida, más (40+1) términos aditivos o *bias*).

En el caso del trabajo de [Pomares, 2000] que utiliza su algoritmo se introducen ahora distintos tipos de funciones de pertenencia, tales como particiones triangulares (PT), funciones triangulares (TL), funciones gausianas (G), y funciones pseudo-gausianas libres (PGL).

Estas funciones también han sido utilizadas en [Castillo, 2001] para comprobar los resultados obtenidos por G-PROP, un algoritmo que evoluciona perceptrones multicapa.

Si se analizan los resultados en general se puede observar, como el algoritmo sigue comportándose bien para todas estas funciones. Así, y de una manera coherente, va disminuyendo el error, cada vez que se va introduciendo una nueva función base. También cabe notar que al contrario de cómo ocurría en las funciones unidimensionales, ahora no suele existir un número de funciones límite, a partir del cual ya no interese introducir más funciones base porque el error no mejora.

Si comparamos nuestros resultados con los conseguidos por [Pomares, 2000], la primera impresión es que para este subconjunto de funciones, este último trabajo, ha conseguido unos resultados bastante ajustados, superando los que conseguía para funciones unidimensionales. Esto se puede deber a la adaptación de su algoritmo para este tipo de funciones, y por la forma de algunas de estas funciones, ya que cuando las funciones tienen tramos planos y con poca variabilidad, su mecanismo de defuzzifcación puede generar planos en el espacio de salida de una forma sencilla y con un mínimo número de parámetros.

Así, por ejemplo, [Pomares, 2000] consigue mejores resultados que nuestro algoritmo para las funciones f_1 y f_4, mientras que sus resultados son peores para f_6, f_7 y f_8. Para f_5 los resultados son similares. De todas formas, las diferencias de resultados entre ambos algoritmos no son muy significativas. En cuanto a los algoritmos de diseño del modelo, también habría que resaltar, que son más o menos similares en cuanto a complejidad y coste computacional.

Con respecto al trabajo de [González, 2001] la comparación con nuestro trabajo tiene unos resultados similares a la comparación con el trabajo anterior, en el campo con los resultados. Así por ejemplo los resultados de [González, 2001] superan a los nuestros para las funciones, f_4 y f_5, mientras que son peores para f_1, f_7 y f_8. Para f_6, se pueden considerar similares, ya que el que un trabajo supere a otro depende del número de funciones base. Igualmente, las diferencias de resultados entre ambos

algoritmos no son muy significativas. Como siempre hay que señalar, que incluso existiendo esta similitud en el campo de los resultados, nuestro algoritmo presenta una menor complejidad y coste computacional que el de [González, 2001].

Los resultados del resto de los algoritmos, tanto los de diseño de sistemas difusos como los de diseño de redes neuronales, se suelen superar con cierta claridad y de forma general, aunque con alguna excepción. Por ejemplo, y prácticamente sólo en la función f_6, el algoritmo de [Cherkassky, 1996], exhibe unos resultados más o menos similares, si se tienen en cuenta el número de parámetros libres de los modelos generados.

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.3267	0.5842	0.3779	0.1023	0.3305	0.5882	0.3819	0.1027
4	0.3248	0.3259	0.3251	0.0004	0.3288	0.3305	0.3291	0.0007
5	0.2834	0.3020	0.2904	0.0054	0.2906	0.3055	0.2953	0.0036
6	0.2401	0.2647	0.2460	0.0057	0.2463	0.2728	0.2512	0.0066
7	0.1765	0.2557	0.2114	0.0190	0.1852	0.2592	0.2175	0.0190
8	0.0358	0.2168	0.1823	0.0348	0.0385	0.2165	0.1864	0.0355
9	0.1220	0.1824	0.1468	0.0080	0.1277	0.1848	0.1507	0.0076
10	0.0859	0.1282	0.1198	0.0066	0.0927	0.1315	0.1243	0.0060
11	0.0838	0.0992	0.0864	0.0038	0.0903	0.1010	0.0912	0.0019
12	0.0182	0.0409	0.0192	0.0041	0.0218	0.0482	0.0228	0.0047
13	0.0162	0.0199	0.0179	0.0010	0.0202	0.0236	0.0214	0.0009
14	0.0150	0.0201	0.0179	0.0012	0.0187	0.0232	0.0212	0.0011
15	0.0129	0.0191	0.0166	0.0016	0.0167	0.0223	0.0197	0.0015
16	0.0089	0.0191	0.0161	0.0023	0.0123	0.0224	0.0189	0.0022
17	0.0092	0.0196	0.0138	0.0025	0.0122	0.0224	0.0164	0.0025
18	0.0089	0.0169	0.0125	0.0020	0.0113	0.0195	0.0149	0.0019
19	0.0057	0.0181	0.0110	0.0031	0.0077	0.0207	0.0133	0.0031
20	0.0053	0.0140	0.0097	0.0026	0.0068	0.0161	0.0118	0.0028
21	0.0047	0.0164	0.0082	0.0024	0.0065	0.0194	0.0104	0.0026
22	0.0043	0.0130	0.0074	0.0023	0.0056	0.0153	0.0094	0.0024
23	0.0041	0.0109	0.0071	0.0016	0.0057	0.0135	0.0091	0.0018
24	0.0039	0.0108	0.0066	0.0016	0.0058	0.0125	0.0085	0.0017
25	0.0039	0.0101	0.0059	0.0017	0.0054	0.0123	0.0077	0.0019
26	0.0038	0.0102	0.0059	0.0015	0.0052	0.0125	0.0076	0.0017
27	0.0034	0.0085	0.0055	0.0013	0.0049	0.0109	0.0073	0.0016
28	0.0035	0.0096	0.0055	0.0015	0.0049	0.0122	0.0072	0.0017
29	0.0026	0.0090	0.0046	0.0015	0.0048	0.0146	0.0076	0.0021
30	0.0029	0.0085	0.0046	0.0014	0.0052	0.0126	0.0077	0.0020

Tabla 4-14: Resultados obtenidos por el algoritmo propuesto para la función f_1

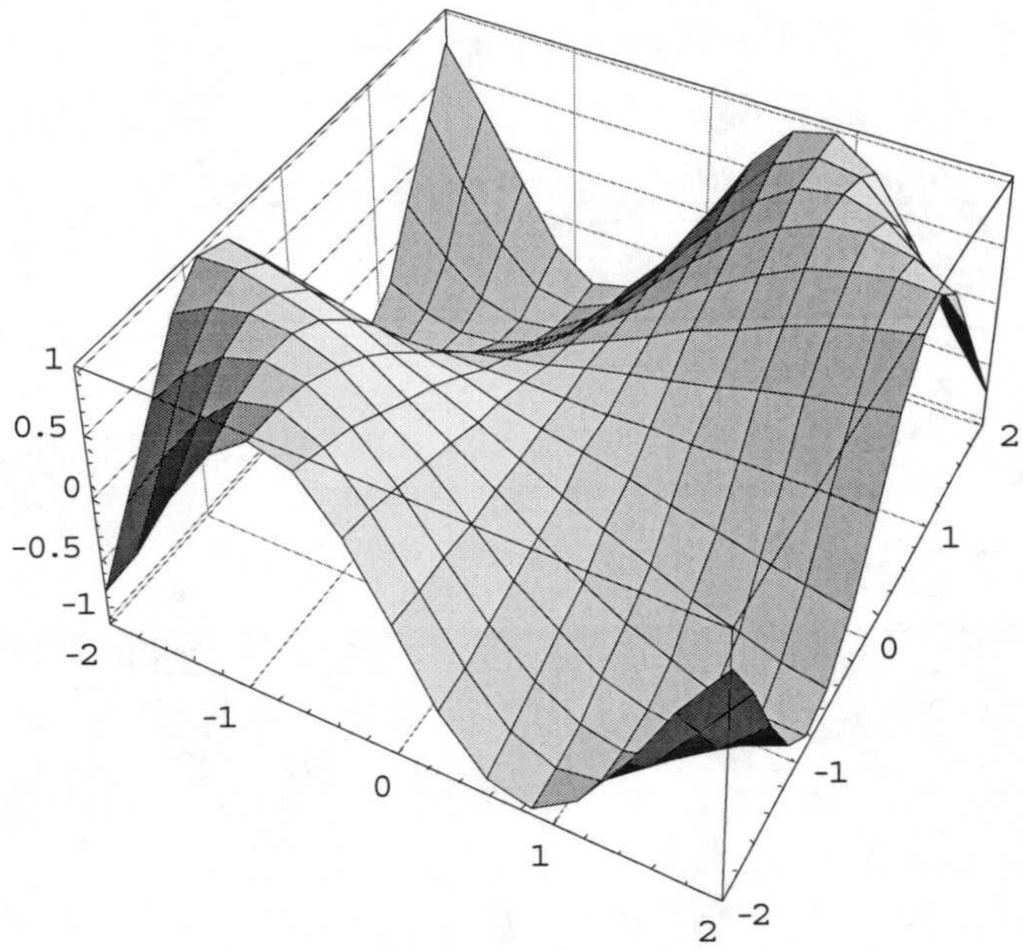

Figura 4-8: Función f_1

Algoritmo	*m*	n_p	*Test NRMSE*
[Cherkassky, 1996]	40	161	0.017
[Pomares, 2000]	12 × 12 (PT)	164	0.017
	7 × 7 (G)	73	0.007
	8 × 8 (PGL)	104	0.003
[González, 2001]	21	84	0.0176 ± 0.0044
	23	92	0.0144 ± 0.0051
	25	100	0.0116 ± 0.0009
	27	108	0.0103 ± 0.0017
	29	116	0.0089 ± 0.0020
Algoritmo propuesto	21	84	0.0104 ± 0.0026
	23	92	0.0091 ± 0.0018
	25	100	0.0077 ± 0.0019
	27	108	0.0073 ± 0.0016
	29	116	0.0076 ± 0.0021

Tabla 4-15: Comparativa de algoritmos para la función f_1

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.8309	0.9638	0.8937	0.0396	0.8326	0.9640	0.8924	0.0388
4	0.7263	0.8675	0.7982	0.0223	0.7315	0.8705	0.7981	0.0225
5	0.6594	0.7303	0.6891	0.0108	0.6621	0.7281	0.6903	0.0101
6	0.5805	0.6106	0.6070	0.0091	0.5906	0.6185	0.6147	0.0091
7	0.4283	0.5394	0.5084	0.0170	0.4318	0.5432	0.5142	0.0173
8	0.3405	0.4936	0.4084	0.0347	0.3488	0.4965	0.4137	0.0346
9	0.2444	0.3571	0.3041	0.0262	0.2505	0.3714	0.3102	0.0271
10	0.1992	0.3278	0.2561	0.0345	0.2075	0.3395	0.2650	0.0343
11	0.1859	0.2519	0.2130	0.0167	0.1992	0.2596	0.2218	0.0163
12	0.1456	0.2202	0.1861	0.0171	0.1542	0.2294	0.1950	0.0179
13	0.1372	0.2093	0.1728	0.0176	0.1468	0.2203	0.1821	0.0187
14	0.0688	0.1734	0.1390	0.0252	0.0740	0.1823	0.1471	0.0257
15	0.0824	0.1585	0.1334	0.0180	0.0916	0.1669	0.1419	0.0182
16	0.0571	0.1378	0.1141	0.0222	0.0634	0.1448	0.1222	0.0224
17	0.0552	0.1253	0.1018	0.0274	0.0621	0.1329	0.1095	0.0280
18	0.0436	0.1301	0.0800	0.0305	0.0510	0.1378	0.0878	0.0307
19	0.0306	0.1091	0.0544	0.0221	0.0380	0.1208	0.0634	0.0223
20	0.0261	0.0958	0.0474	0.0125	0.0336	0.1119	0.0557	0.0135
21	0.0256	0.0522	0.0401	0.0064	0.0338	0.0602	0.0485	0.0065
22	0.0191	0.0462	0.0360	0.0062	0.0250	0.0548	0.0441	0.0065
23	0.0222	0.0434	0.0350	0.0053	0.0295	0.0518	0.0437	0.0056
24	0.0176	0.0432	0.0278	0.0067	0.0236	0.0503	0.0356	0.0071
25	0.0145	0.0526	0.0268	0.0075	0.0202	0.0608	0.0351	0.0078
26	0.0152	0.0383	0.0256	0.0043	0.0212	0.0459	0.0334	0.0048
27	0.0131	0.0415	0.0228	0.0055	0.0210	0.0490	0.0299	0.0057
28	0.0138	0.0308	0.0213	0.0037	0.0180	0.0382	0.0278	0.0041
29	0.0098	0.0278	0.0184	0.0042	0.0151	0.0341	0.0252	0.0046
30	0.0114	0.0280	0.0173	0.0031	0.0170	0.0350	0.0241	0.0032

Tabla 4-16: Resultados obtenidos por nuestro algoritmo para la función f_4

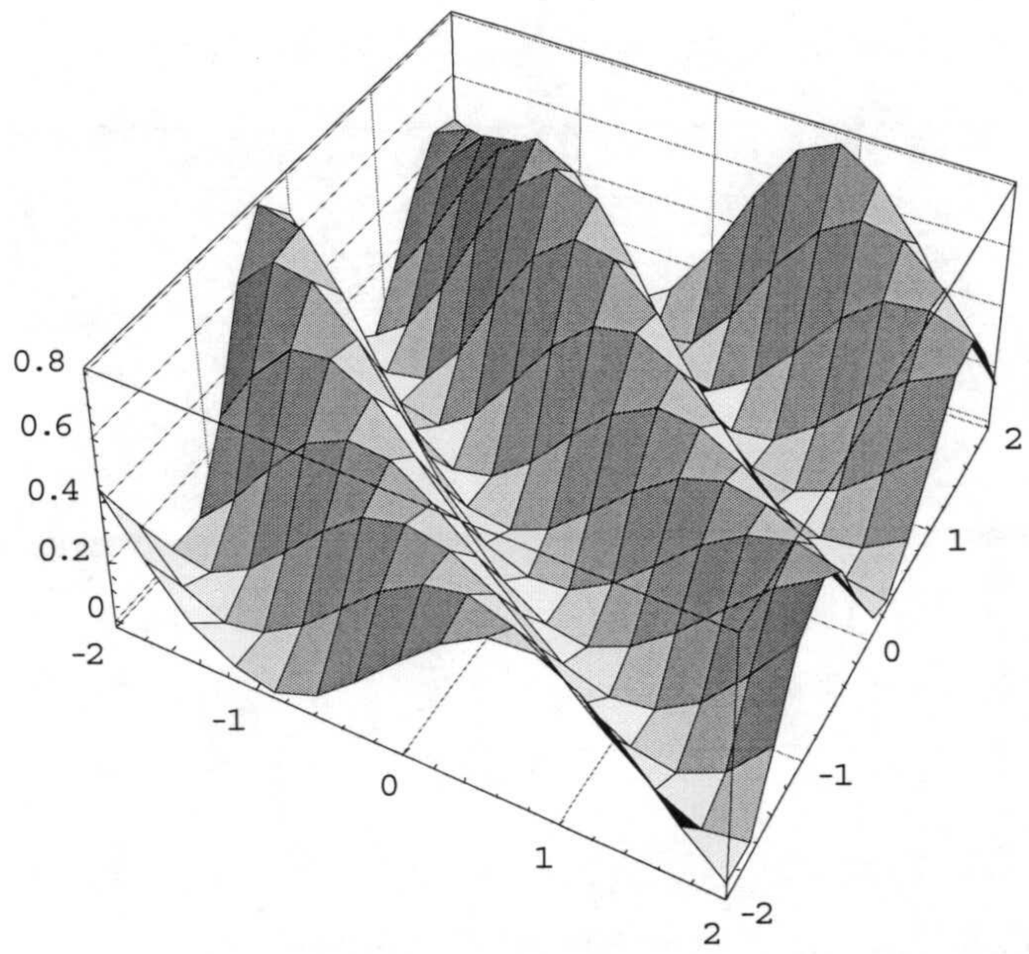

Figura 4-9: Función f_4

Algoritmo	*m*	n_p	*Test NRMSE*
[Cherkassky, 1996]	40	161	0.052
[Pomares, 2000]	6×7 (G)	64	0.061
	7×8 (G)	82	0.028
	9×9 (G)	113	0.015
[González, 2001]	21	84	0.0473 ± 0.0086
	23	92	0.0362 ± 0.0088
	25	100	0.0265 ± 0.0033
	27	108	0.0239 ± 0.0054
	29	116	0.0214 ± 0.0049
Algoritmo propuesto	21	84	0.0485 ± 0.0065
	23	92	0.0437 ± 0.0056
	25	100	0.0351 ± 0.0078
	27	108	0.0299 ± 0.0057
	29	116	0.0252 ± 0.0046

Tabla 4-17: Comparativa de distintos algoritmo para la función f_4

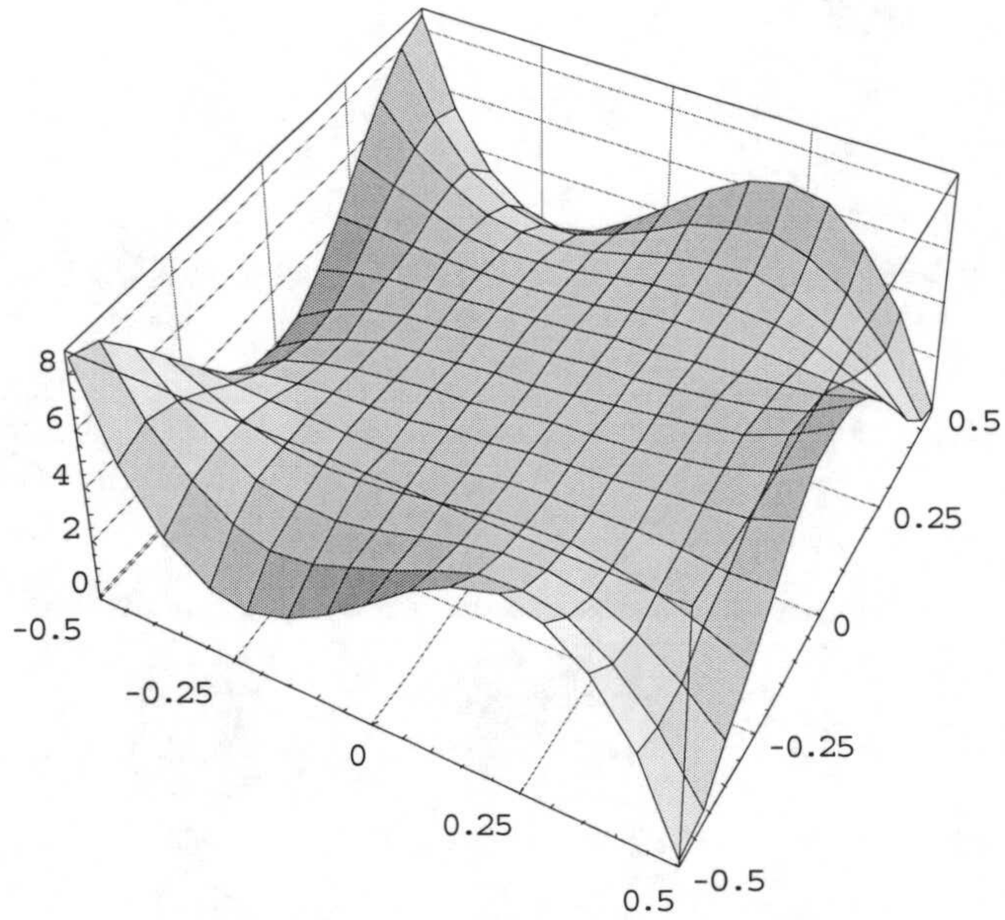

Figura 4-10: Función f_5

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.7024	0.8092	0.7133	0.0320	0.6921	0.8169	0.7048	0.0374
4	0.5936	0.6210	0.5956	0.0048	0.5843	0.6159	0.5882	0.0058
5	0.4800	0.5779	0.5019	0.0319	0.4760	0.5738	0.4964	0.0317
6	0.2789	0.4270	0.2838	0.0266	0.2666	0.4339	0.2722	0.0300
7	0.2345	0.2525	0.2461	0.0051	0.2304	0.2428	0.2363	0.0031
8	0.1789	0.2470	0.2086	0.0189	0.1761	0.2359	0.2015	0.0167
9	0.0502	0.2070	0.1594	0.0343	0.0537	0.1995	0.1551	0.0315
10	0.0307	0.1781	0.1229	0.0390	0.0336	0.1725	0.1176	0.0336
11	0.0173	0.1793	0.0941	0.0364	0.0180	0.1613	0.0899	0.0303
12	0.0219	0.1448	0.0803	0.0347	0.0225	0.1267	0.0757	0.0290
13	0.0160	0.1147	0.0571	0.0237	0.0164	0.1040	0.0558	0.0203
14	0.0081	0.1100	0.0546	0.0274	0.0084	0.1043	0.0527	0.0242
15	0.0115	0.0749	0.0368	0.0171	0.0117	0.0737	0.0368	0.0161
16	0.0152	0.0654	0.0323	0.0134	0.0170	0.0657	0.0326	0.0124
17	0.0097	0.0618	0.0301	0.0150	0.0105	0.0623	0.0309	0.0144
18	0.0040	0.0448	0.0218	0.0087	0.0048	0.0469	0.0232	0.0090
19	0.0059	0.0459	0.0227	0.0102	0.0089	0.0591	0.0289	0.0155
20	0.0054	0.0470	0.0232	0.0114	0.0080	0.0632	0.0304	0.0178

Tabla 4-18: Resultados obtenidos para la función f_5

Algoritmo	*m*	n_p	*Test NRMSE*
MLP [Cherkassky, 1991]	15	61	0.308
PP [Friedman, 1981]	-	-	0.504
CTM [Cherkassky, 1991]	-	-	0.131
MARS [Friedman, 1991)]	-	-	0.190
[Cherkassky, 1996]	40	161	0.038
[Pomares, 2000]	4×5 (PT)	25	0.194
	6×6 (PT)	44	0.090
	8×8 (PT)	76	0.044
[González, 2001]	7	28	0.1705 ± 0.1237
	10	40	0.0689 ± 0.0517
	13	52	0.0235 ± 0.0008
	16	64	0.0168 ± 0.0096
	19	76	0.0121 ± 0.0038
Algoritmo propuesto	7	28	0.2363 ± 0.0031
	10	40	0.1176 ± 0.0336
	13	52	0.0558 ± 0.0203
	16	64	0.0326 ± 0.0124
	19	76	0.0289 ± 0.0155

Tabla 4-19: Comparativa de distintos algoritmos para la función f_5

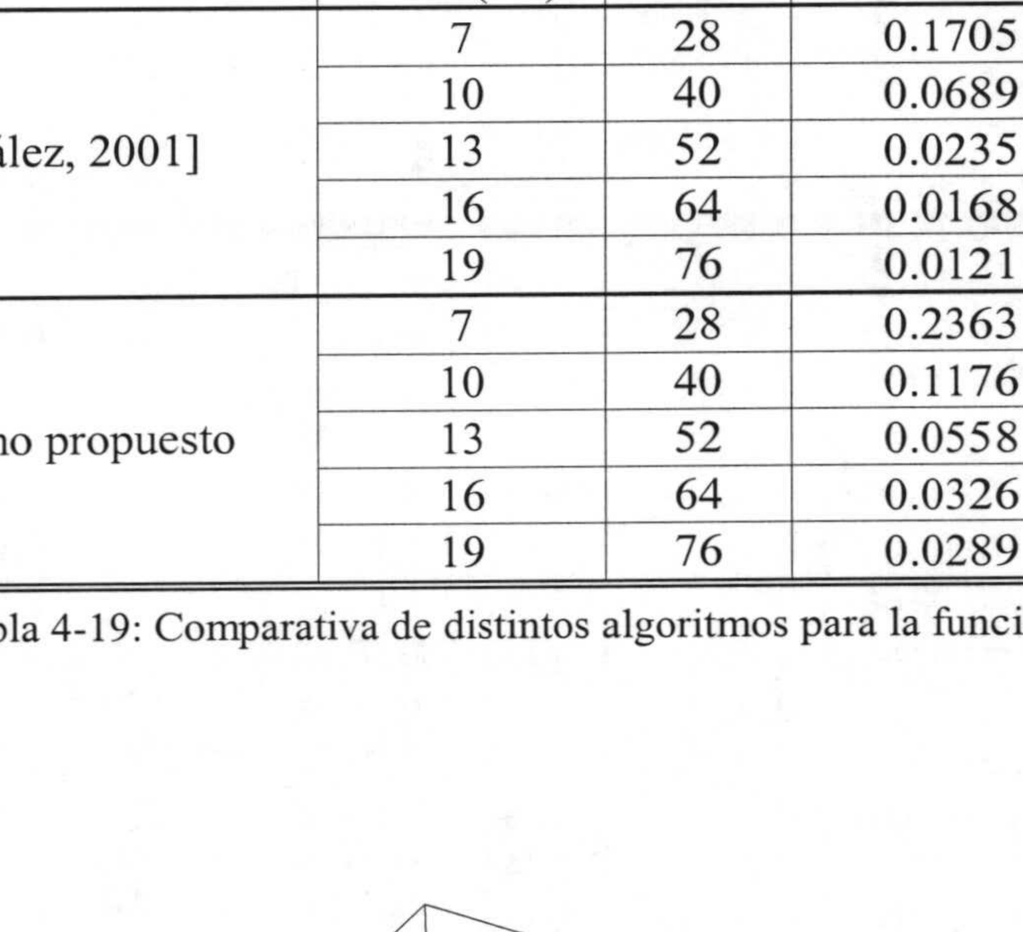

Figura 4-11: Función f_6

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.4713	0.5490	0.5323	0.0311	0.4746	0.5511	0.5348	0.0306
4	0.4071	0.4074	0.4071	0.0001	0.4116	0.4120	0.4118	0.0001
5	0.2736	0.3629	0.3581	0.0167	0.2829	0.3716	0.3666	0.0168
6	0.2967	0.3389	0.3123	0.0123	0.3081	0.3449	0.3225	0.0103
7	0.1567	0.2879	0.2186	0.0280	0.1667	0.2945	0.2254	0.0281
8	0.1491	0.2065	0.1589	0.0098	0.1588	0.2124	0.1657	0.0100
9	0.0730	0.1439	0.1333	0.0145	0.0893	0.1520	0.1412	0.0142
10	0.0739	0.1347	0.1059	0.0117	0.0760	0.1443	0.1108	0.0135
11	0.0493	0.0903	0.0708	0.0161	0.0533	0.0945	0.0756	0.0144
12	0.0353	0.0750	0.0482	0.0121	0.0406	0.0781	0.0535	0.0109
13	0.0283	0.0653	0.0394	0.0093	0.0323	0.0679	0.0445	0.0088
14	0.0196	0.0597	0.0356	0.0095	0.0254	0.0621	0.0402	0.0088
15	0.0186	0.0564	0.0324	0.0101	0.0235	0.0596	0.0368	0.0092
16	0.0160	0.0369	0.0260	0.0060	0.0218	0.0421	0.0312	0.0060
17	0.0133	0.0388	0.0245	0.0063	0.0192	0.0416	0.0295	0.0056
18	0.0117	0.0416	0.0208	0.0072	0.0178	0.0479	0.0262	0.0068
19	0.0102	0.0315	0.0182	0.0048	0.0156	0.0339	0.0229	0.0041
20	0.0100	0.0296	0.0162	0.0051	0.0150	0.0315	0.0211	0.0041
21	0.0090	0.0281	0.0143	0.0041	0.0135	0.0309	0.0191	0.0036
22	0.0083	0.0261	0.0130	0.0038	0.0126	0.0299	0.0183	0.0034
23	0.0092	0.0212	0.0121	0.0028	0.0133	0.0252	0.0171	0.0025
24	0.0081	0.0176	0.0111	0.0021	0.0134	0.0251	0.0162	0.0025
25	0.0069	0.0226	0.0113	0.0031	0.0111	0.0267	0.0160	0.0028
26	0.0771	0.0999	0.0882	0.0059	0.0873	0.1106	0.0976	0.0062
27	0.0067	0.0202	0.0100	0.0024	0.0104	0.0256	0.0151	0.0028
28	0.0067	0.0134	0.0096	0.0015	0.0113	0.0188	0.0148	0.0017
29	0.0060	0.0140	0.0097	0.0020	0.0099	0.0191	0.0147	0.0022
30	0.0114	0.0280	0.0173	0.0031	0.0170	0.0350	0.0241	0.0032

Tabla 4-20: Resultados obtenidos por nuestro algoritmo para la función f_6

Algoritmo	*m*	n_p	*Test NRMSE*
MLP [Cherkassky, 1991]	15	61	0.096
PP [Friedman, 1981]	-	-	0.128
CTM [Cherkassky, 1991]	-	-	0.170
MARS [Friedman, 1991]	-	-	0.063
[Cherkassky, 1996]	40	161	0.008
[Pomares, 2000]	3 × 5 (PT)	19	0.278
	4 × 6 (PT)	30	0.104
	5 × 9 (PT)	55	0.041
[Castillo, 2000]	G-Prop (fn)	174 ± 33	0.071 ± 0.006
	G-Prop (fl)	70 ± 37	0.079 ± 0.008
	G-Prop (fd)	75 ± 14	0.071 ± 0.006
[González, 2001]	5	20	0.3622 ± 0.0268
	10	40	0.1343 ± 0.0261
	15	60	0.0459 ± 0.0096
	21	84	0.0200 ± 0.0054
	25	100	0.0118 ± 0.0049
Algoritmo propuesto	5	20	0.3666 ± 0.0168
	10	40	0.1108 ± 0.0135
	15	60	0.0368 ± 0.0092
	21	84	0.0191 ± 0.0036
	25	100	0.0160 ± 0.0028

Tabla 4-21: Comparativa de distintos algoritmos para la función f_6

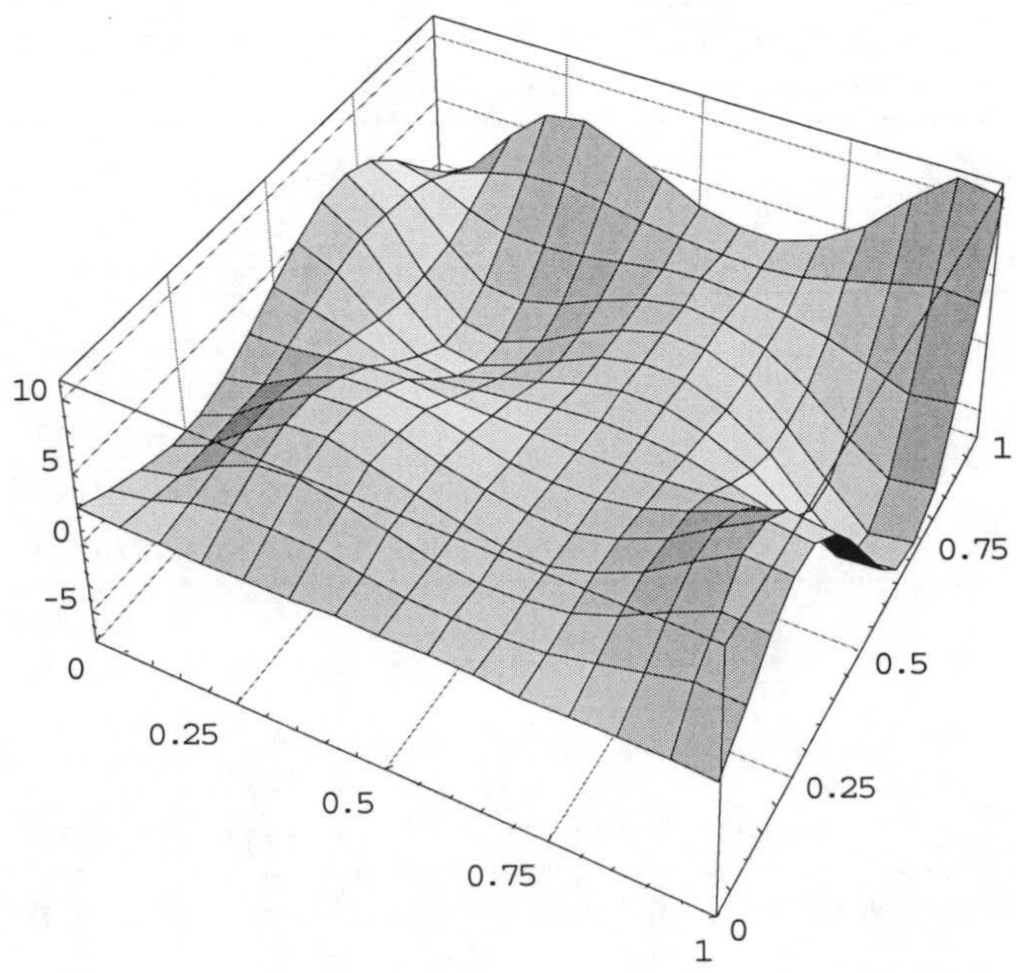

Figura 4-12: Función f_7

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.4577	0.5124	0.4595	0.0098	0.4647	0.5138	0.4663	0.0088
4	0.3219	0.4243	0.3906	0.0202	0.3290	0.4278	0.3913	0.0208
5	0.2481	0.3174	0.2547	0.0164	0.2485	0.3217	0.2553	0.0175
6	0.1406	0.2608	0.1893	0.0237	0.1381	0.2690	0.1881	0.0253
7	0.1161	0.1901	0.1633	0.0196	0.1130	0.1917	0.1621	0.0200
8	0.1008	0.1788	0.1521	0.0228	0.1040	0.1765	0.1513	0.0221
9	0.0976	0.1594	0.1193	0.0139	0.0977	0.1589	0.1195	0.0137
10	0.0557	0.1221	0.0687	0.0201	0.0578	0.1230	0.0700	0.0196
11	0.0384	0.0672	0.0453	0.0075	0.0425	0.0672	0.0471	0.0066
12	0.0311	0.0530	0.0393	0.0037	0.0368	0.0554	0.0418	0.0031
13	0.0244	0.0444	0.0342	0.0042	0.0292	0.0460	0.0372	0.0035
14	0.0224	0.0447	0.0316	0.0047	0.0256	0.0453	0.0350	0.0040
15	0.0197	0.0371	0.0282	0.0042	0.0234	0.0391	0.0321	0.0036
16	0.0175	0.0321	0.0235	0.0037	0.0200	0.0359	0.0277	0.0036
17	0.0150	0.0298	0.0212	0.0036	0.0183	0.0324	0.0254	0.0036
18	0.0139	0.0295	0.0186	0.0031	0.0173	0.0340	0.0227	0.0034
19	0.0139	0.0240	0.0173	0.0026	0.0167	0.0298	0.0215	0.0031
20	0.0103	0.0283	0.0165	0.0037	0.0143	0.0319	0.0207	0.0036

Tabla 4-22: Resultados del algoritmo propuesto para la función f_7

Algoritmo	*m*	n_p	*Test NRMSE*
MLP [Cherkassky, 1991]	15	61	0.227
PP [Friedman, 1981]	-	-	0.206
CTM [Cherkassky, 1991]	-	-	0.197
MARS [Friedman, 1991]	-	-	0.179
[Cherkassky, 1996]	40	161	0.034
[Pomares, 2000]	4 × 4 (PT)	20	0.161
	5 × 5 (PT)	31	0.109
	7 × 6 (PT)	51	0.059
[Castillo, 2000]	G-Prop (fn)	118 ± 39	0.21 ± 0.01
	G-Prop (fl)	105 ± 34	0.23 ± 0.01
	G-Prop (fd)	115 ± 36	0.22 ± 0.02
[González, 2001]	7	28	0.1426 ± 0.0190
	10	40	0.0780 ± 0.0080
	13	52	0.0590 ± 0.0103
	16	64	0.0474 ± 0.0062
	19	76	0.0324 ± 0.0050
Algoritmo propuesto	7	28	0.1621 ± 0.0200
	10	40	0.0700 ± 0.0196
	13	52	0.0372 ± 0.0035
	16	64	0.0277 ± 0.0036
	19	76	0.0215 ± 0.0031

Tabla 4-23: Comparativa de resultados de distintos algoritmos para la función f_7

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.1458	0.1462	0.1460	0.0001	0.1482	0.1485	0.1483	0.0001
4	0.0465	0.1519	0.1146	0.0306	0.0442	0.1542	0.1162	0.0323
5	0.0685	0.1027	0.0894	0.0083	0.0709	0.1049	0.0911	0.0085
6	0.0444	0.0810	0.0596	0.0116	0.0432	0.0819	0.0594	0.0124
7	0.0291	0.0667	0.0369	0.0081	0.0301	0.0669	0.0373	0.0076
8	0.0198	0.0510	0.0300	0.0079	0.0200	0.0504	0.0301	0.0071
9	0.0168	0.0388	0.0241	0.0071	0.0186	0.0369	0.0247	0.0055
10	0.0128	0.0344	0.0179	0.0057	0.0136	0.0325	0.0187	0.0045
11	0.0110	0.0314	0.0156	0.0049	0.0119	0.0317	0.0168	0.0042
12	0.0079	0.0248	0.0136	0.0053	0.0096	0.0225	0.0145	0.0039
13	0.0068	0.0208	0.0103	0.0029	0.0087	0.0192	0.0123	0.0025
14	0.0054	0.0186	0.0090	0.0035	0.0071	0.0190	0.0110	0.0029
15	0.0048	0.0163	0.0079	0.0029	0.0071	0.0184	0.0102	0.0027
16	0.0040	0.0113	0.0065	0.0018	0.0062	0.0135	0.0089	0.0017
17	0.0041	0.0139	0.0068	0.0021	0.0058	0.0159	0.0094	0.0021
18	0.0029	0.0128	0.0061	0.0019	0.0053	0.0177	0.0088	0.0023
19	0.0034	0.0121	0.0058	0.0023	0.0055	0.0171	0.0085	0.0027
20	0.0026	0.0077	0.0049	0.0015	0.0048	0.0104	0.0075	0.0017
21	0.0017	0.0098	0.0047	0.0020	0.0043	0.0125	0.0073	0.0021
22	0.0029	0.0069	0.0045	0.0013	0.0047	0.0097	0.0069	0.0015
23	0.0020	0.0093	0.0046	0.0020	0.0037	0.0127	0.0071	0.0022
24	0.0013	0.0085	0.0041	0.0017	0.0026	0.0134	0.0066	0.0022
25	0.0010	0.0073	0.0034	0.0015	0.0031	0.0125	0.0061	0.0020
26	0.0013	0.0058	0.0033	0.0013	0.0029	0.0085	0.0059	0.0014
27	0.0022	0.0081	0.0041	0.0017	0.0046	0.0112	0.0067	0.0018
28	0.0010	0.0067	0.0034	0.0013	0.0031	0.0098	0.0059	0.0014
29	0.0009	0.0060	0.0029	0.0013	0.0031	0.0112	0.0057	0.0016
30	0.0009	0.0068	0.0033	0.0015	0.0014	0.0094	0.0046	0.0019

Tabla 4-24: Resultados del algoritmo propuesto para función f_8

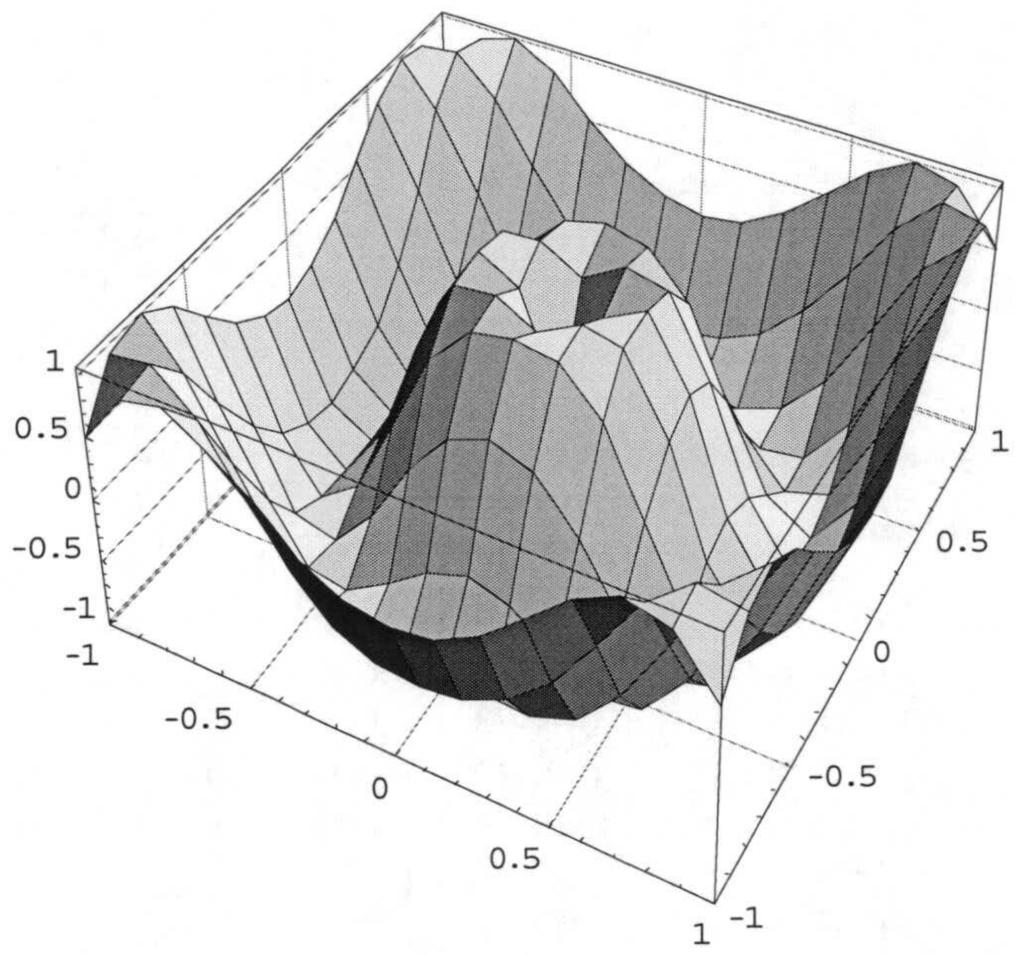

Figura 4-13: Función f_8

Algoritmo	m	n_p	*Test NRMSE*
[Cherkassky, 1996]	40	161	0.049
[Pomares, 2000]	7 × 7 (PGL)	83	0.045
	9 × 9 (TL)	127	0.040
	8 × 8 (G)	92	0.031
[González, 2001]	17	68	0.0561 ± 0.0097
	19	76	0.0551 ± 0.0094
	21	84	0.0457 ± 0.0157
	23	92	0.0303 ± 0.0185
	25	100	0.0271 ± 0.0105
Algoritmo propuesto	17	68	0.0094 ± 0.0021
	19	76	0.0075 ± 0.0017
	21	84	0.0073 ± 0.0021
	23	92	0.0071 ± 0.0022
	25	100	0.0061 ± 0.0020

Tabla 4-25: Comparativa de algoritmos para la función f_8

4.1.3 La serie temporal de Mackey-Glass

Uno de los benchmark más importantes más importantes para los algoritmos de identificación y optimización de modelos son las series temporales basadas en la ecuación diferencial de Mackey-Glass, [Mackey, 1977]. Estas series se generan, esencialmente, a partir de la siguiente ecuación:

$$\frac{ds(t)}{dt} = \alpha \frac{s(t-\tau)}{1+s^{10}(t-\tau)} - \beta s(t) \qquad (4\text{-}14)$$

Los parámetros libres han sido fijados a $\alpha = 0.2$ y $\beta = 0.1$, lo que según el trabajo [Mackey, 1977], convierte a la ecuación diferencial anterior en una serie temporal caótica sin un período claramente definido; no converge ni diverge y es muy sensible a las condiciones iniciales.

Para encontrar una solución numérica a la ecuación diferencial se ha aplicado el método de Runge-Kutta de cuarto orden. Los valores iniciales se han fijado a s(0) = 1.2 y s(t) = 0 cuando t < 0, fijándose $\tau = 17$.

4.1.3.1 Predicción a corto plazo

Según el Embeddig Theorem [Kantz, 1997], toda serie caótica determinística se puede aproximar por una función suave *F*, de la forma:

$$s(t+p) = F(s(t), s(t-p), s(t-2p), \ldots, s(t-dp)) \qquad (4\text{-}15)$$

donde *p* es un retardo y *d* es un parámetro que depende de la dimensión de los atractores caóticos. Siguiendo las convenciones establecidas para predecir esta serie temporal a corto plazo, en esta sección se ejecutará el algoritmo propuesto para buscar redes que predigan el valor $s(t + 6)$ a partir del valor actual $s(t)$ y de valores pasados $s(t - 6)$, $s(t\text{-}12)$, y $s(t\text{-}18)$, empleando por tanto valores de entrenamiento de la forma:

$$[s(t-18), s(t-12), s(t-6), s(t), s(t+6)] \qquad (4\text{-}16)$$

Como en el resto de los trabajos que predicen esta serie temporal, se han usado los primeros 500 vectores para entrenar la red, y los 500 vectores siguientes para validarla. La comparación se realiza con diferentes métodos tanto de diseño de redes neuronales, como de identificación de sistemas difusos [Crowder, 1990][Jang, 1997][Kim, 1997][Lee, 1994][Pomares, 2000][Gonzalez, 2001]

4.1.3.2 Predicción a largo plazo

En este caso se utilizarán los valores $s(t)$, $s(t - 6)$, $s(t - 12)$ y $s(t - 18)$ para estimar el valor de la serie 85 medidas después, es decir, para predecir el valor $s(t + 85)$, por tanto, los vectores de entrenamiento tendrán ahora la forma:

$$[s(t-18), s(t-12), s(t-6), s(t), s(t+85)] \quad (4\text{-}17)$$

Al igual que en el caso anterior, se han usado los 500 vectores para entrenar la red y los 500 vectores siguientes para validarla.

Esta configuración también ha sido utilizada por varios autores para comprobar la capacidad de predicción a largo plazo de los siguientes modelos, como el modelo RAN [Platt, 1991], descrito en el capítulo 3, que va añadiendo funciones base a una red de funciones de base radial basándose en la novedad de los datos de entrada según se supere un umbral de error, o las modificaciones propuestas en [Rosipal, 1998], que incluyen la descomposición QR de Givens (RAN-GQRD) para calcular los pesos de la red y un criterio de poda de funciones base (RAN-P-GQRD) para reducir la complejidad de la red final.

En [Bersini, 1997] se proponen dos algoritmos, uno de fuerza bruta y otro incremental, para entrenar sistemas difusos, y se demuestra que el algoritmo de fuerza bruta, además de presentar un comportamiento inestable conforme se aumenta el número de reglas, no llega a obtener los resultados del algoritmo incremental.

En [De Falco, 1998] se propone el uso de BGAs (Breeder Genetic Algorithms) para entrenar perceptrones multicapa.

4.1.3.3 Resultados

Si se analizan los resultados para la predicción de series temporales, se deduce que se mantiene el buen comportamiento que presentaba nuestro algoritmo para la aproximación de funciones. El error disminuye de una forma lógica, conforme se van introduciendo nuevas funciones base. Además y en estas pruebas, la desviación típica es todavía más pequeña con respecto a la media lo que implica una mayor robustez del algoritmo.

En el caso de la predicción a corto plazo si se comparan los resultados con los de otros métodos, se observa que en general se supera tanto en error, como en complejidad a los modelos generados por otros métodos, con distintos funcionamientos. En este terreno el algoritmo, es tan sólo ligeramente superado por el algoritmo de [González, 2001]

Para la predicción a largo plazo, podemos profundizar un poco más este análisis al incorporarse a la comparativa muchos métodos para el diseño de redes neuronales

en general y RBFNs en particular. Para el único método de identificación de sistemas difusos [Bersini, 1997], se puede observar como aunque hay cierta similitud en los errores obtenidos, el número de parámetros libres de los modelos de éste método es mucho mayor. Si se compara ahora, con un algoritmo para el desarrollo de perceptrones multicapa, se puede afirmar como se supera en general los resultados obtenidos.

Pasamos ahora a realizar la comparativa con los modelos que generan RBFNs, los cuales se describen en el capítulo 2. Para comenzar podemos decir con respecto al algoritmo RAN clásico, los resultados se superan ampliamente. A pesar de que este algoritmo se mejora [Rosipal, 1998], dando como resultado los algoritmos RAN-GQRD y RAN-P-GQRD, los resultados del algoritmo propuesto siguen mejorando a los de estos métodos, aunque la diferencia entre estos ya disminuye bastante. Se considera que las diferencias en coste computacional entre ambos algoritmos no son muy importantes. Por último y con respecto al algoritmo de [González, 2001], aunque las diferencias son mínimas, la balanza se decanta en este caso a nuestro favor, aún si se tiene en cuenta que [González, 2001] usa un algoritmo más complejo.

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.0805	0.1081	0.0877	0.0068	0.0733	0.1064	0.0807	0.0079
4	0.0519	0.0835	0.0665	0.0089	0.0451	0.0779	0.0612	0.0095
5	0.0434	0.0741	0.0548	0.0069	0.0369	0.0710	0.0498	0.0074
6	0.0410	0.1022	0.0505	0.0117	0.0348	0.0960	0.0474	0.0121
7	0.0334	0.0501	0.0415	0.0035	0.0312	0.0480	0.0388	0.0038
8	0.0336	0.0446	0.0390	0.0027	0.0314	0.0422	0.0369	0.0028
9	0.0310	0.0502	0.0368	0.0042	0.0281	0.0457	0.0350	0.0039
10	0.0297	0.0520	0.0344	0.0049	0.0280	0.0471	0.0333	0.0043
11	0.0241	0.0430	0.0314	0.0036	0.0230	0.0405	0.0304	0.0033
12	0.0225	0.0387	0.0282	0.0027	0.0210	0.0385	0.0277	0.0030
13	0.0237	0.0478	0.0284	0.0053	0.0225	0.0466	0.0285	0.0050
14	0.0213	0.0355	0.0255	0.0031	0.0198	0.0346	0.0256	0.0030
15	0.0214	0.0274	0.0243	0.0016	0.0209	0.0275	0.0243	0.0016
16	0.0201	0.0259	0.0232	0.0013	0.0194	0.0261	0.0234	0.0014
17	0.0185	0.0307	0.0231	0.0023	0.0185	0.0309	0.0233	0.0025
18	0.0198	0.0335	0.0227	0.0023	0.0201	0.0317	0.0230	0.0020
19	0.0188	0.0308	0.0219	0.0021	0.0192	0.0286	0.0223	0.0018
20	0.0181	0.0343	0.0202	0.0025	0.0182	0.0349	0.0204	0.0026

Tabla 4-26: Resultados obtenidos por nuestro algoritmo para la predicción de la serie de Mackey-Glass para un valor $s(t+6)$

Algoritmo		*m*	n_p	*Test NRMSE*
Modelo de Predicción Lineal		-	-	0.55
Modelo Auto-Regresivo		-	-	0.19
L-X Wang	T-Norma: Prod.	-	-	0.0907
	T-Norma: Min.	-	-	0.0904
Correlación en Cascada ANN		-	-	0.06
Polinomio de sexto orden		-	-	0.04
D. Kim y C. Kim (Algoritmo genético + Sistema difuso)		$5 \times 5 \times 5 \times 5$ (TL)	665	0.0492
		$7 \times 7 \times 7 \times 7$ (TL)	2457	0.0423
		$9 \times 9 \times 9 \times 9$ (TL)	6633	0.0379
Retropropagación ANN		-	-	0.02
ANFIS (ANN + SLD)		-	-	0.007
[Pomares, 2000]		$3 \times 3 \times 3 \times 3$ (PT)	57	0.0109
		$3 \times 4 \times 4 \times 4$ (PT)	199	0.0071
		$4 \times 4 \times 5 \times 5$ (PT)	410	0.0063
		$2 \times 2 \times 2 \times 2$ (PGL)	24	0.0203
		$3 \times 3 \times 3 \times 3$ (PGL)	101	0.0058
[González, 2001]		4	24	0.0151 ± 0.0019
		7	42	0.0075 ± 0.0009
		10	60	0.0055 ± 0.0010
		13	78	0.0043 ± 0.0011
		16	96	0.0036 ± 0.0002
Algoritmo propuesto		4	24	0.0139 ± 0.0021
		7	42	0.0088 ± 0.0008
		10	60	0.0076 ± 0.0009
		13	78	0.0065 ± 0.0011
		16	96	0.0053 ± 0.0003

Tabla 4-27: Comparativa de resultados para la predicción de la serie de Mackey-Glass para un valor $s(t+6)$

RBFs	*Entrenamiento NRMSE*				*Test NRMSE*			
	Mejor	*Peor*	*Medio*	*Desv.*	*Mejor*	*Peor*	*Medio*	*Desv.*
3	0.4440	0.4665	0.4580	0.0082	0.4227	0.4647	0.4545	0.0146
4	0.4079	0.4502	0.4325	0.0075	0.4037	0.4555	0.4366	0.0083
5	0.3685	0.4297	0.4007	0.0198	0.3684	0.4302	0.3970	0.0238
6	0.3362	0.4376	0.3853	0.0229	0.3357	0.4393	0.3808	0.0250
7	0.3178	0.3824	0.3425	0.0164	0.3064	0.3788	0.3367	0.0180
8	0.2736	0.3721	0.3121	0.0253	0.2615	0.3658	0.3050	0.0266
9	0.2291	0.3524	0.2759	0.0253	0.2233	0.3514	0.2684	0.0265
10	0.2047	0.2902	0.2564	0.0188	0.2003	0.2898	0.2493	0.0207
11	0.1981	0.2751	0.2376	0.0171	0.1874	0.2698	0.2298	0.0187
12	0.1615	0.2494	0.2119	0.0211	0.1656	0.2407	0.2030	0.0191
13	0.1519	0.2281	0.1863	0.0179	0.1400	0.2169	0.1785	0.0169
14	0.1373	0.2491	0.1758	0.0223	0.1324	0.2382	0.1675	0.0210
15	0.1274	0.1831	0.1535	0.0145	0.1305	0.1838	0.1492	0.0135
16	0.1229	0.1774	0.1452	0.0144	0.1215	0.1690	0.1409	0.0126
17	0.1090	0.1532	0.1299	0.0097	0.1077	0.1516	0.1281	0.0091
18	0.1043	0.1321	0.1177	0.0076	0.1046	0.1389	0.1239	0.0084
19	0.0954	0.1284	0.1087	0.0074	0.1064	0.1320	0.1190	0.0070
20	0.0823	0.1247	0.0998	0.0083	0.0843	0.1350	0.1133	0.0125
21	0.0820	0.1131	0.0967	0.0076	0.0960	0.1276	0.1151	0.0082
22	0.0807	0.1138	0.0933	0.0082	0.0789	0.1397	0.1121	0.0113
23	0.0725	0.1553	0.0902	0.0140	0.0841	0.1497	0.1059	0.0134
24	0.0701	0.0980	0.0819	0.0068	0.0834	0.1212	0.1019	0.0095
25	0.0661	0.0920	0.0807	0.0066	0.0758	0.1169	0.0965	0.0106
26	0.0681	0.0870	0.0770	0.0048	0.0708	0.1129	0.0947	0.0101
27	0.0581	0.1047	0.0722	0.0086	0.0695	0.1128	0.0869	0.0095
28	0.0528	0.0783	0.0700	0.0065	0.0632	0.1124	0.0862	0.0119
29	0.0571	0.0802	0.0667	0.0052	0.0651	0.0933	0.0803	0.0066
30	0.0544	0.0758	0.0665	0.0060	0.0620	0.0979	0.0816	0.0096

Tabla 4-28: Resultados obtenidos por nuestro algoritmo para la predicción de la serie de Mackey-Glass para un valor $s(t+85)$

Algoritmo		*m*	n_p	*Test NRMSE*
MLP + BGA [De Falco, 1998]		16	80	0.2666
RAN [Platt, 1991]	$\varepsilon = 0.1$	57	342	0.378
	$\varepsilon = 0.05$	92	552	0.376
	$\varepsilon = 0.02$	113	678	0.373
	$\varepsilon = 0.01$	123	738	0.374
RAN-GQRD [Rosipal, 1998]	$\varepsilon = 0.1$	14	84	0.206
	$\varepsilon = 0.05$	24	144	0.170
	$\varepsilon = 0.02$	44	264	0.172
	$\varepsilon = 0.01$	55	330	0.165
RAN-P-GQRD [Rosipal, 1998]	$\varepsilon = 0.1$	14	84	0.206
	$\varepsilon = 0.05$	24	144	0.174
	$\varepsilon = 0.02$	31	186	0.160
	$\varepsilon = 0.01$	38	228	0.183
Sistemas Difusos [Bersini, 1997]	Fuerza Bruta	10	190	0.1086
		11	206	0.1098
		12	228	0.1026
		13	247	0.2235
		14	266	0.1586
		15	285	0.1028
	Incremental	14	266	0.0965
[González, 2001]		14	84	0.1977 ± 0.0164
		17	102	0.1467 ± 0.0178
		20	120	0.1268 ± 0.0174
		23	138	0.1012 ± 0.0132
		26	156	0.0999 ± 0.0192
		29	174	0.0891 ± 0.0131
Algoritmo propuesto		14	84	0.1675 ± 0.0210
		17	102	0.1281 ± 0.0091
		20	120	0.1133 ± 0.0125
		23	138	0.1059 ± 0.0134
		26	156	0.0947 ± 0.0101
		29	174	0.0803 ± 0.0066

Tabla 4-29: Comparativa de resultados para la predicción de la serie de Mackey-Glass para un valor $s(t+85)$

5. Conclusiones

En este capítulo se ponen de manifiesto una serie de conclusiones extraídas, tras la realización de esta tesis doctoral. En éstas destacarán las aportaciones realizadas, y se señalarán las líneas de trabajo futuro.

Esta memoria presenta en particular un nuevo algoritmo para el diseño y optimización de Redes de Funciones de Base Radial aplicado al problema de la aproximación de funciones. Sin embargo y de una manera más general, representa una estrategia para solventar problemas de búsqueda y optimización.

Para alcanzar nuestro objetivo se aporta una arquitectura híbrida de técnicas bioinspiradas o soft computing. En un nivel inferior de esta arquitectura, tendríamos la red neuronal en sí, que se encargaría del proceso de los datos del problema. En un nivel superior al anterior, se establecería una técnica basada en computación evolutiva, cuya labor consiste en adaptar o configurar de manera automática los parámetros de la red, con el objetivo de buscar un rendimiento óptimo de ésta. Por último se podría distinguir un último nivel, que estaría formado por un sistema de lógica difusa, cuya misión es configurar los mecanismos de aplicación de los operadores de la técnica evolutiva de la capa inferior.

Una vez aplicada esta hibridación de técnicas ya se consigue una solución buena, en el camino adecuado para alcanzar el óptimo global. Para refinar esta solución aún más y conseguir resultados verdaderamente competitivos se recurre al uso de una técnica de minimización del error clásica como es el algoritmo Levenberg-Marquardt. El hecho de que el refinamiento definitivo de la solución lo va a llevar a cabo este método, implica que podemos relajar la intensidad de aplicación de algunas de las fases de la estrategia anterior, con el ahorro en el coste computacional que ello supone. Así pues, la misión de nuestra arquitectura híbrida bioinspirada, se debe definir claramente como aquella que nos va a conducir a obtener una solución robusta, en el sentido de estar en el camino apropiado de alcanzar el óptimo global, sin llegar a obtener una solución demasiado precisa.

Como material introductorio y de referencia para el lector se hace una descripción y estudio en el capítulo 1, de las principales técnicas bioinspiradas o soft computing. Concretamente se analizarán las redes neuronales, la computación evolutiva y la lógica difusa, donde además detallar su funcionamiento, se ponen de manifiesto sus ventajas y algunos ejemplos de uso. Este capítulo se completa con la descripción de las técnicas clásicas de minimización de error, ya que se usarán en la parte final de nuestro algoritmo.

Siguiendo con el material necesario para resolver nuestro problema en el capítulo 2, es estudian las características más importantes de las Redes de Funciones de Base Radial. Para completar este capítulo, se hace un repaso bibliográfico de los algoritmos que se emplean en el diseño de estas redes.

Centrándonos ahora en el algoritmo propuesto, las principales conclusiones y aportaciones que se pueden extraer son:

- Se ha definido un mecanismo de inicialización, con una gran componente aleatoria, para los principales parámetros que caracterizan la red. Sin embargo se respetan unas mínimas reglas, que ahorrarán trabajo a las fases posteriores.

- La estrategia de entrenamiento de los pesos se ha diseñado con características adaptativas y para ahorrar tiempo de computación. Esto es así porque estas fases iniciales no tienen como fin proporcionar una solución con mucha precisión.

- Se ha definido un nuevo marco coevolutivo cooperativo para este tipo de redes, donde cada neurona o función base representa un individuo de la población, constituyendo toda la población, la solución a nuestro problema. Esto individuos deben competir para sobrevivir, ya que si no realizan un trabajo adecuado serán eliminados. Por otra parte también deben cooperar, en el sentido en que cada individuo realiza una aportación a la solución final. Frente a los modelos evolutivos tradicionales, donde se mantiene una población en la que cada individuo representa una red completa, el uso de este marco coevolutivo, también supone un ahorro computacional.

- Como novedad dentro de este marco coevolutivo cooperativo se ha establecido un nuevo esquema de aportación de crédito, en el que se definen tres parámetros que representan la labor que realiza una función base dentro de la red. Así se mide la aportación función base realiza a la salida de la red, el error que se produce dentro de su radio y el grado de solapamiento que tiene con otras funciones base.

- Para establecer un orden entre las funciones base o individuos de la población, se utilizan los fundamentos de las técnicas multiobjetivo que en función de los tres parámetros anteriores definirán una relación de orden entre dos funciones base cualesquiera. Esto nos permitirá diferenciar con más precisión qué funciones base están funcionando mejor y cuáles peor.

- Se han definido tan solo dos operadores para aplicar sobre las funciones base o individuos, lo que simplifica la complejidad del algoritmo y disminuye su coste computacional. Así, se dispone de un primer operador básico de eliminación de funciones base y de un operador de adaptación de éstas. Este último operador representa una nueva aportación de esta memoria, ya que explota la información de la zona donde se encuentra ubicada una función base, para así adaptarla y mejorar su funcionamiento.

- Una de las principales aportaciones realizadas por el algoritmo, es la estrategia de aplicación de los operadores sobre las funciones base o individuos de la población. Para esto se define un sistema de lógica difusa que, a partir de los parámetros que caracterizan una función base en su aportación de crédito, proporcionará las probabilidades de aplicación de cada uno de los operadores sobre una determinada función base.

- Otra importante aportación es el mecanismo diseñado para la introducción de nuevas funciones base. El método tradicional consistía en introducir funciones base en puntos del conjunto de entrenamiento, más o menos aislados y que presentarán un alto índice de error. Nuestro algoritmo propone la definición del concepto de zona o conjunto de puntos contiguos. Las nuevas funciones base se introducirán en las zonas que mayor error tengan. Este mecanismo proporciona una mayor robustez al algoritmo general.

En el capítulo 4, se ha puesto a prueba el método general, aplicándolo al problema de la aproximación funcional, sobre una batería de funciones unidimensionales y bidimensionales. Estas funciones tienen distintas características además del número de variables de entrada como son: la fluctuación de la salida, el rango de definición, etc. El algoritmo también se ha aplicado a la predicción de series temporales, tanto a corto como a largo plazo.

Los resultados obtenidos demuestran un funcionamiento coherente y robusto del algoritmo propuesto. Estos resultados se han comparado con los de otros algoritmos, utilizados también en la aproximación funcional, pero que utilizan diversas estrategias de funcionamiento. Así podemos encontrar tanto algoritmos para el diseño de redes neuronales, como para la identificación de sistemas difusos. Centrándonos en la

comparativa, el algoritmo demuestra que ha alcanzado unos resultados bastante buenos, superando a otros algoritmos más estándares, tanto en el error alcanzado como en la complejidad de los modelos generados.

En cuanto a las líneas de trabajo futuro, aparecen dos de una forma más o menos clara:

- Una segunda línea sería la modificación del algoritmo para que explore de una manera más automática las fronteras de Pareto, definidas por la complejidad de los modelos generados y el error que éstos presentan. La idea sería definir de alguna manera unos rangos entre los que podría variar, la población de funciones base, de modo que con una sola ejecución se delimitara la frontera de Pareto para modelos que tengan una complejidad comprendida entre estos rangos. Las precauciones a tener en cuenta en este desarrollo, van en el sentido de establecer estos rangos de una manera razonable y pensar que cuando se introduce o se elimina una función base de un modelo, seguramente habrá que realizar alguna modificación en las restantes para que se adapten a la nueva situación.

- La primera sería trabajar en una versión concurrente o distribuida de la misma, aprovechando el grado de paralelismo que hay implícito en el problema. El trabajo a realizar en este campo, debe partir del hecho de que se está manteniendo una población en la que los individuos cooperan para proporcionar la salida del sistema. Sin embargo se debe explotar la respuesta localizada que aportan las funciones base para el conjunto de datos de entrada, lo que evidentemente supone para éstas un cierto grado de independencia

Generalizando, y tal y como se ha comentado, en realidad lo que se propone en esta tesis es una arquitectura híbrida de técnicas bioinspiradas para resolver el problemas de búsqueda y optimización. Esto implica que con las debidas adaptaciones, el algoritmo propuesto se podría intentar aplicar a muchos campos. Así, por ejemplo, uno de los problemas más directos a los que se puede comenzar sería el problema de clasificación.

6. Referencias

Alander, J. (2000) "Genetic algorithms and other natural optimizations methods to solve hard problems – A tutorial review." Technical Report 96-1, Department of Information Technology and Production Economics, University of Vaasa, Finland

Andersen, H; Tsoi, C; (1993) "A constructive algorithm for the training of a multilayer
perceptron based on the genetic algorithm" Complex Systems, vol. 7, no. 4, pp. 249-268.

Angeline, P. (1995) "Adaptive and self-adaptive evolutionary computation". In Palaniswami M., Attikiouzel Y., Marks R. J., Fogel D., and Fukuda T., editors, Computational Intelligence: A Dynamic System Perspective, pp. 152-161. IEEE press.

Bäck, T. (1993) "Optimal mutation rates in genetic search" in Proc. 5th Int. Conf. on Genetic Algorithms, S. Forrest, Ed. San Mateo, CA: Morgan Kaufmann, pp. 2-8

Bäck, T. (1994) "Selective pressure in evolutionary algorithms: a characterization of selection mechanism" in Proc. 1st IEEE Conf. on Evolutionary Computation. Piscataway, NJ: IEEE Press, 1994, pp. 57-62

Bäck, T. (1995) "Generalized convergence models for tournament and (μ, λ)-selection. In Eshelman, L. editor, Proceedings of the Sixth International Conference on Genetic Algorithms, pp. 2-8, San Francisco, CA. Morgan Kaufmann.

Bäck, T. (1996). Evolutionary Algorithms in Theory and Practice. Oxford University Press; New York, 1996.

Bäck, T.; Hammel, U.; Schwefel, H.; (1997). Evolutionary computation: comments on the history and current state. IEEE Transactions on Evolutionary Computation, vol. 1 n. 1 April. Pp. 3-17

Balakrishnan, K.; Honavar, V.; (1995) "Evolutionary design of neural architectures". Technical Report CS: TR: 95-01. Departament of computer science. Iowa State University, Ames, IA50011

Bayes, T. (1763). An essay towards solving problem in the doctrine of chances. Philosophical Transactions of the Royal Society of London. 53: pp 370-418.

Beasly, D.; Bull, D.; Martin R.; (1993) "A sequential niche technique for multimodal function optimization." Evolutionary Computation, 1. pp. 101-125.

Belew, R.; McInerney, J. and Schaudolph, N. (1991) "Evolving networks: using genetic algorithm with connectionist learning". Technical Report #CS90-171 (Revised) Computer Science & Engr. Dept. (C-014), Univ. of California at San Diego, La Jolla, CA 92093, USA, February.

Bersini, H.; Duchateau, A.; Bradshaw, N. (1997). "Using incremental learning algorithms in the search for minimal and effective models". In Proceedings of the 6th IEEE International Conference on fuzzy system, pp. 1417-1422, Barcelona, Spain. IEEE Computer Society Press.

Beyer, H.; (1995) "Toward a theory of evolution strategies: On the benefits of sex – the (μ/μ, λ)-theory" Evolutionary Computation, vol. 3, n. 1, pp. 81-111

Bezdek, J.C. (1981) Pattern recognition with fuzzy objective function algorithms. Plenum, New York.

Bonissone, P.; (2000) "Hybrid soft computing system: where are we going?" Proceedings of the 14th European Conference on Artificial Intelligence (ECAI 2000) , Berlin, Germany, pages 739-746, August.

Broomhead, D. and Lowe, D. (1988) "Multivariable functional interpolation and adaptive networks. Complex System. 2. pp 321-355.

Carpenter, G. and Grossberg, S. (1983) "A massively parallel architecture for a self-organizing neural pattern recognition machine" Computer Vision, Graphics, and Image Processing, vol 37, pp 54-115.

Carpenter, G. and Grossberg, S. (1987) "Art 2: Self-organization of stable category recognition codes for analog ouput patterns" Applied Optics, vol. 26, pp 4919-4930, Dec

Carpenter, G. and Grossberg, S. (1990) "Art 3 hierarchical search: Chemical transmitter in self-organizing pattern recognition architectures" in Proc. It. Joint Conf. on Neural Networks, vol. 2, pp. 30-33, Wash., DC, Jan.

Castillo, P.; Arenas, M.; Castellano, J.; Cillero, M.; Merelo, J.; Prieto, A.; Rivas, V.; Romero, G.; (2001). "Function approximation with evolved multilayer perceptrons". In Mastorakis, N. editor, 2001 WSES International Conference on: Neural Networks Applications (NNA ´01). Artificial Inteligence, pp.: 195-200. Puerto de la Cruz, Tenerife, Canary Islands. World Scientific and Engineering Society Press.

Chen, S.; Cowan, C.; Grant, P.; (1991) "Orthogonal least squares learning algorithm for radial basis function networks" IEEE Trans. Neural Networks, 2:302-309

Chen, S.; Chung, E.; Alkadhimi, K.;(1996) "Regularized orthogonal least squares algorithm for constructing radial basis function networks". Int. J. Contr., 64(5) pp. 829-837

Chen, S.; Woo,Y.; Luk, B.; (1999) "Combined genetic algorithms optimization and regularized orthogonal least squares learning for radial basis function networks. IEEE Trans. Neural Netwoks, 10(5):1239-1243

Cherkassky, V.; Lari-Najafi, H. (1991). "Constrained topological mapping for non-parametric regression analysis". Neural networks, 4(1): pp. 27-40.

Cherkassky, V.; Gehring, D.; Muller, F.(1996). "Comparison of adaptative methods for function estimation from samples" IEEE Trans. Neural Networks, 7(4): pp. 969-984.

Chien, C (1990) "Fuzzy logic in control system: fuzzy logic controller" IEEE transactions on system, man and cybernetics vol. 20 n. 2 pp 404-435 March-April

Coello, C.; (1999) "An Updated Survey of Evolutionary Multiobjective Optimization Techniques: State of the art and future trends". Congress on Evolutionary Computation, CEC'99, pp.3-13.

Crowder, R.; (1998) "Predicting the Mackey-Glass Time Series with cascade-correlation learning". In Touretzky, D.; et al. editors, Proceedings of the 1990 Connectionist Models Summer School, pp. 117-123. Carnegie Mellon University. Morgan Kaufmann

Davis, L. (1991). Handbook of genetic algorithms. New York: Van Nostrand Reinhold.

De Falco, I.; Della Cioppa, A.; Iazzetta, A.; Natale, P.; Tarantino, E.; (1998) "Optimizing neural networks for time series prediction". In Roy, R.; et al. editors. Proceedings of the 3rd on line world conference on soft computing (WSC3). Advances in Soft Computing – Engeenering design and Manufacturing Internet. Springer Verlag.

De Jong, K (1975) "An analysis of the behaviour of a class of genetic adaptive system" Ph. D. dissertation, Univ. of Michigan, Ann Arbor, Diss. Abstr. Int. 36(10), 5140B, University Microfilms n. 76-9381.

Dempster, A.; (1967). Upper and lower probabilities induced by multivalued mapping. Annals of Mathematical Statistics, 38:325-339

Dennis, J. E.; Schnabel, R. B. (1983) Numerical Methods for unconstrained optimization and nonlinear equations. Prentice Hall, Englewood Cliffs, N.J.

Dickerson, J.A.; Kosko, B.; (1996): "Fuzzy function approximation with ellipsoidal rules". IEEE Trans. Syst. Man and Cyber., Part B, 26(4):542-560.

Duda, R. O; Hart, P. E; (1973) Pattern classification and scene análisis, Wiley, New York.

Eiben, A.; Hinderting, R.; Michalewicz, Z. (2000) "Parameter control in evolutionary computation" IEEE Trans. on Evolutionary Computation, vol. 3, n. 2, pp. 124-141,

Eshelman, L.; Caruna, R.; Schaffer J.; (1989) "Biases in the crossover landscape" in Proc. 3rd Int. conf. on Genetic Algorithms. San Mateo, CA: Morgan Kaufmann, pp. 10-19

Eshelman, L.; Schaffer, J.;(1993) "Real-coded genetic algorithms and interval-schemata" in Foundations of Genetic Algorithms 2. San Mateo, CA: Morgan Kaufmann, pp. 187-202.

Fahlman, S.; (1988) "Faster-learning variations on back-propagation: an empirical study". In Touretzky, D. et al., editors, Proc. of the 1988. Connectionist models summer school, pp. 38-51. Morgan Kaufmann, San Mateo, CA. 1988.

Fahlman, S.; Lebiere, C.; (1990) "The cascade-correlation learning architecture," in Advances in Neural Information Processing Systems 2 (D. S. Touretzky, ed.), pp. 524-532, Morgan Kaufmann, San Mateo, CA

Fogarty, T. (1989) "Varying the probability of mutation in the genetic algorithm. In J.D. Schaffer, editor, Proceedings of the 3rd. International Conference on Genetic Algorithms, pp 104-109. Morgan Kaufmann

Fogel, L. (1962) "Autonomous automata" Ind. Res. vol. 4 pp 14-19, 1962

Fogel, D. (1993) "On the philosophical differences between evolutionary algorithms and genetic algorithms", in Proc. 2nd Annu. Conf. on Evolutionary Programming Society. pp. 23-29

Fonseca, C.; Fleming, P.; (1993) "Genetic algorithms for multiobjetive optimization. In Forrest ed. Proceedings of the fifth international conference on genetic algorithms. Morgan Kaufmann.

Frean, M.; (1990) "The upstart algorithm: a method for constructing and training feedforward neural networks," Neural Computation, vol. 2, pp. 198 - 209

Friedman, J.; (1981). "Projection pursuit regression". J. Amer. Statist. Assoc., 76:817-823.

Friedman, J.; (1991). "Multivariate adaptative regression splines (with discussion)". Ann. Statist., 19:1-141.

García, N., Hervás, C., Muñoz, J. (2002) "Multi-objetive cooperative coevolution of artificial neural networks". Neural networks, 15, pp. 1259-1278

Geman, S., Bienenstock, E. and Doursat, R. (1992), "Neural Networks and the Bias/Variance Dilemma", Neural Computation, 4, pp 1-58.

Gersho, A.; (1979) "Asymtotically optimal block quatization" IEEE Trans. Inf. Theory, 25(4): 373-380.

Ghost, J.; Deuser, L.; Beck, S.; (1992) "A neural network based hybrid system for detection, characterization and classification of short-duration oceanic signals, IEEE Jl. Of Ocean Enginering, 17(4):351-363, October

Giordana, A, Saitta, L. and Zini, F. (1994). "Learning disjunctive concepts by means of genetic algorithms". In Cohen, W et al., editors, Proceedings of the 11th International Conference on Machine Learning, pp. 96-104, Morgan Kaufmann, San Francisco, California.

Giordana, A., Neri, F. (1996). "Search-intensive concept induction". Evolutionary Computation, 3(4):375-416.

Goldberg, D. E. 1978. Genetic Algorithms. Addison Wesley, Reading, M.A.

Goldberg, D.; Richardson J.; (1987). "Genetic algorithms with sharing for multimodal function optimization. In Grefenstette (ed.), Proceedings of the Second International Conference on Genetic Algorithms, pp. 41-49. Lawrence Erlbaum Asociates.

Golub, G.; Van Loan, C.; (1996) Matriz computations. Johns Hopkins University Press, Baltimore, 3rd ed.

González, J; (2001) Identificación y optimización de redes de funciones de base radial para aproximación funcional. Tesis doctoral. Dpto. Arquitectura y Tecnología de Computadores. Universidad de Granada

González, J.; Rojas, I.; Pomares, H.; Ortega, J.; Prieto, A.; (2002) "A new clustering technique for function approximation" IEEE Trans. Neural Networks. Vol. 13. n.1 Januray. pp: 132-142

Grönroos, M.;(1998) Evolutionary design of neural networks. Master of Science Thesis. University of Turku

Grossberg, S. (1976) "Adaptative pattern classification and universal recording, II: Feedback, expectation, olfaction and illusions" Biolog. Cybernetics, vol 23, pp 187-202, 1976.

Harp, S; Samad, T; Guha, A; (1989) "Towards the genetic synthesis of neural etworks"
in Proc. of the Third Int'l Conf. on Genetic Algorithms and Their Applications (J. D. Schaffer, ed.), pp. 360--369, Morgan Kaufmann, San Mateo, CA.

Hartigan, J. A.; (1975) Clustering Algorithms. Willey, New York, 1975.

Haykin, S.; (1999) Neural Networks. A comprehensive foundation, 2nd. Upper Saddle River, NJ: Prentice-Hall.

Hebb, D. (1949) The organization of behaviour. New York: Wiley.

Hindertin, R.; Michalewicz Z.; Peachey, T.; (1996) Self-adaptive genetic algorithm for numeric functions. In Voight et al. Proceedings of fourth Conference on Parallel Problem Solving from Nature, number 1141 in LNCS. Springer, Berlin.

Holland, J. (1975). Adaptation in natural and artificial system. Ann Arbor, MI: Univ. of Michigan Press.

Holland, J. (1978). Cognitive system based on adaptive algorithms. In Waterman, D. et al. editors, Pattern-Directed Inferenced System. Academic Press, New York.

Hopfield, J. (1982) "Neural networks and physical system with emergent collective computational abilities" Proc. Natl. Acad. Sci. vol. 79, pp. 2554-2558, Apr.

Hopfield, J. (1984) "Neurons with graded response have collective computational properties like those of two-state neurons". Proc Natl Acad. Sci., vol. 81, pp. 3088-3092, May

Jacobs, D.A.; (1977) The state of the art in numerical analysis, chapter III. Academic Press, London.

Janikow, C.; Michalewicz, Z.; "An experimental comparison of binary and floating point representation in genetic algorithms" in Proc. 4th Int. Conf. on Genetic Algorithms. San Mateo, CA, Morgan Kaufman, pp. 31-36

Jang, J.; Sun, C.; Mizutani, E.; (1997). Neuro-fuzzy and Soft computing. Pretice Hall, Upper Saddle River, N. J.

Kadirkamanathan, V.; (1993) A function estimation approach to sequential learning with neural networks. Neural computation, 5, 954-975.

Kanjilal, P.; Banerjee, D.; (1995) "On the application of orthogonal transformation for the design and analysis of feedforward networks. IEEE Trans. neural networks, 6(5): 1061-1070

Kantz, H.; Schreider, T.;(1977). Nolinear time series analysis. Cambridge University Press

Karayiannis, N. B; Mi, G. W. (1997) "Growing radial basis networks: merging supervised and unsupervised learning with networks growth techniches. IEEE Trans. Neural Networks, 8(6):1492-1506, Nov.

Kauffman, S; Johnsen (1991). "Co-evolution to the edge of chaos: Coupled fitness landscapes, poised states, and co-evolutionary avalanches. In C. G. Langton et al. editors. Artificial Life II, SFI Studies in Sciences of Complexity, vol. 10, pp. 325-369. Addison-Wesley

Kim, D.; Kim, C.;(1997). "Forecasting time series with genetic fuzzy predictor ensemble. IEEE Trans. Fuzzy System, 5(4), pp. 523-535

Kohonen, T (1984), "Self-Organization and Associative Memory" Berlin: Springer-Verlag.

Kohonen, T (1988), "Learning Vector Quantization," Neural Networks, 1 (suppl 1), 303.

Kowalski, J.; Hartman, E.; Keeler, J. (1990) "Layered neural networks with Gaussian hidden units as universal approximators" Neural Computation, 2: 210-215.

Kursawe, F.; (1995) "Toward self-adapting evolution strategies" in Proc. 2nd IEEE Conf. Evolutionary Computation, Perth, Australia. Piscataway, NJ: IEEE Press, pp. 283-288

Larsen, P. (1980) "Industrial applications of fuzzy logic control" Int. J. Man, Mach Studies, vol 12, n. 1, pp 3-10.

Lee, S.; and Rhee, M. (1988) "Multilayer feedforward potential function network" In Proceedings of the Second International Conference on Neural Networks. Pp. 161-171

Lee, S.; Kim, I.; (1994). "Time series analysis using fuzzy learning". In Kim, W. et al. editors. Proceeding of the International Conference on Neural Information Processing, ICONIP'94, volume 6, pp. 1577-1582, Seoul, Korea.

Light, W.(1992) "Some aspects of radial basis function approximation. In S.P. Singh, editor, Approximation Theory, Spline Functions and Applications, pp. 163-190. NATO ASI Series Vol. 256, Kluwer Acad., Boston.

Lipmann, R (1987). "An introduction to computing with neural nets". IEEE Transactions on Acoustics, Speech, and Signal Processing, 2(4), 4-22

Mackey, M.; Glass, L.; "Oscillation and chaos in physiological control system" Sci vol. 197, pp. 287-289, 1.977.

Mandani, E. (1974). "Applications of fuzzy algorithms for symple dinamycs plant". Proc. IEEE, vol 121, pp. 1585-1588

Mandani, E.; Assilian, S. (1975) "An experiment in linguistic synthesis with a fuzzy controller" Int. J. Mach Studies vol. 7 n. 1 pp. 1-13.

Marquardt, D.; (1963) Journal of the society for industrial and applied mathematics, vol. 11, pp. 431-441.

McCulloch, W. and Pitts, W. (1943). "A logical calculus of the ideas immanent in nervous activity". Bulletin of Mathematical Biophysics, 7. pp115-133

Megdassy, P. (1961) Decomposition of superposition of distributed functions. Hungarian Academy of Sciences, Budapest.

Michalewicz, Z.; Krawczyk, J.; Kazemi, M.; Janikow, C (1990) "Genetic algorithms and optimal control problems". In proceedings of the 29th IEEE Conference on Decision an Control, pp 1664-1666. IEEE Computer Society Press

Michalewicz, Z. (1993) "A hierarchy of evolution programs: An experimental study" Evolutionary computation. vol 1 n. 1 pp. 51-76

Micchelli, C. (1986) "Interpolation of scattered data: distance matrices and conditionally positive definite functions" Constructive Approximation, 2: 11-22.

Minsky, Papert (1969) Perceptrons, An introduction to computational geometry, MIT press, expanded edition.

Moller, M.; (1993) "A scaled conjugate gradient algorithm for fast supervised learning". Neural Networks, 6:pp. 525-533.

Moody, J; Darken, C. (1989) "Fast learning networks of locally-tuned processing units," Neural Computation, vol. 3 n. 4 pp.579-588.

Moriarty, D., Miikkulainen, R. (1997). Forming neural networks through efficient and adaptive coevolution. Evolutionary computation, 5(4): 373-399.

Musavi, M.; Ahmed, W.; Chan, K.; Faris, K.; Hummels, D.; (1992) "On the training of radial basis functions classifiers" Neural Networks, 5(4): pp 595-603

Nie, J.H.; Lee, T.H.;(1996). "Rule based modeling: Fast construction and optimal manipulation". IEEE Trans. Syst. Man Cyber., Part A, 26(6).

Nix, A.; Vose, M.; (1992) "Modelling genetic algorithms with Markov chains" Ann. Math. Artif. Intell. vol. 5, pp. 79-88

Orr, M.; (1993) "Regularized center recruitment in radial basis function networks". Technical report 59. Center for cognitive science. University of Edinburgh.

Orr, M.; (1996) "Introduction to radial basis function networks". Technical report. Center for cognitive science. University of Edinburgh.

Park, J.; Sandberg, I.; (1991) "Universal approximation using radial basis function networks". Neural Computation, 3(2):246-257.

Park, J.; Sandberg, I.; (1993) "Universal approximation and radial basis function networks". Neural Computation, 5(2):305-316.

Pareto, V.; (1896) Cours d'economie politique. Vol I et II. F. Rouge. Lausanne.

Pedrycz, W.; (1998) "Conditional fuzzy clustering in the design of radial basis function neural networks". IEEE Trans. Neural Networks. 9(1):601-612

Platt, J.; (1991) "A resource-allocating network for function interpolation", Neural Computation 3, 213-225

Poggio, T.; Girosi. F. (1990) "Networks for approximation and learning. Proc. IEEE, 78(9): 1481-1497, Sept.

Polak, E.; (1971) Computational methods in optimization. Academic Press. New York

Pomares, H.; Rojas, I.; Ortega, J.; González, J.; Prieto, A.;(2000). "A systematic approach to Self-Generating Fuzzy Rule-Table for Function Approximation". IEEE Trans. Syst., Man, and Cyber., Part B, 30(3):431-447.

Potter, M., De Jong, K, (2000) "Cooperative Coevolution: an architecture for evolving coadapted subcomponents. Evolutionary Computation, 8(1), 1-29.

Powell, M. (1985). "Radial basis functions for multivariable interpolation: A review". In IMA. Conf. on Algorithms for the approximation of functions and data, pp. 143-167.

Prados, D. (1992). "Training multilayered neural networks by replacing the least fit hidden neurons. In Proc. of IEEE SOUTHEASTCON'92, vol. 2, pp. 634-637. IEEE Press, New York, NY.

Press, W.; Vettering, W.; Teukolsky, S.; Flannery, B.; (1994) Numerical recipes in C. Cambridge University Press, 2nd edition.

Procyk, R.; Mandani E.(1979) "A linguistic self-organizing process controller" Automat. vol 15 n. 1, pp 15-30.

Rechenberg, I. (1973) Evolutionsstrategies: Optimierung technischer systeme nach prinzipien der biologischen evolution. Sttutgart, Germany: Frommann-Holzboog.

Renders, J.; Flasse, S.; (1996) "Hybrid methods using genetics algorithms for global optimization" IEEE Trans. Syst., Man, Cybern. B, vol. 26 n. 2, pp. 243-258

Rivas, V.; Castillo, P.; Merelo, J.; (2001) "Evolving RBF Networks" In Mira, J. and Prieto, A. (eds): IWANN 2001, LNCS 2084, pp. 506-513. Springer-Verlag.

Rojas,I.; Valenzuela,O.; Prieto,A.(1997): "Statistical Analysis of the Main Parameters in the Definition of Radial Basic Function Networks", Lecture Notes in Computer Science, Vol.1240, pp. 882-891, Springer-Verlag, June.

Rojas, I.; Pomares, H.; Ortega, J.; Prieto, A.; (2000a) "Self-Organized fuzzy system generation from training examples". IEEE Trans. On Fuzzy System, vol.8. n. 1 pp. 23-36. February

Rojas, I.; González, J.; Cañas, A.; Diaz, A.; Rojas, F.; Rodríguez, M.; (2000b) "Short-term prediction of chaotic time series by using RBF networks with regresión weiths". Int. Journal of Neural System, 10(5): 353-364

Rosenblatt, F. (1962) Principles of Neurodynamics: Percentrons and the Theory of Brain Mechanisms. Washintong, DC: Spartan Books.

Rosipal, R.; Koska, M.; Farkaš, I.; (1998). "Prediction of chaotic time series with a resource allocating rbf networks". Neural Processin Letters, 7:185-197.

Rovatti, R.; Guerrieri, R.; (1996). "Fuzzy sets of rules for sistem identification". IEEE Trans. Fuzzy Sistem, 4(2):pp. 89-102

Rumelhart, D.; Hinton, G.; Willians, R.; (1986) "Learning internal representation by error propagation" in Rumelhart & McClelland (Eds), Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations. MIT Press

Runkler, T. A.; Bezdek, J. C. (1999) "Alternating cluster estimation: A new tool for clustering and function approximation. IEEE Trans. Fuzzy System, 7(4):377-393, Aug.

Russo, M.; Patané, G.; (1999) "Improving the LBG alorithm", volume 1606 of Lecture Notes on Computer Science, pp. 621-630. ISSN: 0302-9743, Springer-Verlag, 1999

Salmerón, M.; Ortega, J.; Puntonet, C.; Prieto, A.; (2001) "Improved RAN sequential prediction using orthogonal techniques. Neurocomputing. In press. Paper code Neucom 2001

Shafer, G.; (1976). A Mathematical Theory of Evidence. Princenton University Press, Princenton, NJ.

Schwefel, H (1975) Evolutionsstrategie und numerische optimierung dissertation. Technische Universität Berlin, Germany. May

Schwefel, H (1977) Numerische optimierung von computer modellen mittels der evolutionsstrategie, vol. 26 of interdisciplinary system research. Birhäuser, Basel.

Schwefel, H (1995) "Contemporary evolution strategies" in Advances in Artificial Life. 3rd Int. Conf. on Artificial Life (Lecture Notes in Artificial Intelligence, vol. 929). F. Morán, et al. Eds. Berlin, Germany: Springer, pp. 893-907

Smith, J.; Fogarty T.; (1996) "Adaptively parameterised evolutionary system: Self adaptive recombination and mutation in a genetic algorithm.

Spears, W.; (1995) "Adapting crossover in evolutionary algorithms. In McDonnell et al. Proceedings of the Fourth Annual Conference on Evolutionary Programing pp. 367-384.

Srinivas, N.; Dev, K.; (1995) Multiobjetive optimization using nondominated sorting in genetic algorithms. Evolutionary computation, 2(3):221-248

Sudkamp, T.; Hammell, R.J.;(1994). "Interpolation, completion, and Learnig Fuzzy Rules". IEEE Trans. Syst. Man and Cybernetics, 24(2):332-343.

Sugeno, M.; Kang, G. (1988) "Structure identification of fuzzy model" Fuzzy Sets and System, vol. 28 pp. 15-33.

Syswerda, G. (1989) "Uniform crossover in genetic algorithms" in Proc. 3rd Int. Conf. on Genetic Algorithms. San Mateo, CA: Morgan Kaufman. pp. 2-9

Tsukamoto, Y. (1979) "An approach to fuzzy reasoning method" in Advances in Fuzzy Set Theory and Applications , Madan M. Gupta, Rammohan K. Ragade, and Ronald R. Yager, Eds. Amsterdam: North-Holland, pp. 137-149

Uhr, L.; (1973) Pattern Recognition, Learning, and Thought. Englewood Cliffs, NJ: Prentice Hall

Wang, L.X.; Mendel, J.M.;(1992). "Generating Fuzzy rules by learning from examples". IEEEb Trans. Syst. Man and Cybernetics, 22(6):1414-1427.

Werbos, P (1974). "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences". Unpublished doctoral dissertation, Harvard University.

Whitehead, B; Choate, T.;(1996) "Cooperative-competitive genetic evolution of Radial Basis Function centers and widths for time series prediction". IEEE Trans. on Neural Networks, Vol.7, No.4, pp.869-880. July.

Windrow, B.; Hoff, M. (1960). Adaptive switching circuits. In 1960 IRE WESCON Convention, pp 96-104. IRE

Windrow, B.; Stearns, S. (1985). Adaptive signal processing. Engleewood Cliffs, NJ: Prentice-Hall.

Widrow, B.; Lehr, M.A. (1990) 30 Years of adaptative neural networks: perceptron, madaline and backpropagation. Proceedings of the IEEE, Vol. 78 n. 9, september

Wolpert, D.H. (ed.) (1995b), The Mathematics of Generalization: The Proceedings of the SFI/CNLS Workshop on Formal Approaches to Supervised Learning, Santa Fe Institute Studies in the Sciences of Complexity, Volume XX, Reading, MA: Addison-Wesley.

Yao, X (1999) "Evolutionary artificial neural networks" Proceedings of the IEEE, 87 (9): pp. 1423-1447, September

Yingwei, L.; Sundarajan, N.; Saratchandran, P.; (1997) "A sequential learning scheme for function approximation using minimal radial basis function neural networks" Neural Computation, 9:361-478

Zadeh, L. (1965) "Fuzzy sets", Inform. And Contr., vol. 8, pp. 338-353.

Zadeh, L. (1973) "Outline of a new approach to the analysis complex system and decision processes" IEEE Trans. Syst. Man Cybern., vol. SMC-3. pp 28-44.

Zadeh, L. (1994) "Fuzzy logic and soft computing: issues, contentions and perspectives". In Iizuka 94: 3rd International conference on fuzzy logic, neural nets and soft computing, pp: 1-2, Iizuka, Japan.