Redes de Neuronas Artificiales

Curso: 2001/2002
Cuatrimestre: Primero
Alumna: Laura M. Castro Souto
Profesores: Antonino Sánchez del Riego
Julián Dorado de la Calle

Índice general

Parte I

Fundamentos Biológicos de las Redes de Neuronas Artificiales

Capítulo 1

Introducción a la neurocomputación

Entendemos por **neurocomputación** la utilización de los *sistemas conexionistas* o *redes de neuronas artificiales* para realizar una computación o procesado de información.

El principal componente de la **neurocomputación** son, pues, los *sistemas conexionistas* o *redes de neuronas artificiales*, términos análogos, y su meta fundamental, atacar *problemas reales*. Este tipo de problemas se caracteriza por su incertidumbre, ruido e imprecisión en los datos, etc. Por tanto, los sistemas conexionistas deben reflejar cierta tolerancia a dicha imprecisión e incertidumbre. Es decir, las RNA[1] *no* se aplican a algo de tipo algorítmico sino todo lo contrario, se vuelven útiles frente a problemas para los que no existe o no se conoce un algoritmo que los aborde, o bien cuando dicho algoritmo existe pero resulta computacionalmente demasiado lento o caro[2] (las RNA suelen ser muy rápidas en funcionamiento).

Las RNA están inspiradas en las *redes de neuronas biológicas*, tanto en su estructura como en su comportamiento (algoritmos genéticos, computación molecular). Ahora bien, una neurona biológica, comparada con cualquier otro dispositivo eléctrico, es muy lenta. ¿Por qué, entonces, funcionan y dan tan buen resultado las RNA? Debido a su *alto grado de paralelismo*; su lentitud se compensa con el gran número de componentes que las forman y trabajan al unísono.

Los **neurocomputadores** introducen, por tanto, inspirados en esta idea, el paralelismo en su arquitectura, sustituyendo a la clásica arquitectura secuencial de Vonn Neumann.

Las RNB[3] pueden verse, siguiendo la idea anterior, como dispositivos no lineales con una alta capacidad de paralelismo caracterizados sobre todo por una gran robustez y tolerancia a fallos[4]. Resultan excepcionalmente buenas en tareas tales como:

[1] Abreviaremos de este modo "*redes de neuronas artificiales*".

[2] Algo que, por otra parte, cada vez ocurre menos.

[3] Abreviaremos así "*redes de neuronas biológicas*".

[4] A pesar de que mueren continuamente, las funciones que realizan globalmente no se ven afectadas por ello. La cuestión de si se reproducen o no suscita aún muchas controversias.

- Procesos de inferencia.
- Tareas de sentido común.
- Procesado de imágenes.

y es por ello por lo que las RNA intentan y pretenden *emular* las RNB. Son tareas que el ser humano lleva a cabo muy bien y que pretenden ser automatizadas (*dominios de aplicación*: reconocimiento de patrones, predicción, optimización, control, memorias asociativas, etc.).

Deben distinguirse bien los conceptos de **emular** y **simular**, que es aplicable a los *sistemas expertos* (parte simbólica). Ambos significan un intento de reproducir el comportamiento, pero en la *emulación*, además, se trata de imitar no sólo unas reglas de funcionamiento, sino la estructura interna de algo.

En las RNB, el aprendizaje es el resultado de un complejo proceso químico-eléctrico en la sinapsis, desatado a raíz de información que llega proveniente del exterior por los sentidos. Las RNA lo hacen *modificando sus pesos sinápticos* como consecuencia de la introducción de información de diversa índole: imágenes, probabilística (con ruido, difusa, imprecisa...), etc.

Pero, ¿qué es **aprender**? Consideramos que el proceso de *aprendizaje* consiste en, a partir de mucha y diversa información que llega por todo tipo de medios, ser capaz de integrarla y abstraerla para generar nueva información y conocimientos aplicables a situaciones nuevas. En suma, es *capacidad de generalización*.

1.1. RNA vs. Programación Secuencial

Las RNA almacenan toda la información en los *pesos sinápticos*. Los elementos principales de las RNA, llamados **elementos de proceso** o **procesado** (**EP**), disponen de un conjunto de primitivas que definen su comportamiento y que pueden ser definidas y pueden ser asimétricas (no tienen por qué ser iguales en todos los PE de todas las capas de una RNA), como puede ser por ejemplo la *función umbral*, cuyos valores más típicos son:

$$\sum_i x_i w_{ij}$$

$$f\left(\sum_i x_i w_{ij}\right) \quad \text{donde} \quad f \text{ función umbral} \begin{cases} \text{sigmoidal} \\ \text{tangente hiperbólica} \\ \text{sinoidal} \\ \text{logarítmica} \end{cases}$$

Otros parámetros que caracterizan a las RNA son:

√ Topología.

√ Propiedades de los EP.

√ Regla o Esquema de aprendizaje utilizado (algoritmo).

√ Validación (mediante el uso de índices o indicadores para medir ratios de error):

- Análisis del error.
- Capacidad de generalización.

√ Ejecución.

Si comparamos una *red neuronal* (de cualquier tipo) con una *máquina de Von Neumann* veremos que cuenta con:

- ◦ Mayor grado de paralelismo.
- ◦ Representación, computación distribuida.
- ◦ Capacidad de aprendizaje.
- ◦ Capacidad de generalización.
- ◦ Capacidad de adaptación.
- ◦ Tolerancia a fallos.
- ◦ Dominios de aplicación con problemas perceptuales[5] complejos, problemas de sentido común,... frente al dominio de computación numérica.

Más detalladamente puede verse en la tabla 1.1 (página 10).

Hoy por hoy, en el estudio de las RNA, RNB y la computación, colaboran las siguientes ramas de la ciencia:

- ∗ neurofisiología[6]
- ∗ ciencia cognitiva, psicología
- ∗ física (mecánica, termodinámica)
- ∗ teoría de control
- ∗ informática, computación
- ∗ inteligencia artificial
- ∗ estadística, matemática

Los campos sobre los que se vaticina mayor proyección de futuro:

- ▶ computadores neuronales
- ▶ programación inteligente de propósito general
- ▶ computación molecular

[5] Perceptivos.

[6] Los estudios sobre neonatos son muy intensivos y abundantes.

	COMPUTACIÓN VON NEUMANN	MODELO RN
PROCESADOR	▷ complejo ▷ alta velocidad ▷ uno (generalmente)	▷ simple ▷ lento ▷ altamente paralelo (gran número)
MEMORIA	▷ separada del procesador ▷ localizada ▷ no direccionable por contenido	▷ integrada en el procesador ▷ distribuida ▷ direccionable por contenido
COMPUTACIÓN	▷ centralizada ▷ secuencial ▷ programa almacenado	▷ distribuida ▷ paralela ▷ autoaprendida
TOLERANCIA A FALLOS	▷ muy vulnerable	▷ robusto
DOMINIO DE APLICACIÓN	▷ manipulación numérica y simbólica	▷ problemas perceptuales y de sentido común
ENTORNO DE OPERACIÓN	▷ muy bien definido y construido	▷ pobremente definido y construido

Cuadro 1.1: Modelo Von Neumann vs. Modelo RN.

1.2. Breve historia de las RNA

Haremos unas breves pinceladas de los hechos históricos más importantes en la historia de las redes neuronales[7]

Época de aparición Trabajos de McCulloch & Pitts (1940) y Ramón y Cajal.

Época de decaimiento Trabajos de Rosenblatt sobre el perceptrón (1960); críticas de Minsky & Paper.

Época de resurgimiento Trabajos de Hopfield (1980); algoritmo *backpropagation* propuesto por Werbs y popularizado por Rumelhart (1981).

1.3. Esquemas o Paradigmas de Aprendizaje

Existen tres *esquemas* o *paradigmas* típicos para el aprendizaje o entrenamiento de una red de neuronas artificial, que veremos a continuación.

1.3.1. Aprendizaje Supervisado

Tal vez el esquema más utilizado a lo largo de la historia, este paradigma de aprendizaje[8] está en función de **salida deseada vs salida obtenida**.

[7]El libro *Neurocomputing: foundations of research* de J.A. Anderson & E. Rosenfeld de MIT PRESS, 1990, es una gran referencia para conocer mejor esta historia.

[8]Proceso que produce la modificación de los pesos sinápticos.

Se analiza el error y se realimenta el algoritmo de aprendizaje. Los patrones se introducen repetidamente hasta disminuir lo máximo posible el error global.

Una variante es el **aprendizaje por refuerzo**, en el que no existe salida deseada. Lo que se hace es, mediante algún tipo de ecuación, se determina la corrección de la salida (no la respuesta correcta) y se refuerza la salida en función de dicha corrección.

1.3.2. Aprendizaje No Supervisado

Se presentan a la red grandes cantidades de datos[9], en los que estos algoritmos buscan regularidades estadísticas, definiendo en base a ellas *categorías* o *clústeres*.

1.3.3. Paradigma Híbrido de Aprendizaje

Utiliza una combinación de los dos anteriores, bien en un orden, bien en el otro. Ejemplo: redes RBF[10].

1.4. Las RNA más utilizadas

Las redes de neuronas que más se usan son:

- Redes de Hopfield.
- Perceptrón Multicapa.
- Mapas Autoorganizativos (Self-Organizing maps, SOM).
- RBF (Radial Basic Function).
- ART (Adaptive Resonant Theory).

Veremos algunas de ellas en las subsecciones siguientes.

1.4.1. Redes de Hopfield

Las **redes de Hopfield** son redes *recurrentes* de interconexión completa (ver figura 1.4.1, página 12).

La información del exterior puede llegar tanto a un único PE como a todos ellos. Mientras que en la mayoría de las redes el estado de los PE se actualiza de modo síncrono, capa por capa (**feedforward**[11]), en estas redes los EP se pueden actualizar en paralelo.

Las **redes de Hopfield** funcionan como *memorias asociativas*, esto es, buscan la reconstrucción de patrones incompletos o con ruido.

[9] El aprendizaje no supervisado suele necesitar muchos más *patrones de entrenamiento* que el supervisado.

[10] En este punto puede resultar conveniente la consulta de las referencias [1] y [2] de la bibliografía.

[11] Cada capa puede ser un proceso, comunicarse por mensajes, ser threads,...

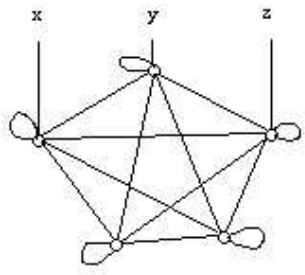


Figura 1.1: Ejemplo de red de Hopfield.

Máquinas de Boltzman

Están basadas en funciones de la *termodinámica*, funciones de energía que según el problema han de ser maximizadas o minimizadas.

Aplicaciones de las redes Hopfield

Entre otras, las redes Hopfield se usan para:

- Segmentación y filtrado de imágenes.
- Optimización (problemas como el del viajante).
- Memorias direccionables por contenido.

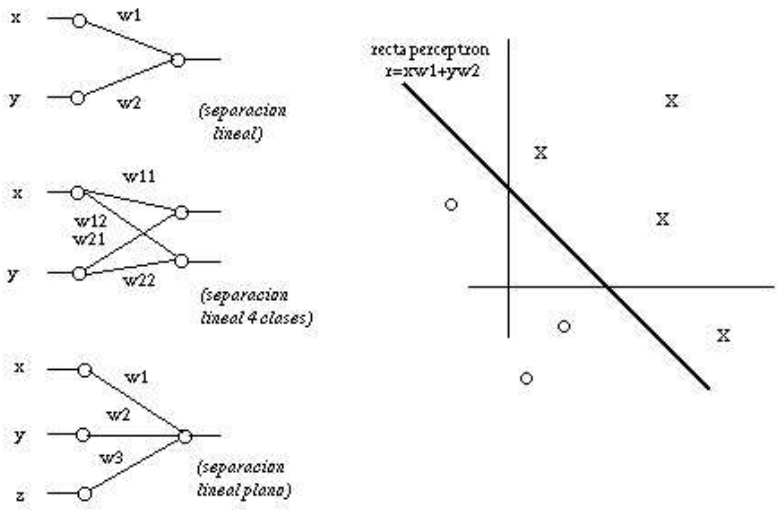
1.4.2. Perceptrón Multicapa (Multilayer Perceptron)

Una de las primeras RNA que se desarrollaron, sus aplicaciones residen en el campo del reconocimiento y la clasificación (ver figura 1.4.2, página 13).

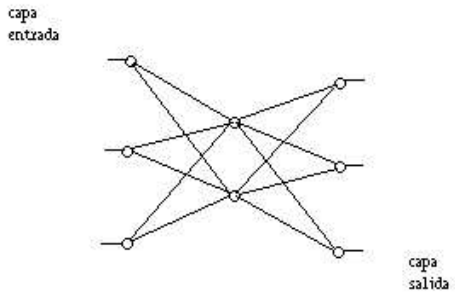
En origen, se habla del **perceptrón simple**, una RNA de una capa capaz únicamente de diferenciar tantas clases como PE se colocasen en la capa de salida, esto es, de establece una *separación lineal* en el espacio en que nos encontremos (marcado por el número de PE en la capa de entrada).

Por tanto, es incapaz de implementar operaciones como el `XOR`, careciendo, pues, de dominios de aplicación reales, ya que lo normal en dominios reales es que los patrones no sean separables linealmente.

Por esta razón se evolucionó al **perceptrón multicapa**, una RNA de interconexión completa con aprendizaje supervisado.



(a) Simple.



(b) Multicapa.

Figura 1.2: Perceptrón.

1.5. Clasificación general

El que sigue es un esquema general de clasficación de las RNA:

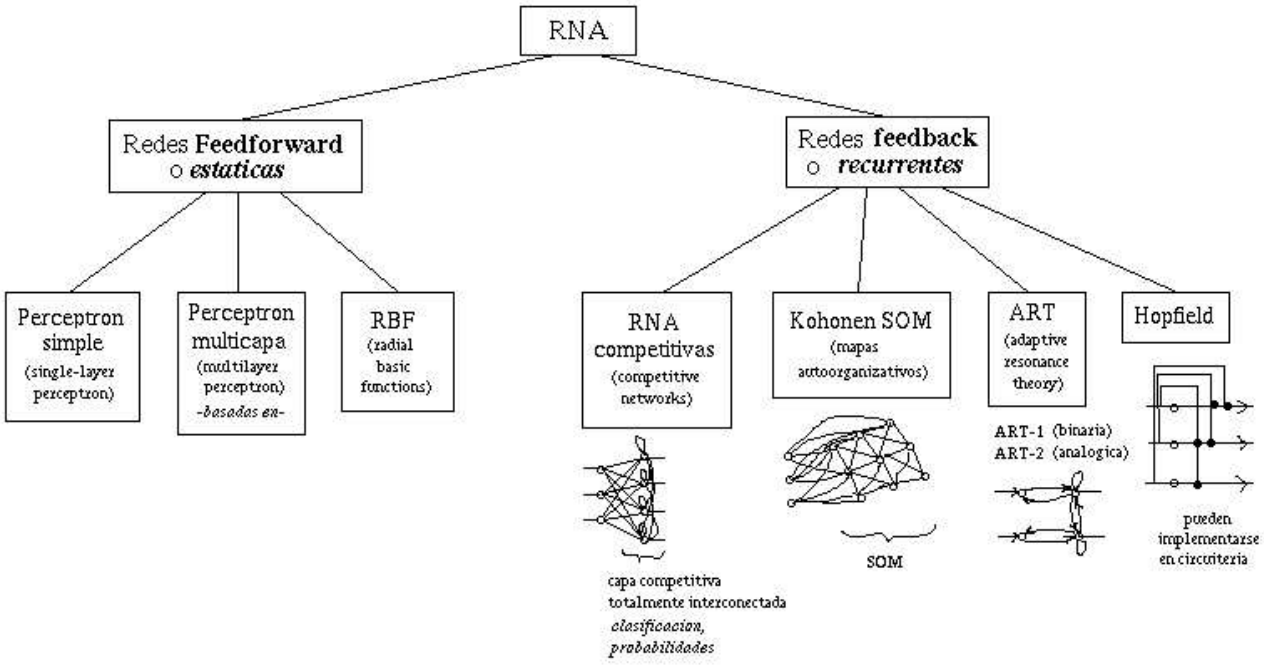


Figura 1.3: Clasificación de las RNA.

Las **redes feedforward** se caracterizan por que las interconexiones de sus PE son siempre *unidireccionales*, al contrario que en las **redes feedback**, en las que hay enlaces de capas posteriores hacia capas anteriores.

Las **redes feedforward** se denominan también **estáticas** porque con una entrada determinada producen una única salida, no se comportan como memorias, esto es, la respuesta no es función de estados previos. Por esta razón se las compara en ocasiones con *circuitos combinacionales*. Por su parte, las **redes feedback** se llaman también **recurrentes** o **dinámicas** por la razón contraria: pueden tener bucles y su salida resulta de una evolución a través de una serie de estados tras la presentación de la entrada. Se las compara, pues, con *circuitos secuenciales* (con biestables y capacidad de memoria).

Existen *algoritmos de aprendizaje* específicos para cada tipo de RNA, y también los hay aplicables a los diferentes tipos. Veremos una taxonomía en la sección siguiente.

1.6. Algoritmos de aprendizaje. Clasificación

Los principales aspectos a considerar en un proceso de aprendizaje son:

- ¿Cuál es el modelo del problema?

- ¿Cuál es el dominio de trabajo? (sus restricciones,...)
- ¿Qué información está disponible?
- Entender cómo se actualizan los pesos, qué son y cómo funcionan las reglas distintas de aprendizaje (algoritmos) –fundamental–.
- Capacidad (número de patrones que puede almacenar la red).
- Complejidad del conjunto de entrenamiento (determinar el número de patrones necesario para que la red generalice bien[12]).
- Complejidad computacional (tiempo requerido por un algoritmo de aprendizaje determinado)[13].

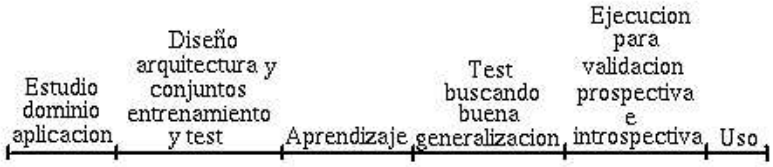


Figura 1.4: Ciclo de vida de una RNA.

Como se puede observar en la tabla 1.6 (página 16), existen 4 tipos básicos de reglas de aprendizaje:

1. ***Por corrección de error***.
2. ***De Boltzmann***.
3. ***De Hebb***.
4. ***Aprendizaje competitivo***.

1.7. Reglas de aprendizaje

Son el conjunto de "normas" que marcan cómo se van modificando los pesos a través de los diferentes pasos del entrenamiento.

[12]Si los patrones son escasos en número, la red puede aprender pero no generalizará.
[13]Existen *modificaciones rápidas* de algunos de los principales algoritmos.

PARADIGMA	REGLA APRENDIZAJE	ARQUITECTURA	ALGORITMO APRENDIZAJE	DOMINIO APLICACIÓN
Supervisado	*Corrección del error*	Perceptrón simple, multicapa	Algoritmo del perceptrón, backpropagation, adaline, madaline	Clasificación de patrones, aproximación de funciones, predicción (señales), control (centrales, pacientes).
	Boltzmann	Recurrentes	Algoritmo de Boltzmann	Clasificación de patrones.
	Hebbian	Multicapa feedforward	Algoritmo de análisis discriminal	Análisis de datos, clasificación de patrones.
	Competitivas	Competitivas	Cuantificación de vectores (LVQ)	Clasficación, categorización, definición de clases; compresión de datos.
No Supervisado	*Corrección del error*	Multicapa	Proyecciones de Shannon	Análisis de datos.
	Hebbian	Feedforward, competitivas	Análisis de componente principal	Análisis de datos, compresión.
		Hopfield	Aprendizaje memorias asociativas	Memorias
	Competitivas	Competitivas	Cuantificación de vectores (LVQ)	Categorías
		SOM	Mapas autoorganizativos	
		ART	Algoritmos ART-1 y ART-2	
Híbridos	*Corrección del error*	RBF	Algoritmo RBF	Clasificación de patrones, aproximación de funciones, predicción, control.

Cuadro 1.2: Taxonomía de los algoritmos de aprendizaje.

1.7.1. Regla de Hebb

Se basa en el siguiente principio:

> *"Una sinapsis incrementa su eficacia si las dos neuronas interconectadas por ella tienden a estar activas o inactivas al mismo tiempo. En caso contrario, se intenta decrementar o atenuar esa conexión."*

El "incremento de la eficacia de una conexión" se traduce en RNA en el aumento del peso asociado a esa conexión entre dos PE concretos.

El **esquema de aprendizaje de Hebb** es, pues, un tipo de aprendizaje supervisado que se rige por:

$$\Delta w_{ij} = \mu \times \Delta_i(t) \times O_j(t)$$

donde

Δw_{ij}	es el peso asociado a la conexión entre los PE i y j
μ	es la ***tasa de aprendizaje***
$\Delta_i(t)$	es la activación de i en t
$O_j(t)$	es la señal que emite j en t

reforzando así los pesos que habitualmente están activos al mismo tiempo.

Puede verse un ejemplo de la aplicación de esta regla en las anotaciones del tema, aunque a efectos prácticos no es muy útil; como el perceptrón (que veremos en la sección siguiente), divide el espacio de entradas (dominio de los patrones de entrenamiento) en regiones buscando la dirección de máxima variación de la población. Por ello, se dice que esta regla presenta una **orientación selectiva**.

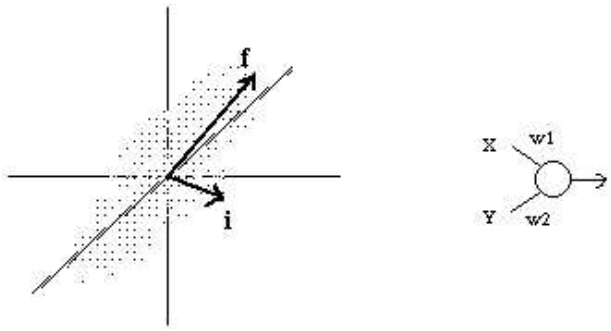


Figura 1.5: *Orientación selectiva* de la regla de aprendizaje de Hebb.

1.7.2. Regla de Boltzmann

Las redes sobre las que se aplica este tipo de aprendizaje, el aprendizaje de **Boltzmann**, son redes recurrentes simétricas con elementos de proceso binarios[14]:

$$\begin{aligned} \text{simétricas} &\Rightarrow w_{ij} = w_{ji} \\ \text{PE binarios} &\Rightarrow PE \in \{\pm 1\} \end{aligned}$$

Estas redes no tienen distinción entre capa de entrada/oculta/salida como tales, sino que sus PE pertenecen a uno de dos posibles conjuntos: **conjunto de elementos de procesado *visibles***, que interaccionan con el entorno (reciben información del exterior) o **conjunto de elementos de procesado *no visibles* u *ocultos***, que no lo hacen[15].

Los PE usan como función de activación la *distribución de Boltzmann*, que genera salidas ± 1 y está basada en la Teoría de la Información (concretamente en el concepto de entropía) y la termodinámica. Los cambios en los pesos se realizan:

$$\Delta w_{ij} = \mu \cdot (\bar{\rho_{ij}} - \rho_{ij})$$

donde

Δw_{ij}	variación del peso w_{ij}
μ	tasa de aprendizaje
$\bar{\rho_{ij}}$	correlación entre el estado de la unidad i y el de la unidad j cuando la RN de Boltzmann está en *modo visible* (le llegan datos del exterior)
ρ_{ij}	correlación entre los estados de i y j cuando la red está en modo *no visible* (opera libremente, sin datos del exterior)

Se puede considerar esta regla un caso particular del aprendizaje por corrección de error.

1.7.3. Regla del Perceptrón

La **regla del perceptrón** fue propuesta por Rosenblatt en un intento por mejorar la regla anterior. Usa una *función de activación* lineal y una *función de transferencia* tipo umbral.

La arquitectura de red sobre la que se aplica consiste en una capa de entrada, una capa asociativa (oculta) y una capa de respuesta o capa de salida. La capa de entrada y la asociativa se conectan totalmente con pesos que permanecen fijos desde el principio sin ser modificados. Los pesos que varían, y por tanto que realmente encarnan el proceso de aprendizaje, son los que interconectan la capa asociativa y la de salida.

[14] Las redes de Hopfield son una generalización de este caso.

[15] En el caso de una red Hopfield, todos los elementos son visibles.

El modo de comportamiento de esta regla es sencillo: se presenta un patrón de entrada p y para cada PE de la capa de respuesta se computa la salida (los O_i tomarán, pues, valores 0 ó 1), que será correcta o no (es supervisado).

El ajuste de los pesos se realiza de la siguiente manera:

- √ Si O_i es *correcta*, no se modifica ningún peso conectado a ese PE.
- √ Si O_i *no* es *correcta*, se modifican los pesos conectados a ese PE de acuerdo con la expresión:

$$\Delta w_{ij} = \mu_i \times (P_{ip} - O_{ip}) \times O_{jp}$$

donde

P_{ip}	salida deseada
O_{ip}	salida obtenida
O_{jp}	salida del PE de la capa asociativa
$(P_i - O_i)$	error

para todo elemento de la capa asociativa conectado con ese PE. Entonces:

$$w_{ij}(n) = w_{ij}(a) + \Delta w_{ij}$$

donde

$w_{ij}(n)$	nuevos pesos
$w_{ij}(a)$	pesos anteriores
Δw_{ij}	incremento

Pseudocódigo

En realidad, el algoritmo que se aplica en la práctica es ligeramente diferente:

```
para (cada patron en el conjunto de entrenamiento) hacer
  aplicar patron al perceptron
  obtener salida
  si (respuesta correcta) y (respuesta=+1) entonces
    nuevo_vector_pesos=vector_pesos_antiguo+vector_patron_entrada
  sino si (respuesta correcta) y (respuesta=-1) entonces
    nuevo_vector_pesos=vector_pesos_antiguo-vector_patron_entrada
  sino si (respuesta no correcta) y (respuesta=+1) entonces
    nuevo_vector_pesos=vector_pesos_antiguo-vector_patron_entrada
  sino si (respuesta no correcta) y (respuesta=-1) entonces
    nuevo_vector_pesos=vector_pesos_antiguo+vector_patron_entrada
fin-para
```

Los dos primeros condicionantes representan el *refuerzo en acierto*, mientras que los dos últimos representan la *penalización en fallo*.

Puede verse un ejemplo de la aplicación de esta regla en las anotaciones.

1.7.4. Regla Delta Generalizada - Backpropagation (Retropropagación del error)

Nos encontramos en esta ocasión con un aprendizaje supervisado, que necesita consecuentemente *patron_entrada+salida_deseada* y donde el error se define *salida_obtenida-salida_deseada*, del cual se obtiene una fórmula para *propagar el error a capas anteriores.*

Hasta la aparición de esta regla, el mundo conexionista había caído en desgracia condenado por los artículos de Marvin Minsky, ya que el perceptrón no presentaba la posibilidad de propagar el error a capas ocultas.

El padre de esta regla, controvertidamente, parece ser Rumelhart (1983). La modelización matemática de la misma corrió a cargo de Parker Le Cun (1985).

La arquitectura de red sobre la que se aplica **backpropagation** se corresponde con un modelo feedforward de interconexión completa.

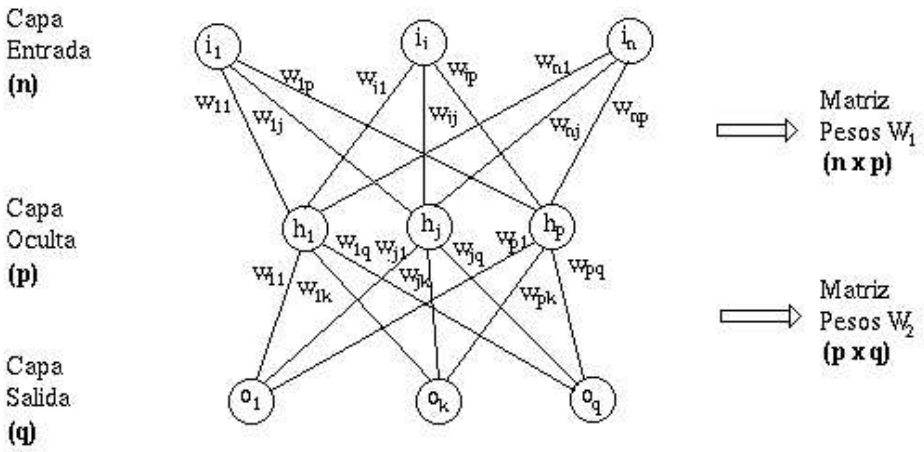


Figura 1.6: Red tipo backpropagation de 3 capas (caso más sencillo).

Parámetros

Los principales parámetros de la **regla delta generalizada** son:

- La **tasa de aprendizaje** μ, con valores típicos en el intervalo $[0'01 \dots 1]$.
- El **momento** Θ.
- Los valores iniciales de los pesos sinápticos, que suelen ser aleatorios y caer en el intervalo $[0 \dots 1]$ para evitar que la red se sature[16].

[16]Se dice que una RNA está *saturada* cuando el valor de alguno (o varios) de los pesos asociados a las interconexiones de sus PE resulta exageradamente elevado (en valor absoluto).

- Número de elementos de la(s) capa(s) oculta(s). Así como $\boldsymbol{n}$ y $\boldsymbol{q}$ son función del problema, $\boldsymbol{p}$ es variable; suele probarse con $\frac{n+q}{2}$. Este número influye directamente en la capacidad de generalización de la red, pues en general:
  - si $\boldsymbol{p}$ es *pequeño*, se necesitan más casos para entrenar y que estos sean realmente representativos
  - si $\boldsymbol{p}$ es *grande*, basta con un número de casos menor, pero el entrenamiento es más caro computacionalmente y es más fácil que la red se vuelva una memoria y no generalice correctamente (no obstante, hay casos concretos en los que funciona bien, como por ejemplo en procesado de imágenes)

  También influye en la existencia de mínimos locales.
- La **función de activación** (ver figura 1.7.4, página 22).
- El **conjunto de entrenamiento**: *(n variables, q variables)* para obtener *(salida producida-deseada)*, $o_s - d_s$.

Una RNA se puede interpretar como un método de ajuste paramétrico (estadístico).

Algoritmo

Sigue los pasos:

1. Asignar valores aleatorios a los pesos (matrices W_1 y W_2), habitualmente entre $[0 \dots 1]$ o $[-1 \dots 1]$. Pueden aplicarse umbrales para cada PE.
2. Pasos progresivos (tantos como capas):

   a) Computar activaciones de la capa oculta:

   $$O_h = F\left(I \times W_1\right)$$

   donde

O_h	vector salida capa oculta
F	función activación
I	vector entrada
W_1	matriz de pesos entre la capa de entrada y la capa oculta

   b) Computar activación de la capa de salida:

   $$O_s = F\left(O_h \times W_2\right)$$

   donde

O_s	vector salida capa salida
F	función activación
O_h	vector salida capa oculta
W_2	matriz de pesos entre la capa oculta y la capa de salida

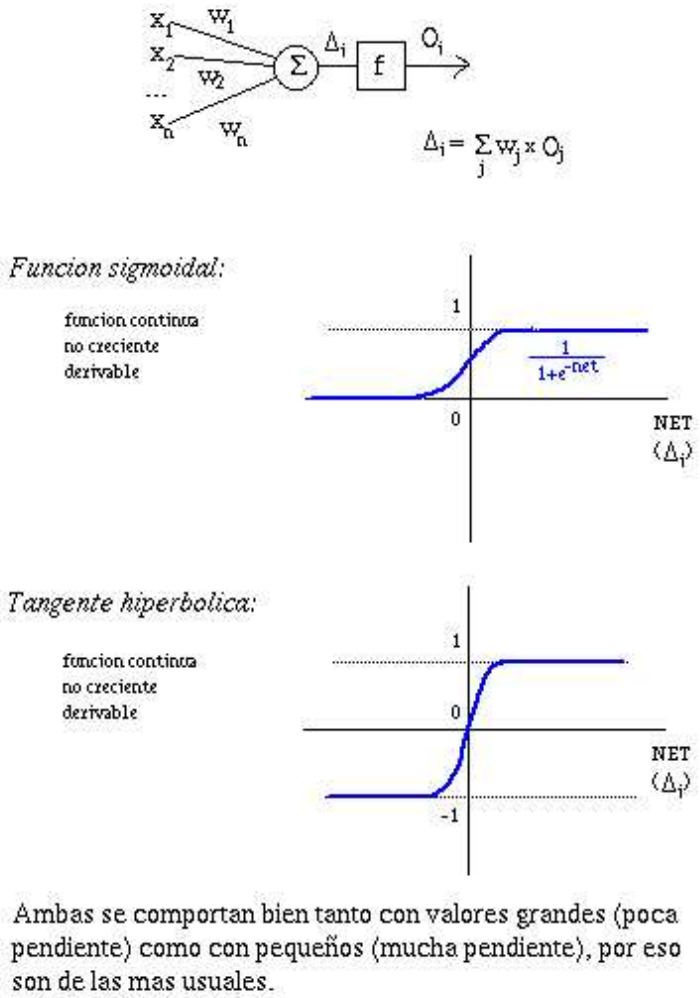


Figura 1.7: Función de activación (ejemplos).

3. Pasos regresivos (tantos como capas):

   a) Computar error en la capa de salida[17]:

$$e_s = O_s \times (1 - O_s) \times (d_s - O_s)$$

   donde

e_s	vector de errores capa salida
O_s	vector salida, $O_s \in [0 \dots 1]$
d_s	vector salida deseada
$O_s \times (1 - O_s)$	factor
$(d_s - O_s)$	error

   b) Computar error en la capa oculta:

$$e_h = O_h \times (1 - O_h) \times W_2 \times e_s$$

   donde

e_h	vector de errores capa oculta
O_h	vector salida capa oculta
W_2	matriz de pesos entre la capa oculta y la capa de salida
e_s	vector de errores capa salida

   c) Ajustar los pesos entre la capa oculta y la capa de salida (W_2):

$$W_2(t) = W_2(t-1) + \Delta W_2(t)$$

   con

$$\Delta W_2(t) = \mu \times O_h \times e_s + \Theta \times \Delta W_2(t-1)$$

   donde

μ	tasa de aprendizaje
O_h	vector salida capa oculta
e_s	vector de errores capa salida
Θ	momento

   d) Ajustar los pesos entre la capa de entrada y la capa oculta (W_1):

$$W_1(t) = W_1(t-1) + \Delta W_1(t)$$

   con

$$\Delta W_1(t) = \mu \times i \times e_h + \Theta \times \Delta W_1(t-1)$$

   donde

μ	tasa de aprendizaje
i	vector de entrada
e_h	vector errores capa oculta
Θ	momento

[17] Como es habitual, *error=salida deseada-salida obtenida.*

¿Cómo seleccionar μ?

Hay diferentes cosas que pueden tenerse en cuenta a la hora de seleccionar el parámetro *velocidad de aprendizaje* μ:

- $\surd$ Harry & Eatom recomiendan la realización de estudios previos de los datos en busca de clústeres. En la mayoría de los casos, puede llegarse a una conclusión similar a:

$$\mu = \frac{1'5}{\sqrt{{N_1}^2 + {N_2}^2 + \ldots + {N_n}^2}} \qquad N_i \equiv \text{ patrones de cada clúster de datos}$$

  Haciendo algo similar con el *momento* Θ eligen $\Theta = 0'9$ o menor.

- $\surd$ Puede realizarse un decremento progresivo según:

$$\mu(t) = \frac{\mu(0)}{1 + \frac{t}{r}} \qquad r \equiv \text{ ajuste, acelerador (parámetro exterior)}$$

- $\surd$ Puede haber una μ hasta por cada PE particular.

- $\surd$ Silva y Almeida sugieren que la *velocidad de aprendizaje* puede variar dependiendo de si el error aumenta o disminuye, teniendo en cuenta la evolución que ha seguido en los últimos ciclos (además de ser distinta para cada PE, también).

- $\surd$ Pueden usarse diferentes μ según la etapa del proceso de aprendizaje en la que nos encontremos.

- $\surd$ Pueden utilizarse para determinarla algoritmos genéticos, funciones difusas...

1.8. Estimación del *perfomance* de un Sistema de Aprendizaje

El objetivo de una RNA, como el de todo *sistema de aprendizaje*, sobre todo de los **socráticos** (aprendizaje a partir de ejemplos), es conseguir buenos resultados con casos no usados para el entrenamiento[18], y poder de este modo enfrentar con cierta seguridad labores de predicción y clasificación.

Una medida común de dicha "seguridad" es, por ejemplo:

$$\text{RATIO DE ERROR} = \frac{\text{nº de errores}}{\text{nº de casos}} \qquad \text{(tanto por 1)}$$

El problema del **ratio de error** es que no representa medida *verdadera* en cuanto que para que lo fuese tendría que estar en función de toda la población (que, en general,

[18] Capacidad de generalización.

tenderá a ser infinita), de modo que lo que realmente obtenemos no es más que una *estimación*, que puede ser optimista o pesimista. A raíz de esto se definen otras medidas, como por ejemplo el **true error rate** (*ratio de error verdadero*), que se define como *el ratio de error de un sistema de clasificación sobre un gran número de casos que convergen en el límite a la distribución de la población actual.*

Algo que también es conveniente a la hora de enfrentar el *diagnóstico*, es diferenciar claramente el tipo de error:

Falsos negativos Son los errores más serios y problemáticos; generalmente querremos que sean lo más raros que nos sea posible. Se trataría, por ejemplo, de un sistema que nos dice que un paciente está sano cuando realmente está enfermo.

Falsos positivos Aunque no dejan de ser errores, son menos problemáticos y siempre mejores que los anteriores. Es el caso del enfermo sano al que se le cree enfermo.

Otra medida que nos da mucha más información es la **confussion matrix**, que nos permite definir y caracterizar distintos tipos de errores. Por ejemplo:

	TRUE CLASS		
PREDICTED CLASS	**C-1**	**C-2**	**C-3**
C-1	50	0	0
C-2	0	48	5
C-3	0	2	45

Cuadro 1.3: Ejemplo de *confussion matrix.*

Obsérvese que en la diagonal obtenemos las clasificaciones correctas de nuestro sistema, y recuérdese que en esta sección trabajamos siempre sobre el conjunto de test.

	CLASE POSITIVA (C+)	CLASE NEGATIVA (C–)
PREDICCION POSITIVA (R+)	VERDADEROS POSITIVOS (TP)	FALSOS POSITIVOS (FP)
PREDICCION NEGATIVA (R–)	FALSOS NEGATIVOS (FN)	VERDADEROS POSITIVOS (TN)

Cuadro 1.4: Posibilidades en una predicción.

$$\textbf{sensitivity} = \frac{TP}{C+} \qquad \textbf{specifity} = \frac{TN}{C-}$$

$$\textbf{predictive\_value}(+) = \frac{TP}{R-} \qquad \textbf{predictive\_value}(-) = \frac{TN}{R+}$$

$$\textbf{accuracy}(\textbf{precission}) = \frac{TP + TN}{(C+) + (C-)}$$

$$\textbf{error} = 1 - \text{accuracy}$$

Si tenemos un sistema con *alta sensitividad* para el diagnóstico de una enfermedad, por ejemplo, entonces si el diagnóstico es positivo, es muy probable que sea cierto. Claro que si el mismo sistema tiene al mismo tiempo *baja especifidad*, significaría que a los individuos sanos ¡también les está diagnosticando enfermedad!

A los diferentes tipos de errores se les puede asociar un **peso** o **coste**. En este caso, además de tener la matriz de errores tendríamo una *matriz de costes:*

	TRUE CLASS		
PREDICTED CLASS	**C-1**	**C-2**	**C-3**
C-1	0	1	1
C-2	2	0	1
C-3	5	3	0

Cuadro 1.5: Ejemplo de *matriz de costes*.

De nuevo, la diagonal tiene significado: en este caso, que acertar no tiene coste alguno, como es lógico. En el ejemplo anterior, el mayor coste está asociado a fallar en la predicción de la clase 1 estableciéndola en su lugar como de clase 3.

Los costes pueden ser tanto positivos como negativos (según la interpretación del problema). Ambas matrices (la de errores y la de costes), se utilizarían conjuntamente para obtener los índices de seguridad en la predicción con el sistema de aprendizaje en cuestión:

$$\textbf{coste total} = \sum_{i=1}^{n} \sum_{j=1}^{n} E_{ij} \times C_{ij}$$

donde

$$\begin{array}{lcll} E_{ij} & \equiv & \text{nº de errores} & \text{(de la matriz de errores)} \\ C_{ij} & \equiv & \text{coste} & \text{(de la matriz de costes)} \end{array}$$

Este **coste total** podría asignarse a cada par *(arquitectura conexionista, conjunto de entrenamiento)* para así tener un criterio de decisión y/o elección. También se puede trabajar con el **coste medio**:

$$\textbf{coste medio de decision} = \frac{\text{coste total}}{\text{total casos}}$$

Otras medidas:

$$\textbf{riesgos} = \sum_{i=1}^{n}\sum_{j=1}^{n} E_{ij} \times R_{ij}$$

donde R_{ij} son los costes;

$$\textbf{utilidad} = \sum_{i=1}^{n}\sum_{j=1}^{n} E_{ij}U(R_{ij})$$

donde U es una función que modifica R_{ij}.

Por último, teniendo E, *ratio de error* sobre un conjunto de test independiente y aleatorio, también es útil:

$$\textbf{desviación estándar} = \sqrt{\frac{E(1-E)}{n}}$$

Capítulo 2

Visión general de algunos tipos de RNA

2.1. Memorias asociativas

2.1.1. Conceptos

Muchas redes que siguen el esquema de **memorias asociativas** utilizan el concepto de **distancia de Hamming**, que veremos a continuación, para la clasificación de patrones.

Situados en el espacio *n-dimensional*, el **cubo de hamming n-dimensional** tiene 2^n puntos equidistantes del origen del espacio euclídeo. La figura 2.1 es la representación del cubo en el caso del espacio tridimensional.

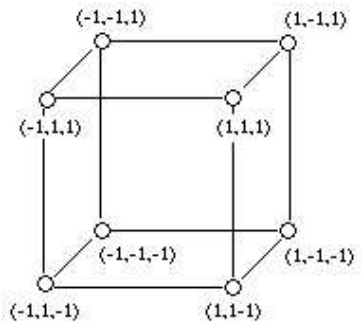


Figura 2.1: Cubo de Hamming 3D.

La distancia entre dos puntos en el espacio n-dimensional se calcula:

$$x = \{x_1, x_2, \ldots x_n\}$$
$$y = \{y_1, y_2, \ldots y_n\}$$

$$\mathbf{d} = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots (x_n - y_n)^2}$$

como $x_i, y_i \in \{\pm 1\}$, se tiene que $(x_i - y_i)^2 \in \{0, 4\}$ y además

$$\begin{array}{l} (x_i - y_i) = 0 \quad \text{si} \quad x_i = y_i \\ (x_i - y_i) = 4 \quad \text{si} \quad x_i \neq y_i \end{array}$$

de modo que podemos escribir

$$\mathbf{d} = \sqrt{4 \times \text{nº de componentes distintas}}$$

A ese *número de componentes distintas* es a lo que llamamos precisamente **distancia de Hamming** (**h**), así que:

$$\mathbf{d} = 2\sqrt{\mathbf{h}} \qquad \Rightarrow \qquad \mathbf{h} = \frac{d^2}{4}$$

2.1.2. Tipos

Memorias heteroasociativas En este tipo de redes se establece una correspondencia

$$\begin{array}{ccc} \Omega(X) & \longrightarrow & Y \\ \Omega(x_i) & \rightsquigarrow & y_i \end{array}$$

Si se proporciona un x_i que no está en la memoria, el y_i que se obtiene es el correspondiente al x_j almacenado de distancia hamming menor con x_i.

Memorias interpoladoras Realizan lo mismo que las anteriores, pero devuelven el y_j desplazado lo mismo que lo está el x_i respecto al x_j.

Memorias autoasociativas Asocian cada x_i consigo mismo (en lugar de con otro par), de modo que al presentar un x_i desconocido es *reconstruido* en función de la mínima distancia hamming.

2.1.3. Trabajos de Bart Kosho

El conexionista Bart Kosho empieza a estudiar las memorias asociativas en torno a 1958. Trabaja con p pares de patrones (A_i, B_i) donde $A_i, B_i \in \{\pm 1\}$ y A_i, B_i tienen dimensiones diferentes (A_i dimensión m, B_i dimensión n).

La matriz de distancias hamming es, pues, $m \times n$. Este esquema conlleva una serie de limitaciones en especio que hacen que como máximo se puedan almacenar $S < mín(m, n)$ pares de patrones.

$$w = y_1 \times x_1^t + y_2 \times x_2^t + \ldots + y_L \times x_L^t$$

La matriz de pesos se obtiene:

$$M = \sum_{i=1}^{L} A_i^t \times B_i$$

donde L es el número de pares de patrones. Se cumple que:

$$F(A_i \times M) = B_i$$
$$F(B_i \times M^t) = A_i$$

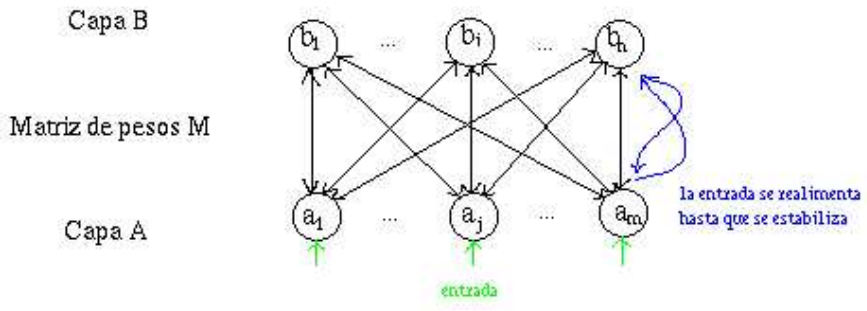


Figura 2.2: Arquitectura de una memoria asociativa.

Codificación de patrones

El proceso de "memorización" de patrones tiene lugar de la siguiente manera:

1. Se calcula la matriz de pesos M_1.
2. Si almacena todos los pares (sin desplazamiento) $\rightarrow$ FIN.
3. sino:
   - *a*) Se borran de M_1 los pares que son mal clasificados.
   - *b*) Se calcula M_2 con los pares mal clasificados.
   - *c*) Si con M_1, M_2 tenemos todos los patrones bien clasificados $\rightarrow$ FIN.
   - *d*) sino REPETIR.

En funcionamiento

El proceso es el siguiente:

1. Se calcula la energía asociada a cada uno de los i pares (A, Y_i) (para un patrón A, hay un Y_i por cada M_i). Dicha energía se calcula:

   $$E = -A \times M_i \times Y_i^t$$

   donde A tiene dimensión $1 \times m$, M_i es $m \times n$ y Y_i^t $n \times 1$, de modo que se obtiene un número.

2. Nos quedamos con el Y_i cuya energía esté más próxima a la energía de la matriz M *ortogonal*, que es:

   $$E^* = -n \times m$$

2.1.4. Redes de Hopfield

Hopfield comienza a trabajar en el modelo que después llevaría su nombre alrededor de 1985. Las características generales de las redes Hopfield son:

- Una única capa (construida y posteriormente entrenada).
- Interconexión completa, retroalimentación.
- Respuesta dinámica: la red progresa con la entrada hasta alcanzar un estado estable (hasta que la salida deja de cambiar).
- Aplicaciones: reconocimiento de patrones, almacenados en sus pesos sinápticos. El número de patrones que puede almacenar es:

$$\text{nº patrones} \leq \frac{N}{4 \times \log_2(N)}$$

con N el número de EP de la RNA.

Los más conservadores afirman que en realidad el límite es:

$$\text{nº patrones} < 0'15 \times N$$

Mientras que los más optimistas sostienen que pueden llegar a conseguirse:

$$\text{nº patrones} \leq N$$

El esquema supervisado (con entrada y salida deseada), aunque es viable, no suele utilizarse con este tipo de red; es más corriente emplear conjuntos de entrenamiento donde sólo se presenten entradas, y asumir como salidas deseadas los propios patrones de entrada. Así, se presentan más tarde en funcionamiento patrones incompletos o parcialmente erróneos para su reconstrucción.

La función de transferencia es la siguiente:

$$\text{OUT}\,(t) = \begin{cases} 1 & \text{si} \quad \text{NET} > \text{UMBRAL}_i \\ 0 & \text{si} \quad \text{NET} < \text{UMBRAL}_i \\ \text{OUT}\,(t-1) & \text{si} \quad \text{NET} = \text{UMBRAL}_i \end{cases}$$

donde

$$\text{NET} = \sum_{i=1}^{n} O_i \times w_{ij}$$

De modo que una red Hopfield se puede *resumir* en una simple matriz $n \times n$.

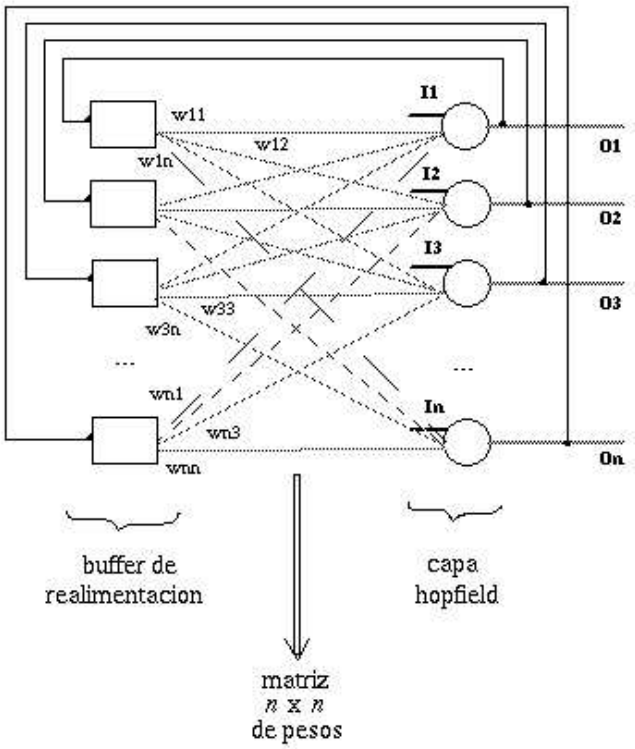


Figura 2.3: Arquitectura de una red Hopfield.

Teorema de Estabilidad

Una RNA de Hopfield es ***estable*** *si la matriz de pesos que le corresponde es simétrica y con ceros en la diagonal principal, es decir, siempre que*

$$\begin{aligned} w_{ij} &= w_{ji} \qquad i \neq j \\ w_{ij} &= 0 \qquad i = j \end{aligned}$$

Que sea **estable** significa que con cualquier entrada, la salida se estabiliza en un número finito de "ciclos" o iteraciones de la red. Este teorema establece una *condición suficiente* pero *no necesaria*.

Construcción de la matriz de pesos

De acuerdo con lo anterior, podemos asegurarnos la estabilidad de nuestra red Hopfield construyendo cada elemento de la matriz de pesos de la siguiente manera:

$$w_{ij} = \begin{cases} \sum_{i=0}^{N-1} x_i^S \times x_j^S & i \neq j \\ 0 & i = j \end{cases}$$

donde

$$\begin{array}{lcll} N & & \text{número de patrones} & \\ & \equiv & & \\ x_i^S & \equiv & \text{elemento } i \text{ del patrón } S, & x_i^S \in \{\pm 1\} \end{array}$$

Eliminar o dar de alta nuevos patrones es tan sencillo como modificar los pesos sumando o restando el valor del patrón a las posiciones correspondientes de la matriz.

Aplicaciones

Algunos campos de aplicación de las redes de Hopfield son:

- Reconocimiento de voz, escritura,...
- Medicina.
- Gestión de sistemas, routing.

2.2. Redes Competitivas

La principal característica de las redes competitivas y su comportamiento en aprendizaje es que los PE de su capa de salida *compiten* entre sí de tal forma que el que resulta ganador se activa e identifica la clase a la que pertenece el patrón de entrada (se definen tantos PE de salida como clases o clústeres quieran obtenerse), habiendo únicamente un PE que se activa en cada ciclo.

Esta es una gran aproximación al comportamiento de las redes de neuronas biológicas, pues está demostrado que en la zona del córtex cerebral existen excitaciones e inhibiciones entre neuronas vecinas que reproducen un comportamiento similar al que acabamos de esbozar.

Este tipo de reglas (y sus algoritmos de aprendizaje) se orientan, pues, fundamentalmente a la clasificación o clusterización de patrones similares del conjunto de entrenamiento en función de las correlaciones entre los mismos.

El modelo de RNA competitiva más simple (una sola capa, interconexión completa) puede verse en la figura 2.2.

El PE que gana (el que tiene mayor entrada) se activa y además inhibe a sus vecinos:

$$w_{i^*} \times x \geq w_i \times x \quad \forall i \quad \text{donde } i^* \text{ es el ganador}$$

$$\Delta w_{ij} \begin{cases} \mu \left({x_j}^n - w_{ij} \right) & i = i^* \\ 0 & i \neq i^* \end{cases}$$

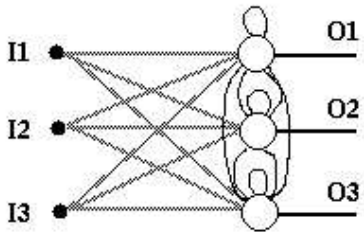
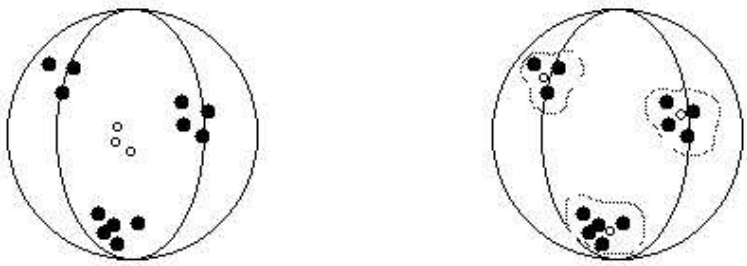


Figura 2.4: Arquitectura de una RNA competitiva sencilla.

Como se puede ver, el ganador intenta desplazar su peso hacia la coordenada de entrada (si $x > w$, $w(t-1) \rightarrow x$; si $x < w$, ídem); se acerca el vector de pesos al vector media de las entradas que clasifica su PE asociado. En otras palabras, mueve el patrón almacenado en la unidad ganadora (pesos) hacia el patrón de entrada.



(a) Antes del entrenamiento.

(b) Después del entrenamiento.

Figura 2.5: Distribución de pesos en aprendizaje competitivo.

En estas redes el aprendizaje nunca finaliza, en realidad, suele irse decrementando la tasa de aprendizaje y detener el proceso cuando llega a valor cero, o bien cuando la red y sus poblaciones se estabilizan. Suelen utilizarse en aplicaciones tales como procesado de voz, por ejemplo.

2.2.1. Interacción lateral

Muchas de las ideas que estamos viendo en redes competitivas surgen de similitudes con las RNB derivadas de estudios del *neocórtex* y el *córtex cerebral*, cuyas características son:

- capas bidimensionales
- neuronas activas producen excitación en las más próximas
- neuronas activas producen inhibición en las más alejadas

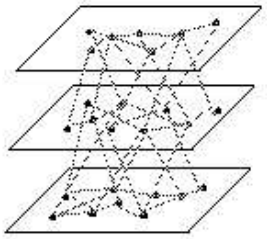


Figura 2.6: Esquema del neocórtex.

- las relaciones entre neuronas se debilitan con la distancia

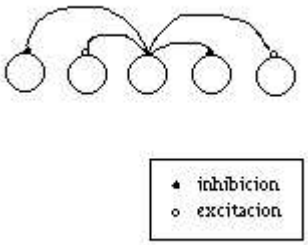


Figura 2.7: Representación excitaciones/inhibiciones neuronales.

La función que mejor modela este comportamiento es el conocido **sombrero mexicano**:

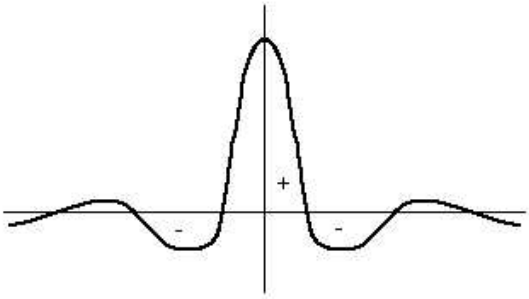


Figura 2.8: Sombrero mexicano.

2.3. Mapas Autoorganizativos de Kohonen (SOM)

Este tipo de arquitecturas presenta una propiedad característica que es el **mantenimiento de la topología**. Surge la idea de nuevo de la estructura del *neocórtex* cerebral.

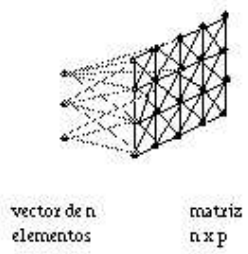


Figura 2.9: Arquitectura básica clásica de un mapa autoorganizativo.

En este contexto cambia la notación y ahora w_{ij} es el vector de n elementos asociado con la unidad de la posición (i, j) de la matriz $x \times p$.

Cada PE calcula la distancia euclídea[1] entre el vector de entrada x y el vector w_{ij} de cada elemento de proceso. De nuevo, al igual que en los esquemas competitivos, ganará el más próximo. Hay arquitecturas en las que puede ganar más de un PE y en ellas surge el concepto de **vecindad**.

Algoritmo de funcionamiento

1. Inicializar los pesos sinápticos con valores aleatorios pequeños[2]. Se selecciona la tasa de aprendizaje, la vecindad (pueden solaparse).
2. Presentar un patrón y evaluar la salida.
3. Seleccionar el PE (c_i, c_j) con *salida mínima* (distancia mínima entre el vector de pesos y la entrada).
4. Actualización de pesos:

$$w_{ij}(t+1) = \begin{cases} w_{ij}(t) + \mu\,(x(t) - w_{ij}(t)) & (i,j) \in N_{(c_i,c_j)}(t) \\ w_{ij} & \text{en otro caso} \end{cases}$$

   donde

$$N_{(c_i,c_j)} \text{ son los vecinos de la unidad } (c_i, c_j) \text{ (ganador) en } t$$

5. Decrementar la tasa de aprendizaje μ (cada ciclo o cada x ciclos) y reducir el número de vecions, el tamaño de la vecindad[3].
6. Repetir del 2 al 5 con todos los patrones del conjunto de entrenamiento hasta un número predeterminado de iteraciones o algún criterio de convergencia.

Los campos de aplicación más típicos de los mapas autoorganizativos de Kohonen son el clústering, el análisis de datos multivariables, reconocimiento de voz, procesado de imagen, robótica, control, etc.

[1] La distancia Hamming es equivalente a la distancia euclídea cuando se trabaja con ceros y unos.

[2] Para no condicionar previamente la distribución del SOM.

[3] Los *vecinos* pueden (o suelen) empezar siendo la mitad o una tercera parte de los PE de la red e ir decreciendo según una función exponencial decreciente.

2.4. Radial Basic Functions

Las **redes RBF**, que ya mencionamos cuando hablamos del aprendizaje híbrido, son una clase especial de redes feedforward, con dos capas generalmente (entrada, oculta, salida).

Los PE de la capa oculta usan como función de activación una *función de base radial* (por ejemplo, la función de Gauss); por tanto, tenemos tantas funciones de base radial como PE tenga la capa oculta, con centro y longitud definidas por su vector de pesos asociado. La última capa realiza una interpolación, de modo que una red RBF lo que hace es buscar una función que se ajuste al espacio de patrones de entrada.

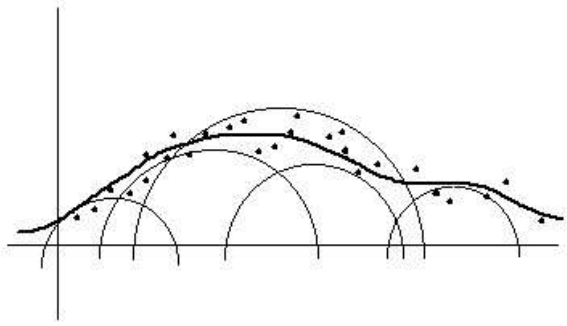


Figura 2.10: Resultado de una red RBF.

Su aprendizaje, como decimos, es *híbrido*, y su convergencia es muy rápida (más aún que la de redes como las backpropagation), aunque necesitan muchos elementos de procesado en la capa oculta para obtener una buena interpolación, lo que hace que sus requisitos computacionales sean elevados.

Su uso está empezando a extenderse sobre todo en predicción de señales.

2.5. Redes de contrapropagación

Sus creadores fueron Hecht y Nielsen, allá por el año 1987. Son unas redes de rápido entrenamiento que generalizan muy bien y son capaces de manejar información incompleta, parcialmente errónea, etc.

Su arquitectura es una "mezcla" de dos modelos:

- *Redes de Kohonen* (autoorganizativas), esquema que le otorga su capacidad de generalización y autoorganización.
- *Redes Grossberg* (codificadores de), que le aportan el soporte que presentan para información incompleta y/o errónea.

Su topología es la de una red de tres capas (entrada, oculta y de salida), con dos matrices de pesos. La capa competitiva es la que queda entre la entrada y la capa oculta, siendo la de Grossberg la capa oculta-salida.

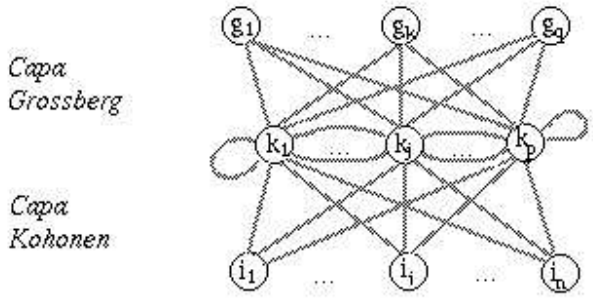


Figura 2.11: Arquitectura básica de una red CNP.

La capa autoorganizativa sigue el esquema general que ya vimos para este tipo de redes; la capa Grossberg puede funcionar con un esquema supervisado o no supervisado:

↪ *Modificación de pesos capa autoorganizativa.-* Se usa un **vector distancia** d para determinar el ganador de la capa competitiva:

$$d = \sum_{i=1}^{n} (A_i - B)^2$$

El ajuste de pesos responde a:

$$W(t) = W(t-1) - \Delta W$$

$$\Delta W = \mu \times h \times d$$

Por supuesto, podría tenerse en cuenta un término *momento*.

↪ *Modificación de pesos capa Grossberg.-* En el modelo básico, la capa Grossberg se actualiza sólo en las conexiones con el ganador de la capa oculta. Dicho ajuste se lleva a cabo según:

$$W(t) = W(t-1) + \Delta W$$

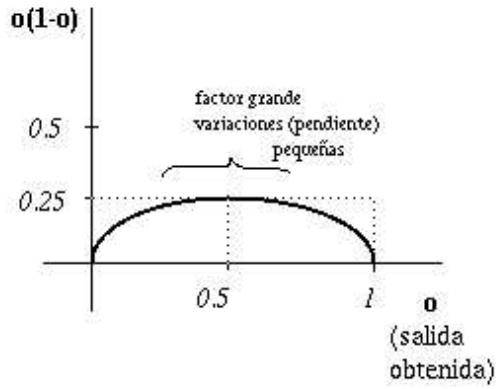
$$\Delta W = \mu \times h \times d$$

donde d ahora es:

$$d = o(1-o)(t-o)$$

con

o = vector salida
d = vector errores
t = vector salida deseada



El funcionamiento de estas redes es, como de costumbre, sencillo. Simplemente (aunque hay variantes) consiste en presentar un patrón, calcular las activaciones de la capa oculta, designar un ganador (el PE de mayor activación) y calcular la salida propagando señal desde la capa oculta poniendo a uno el PE ganador y los otros a cero.

Entre las ventajas de este tipo de red podemos contar su capacidad para ser entrenadas mediante un paradigma tanto supervisado como no supervisado, e incluso con una mezcla de ambos. Existen diseños en que se designan varios PE ganadores en la capa oculta, e incluso modelos en los que esta capa es bidimensional.

En cuanto a las aplicaciones, podemos citar, entre otras:

√ Clasificación de patrones (dado un patrón de entrada, la salida designa una de entre varias posibles clases), clasificación de señales (*ECG*, *PIC*,...).

√ Reconocimiento de caracteres, reconocimiento de voz.

√ Clustering (clasificación de patrones no supervisado), análisis de similitudes.

√ Compresión de datos, análisis de datos médicos.

√ Aproximación de funciones (la RNA funciona como una *caja negra*), modelización, estimación.

√ Predicción: dada una serie temporal de n muestras, predecir $n + 1$. Utililidad en toma de decisiones en negocios (bolsa, ciencia,...).

√ Optimización (obtención de una solución óptima a partir de una solución satisfactoria atendiendo a una serie de restricciones o reglas). Se busca maximizar/minimizar una función objetivo.

√ Memorias direccionables por contenido (información multimedia, sistemas de control).

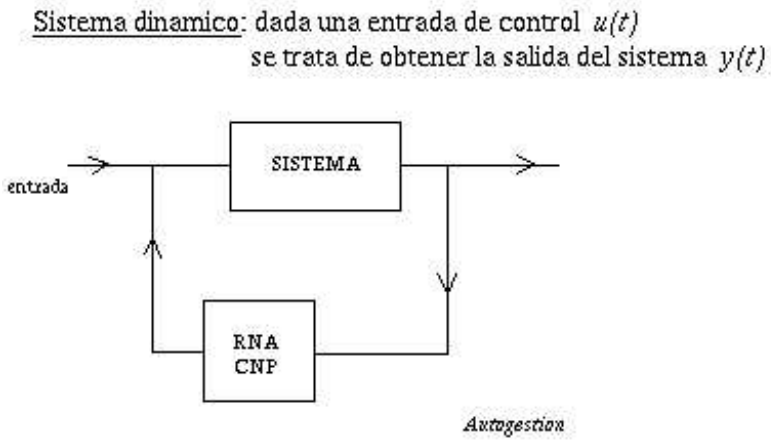


Figura 2.12: Sistema de control implementado con una RNA CNP.

Parte II

Modelos Avanzados de Redes de Neuronas Artificiales

Capítulo 3

Procesado Temporal

Una **serie temporal** es una secuencia ordenada de observaciones (normalmente el orden se refiere al *tiempo*) cuya característica fundamental es que dichas observaciones están *correlacionadas*[1].

Podemos encontrar dos *enfoques* en lo que a **series temporales** se refiere:

- El *objetivo* es el **control** del proceso (control industrial).
- El *objetivo* es la **predicción** (extracción de la función que subyace); es el más frecuente, puesto que se usa como apoyo a la toma de decisiones en campos tan variados como el económico, control industrial, medicina, metorología,...

Existen dos métodos principales de **predicción**:

- **Cualitativos**: se usan cuando no se tiene información del pasado, sino del modelo (por ejemplo, para hacer predicción de ventas de un nuevo producto). Reciben también el nombre de ***predicción basada en modelos***, y son por naturaleza inexactos.
- **Cuantitativos**: se usan cuando se dispone de información histórica (típicamente, una serie temporal). Serán en los que nos centraremos.

Una **serie temporal** tiene dos componentes básicas:

$$Y_i = D_i + A_i$$

1. **Determinística**, la que podremos predecir. Se compone a su vez de:

$$D_i = T_i + C_i + E_i + I_i$$

   a) ***Tendencia***, que aporta una idea general de si la serie es creciente, decreciente o estacionaria.

[1] Contrasta con lo que hemos visto hasta ahora en RNA: estamos acostumbrados a que el orden –de presentación de patrones, por ejemplo– no deba influir.

   b) ***Factor cíclico***, una componente de períodos grandes.

   c) ***Factor estacional***, una componente de períodos cortos.

   d) ***Factor irregular***, que no aleatorio: tiene unas causas puntuales que si se conociesen permitirían su predicción.

2. **Aleatoria**, que en el mejor de los casos, será *ruido blanco*, y determina el nivel de error que obtendremos en la predicción.

Consideraremos cuatro tipos de series temporales:

- **Estacionarias**: su media no aumenta, no tienen tendencia y la varianza de los puntos no cambia.
- **No estacionarias en la media**: son series con tendencia, pero las varianzas dentro de los ciclos estacionarios son iguales.
- **No estacionarias en la varianza**: son series con tendencia en las que la varianza se va haciendo más acusada en cada ciclo.
- **Caóticas**: las más complejas de predecir, se rigen por la *Teoría del Caos*, con ecuaciones de funciones recursivas con gran error asociado.

Las series reales normalmente no son estacionarias ni en media ni en varianza. Con RNA puede abordarse cualquier serie, pero es importante detectar con cuál tipo nos enfrentamos. La **Estadística** puede abordar los dos primeros tipos con métodos como el *Análisis de series temporales* o los *Métodos ARIMA*[2], desarrollados por Box-Jenkins (1976) y cuyo proceder se estructura de la siguiente forma:

1. Cálculo de estadísticos de la serie.
2. Comprobación de la estacionalidad o bien transformación de la serie para que lo sea y recálculo de estadísticos.
3. Identificar un modelo de datos para la serie temporal a partir de los estadísticos.
4. Validación (predicción en casos reales).
5. Vuelta a (1) si es necesario, o finalización del proceso.

Los principales problemas son la falta de un mecanismo, unas reglas fijas para el ajuste de los parámetros del modelo ARIMA (se usan gráficas de correlaciones entre los datos para *estimarlos*), lo que hace que se necesite personal experto, y ls aplicación reducida a los modelos estacionarios (los únicos en los que la Estadística funciona realmente bien) y los no estacionarios sobre los que puedan aplicarse transformaciones.

[2] *Autoregressive Integrative Moving Average*, un método modular –las series estacionarias se clasifican como AR, existen series ARMA, las ARIMA son las más complejas...–

3.1. Análisis de Series Temporales con RNA

Lo primero que debe hacerse es preparar el conjunto de entrenamiento: *visualizar* la serie (para intentar hallar pistas sobre qué tipo de sistema tenemos y escoger la RNA más adecuada), *preparar* y *seleccionar* los datos.

La ***preparación de los datos*** conlleva un análisis estadístico para ver con qué tipo de datos trabajamos (analizar valores erróneos –fuera del rango normal, "blancos"...–) y posteriormente un estudio de correlaciones entre las variables de entrada[3] (y de salida).

Tras ello filtraremos la entrada (datos) para eliminar en cierta medida el ruido (aunque sabemos que las RNA trabajan bien en presencia de ruido, son capaces de extraer igualmente el modelo). Se normalizarán los valores (dependiendo de la función de transferencia que se pretenda usar, al rango [0.,1] ó al [−1.,1]).

Hay que tener en cuenta el tipo de predicción que se pretende realizar:

$\surd$ de valores

$\surd$ de componentes (tendencia, variaciones de ciclo estacional, ...)

pues nos hará preparar los datos de una forma diferente en cada caso. También podría dividirse la serie temporal en sus componentes y usar una RNA para el tratamiento de cada una.

En la ***selección de datos*** se separa un conjunto para test y un conjunto para entrenamiento; en éste se busca equilibrar el número de ejemplos por el número de tipos de salida (aunque se tengan muchos más de un tipo que de otro). Normalmente, se toma la serie temporal y a partir de un determinado punto se determina que será para test (no se puede hacer como en otras RNA vistas en la primera parte y coger "trozos" aleatorios para cada tarea).

3.1.1. Problemas de series temporales abordables con RNA

Hay tres tipos principales de problemas de series temporales abordables con RNA:

* *Reconocimiento*:
$$\text{S.T} \xrightarrow{in} \text{RNA} \xrightarrow{out} \text{valor identificativo}$$

* *Producción*:
$$\text{valor} \xrightarrow{in} \text{RNA} \xrightarrow{out} \text{S.T.}$$

* *Asociación*:
$$\text{S.T} \xrightarrow{in} \text{RNA} \xrightarrow{out} \text{S.T}$$

[3]Cualquier RNA trabajará mejor cuanto más concreta sea la información que le suministremos.

Consideraremos la *predicción* un caso particular de la *asociación*. La *predicción* puede ser:

- ▷ *A corto plazo*: sin realimentación entre la entrada y la salida. La red predice n valores fijos a partir de cada entrada dada.
- ▷ *A largo plazo*: se presenta a la red una entrada y la salida de la misma se realimenta a la entrada de nuevo. Se repite el proceso hasta obtener el momento deseado.

Una red de predicción a corto plazo, realmente sólo predice una tendencia, por ello no funcionarán bien aplicadas a predicción a largo plazo (que es mucho más compleja).

3.2. Tipos de RNA aplicables a series temporales

Las redes para procesado temporal aparecieron ya con McCulloch y Pitts (1943), que intentaron desarrollar modelos de neurona y de red capaces de realizar procesados temporales. El uso de redes recurrentes se popularizó con Hopfield, que las aplicó al tratamiento de datos estáticos, y Minsky & Papert, que pusieron las ideas de cómo deberían funcionar los algoritmos de entrenamiento para tratar la componente temporal en redes.

Por ARQUITECTURA, sabemos que tenemos:

- Redes alimentadas hacia delante (*feedforward*), cuyas conexiones nunca van hacia la misma capa ni hacia capas anteriores. Se dice que tienen una ejecución *por capas*
- Redes recurrentes, en las que no hay la restricción anterior y es posible una interconexión total. No existe el concepto de capa porque todas las neuronas pueden ser entrada/salida o ambas a un tiempo. Se dice que tienen una ejecución *por instante de tiempo*.

Las redes recurrentes presentan la ventaja de poder tener más conexiones (que, al fin y al cabo, es donde se almacena la información) y poder crear ciclos (pudiendo, pues, generar un comportamiento en el tiempo). Sin embargo, tener más conexiones también implica un mayor tiempo de entrenamiento y mayores dificultades para la convergencia (menor estabilidad).

Modelos de RNA aplicables a ST

- Redes feedforward con algoritmo de entrenamiento backpropagation (lo más sencillo, válido sólo para series estacionarias).
- Redes feedforward con algoritmos de entrenamiento para series temporales (*recurrent backpropagation*).
- Redes feedforward con retardos (algoritmo *TDNN*[4]).

[4] *Time Delay Neural Network*, se caracteriza por tener varias conexiones entre dos mismas neuronas simulando una especie de buffer.

- Redes recurrentes con algoritmo *BPTT*[5].
- Redes parcialmente recurrentes (con una capa de contexto donde la salida de un instante es entrada en el siguiente).
- Redes recurrentes con algoritmo *RTRL*[6].

En todos los modelos a partir del primero, se busca implementar una *memoria* (contexto) de las activaciones anteriores, que matice lo que significan los datos de entrada en cada instante, de forma que la red tenga en cuenta un estado de activación interno/anterior además del dato actual.

3.2.1. Redes Feedforward

Ya conocemos su arquitectura y aprendizaje. Suelen usarse con series estacionarias, en predicción a corto plazo.

Entre las consideraciones que deben tomarse está, como ya hemos comentado, la preparación del conjunto de entrenamiento: se define para ello una **ventana temporal** (número de observaciones en el pasado que se van a tener en cuenta para predecir el futuro). Normalmente, para cualquier tipo de red y predicción a corto plazo, la *ventana temporal* suele ser pequeña (entre 2-7 valores, con frecuencia en torno a 3).

Los valores de la *ventana* son la entrada y el siguiente valor (futuro) será la salida deseada. Puede hacerse que la salida no sea uno, sino también varios valores.

En estos casos es interesante la utilización de varias series (a la entrada), relacionadas, claro, para la predicción de una sola (a la salida), porque el manejo de más información –correlaciones– puede ayudar a la red en su cometido.

3.2.2. Redes Temporales

El modelo de neurona para redes temporales difiere ligeramente del modelo que manejan las redes feedforward:

MODELO ESTÁTICO	MODELO TEMPORAL
$x_i = f(net_i)$ $net_i = \sum_j w_{ij}x_j$	$x_i(t+1) = f\left(net_i\left(t+1\right)\right)$ $net_i(t+1) = \sum_j w_{ij}x_j(t)$

Cuadro 3.1: Diferencias entre neuronas feedforward y neuronas temporales.

[5] *Backpropagation Through Time*, desenvuelve la red recurrente hacia delante hasta hacerla feedforward y entrenarla como tal.

[6] *Real Time Recurrence Learning.*

donde f es la función de activación. En las redes temporales no existe el concepto de *capa* y la activación es por *tiempo*.

En cuanto al error:

$$E_{total}(t_0, t_1) = \sum_{t_0+I}^{t_1} E(t)$$

donde t_0 es el instante inicial, t_1 el instante final y $t_0 + I$ el instante en que se empieza a obtener/generar salida (usualmente será $t_0 + 1$). $E(t)$ es el *error cuadrático*:

$$E(t) = \frac{1}{2} \sum_{i \in \text{OUT}} e_i(t)^2$$

donde $e_i(t)^2$ es el error para una única neurona concreta:

$$e_i(t) = \begin{cases} d_i(t) - x_i(t) & i \in \text{OUT} \\ 0 & i \notin \text{OUT} \end{cases}$$

donde el conjunto de salida (OUT) puede ser diferente en distintos momentos del tiempo (OUT(t)).

Algoritmos de gradiente en redes temporales

Por lotes (el conjunto de pesos de la red se ajusta cada cierto número de *pasos*):

$$\Delta w_{ij} = -\mu \frac{\delta E_{total}(t_0, t_1)}{\delta w_{ij}}$$

Continuo (el conjunto de pesos de la red se ajusta tras cada presentación de entradas):

$$\Delta w_{ij} = -\mu \frac{\delta E(t)}{\delta w_{ij}}$$

3.2.3. Time Delay Neural Networks (*TDNN*)

Buscando alternativas a las tradicionales redes feedforward, la primera idea que se manejó para implementar una memoria fueron los *retardos*. En el modelo feedforward, realmente, ya se estaban usando, aunque sólo en la capa de entrada; en esta nueva arquitectura se pretende incorporarlos en todos los elementos de proceso. Para ello, lo que se hace es *duplicar las conexiones* entre neuronas (ver figura 3.2.3, página 51).

De este modo, en cada instante puede considerarse el peso de índice asociado al retardo que se pretenda tener en cuenta. Realmente, la situación es la que se refleja en la figura 3.2.3 (página 51).

Es una forma de implementar un *buffer de activaciones anteriores*, que es lo que se hacía en las redes feedforward con la capa de entrada, como ya hemos mencionado.

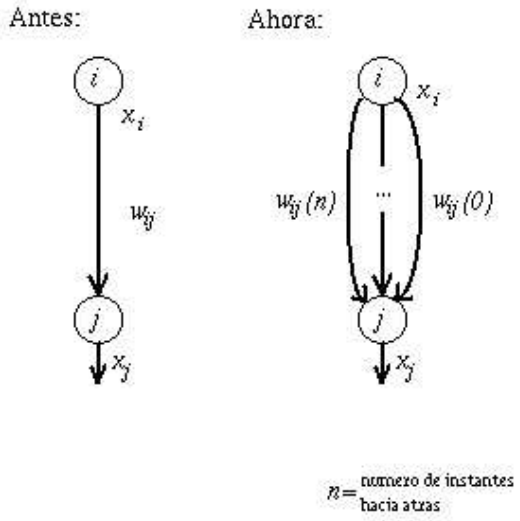


Figura 3.1: Modelos TDNN.

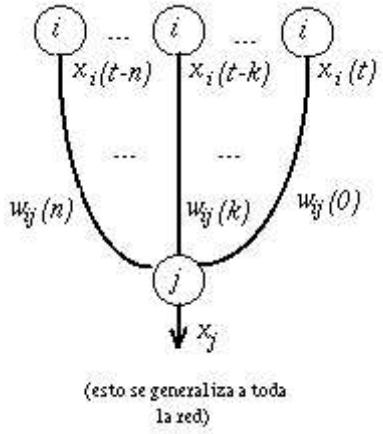


Figura 3.2: Modelos TDNN (general).

Algoritmos de aprendizaje: gradiente y variaciones backpropagation (temporal backpropagation)

Como hemos repartido los retardos en las capas, los algoritmos van a actuar *por capas* y no por instantes de tiempo, a pesar de estar en un esquema temporal donde el concepto de capa, como hemos visto, se diluye.

$$\begin{aligned} x_i(n) &= f(net_i(n)) \\ net_i(n) &= \sum_{l=0}^{n} w_{ij}(l) x_j(n-l) \end{aligned}$$

donde n es el tamaño de la *ventana temporal*. Los pesos se modifican:

$$w_{ij}(n+1) = w_{ij}(n) + \mu \delta_i(n) \cdot x_i(n)$$

donde

$$delta_i(n) = \begin{cases} e_i(n) \cdot f'(net_i(n)) & \text{si PE } \in \text{OUT} \\ f'(net_i(n)) \sum_{m \in A} \sum_{n=0}^{M} \delta_m(n+l) \cdot w_{mi}(n) & \text{si PE } \notin \text{OUT} \end{cases}$$

donde f' es la derivada de la función de activación y A es el conjunto de elementos conectados a la salida de x_i. Como se puede observar, el peso correspondiente al instante 0 no se actualiza nunca, es como una especie de BIAS.

Esto que hemos visto hasta ahora son TDNN con **retardo temporal fijo**. También se han desarrollado TDNN con **retardo temporal variable**; el buffer de activaciones en las primeras se presenta ordenado, mientras que en este último caso se modifica:

$$net_i(n) = \sum_{l=0}^{n} w_{ij} \cdot x_j(n - \tau_{\mathbf{ij}}(\mathbf{l}))$$

donde $\tau_{ij}(l)$ cambia con el tiempo.

En TDNN con retardo temporal fijo a cada conexión le corresponde una activación en el tiempo concreta (índice $n-l$); en TDNN con retardo temporal variable para cada instante l toma diferentes valores ($\tau_{ij}(l)$).

Lo interesante es que $\tau_{ij}(l)$ no es fijo sino que es *entrenable*, además de serlo los pesos $w_{ij}(n)$ como es común. Esto nos permitiría desordenar, eliminar o sobreponderar algunas activaciones frente a otras, quitando importancia al valor de la ventana temporal, que es lo que se persigue:

$$\tau_{ij}(n+1) = \tau_{ij}(n) + \mu \delta_i(n) \cdot x_i(n)$$

3.2.4. BackPropagation Through Time

Las redes **BPTT** son un tipo de redes recurrentes que para entrenarlas se desenvuelven en el tiempo (cada instante t da lugar a una capa), y se utiliza un algoritmo de retropropagación.

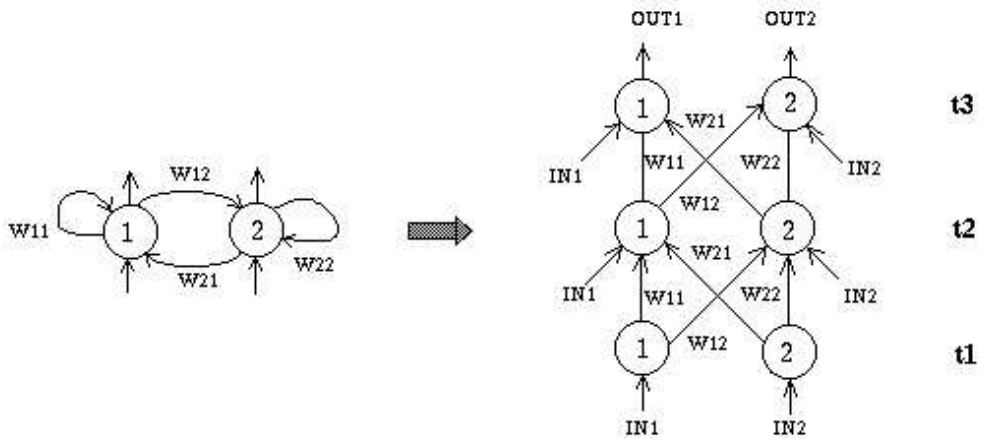


Figura 3.3: Desenvolvimiento de una red BPTT.

La ventaja que presentan con respecto a las redes TDNN es que aquí los pesos son los mismos entre cada par de capas, de modo que hay mucho menos que modificar.

Las líneas maestras del algoritmo BPTT, publicado en 1988 por Werbos, son las siguientes:

$$\Delta w_{ij} = \frac{\delta E(t)}{\delta w_{ij}} = \sum_{T=t_0+1}^{t} \delta_i(t) \cdot x_j(t-1)$$

donde

$$\delta_k(t) = f'(net_k(t)) \cdot \varepsilon_k(t)$$

con

$$\varepsilon_k(t) = \begin{cases} e_k(t) & T = t \\ \sum_{l \in \text{OUT}} w_{lk}\delta_l(t+1) & T < t \end{cases}$$

Como es un algoritmo que sigue la filosofía *backpropagation*, los sumatorios se empiezan a calcular en el valor del índice más alto hasta el más pequeño.

Si el número de t, o bien el de neuronas, es muy grande, este modelo se hace bastante impracticable. Una idea intuitiva para hacerlo más viable es restringir el número de instantes de t. Lo que se suele hacer es, pues, un *cálculo truncado*: se tienen en cuenta sólo h instantes de tiempo hacia atrás. Las fórmulas a aplicar entonces son las mismas, con una única variación en el subíndice inferior del sumatorio en la fórmula de modificación de pesos:

$$\Delta w_{ij} = \frac{\delta E(t)}{\delta w_{ij}} = \sum_{T=t-h+1}^{t} \delta_i(t) \cdot x_j(t-1)$$

de este modo sólo tendrían que almacenarse h valores de t del pasado.

Las ventajas de este modelo son, entre otras, que al ser ya un modelo recurrente, tiene memoria. Entre las desventajas, el hecho de que se está usando un método adaptado de redes que no son recurrentes (backpropagation), lo que nos obliga, como hemos mencionado, a guardar muchos valores para realizar los cálculos.

3.2.5. Redes parcialmente recurrentes

Las redes **parcialmente recurrentes** tienen una arquitectura alimentada hacia delante pero implementan una *capa de contexto* que es la que implementa la recurrencia. Hay diferentes tipos, que pueden observarse en la figura 3.4 (página 55).

Estas redes se entrenan con un algoritmo de retropropagación normal. Las conexiones entre la capa de contexto y la oculta (o, en general, la que esté conectada a ella) no se entrenan, son más bien un parámetro. Lo único que se modifica son las activaciones de la capa oculta, que lo hacen según:

$$x_i = f\left(\sum_j w_{ij} \cdot x_j + \sum_k w_{ik} \cdot c_k\right)$$

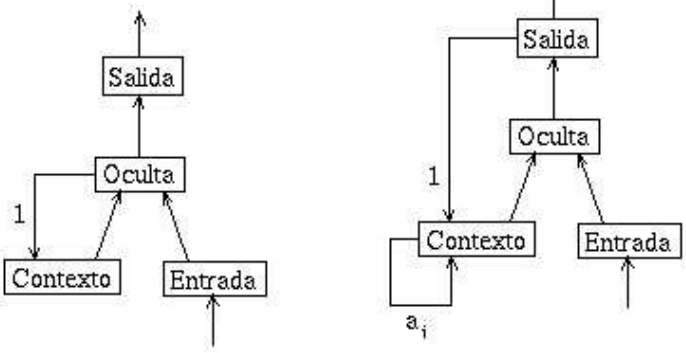
donde x_j representa la capa de entrada y c_k la capa de contexto.

Mas de nuevo estamos ante un "apaño" y no un esquema puro, que como ventajas presenta la facilidad de implementación y entrenamiento (presenta poco cálculo), pero, por contra, sólo es válida para series temporales simples, pues si la serie es compleja no es capaz de generalizar el comportamiento de la misma.

3.2.6. RealTime Recurrent Learning

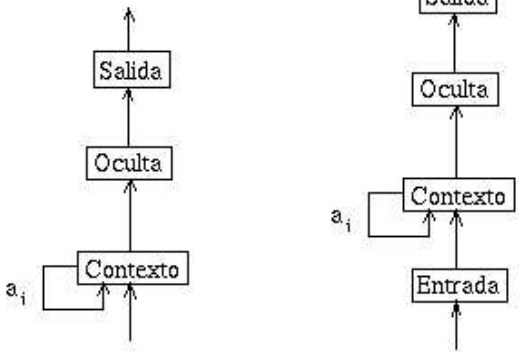
El algoritmo **RTRL** se aplica a redes "realmente" recurrentes (sin el concepto de capa, con activaciones por tiempo) y fue desarrollado por varios autores, entre ellos Zipser, y publicado en 1989.

Otra de sus características principales es que la *propagación del error es hacia delante*:



(a) Clásica.

(b) Variante típica.



(c) Como ventana temporal=2.

(d) Variante recurrente.

Figura 3.4: Tipos de redes parcialmente recurrentes.

$$\Delta w_{ij} = \frac{\delta E(t)}{\delta w_{ij}} = \sum_{k \in \text{OUT}} e_k(t) \cdot \Pi_{ij}^k(t)$$

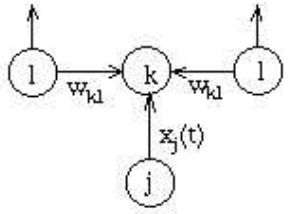
donde el parámetro $\Pi_{ij}^k(t)$ representa el ajuste del peso w_{ij} calculando parcialmente cómo influye esa conexión con respecto a los valores de salida (cada uno de ellos), $k \in \text{OUT}$. Es por ello que se podera el error $e_k(t)$ por medio de este parámetro, lo que influye cada conexión w_{ij} en la salida k.

$$\Pi_{ij}^k(t+1) = f'(net_k(t+1)) \cdot \left(\sum_{l \in \text{OUT}} w_{kl} \cdot \Pi_{ij}^l(t) + \delta_{ik} x_j(t) \right)$$

donde $\Pi_{ij}^k(0) = 0$. La función δ_{ik} en esta ocasión representa la denominada *delta de Kroneker*:

$$\delta_{ik} = \begin{cases} 1 & i = k \\ 0 & i \neq k \end{cases}$$

Los cálculos de este modelo son sencillos, comparados con otros de los ya vistos.



En cuanto a su funcionamiento, como no podemos hablar de capas, para cada instante t se calculan todos los valores $x_j(t)$ a partir de los $x_j(t-1)$, sin importar el orden de cálculo. Una vez que se tienen todas las activaciones, se calculan los $\Pi_{ij}^k(t)$ a partir de los errores en el instante t, $e_j(t)$, que ya se pueden obtener porque se dispone de los $x_j(t)$ y los $\Pi_{ij}^k(t-1)$. Por último, se modifican los pesos, Δw_{ij}.

La ventaja de este modelo se encuentra en la propagación hacia delante de los errores y la simplicidad que ello aporta frente a esquemas backpropagation, mientras que, como desventaja, encontramos la situación de que si la red tiene muchos elementos de procesado hay un número elevado de parámetros Π que calcular (para cada conexión, tantos como salidas tenga la red).

Las redes RTRL son las más adecuadas para predicción a largo plazo y tratamiento de series complejas como caóticas o no estacionarias. No obstante, lo que realmente se suele hacer en la práctica con este tipo de series en las que no se puede aplicar una red alimentada hacia delante con ventana temporal, es simplificarlas. Se evita la utilización de redes RTRL por problemas de convergencia, frente a la sencillez y rapidez de entrenamiento de

una red feedforward. Claro que si la serie es muy compleja y no puede ser aproximada, sí se hace necesario el componente de memoria que aportan.

Capítulo 4

Computación Evolutiva

4.1. Introducción

En muchos problemas del mundo real se utilizan redes de neuronas artificiales junto con diferentes sistemas de otras ramas de la informática, como por ejemplo:

√ Sistemas Expertos, que aportan capacidad de explicación

√ Bases de Datos, en sistemas de *data mining*

√ etc.

Estas conjunciones reciben el nombre de **sistemas híbridos**, y la **Computación Evolutiva** es también uno de ellos.

Las técnicas de **Computación Evolutiva** son técnicas de búsqueda que se basan en la evolución natural. La *Naturaleza* y la *Informática* ya se han relacionado antes: genética, ecología, inteligencia artificial... y no gratuitamente, pues no en vano los sistemas más complejos que se conocen son los seres vivos (y sus mecanismos de evolución). Además, en contraposición con las máquinas, no sólo presentan una complejidad estructural mucho mayor, sino que contienen autoexplicación sobre ellos mismos (código genético). La gran pregunta es: ¿cómo se llega a ese punto, a ese estado de evolución y complejidad a la vez tan armoniosa y organizada? La respuesta está en el *proceso de optimización natural*.

Ha habido grandes figuras, por todos conocidas, que han aportado su granito de arena a la teoría de la selección de las especies:

- *Lamark*, con su teoría de la herencia de los caracteres adquiridos.
- *Darwin*, con su libro *El Origen de las Especies*, sobre la selección natural.
- *Mendel*, con sus teorías sobre la herencia (caracteres dominantes, recesivos...).
- *De Vries*, con su teoría sobre las mutaciones y los cambios abruptos.

La recopilación de todas sus aportaciones ha dado lugar al *Neo-darwinismo*, más o menos vigente en la actualidad.

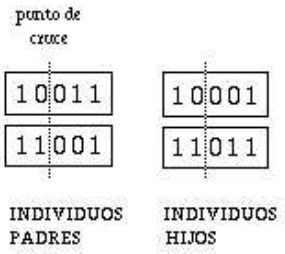
La **Computación Evolutiva** considera a los *individuos* de una población como posibles *soluciones* de un problema dado. Así, dos cosas imprescindibles en el ámbito de la C.E.[1] son la definición de un mecanismo de *codificación* (consideraremos los individuos como cadenas de bits/valores) y un modo de *evaluación* de dichos individuos.

A partir de ahí, la población inicial se desarrolla al azar, y cada individuo se valora independientemente. En analogía con la naturaleza, se escogerá a los "mejores", que son los que con toda probabilidad más oportunidades de sobrevivir han de tener.

En cuanto a los *operadores* aplicables sobre la población, los principales son dos:

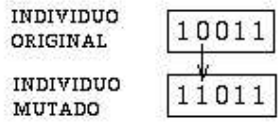
1. **Cruce**, cuya tarea es mezclar, combinar el material genético con el objetivo de reunir en un sólo individuo (solución) lo mejor (mejores características) de la población.

   A partir de dos individuos progenitores y de su *genotipo* se define al azar un **punto de cruce**, y se intercambian sus informaciones genéticas:



2. **Mutación**, que persigue la variabilidad, generar valores nuevos que no estén ya en la población.

   En el caso binario, supondría el cambio de un 1 por un 0 o viceversa:



Entre los métodos de selección, podemos citar el *Roulette Wheel Selection*.

[1] Abreviaremos así a partir de ahora *Computación Evolutiva*.

4.2. Técnicas de Computación Evolutiva

Las tres técnicas más importantes dentro de la C.E. son:

- **Algoritmos genéticos**
- **Programación genética**
- **Vida Artificial**

4.2.1. Algoritmos Genéticos

Obra de John Holland (1975), la principal aplicación de los **algoritmos genéticos** es la *optimización de parámetros*[2]. La población de soluciones se ve como una cadena de posiciones con parámetros, de valores binarios o reales, y los operadores que se aplican son los ya conocidos: cruce y mutación.

Los algoritmos genéticos son buenos localizando *zonas* óptimas, no tanto *valores* óptimos. La ventaja es que no necesitan conocer demasiado el problema, sólo la forma de evaluar los individuos. Claramente, son un método *no determinista*, ya que cuentan con muchas componentes azarosas: generación de la población, selección de los individuos,... y por ello no deben nunca aplicarse si ya existe un algoritmo matemático que resuelva el problema de manera exacta y directa. En caso contrario, los algoritmos genéticos son una buena opción.

Entre los ejemplos de su utilización podemos encontrar:

Optimización

Investigación Operativa

Transporte: volumen, tamaño, coste, rapidez,...

Sistemas clasificadores

Descubrimiento de características para determinar

Rutas

Problema del viajante

4.2.2. Programación Genética

La **programación genética** fue iniciada por John Koza en 1980 para el desarrollo de algoritmos sobre poblaciones de soluciones, dando lugar a la consideración de estructuras como *árboles* (los individuos se representan, en lugar de por tiras de bits, por árboles,

[2]De ahí su aplicación al entrenamiento de RNA, pues se puede ver como un proceso de optimización de los valores de los pesos de las conexiones entre elementos de procesado, como estudiaremos en secciones subsiguientes. Además, presentan la ventaja de no estancarse en mínimos locales, como le sucede a algoritmos de tipo gradiente.

donde las hojas son constantes o variables y las ramas, operadores) y a la definición de funciones y terminales.

Las operaciones de cruce y mutación sobre árboles también sufren la modificación necesaria: el cruce pasa ahora a consistir en el intercambio de 2 ramas entre 2 árboles (individuos padres) y en la mutación se selecciona un nodo y, o bien se cambia su valor, o bien se elimina y se genera una nueva rama en su lugar.

Ejemplos de su utilización son:

Regresión simbólica

Cálculo de funciones que pasen por una serie de puntos

Desarrollo de programas

Control del caminar de un robot de 6 patas

Sistemas basados en reglas

"Sistemas expertos" sin necesitar al experto en campos reducidos

4.2.3. Vida Artificial

Lo que se denomina **vida artificial** es una variante de la C.E. en la que se estudia la población de un modo más global: se considera que los individuos viven sobre un mundo (*rejilla*) en el que desarrollan comportamientos simples. Se estudian entonces las interacciones por proximidad que se producen entre ellos, buscando **comportamientos emergentes**, es decir, comportamientos complejos (difíciles de programar) que surgen como resultado de la realización combinada de comportamientos simples individuales[3].

Ejemplos de utilización de técnicas de vida artificial los encontramos en:

Ant Colony Optimization (A.C.O.)

Enrutamiento, búsqueda de caminos óptimos[4]

Juego de Conway[5]

Agentes

Conjuntos de reglas, pequeños sistemas expertos que realizan tareas simples

Colaborando, son capaces de realizar tareas más complejas

Extracción de reglas, clasificación,...

[3]El ejemplo más representativo lo encontramos en las hormigas.

[4]En imitación a la conducta de las hormigas guiadas por sus feromonas.

[5]Su funcionamiento, recordemos, era simple: en una población dispuesta matricialmente, una celda vacía era ocupada en la generación siguiente si estaba rodeada por al menos 3 celdas ocupadas y una ocupada lo seguía estando si estaba rodeada por 2 ó 3.

Otras aplicaciones

Naturaleza y Biología

Genética: codificación y combinación

Ecología: estudio de poblaciones

Hardware evolutivo

Robótica

F.P.G.A.

4.3. Aplicación de técnicas de Computación Evolutiva en RNA

En RNA los mayores problemas se presentan con respecto a la utilización de las redes, comenzando por la selección adecuada de las variables relevantes del problema, de los conjuntos de entrenamiento y test, la elección del modelo, definición de parámetros (número de capas, número de elementos de procesado por capa, número de conexiones...), determinación del algoritmo de entrenamiento (cada uno con sus problemas: mínimos locales, lenta convergencia,...), etc.

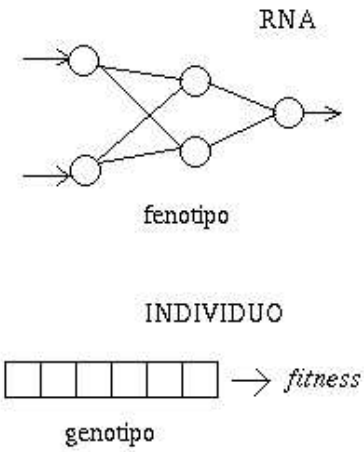
Sobre todas estas cuestiones hay estudios que pueden orientarnos, pero no un método matemático que nos garantice resultados, lo que hace del proceso una larga iteración en modo *prueba-y-error*: métodos incrementales/decrementales para determinar el número de capas y elementos de proceso (que, además, sólo son válidos en problemas de clasificación y son muy dependientes de la arquitectura original), prueba sucesiva de diferentes algoritmos de entrenamiento (muchos de los cuales no pueden paralelizarse, lo que obliga a realizar las pruebas una tras otra, en secuencia), incluso la selección de variables, a pesar de poder apoyarse en la estadística, resulta compleja.

La **Computación Evolutiva** (sobre todo los algoritmos genéticos y la programación genética) representan un novedoso aporte al mejor funcionamiento de las RNA. Resultan útiles, sobre todo, en la selección de variables e incluso como algoritmos de entrenamiento[6]. Lo veremos en los apartados siguientes.

4.3.1. Algoritmos genéticos para el entrenamiento de RNA

Tras fijar una arquitectura, se conocen el número de elementos que tenemos que ajustar (número de conexiones), lo que nos da la "longitud" de los individuos (de su *genotipo*). Se genera entonces al azar una población de este tipo, para posteriormente evaluar el nivel de error de cada individuo (*nivel de ajuste*, $\sum e_k$, también llamado *fitness*), nivel de error que se calcula de igual modo que con los algoritmos de gradiente usuales.

[6]Representan una alternativa a los métodos de gradiente que no tiene que lidiar con mínimos locales y que es paralelizable.



El *fitness* para cada individuo representa lo bueno que es el conjunto de pesos que encarna para la resolución de nuestro problema. Así pues, el paso siguiente es escoger, dependiendo del ajuste, individuos para cruzar e individuos para mutar. Con n generaciones (normalmente pocas) los valores de error irán disminuyendo hasta alcanzar una cota aceptable.

4.3.2. Aplicaciones prácticas de Computación Evolutiva en RNA

Las técnicas de computación evolutiva se suelen utilizar sobre todo en algoritmos de búsqueda. En conjunción con las RNA, son un método único a la hora de evaluar, elegir y/o estudiar la conveniencia de utilizar una u otra arquitectura de red para la resolución de un determinado problema, realizado simultáneamente junto con el ajuste de pesos.

Lo que se hace es seguir las etapas básicas de un algoritmo genético, ya comentadas, (generar n soluciones válidas, evaluarlas eliminando las peores, cruzarlas, mutarlas y repetir el proceso hasta alcanzar un buen ajuste) donde en la población los individuos representan redes (cuyos cromosomas –valor– pueden representar número de PE por capa, por ejemplo). El fitness de cada individuo será entonces el ECM del mejor representante de esa arquitectura (subproblema de ajuste de otra subpoblación). El punto de cruce e intercambio se elige al azar en cada generación, realizándose normalmente un 90 % de cruces y un 10 % de mutaciones, que suelen consistir en la generación aleatoria de un nuevo gen (capa).

Por su parte, la programación genética (que se diferencia en su representación arborescente de los individuos), se utiliza en la construcción automática de programas.

4.3.3. RNA y algoritmos genéticos: ventajas y problemas

Los principales problemas suelen encontrarse en cuestiones referentes al ajuste de arquitecturas RNA comentado en la subsección anterior, entrenamientos, tiempo,... Algunas de las soluciones que se esgrimen al respecto se esbozan a continuación.

Podaje de conexiones (Pruning)

El **podaje de conexiones** consiste en la eliminación, tras el entrenamiento, de conexiones inútiles (poco influyentes, con pesos cercanos a cero), a fin de simplificar la estructura de la RNA.

Con algoritmos genéticos, esto puede implementarse redefiniendo el fitness de modo que no sólo sea el error asociado a la RNA, sino que se añada un factor que represente una penalización proporcional al número de conexiones (a mayor número de conexiones, mayor penalización):

$$\frac{\text{nºconex.} \neq 0}{\text{nºconex.totales}}$$

La filosofía de las penalizaciones suele usarse en AG porque en lugar de eliminar de buenas a primeras los individuos con valores no deseados (una dura restricción), un individuo malo (muy penalizado) puede tener, no obstante, alguna característica buena que podrá ser así incorporada a una versión mejor del sujeto tras un cruce o mutación.

Mínimos locales

Como ya hemos dicho, el entrenamiento de RNA basado en AG es menos dado a caer en mínimos locales, sobre todo si se incrementa el ratio de mutación.

Aplicabilidad

El entrenamiento mediante AG es válido para distintos modelos de red neuronal (feed-forward, recurrentes) tradicionales, incluso para aquéllos no soportados por algoritmos de gradiente (con conexiones a diferentes capas, por ejemplo).

Codificación binaria vs. real

La codificación de los individuos como cadenas de bits simplifica las operaciones (por ejemplo, la mutación consiste sólo en cambiar el valor de una posición aleatoria dentro del cromosoma, de 0 a 1 o viceversa), pero reduce la precisión, puesto que ésta viene determinada por el número de bits utilizado en la codificación.

Una codificación real evita la traducción a binario de valores presentes en el campo de estudio, a costa de cambiar la filosofía del cruce (en binario podría llegar a "partirse" un "peso" por la mitad) y hacer más compleja la mutación (que pasa ahora por generar un número al azar y sustituir o bien operar con el peso ya existente).

Tiempo

El entrenamiento de RNA con AG suele funcionar bien y puede ser incluso más rápido que un entrenamiento tradicional con algoritmos de descenso de gradiente.

Selección de arquitecturas

La selección de arquitecturas de RNA implica tanto la selección del número de capas y el número de neuronas por capa como el podaje (selección) de conexiones. Como ya se ha comentado, cada individuo representa una arquitectura y se define una matriz de conexión $n \times n$ binaria, donde n es el número máximo de PE por red, parámetro que sí hay que proporcionar (basándose en estudios sobre el problema, por ejemplo).

El ajuste en este caso se obtiene tras entrenar la arquitectura representada por cada individuo (red), pudiendo ser el error alcanzado en el entrenamiento de dicha red (que puede hacerse, a su vez, por AG o con métodos clásicos).

Evidentemente, el punto fuerte de esta aplicación no es el tiempo de entrenamiento. Para paliarlo, lo que se hace es no llevar el entrenamiento de cada red-individuo hasta el final, ya que sólo se necesita una idea de cómo se comporta en términos generales. Ha de definirse, pues, un *criterio* de "cuánto" se van a entrenar las redes, que puede ser:

- $\surd$ número de ciclos (ha de ser suficiente para que las redes grandes entrenen, a las pequeñas forzosamente les sobrará)
- $\surd$ nivel de error (presenta un grave problema ante la no convergencia de algunos individuos)
- $\surd$ tiempo (muy subjetivo)

El resto del proceso (mutaciones, cruces) es el habitual, aunque con matices: hay que tener cuidado al escoger el punto de cruce, puesto que los individuos son de longitudes diferentes, mientras que las mutaciones pueden, generalmente, consistir en añadir o quitar capas.

Pese a todo, al margen de la lentitud de esta opción, la realidad es que no hay otra que la suplante en su tarea, es la única forma de hacerlo. Es por ello que se han buscado múltiples formas de optimizarla hasta el extremo, pasando, por supuesto por alternativas que plantean la **paralelización** (dividir la población, esto es, se paraleliza no el algoritmo, sino la evaluación de los individuos):

Migración Se tienen varios procesadores, una población por procesador. Cada x generaciones se intercambian individuos, con el mismo criterio que el de selección para el cruce. Reduce mucho el riesgo de mínimos locales.

Difusión Permite sólo cruces entre individuos próximos –de ajuste, genoma parecido–, generando clusteres en base a las similitudes y repartiendo cada cluster en un procesador diferente. Cuando un individuo muta y deja de parecerse a los de su grupo, cambia de procesador.

Farming Existe un equipo maestro y varios equipos esclavos, cada uno en un procesador diferente. El equipo maestro rige la población al completo (hace la selección), y los esclavos realizan los cruces y mutaciones.

4.4. Sistemas híbridos

A parte de los ya comentados alguna vez (mezcla entre sistemas expertos y RNA), existe un tipo de **sistemas híbridos** que combinan *algoritmos genéticos* y *lógica difusa*, aplicando ésta para que la pertenencia de elementos a conjuntos no sea estricta (*fuzzy sets*) sobre los conjuntos de entrenamiento, habilitando de este modo la resolución de problemas inabordables de otro modo (más próximos a la vida real). La idea fue originaria del japonés Lofti Zadeh.

4.4.1. Definición de conjunto borroso

Un **conjunto borroso** tiene una *función de pertenencia*, que indicará si un elemento pertenece o no al conjunto, que devolverá un número entre 0 (no pertenecia) y 1 (pertenencia) que representa la probabilidad de pertenencia asociada al elemento con respecto al conjunto. Dicha función suele nombrarse Π.

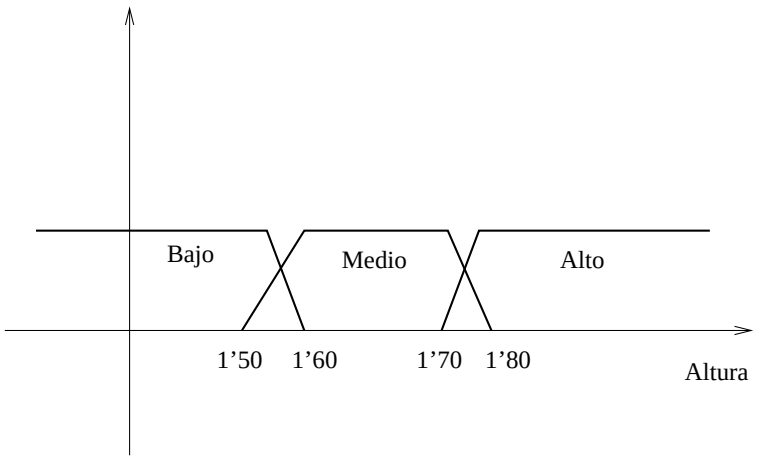


Figura 4.1: Ejemplo de función de pertenencia a un conjunto difuso.

Suelen ser, como en la figura, funciones de forma trapezoidal o triangular.

También se usan *etiquetas semánticas* estándar (*muy*, *bastante*, *más o menos*,...) que se ajustan con respecto a los valores de la función de pertenencia. Esto es lo que permite usar valores reales en lógica difusa.

Se aconseja una partición completa del dominio, es decir, definir las funciones de pertenencia para obtener los conjuntos borrosos de una variable de forma que éstos cubran todos sus posibles valores, su rango al completo, siendo su área de solapamiento no muy grande (entre 20-30 %).

A partir de aquí, para lo que se usa la lógica difusa es para definir *reglas borrosas*, muy utilizadas en sistemas expertos y control industrial. Estas reglas admiten valores difusos de entrada y proporcionan valores difusos de salida. También hay mecanismos de

"traducción" de valores *fuzzy* a *crisp* y viceversa.

Dentro de las RNA hay tres aproximaciones para usar lógica difusa, siendo las dos últimas las más interesantes y la más sencilla la primera (la que primero surgió):

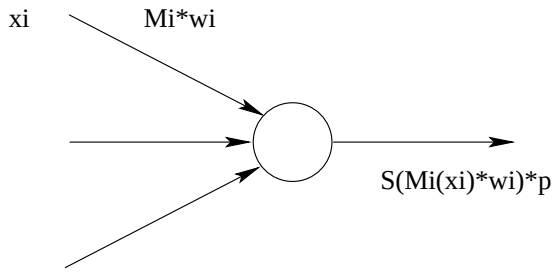
- usar entradas difusas
- usar pesos difusos (sus valores)
- las dos anteriores (una combinación)

La utilización de alguna de ellas da lugar a las **fuzzy artificial neural networks** (**FANN**).

4.4.2. Neurona de procesado difuso

El modelo de PE difuso modifica el modelo normal de neurona, pues sus pesos van a tener ahora un valor difuso. Esto se implementa normalmente teniendo una variable (conjunto) difusa que se multiplica por un peso de valor real tradicional.

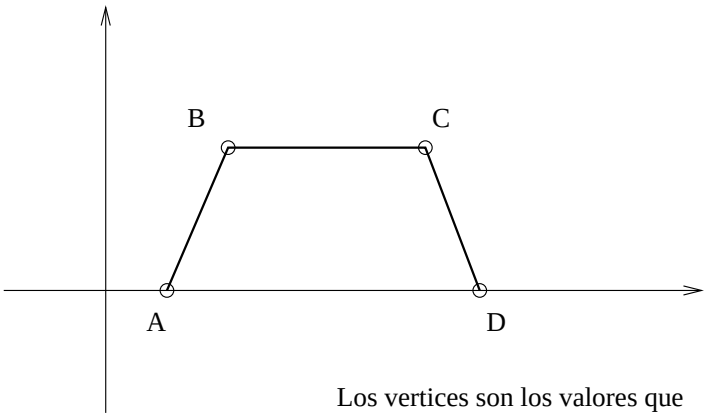
Las conexiones tendrán entradas que serán validadas por el conjunto difuso:



Se evalúa la pertenencia de la entrada al conjunto y el valor obtenido se multiplica por el peso (funciona como una especie de regla difusa ponderada). Toda la información de las entradas se integra del modo usual y se pondera una vez más por un *valor de atenuación*. Al valor de salida, en lugar de aplicarle una función de activación se le aplica nuevamente una regla difusa (que puede ser simplemente otra función de pertenencia o bien una combinación más compleja). El valor obtenido es un valor difuso que puede ser traducido a un valor real (si la neurona se encuentra en una capa de salida).

4.4.3. Entrenamiento

Este tipo de redes no está muy estandarizado y no se usa aún demasiado. Hay diversos algoritmos para su entrenamiento, todos basados en descenso de gradiente (modificaciones del algoritmo de retropropagación), donde primero se ajustan los pesos y después las funciones de pertenencia asociadas a las conexiones.



4.4.4. Aplicaciones de *fuzzy* RNA

El campo principal en el que se están usando las FANN actualmente es en control industrial. Se usan en los bucles de realimentación para controlar la ejecución de procesos. También se usan en clasificación, dentro del ámbito en el que son más útiles, es decir, cuando no podemos precisar los datos de entrada (variables difusas, imprecisas).

Índice alfabético

Índice de figuras

Índice de cuadros

Bibliografía

[1] González Penedo, Manuel.
Apuntes de Sistemas Conexionistas *http://carpanta.dc.fi.udc.es/%7Ecipenedo/cursos/scx/scx.html*, 2001.