Computación neuronal. Universidade de Santiago de Compostela, 1995 (267-300)

X

SUPERCOMPUTACIÓN VECTORIAL Y REDES NEURONALES ARTIFICIALES

Eduardo Sánchez; Senén Barro; Manuel Lama
Universidade de Santiago de Compostela

Carlos V. Regueiro
Universidad de A Coruña

1. INTRODUCCIÓN

Los nuevos proyectos de investigación que involucran Redes Neuronales Artificiales (RNA), así como sus modelos, se vuelven cada vez más y más ambiciosos. Esto demanda nuevas proyecciones tecnológicas, más flexibles y potentes, de las RNA. En este punto podemos distinguir básicamente entre dos tipos: las implementaciones y las simulaciones. Dentro de las primeras nos encontramos con las implementaciones electrónicas, tecnología limitada a un bajo nivel de conexionismo, las implementaciones ópticas, menos desarrolladas que las otras pero con una alta densidad conexionista debido a su naturaleza 3-D, y las implementaciones basadas en el desarrollo de neurocomputadores de propósito más general.

En lo referente a las simulaciones nos encontramos con las proyecciones de RNA en computadores de propósito general [*Parallel Computing,* 1990], las cuales son particularmente interesantes cuando se trata con arquitecturas especiales como las paralelas, vectoriales o sistólicas [Jai-Hoon y col., 1991; Millán y Bofill, 1989]. Esto es debido a que estas arquitecturas ofrecen la posibilidad de aprovechar en mayor o menor grado el masivo cálculo paralelo inherente a las propias redes.

Resulta obvio que las implementaciones tecnológicas son más potentes que las simulaciones porque consiguen un paralelismo real en el funcionamiento de la red y una velocidad muy superior. A pesar de ello son pocas todavía las

implementaciones neuronales en la tecnología VLSI [Valderrama y Carrabina, 1995; Cabestany y col., 1995] y anecdóticas en las implementaciones ópticas. Esto es debido a que, desafortunadamente, este *hardware* de propósito específico es poco flexible y de alto coste.

Para resolver este problema disponemos de las simulaciones, que consiguen la flexibilidad necesaria durante las primeras fases de la investigación y el desarrollo de modelos y aplicaciones. En estas situaciones, donde se demanda velocidad por un lado, y flexibilidad por otro, los grandes supercomputadores, que incorporan arquitecturas optimizadas para mejorar la potencia de cálculo, suponen una solución nada despreciable.

Este capítulo está dedicado a la proyección de RNA sobre arquitecturas vectoriales. El objetivo que se persigue es mostrar diferentes estrategias de programación que permitan obtener la máxima eficiencia en este tipo de supercomputadores. Como apartado previo realizaremos una introducción al mundo de los supercomputadores vectoriales, donde hablaremos de sus principales características y de sus mecanismos de funcionamiento. Dentro del mismo bloque analizaremos las conexiones entre las potencialidades de un vectorial y las necesidades de cálculo de las RNA. A continuación, estudiaremos las propiedades topológicas de las redes en busca de paralelismos espaciales y temporales, ya que estos son susceptibles de una vectorización eficiente. Una vez realizado el análisis global de las RNA, nos detendremos a estudiar la vectorización de los elementos que las componen. Por último, comentaremos un ejemplo concreto de implementación: la Red *Perceptron* Multicapa (RPM) con el algoritmo *BackPropagation* (BP) [Sánchez y col., 1994].

2. ARQUITECTURAS VECTORIALES Y LAS RNA

2.1. Supercomputadores

Comenzamos este apartado con una breve introducción al campo de los supercomputadores, tratando de aportar al lector una visión de conjunto.

El término «supercomputador» es atribuido al computador científico más potente disponible en un momento dado. Estas máquinas tienen un coste muy elevado por su complejo diseño y por el alto grado de investigación y desarrollo que precisan. El campo donde mejor se explota la capacidad de los supercomputadores, y en donde mejor se amortiza su inversión, es el campo científico, en el que es necesario resolver problemas con un gran número de operaciones. Hay que tener en cuenta que los sistemas actuales pueden calcular varios billones de números por segundo con una elevada precisión [Zima, 1991].

Dentro del campo de los supercomputadores nos podemos encontrar con máquinas basadas en diferentes arquitecturas: paralelas, vectoriales, sistólicas, etc. Para obtener un óptimo rendimiento se hace necesario evaluar qué aplicación se adapta mejor a cada arquitectura. Dicho rendimiento o eficiencia tiene que reflejar la potencia del computador. Ésta se puede calibrar por su velocidad en el cálculo de números reales en punto flotante, por el rango de los números que puede calcular, y el tiempo de transporte de los números calculados hacia y desde las diferentes partes del sistema (ancho de banda).

Los actuales supercomputadores han tenido un impacto profundo en todas las ramas de la ciencia y la ingeniería: ingeniería nuclear, desarrollo de dispositivos militares, astrofísica, física de partículas, química cuántica, cristalografía, oceanografía, meteorología y muchos otros. En el campo comercial, se ha encontrado aplicación en el diseño de aviones, diseño de motores para coches, producción de películas, exploración minera, etc.

La búsqueda de una mayor potencia de cálculo para resolver los problemas asociados a las áreas que hemos comentado, ha impulsado la investigación y el desarrollo de arquitecturas que permiten realizar tareas y operaciones en paralelo. Las aplicaciones proyectadas sobre estas arquitecturas serán eficientes en tanto en cuanto sea posible su subdivisión en tareas procesables de modo independiente y en el mismo ciclo de reloj. Esto ha motivado la investigación y el desarrollo de algoritmos paralelos donde se intenta reducir al máximo la dependencia posible entre los cálculos a realizar. A pesar de que esto permite a cada procesador trabajar sin necesidad de intercambiar información con otros procesadores (o intercambiar muy poca), uno de los mayores problemas radica en el protocolo de comunicación procesador-memoria. El desarrollo de los supercomputadores vectoriales, basados en arquitecturas donde las unidades funcionales del computador están segmentadas, ha supuesto un paso adelante. Las arquitecturas segmentadas consiguen un alto grado de paralelismo (no obstante inferior al conseguido por una arquitectura paralela), y, lo que es más importante, eliminan el problema de la comunicación procesador-memoria intrínseco a las arquitecturas paralelas, implementando un único flujo de datos entre cada unidad funcional y la memoria. Esto hace de los supercomputadores vectoriales unas máquinas potentes y a la vez más flexibles de programar que las paralelas, motivo que explica su creciente implantación en todo el mundo.

2.2. Características de una arquitectura vectorial. El procesamiento segmentado

Vamos a centrarnos ahora en los principales aspectos que definen una arquitectura vectorial, para pasar a continuación a explicar brevemente el modo de

funcionamiento. Esto nos será de gran utilidad posteriormente, para entender mejor la proyección de las RNA en estas máquinas.

La potencia de cálculo de un supercomputador vectorial es el resultado de un conjunto de características de su arquitectura [Dasgupta, 1989; Lazou, 1988]:

a) un período de reloj muy pequeño, inferior a 10 nanosegundos,
b) un conjunto de unidades funcionales que no se comunican entre si,
c) segmentación de las unidades funcionales del computador,
d) un conjunto de registros vectoriales de alta velocidad,
e) una alta capacidad de almacenamiento,
f) una compartición de datos y de comunicación conseguida a través de la compartición de memoria y de los registros vectoriales,
g) una única cadena de tráfico de datos a la interfase procesador-memoria.

Todas estas características interaccionan para conseguir la alta eficiencia esperada en un supercomputador y obtener un alto grado de concurrencia en el cálculo. Aún así, dicha eficiencia varia con el tamaño y tipo de aplicación, así como con la habilidad del *software* para explotar al máximo las capacidades de la máquina. En la figura 1 se esquematizan algunas de las características antes nombradas, y se comparan las principales diferencias entre las arquitecturas vectoriales y las paralelas.

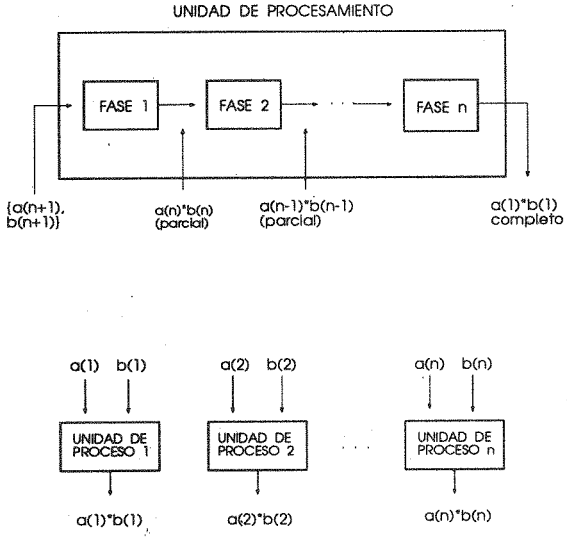


Figura 1. Unidad de procesamiento segmentado (parte superior). Unidades de procesamiento paralelo (parte inferior).

El concepto clave que define a un supercomputador vectorial es el de segmentación (*pipelining*). Segmentación, a nivel hardware, significa que las unidades funcionales del computador están divididas en varios bloques, es decir, la ejecución de cada proceso se realiza en varios pasos tal y como se ilustra en la figura 2. Para empezar a obtener resultados se requiere que el *pipeline* sea rellenado. El tiempo que tarda el *pipeline* en rellenarse se denomina tiempo de arranque. La velocidad con la que se obtiene un resultado, una vez rellenado el *pipeline*, se la denomina velocidad de iniciación.

El rendimiento de una aplicación genérica sobre una arquitectura vectorial depende directamente de la relación entre el producto velocidad de iniciación-número de operaciones, denominado tiempo total de iniciación (base del *pipeline*), y el tiempo de arranque (altura del *pipeline*). Si el primero es mucho mayor que el segundo, de tal manera que podamos considerar despreciable el tiempo necesario para empezar a obtener resultados comparado con el tiempo total de ejecución, nuestra aplicación obtendrá un buen rendimiento. Con este sistema la velocidad con la que se obtienen los resultados puede ser superior a un resultado por ciclo de reloj. Todo esto se ilustra en la figura 2.

¿Cómo se alcanza el paralelismo en estas arquitecturas?. El paralelismo se consigue en los *pipelines* del *hardware*, cuando estos han sido rellenados con diferentes datos. En la figura 2 se observa que en un momento dado, se están realizando n procesos paralelos diferentes (siendo n el número de segmentos del *pipeline*) con n operandos diferentes. Por tanto, para obtener un mayor paralelismo nos interesará una elevada segmentación de las unidades funcionales. Este paralelismo a nivel de *pipeline* es el que vamos a explotar cuando hablemos de la topología de las RNA y de las diferentes estrategias de programación.

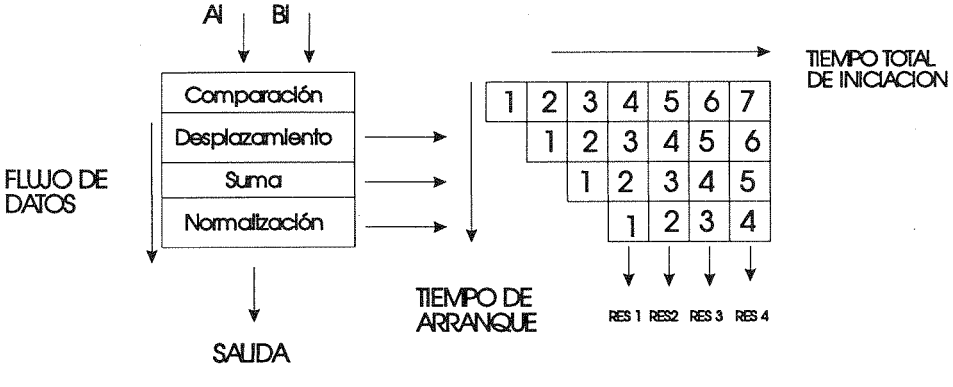


Figura 2. *Pipeline* y tiempo de arranque para una operación de suma en punto flotante.

2.3. El concepto de vectorización

La mayoría de los actuales ordenadores de propósito general realizan operaciones escalares. Este tipo de operaciones se definen como aquellas que trabajan con un par de operandos a la vez y producen un único resultado. En este contexto, los operandos son valores escalares, es decir, cantidades individuales. Por el contrario, los supercomputadores vectoriales realizan operaciones vectoriales, que involucran a más de un par de operandos. Además, estos operandos no son escalares sino vectores, conjuntos de datos que son accesibles bajo ciertas condiciones dependientes de la máquina. Comparándolo con el caso escalar, una operación vectorial implicaría el tratamiento de un par de vectores para producir otro vector como resultado.

Los supercomputadores vectoriales ofrecen al programador dos formas diferentes de sacar provecho a las capacidades del procesador: mediante la vectorización automática y mediante la vectorización explícita. La opción automática depende de la habilidad del compilador del lenguaje correspondiente de cada máquina (generalmente FORTRAN o C) para reconocer estructuras vectorizables dentro de la sintaxis del lenguaje y generar instrucciones vectoriales automáticamente en la fase de compilación del código. Debe quedar claro que los diferentes compiladores trabajan con diferentes grados de éxito. Esto depende de los diseñadores de compiladores, que continuamente intentan mejorar sus técnicas en esta área anunciando mejoras en cada nueva versión. Como extensión a la vectorización automática nos encontramos con la vectorización explícita. Ésta depende de la habilidad del programador a la hora de realizar sus programas y de su conocimiento de la máquina. La vectorización explícita se puede realizar de dos formas:

a) Mediante directivas de compilación, introducidas en el código fuente del FORTRAN o del C. Estas directivas son instrucciones específicas que indican al compilador que un lazo particular debería ser ejecutado de una determinada manera.

b) Mediante técnicas de programación específicas que permitan optimizar el funcionamiento de un programa a través de la manipulación de bloques de código inicialmente no vectorizable, como por ejemplo en el caso de dependencias, recursividades, saltos, etc.

A la hora de preocuparnos por el grado de vectorización de nuestra aplicación, tenemos que tener en cuenta qué instrucciones son vectorizables por la máquina. En las tablas I y II se indican las instrucciones vectorizables y aquellas que no lo son en un supercomputador para el lenguaje FORTRAN. En la tabla III se indican, a modo descriptivo, las técnicas de vectorización de que dispone el programador para optimizar su código.

Tabla I. Código vectorizable.

Rango del código objeto	Lazos DO
Tipo de datos	LOGICAL*4 INTEGER*4 REAL*4 REAL*8 COMPLEX*8 COMPLEX*16
Factores de las instrucciones	Expresiones aritméticas Expresiones lógicas Referencias a funciones Referencias a funciones intrínsecas
Instrucciones	Asignación aritmética Asignación lógica IF aritmético IF lógico ELSE END GO TO simple CONTINUE
Definición y referencia de datos	Referencia de datos no recursiva, excepto recursividad de primer orden

Tabla II. Código no vectorizable.

Rango del código objeto	Lazos DO WHILE Lazos DO UNIT Lazos IF/GO TO
Tipo de datos	LOGICAL*1 INTEGER*2 REAL*16 COMPLEX*32 CHARACTER
Factores de las instrucciones	Referencias a funciones externas
Instrucciones	CALL Instrucciones de E/S GO TO computado GO TO asignado Asignación de un número de instrucción PAUSE RETURN STOP END
Definición y referencia de datos	Referencia de datos recursiva, excepto recursividad de primer orden

Es importante comentar que las técnicas expuestas en este apartado son técnicas generales aplicables en un principio a cualquier código que se quiera vectorizar. Con estas herramientas se persigue la máxima utilización de los recursos de un supercomputador vectorial, independientemente del problema que se esté tratando. Pero las implementaciones basadas únicamente en este tipo de técnicas no conseguirán obtener la máxima eficiencia, ya que no se tienen en cuenta las propias características del problema en cuestión. Lo ideal sería entrever aquellas operaciones inherentes al problema que son susceptibles de ser vectorizadas y proceder a las transformaciones necesarias en el código utilizado para la proyección. Esto es a lo que nos dedicaremos básicamente en el apartado siguiente cuando tengamos en cuenta la topología de la RNA.

Tabla III. Técnicas de vectorización básicas.

Técnicas de vectorización	Comentario
Expansión escalar	Consiste en una trasnsformación del código aplicable a variables escalares dentro de un lazo. Se crea una nueva variable, un vector unidimensional, que sustituye a la original.
Intercambio de lazos	Se basa en el intercambio de lazos anidados para conseguir, ya sea una correcta lectura de matrices o la ejecución del lazo largo cómo lazo interno.
Unión de lazos	Cuando tenemos lazos que poseen los mismos límites en sus índices resulta interesante unirlos en uno solo para utilizar lo máximo posible los *pipelines* del *hardware*.
Unrolling	Consiste en la repetición del cuerpo del lazo, modificando el recorrido de los índices apropiadamente, para ejecutar dos o más instrucciones iguales sobre diferentes datos en una misma iteración.

2.4. Las RNA y el interés por su vectorización

Llegados a este punto, donde ya hemos descrito las potencialidades de las arquitecturas vectoriales, la cuestión radica en saber si es o no interesante la proyección de RNA sobre un supercomputador de esas características. Concretando, debemos analizar si la ganancia potencial que implican los supercomputadores y la vectorización garantiza el coste y el esfuerzo en la programación que ello supone [Thomborson, 1993]. Para responder a esta pregunta, vamos a exponer las características más importantes que definen la simulación de RNA en ordenadores de

propósito general, y cómo se utilizarían los recursos de un supercomputador vectorial para obtener la máxima eficiencia en la simulación:

a) cálculo paralelo masivo: el cálculo correspondiente a las neuronas de una misma capa puede realizarse de modo independiente y en el mismo instante t. En este sentido puede utilizarse toda la potencia de los *pipelines* después del tiempo de arranque.

b) bifurcaciones y decisiones en la implementación: las decisiones que hay que tener en cuenta serán solamente las correspondientes a las condiciones de parada y las correspondientes, por ejemplo, a qué valores toma la función salto, cuando se utiliza cómo función de activación. Esto nos permite la vectorización de la gran mayoría de los lazos de cálculo de la proyección sin tener que resolver los problemas de interrupción que plantean los saltos condicionales.

c) tamaño de los operandos: las redes utilizadas para afrontar algunos problemas importantes de clasificación, por ejemplo, pueden trabajar con matrices de tamaño elevado. Cuanto mayores sean los operandos mayor será la relación entre el tiempo total de iniciación y el tiempo de arranque, con lo que el rendimiento conseguido será muy alto.

d) necesidad de grandes cantidades de memoria: para almacenar los datos, especialmente durante el proceso de aprendizaje, se necesita una gran capacidad de almacenamiento. En este sentido la alta capacidad de almacenamiento de un supercomputador (del orden de Gbytes) resulta una característica fundamental.

e) tipos de aplicaciones en las que se utilizan: entornos donde se necesita trabajar en tiempo real.

f) tipos de operaciones implicadas en la implementación: básicamente sumas, restas, multiplicaciones y divisiones, que son procesables de una manera muy eficiente aprovechando los *pipelines* de un supercomputador vectorial.

Con todos estos puntos expuestos, podemos llegar a la conclusión de que las arquitecturas vectoriales y la vectorización representan una vía adecuada para solucionar algunos problemas asociados al coste computacional de las RNA. A partir de ahora vamos a analizar qué hay de vectorizable en las redes a nivel de topología, para bajar posteriormente a la vectorización de sus componentes.

3. LA TOPOLOGÍA DE LAS RNA Y SUS CARACTERÍSTICAS: EL CÁLCULO PARALELO

Analizadas las particularidades de las arquitecturas vectoriales y cómo éstas pueden dar, a priori, respuesta a las necesidades computacionales de las RNA, vamos

a adentrarnos en las características topológicas de las redes. El estudio se realizará para redes cuyas neuronas están organizadas en capas (no recurrentes). La metodología a seguir será la siguiente: primero buscaremos aquellos procesos de las RNA que se puedan realizar en un mismo instante de tiempo (paralelismos); en segundo lugar, proyectaremos dichos procesos en un lenguaje de programación (FORTRAN) aglutinando los paralelismos en diferentes lazos. Con esto conseguimos que, dado el carácter concurrente de las operaciones involucradas, dichos lazos sean fácilmente vectorizables.

Teniendo esto en cuenta, vamos a introducir en este apartado una definición de lo que entendemos por «procesos paralelos» cuando estamos trabajando con procesadores vectoriales. En la figura 3 exponemos los paralelismos evidentes que se pueden tener en cuenta a la hora de programar redes genéricas.

Estos procesos serían fácilmente implementables en arquitecturas paralelas, pero la explotación de su paralelismo no es tan evidente en una proyección vectorial.

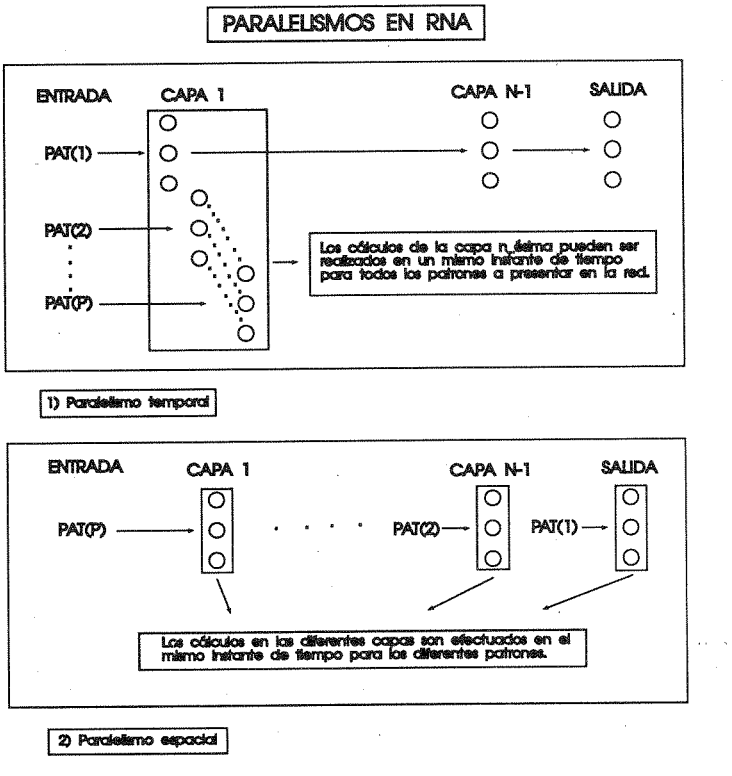


Figura 3. Paralelismos en RNA.

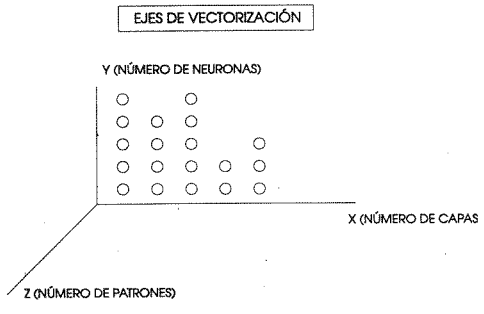
Después mostraremos algunas técnicas que permiten solucionar este problema. Pero antes, vamos a introducir unos conceptos que nos parecen interesantes para entender dichas técnicas: los ejes de vectorización, el «paralelismo» en arquitecturas vectoriales y las regularidades en RNA.

3.1. Conceptos: ejes de vectorización, «paralelismo» en arquitecturas vectoriales y regularidades en RNA

a) Ejes de vectorización.

Dada una red genérica distribuida por capas, entre los parámetros que definen su topología están el número de capas y el número de neuronas por capa. Estas características definen a nivel espacial una red. Se podría también decir que a nivel temporal una red podría estar definida por el número de patrones que van a ser procesados por la estructura. Si representáramos el nivel «espacial» (determinado por el número de capas y el número de neuronas por capa) y el nivel «temporal» (determinado por el número de patrones) en unos ejes coordenados, nos encontraríamos en una situación como la que exponemos en la figura 4.

Figura 4. Ejes de vectorización.



En estos ejes coordenados, en el eje X representaríamos el número de capas de una red, en el eje Y el número de neuronas por capa, y en el eje Z el número de patrones a procesar. A estos tres ejes los vamos a denominar ejes de vectorización, y nos serán de gran ayuda para visualizar cómo trabajan las dos técnicas de programación ideadas para explotar el paralelismo de las RNA.

b) «Paralelismo» en arquitecturas vectoriales.

A la hora de tratar con arquitecturas vectoriales entenderemos como «procesos paralelos» aquellos cálculos que a nivel de programación se realizan en un mismo lazo.

Tenemos que matizar que en una situación de varios lazos anidados, el «paralelismo» sólo se consigue en el lazo más interno y en casos excepcionales en los dos más internos. Además, la dependencia entre los lazos de un mismo grado puede deteriorar el grado de paralelismo de las operaciones. Aún así, la definición es coherente desde el punto de vista vectorial, teniendo en cuenta que los cálculos a realizar en un mismo lazo van a aprovechar los *pipelines* de la arquitectura, en los cuales, sí que se realiza de alguna manera cálculo concurrente después del rellenado de dichos *pipelines*.

Es interesante observar la conexión, a través de los lazos del código utilizado, entre el paralelismo real de la red (conjunto de operaciones que se realizan en el mismo instante t) y el «paralelismo» virtual en un supercomputador vectorial (conjunto de operaciones diferentes correspondientes a los segmentos de una unidad funcional, que trabajan sobre diferentes datos en un mismo instante de tiempo). Esto se expone en la figura 5.

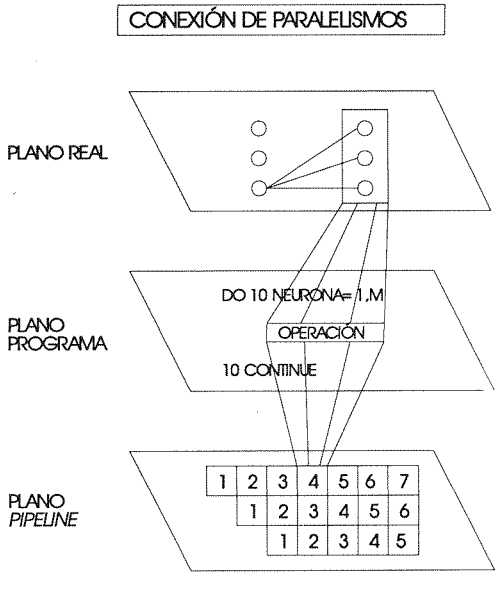


Figura 5. Conexión entre los paralelismos de la RNA y de un *pipeline*.

c) Regularidades en RNA.

Las regularidades de las RNA nos dan información acerca de las características topológicas de las redes. El concepto de regularidad está íntimamente asociado a la relación entre la cantidad de neuronas existente en cada capa. Introduciremos dos definiciones:

Redes Regulares: Son aquellas que poseen un número similar de neuronas por capa.

Redes Irregulares: Son aquellas que poseen un número dispar de neuronas por capa.

3.2. Estrategias de programación atendiendo a la topología de la red

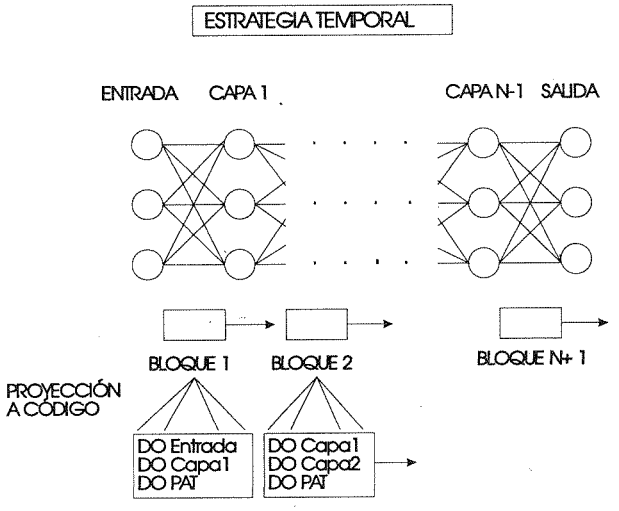
Después de ilustrar los paralelismos que podemos encontrar en una red genérica y de introducir una serie de conceptos, vamos a explicar cómo se pueden explotar dichos paralelismos a través de una arquitectura vectorial. Como comentamos en la introducción a este apartado y teniendo en cuenta la definición de «procesos paralelos», hablar de paralelismos en las RNA es lo mismo que hablar de cálculos realizados en el interior de un lazo en un lenguaje de programación (FORTRAN para nuestro caso) sobre una arquitectura vectorial. Se perseguirá que los paralelismos se extiendan al mayor número de procesos, ya que así los lazos correspondientes a su proyección *software* trabajarán con índices mayores. Trabajar con índices grandes en un lazo significa realizar operaciones con vectores de gran tamaño. De esta manera, se pueden conseguir óptimas eficiencias a nivel de los *pipelines* de la máquina (ver apartado 2.3).

Teniendo estas premisas en cuenta vamos a exponer dos tipos de estrategias que dependen del eje o los ejes sobre los cuales estamos vectorizando. Las denominamos: estrategia temporal y estrategia espacial, siguiendo la línea argumental introducida al comienzo del apartado 3.

a) Estrategia temporal

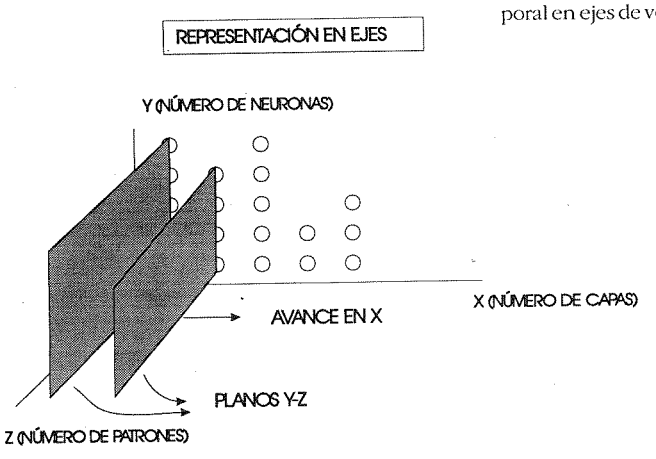
Consiste básicamente en aprovechar el paralelismo temporal, determinado por el número de patrones, vectorizando sobre los planos Y-Z para avanzar posteriormente a lo largo del eje X. Dicho con otras palabras, consiste en vectorizar todos los cálculos para una capa con los diferentes patrones de entrada a la red antes de realizar operaciones en capas subsiguientes. Se podría decir entonces que estamos planteando una estrategia secuencial a nivel global de toda la red (cálculo para la primera capa, cálculo para la segunda capa, ...), con cada módulo de la secuencia convenientemente «paralelizado». Esto se representa en la figura 6. En esta ilustración se expone también como tendría que realizarse la proyección a nivel de un lenguaje de programación. Para recorrer la red hemos utilizado tres índices: I, NEURONA y PAT. Los dos primeros determinan la capa que estamos procesando, fijando un valor en el eje X, y recorren el eje Y para barrer todas las neuronas de cada capa. El último determina el patrón que estamos utilizando y recorre el eje Z. Se observa que los lazos tendrían que anidarse de tal manera que los correspondientes a los índices PAT y NEURONA fueran los más internos. Esto es fácil de entender si tenemos en cuenta que lo que queremos «paralelizar» son los cálculos relacionados con el plano Y-Z, y sólo los lazos internos son aquellos que hemos definido como «paralelizables».

Figura 6. Estrategia temporal.



A nivel de representación, ya sólo nos queda proyectar la estrategia temporal en nuestros ejes de vectorización. El dibujo se representa en la figura 7.

Figura 7. La estrategia temporal en ejes de vectorización.



En primer lugar fijamos el valor 1 en el eje X (número de capas de la red). Este número nos indica que vamos a empezar por procesar la capa número 1. A continuación fijamos el valor 1 en el eje Y (número de neuronas por capa), que nos indica que vamos a comenzar con la primera neurona de la capa antes fijada. Por último, recorremos el eje Z (patrones de la red) para operar con todos los patrones con los que estamos trabajando. El paso siguiente sería saltar al valor Y=2 (segunda

neurona de la capa 1), calcular para todos los patrones en el eje Z, y así sucesivamente. De esta manera vamos rellenando el plano X=1, paralelo al Y-Z, con líneas que recorren el eje Z, cubriendo el número total de patrones. Una vez terminado este proceso (bloque 1 de la figura 6), fijamos el valor de X en 2 y repetimos lo explicado anteriormente. Este proceso se propagaría en la dirección X hasta llegar a la capa de salida.

Por último, solamente nos queda estudiar la dependencia de esta estrategia con la regularidad de la red. Por una parte sabemos que la regularidad de la red es una característica puramente espacial. Por otra parte nuestra estrategia temporal está basada en la vectorización en el eje Z, donde hemos situado el índice PAT. Por tanto, la eficiencia de esta técnica va a ser independiente del grado de regularidad de la red[1].

Como resumen se puede observar que lo que hemos tratado de conseguir ha sido una vectorización sobre el eje de patrones, es decir, sobre el lazo que recorre el índice PAT. La explicación radica en que normalmente el número de patrones es mucho mayor que el número de capas o de neuronas por capa (ejes X e Y), con lo que la eficiencia en la vectorización será mayor en el lazo que recorre PAT porque es el lazo de mayor longitud. También se puede comentar que se pueden aplicar las técnicas de vectorización como el «*unrolling*» y la de «expansión escalar» en el interior del lazo PAT. Pero esto lo analizaremos más en detalle cuando hablemos de cómo vectorizar las componentes de las RNA.

A modo de comentario final decir que la estrategia temporal aquí explicada es la más intuitiva de implementar, y explota el paralelismo más evidente, como es el relacionado con los patrones. Además, resulta la más interesante cuando se disponen de todos los patrones antes de realizar los cálculos en la red. En situaciones en las cuales no disponemos de todos los patrones a procesar en el instante inicial (por ejemplo, en situaciones de tratamiento de señales en tiempo real), la estrategia temporal no sería la más óptima ya que tendríamos que esperar por todos los patrones para propagar el procesamiento capa a capa. En este caso cabe plantearse una nueva estrategia: la estrategia espacial.

b) Estrategia espacial

En contraposición a la estrategia temporal, la estrategia espacial va a explotar el paralelismo del mismo nombre, vectorizando sobre los planos X-Y para después

---

1 Cuando comparamos la eficiencia entre dos redes estamos comparando el tiempo que se tarda en realizar el mismo proceso sobre ambas redes, suponiendo que el número de operaciones involucrado es constante. Por ejemplo, una red regular 4-4-4 implica (4*4)+(4*4)=32 operaciones, y se podría comparar con una red irregular 2-8-2, también de (2*8)+(2*8)=32 operaciones; la similitud es válida siempre y cuando consideremos la aproximación de que los tiempos de computación para la función de activacion y la función de combinación son similares.

avanzar sobre el eje Z a modo de *pipeline*. El objetivo de esta técnica radica en conseguir que las diferentes capas de la red estén procesando datos simultáneamente, aunque sobre diferentes patrones para cada una de las capas. La implementación se llevaría a cabo en un único bloque de lazos anidados, donde se tendría en cuenta la capa a procesar para conocer el patrón correspondiente. La estrategia así planteada estaría segmentada básicamente a lo largo del eje Z correspondiente al número de patrones. Esto se ilustra en la figura 8.

Vemos que el tiempo de arranque de nuestro *pipeline* coincidirá con el número de capas de nuestra red. El paralelismo se consigue si tenemos en cuenta que, rellenado el *pipeline* (o, visto de otro modo, rellenada nuestra red con los primeros patrones), los cálculos se van a realizar en un mismo instante de tiempo para cada uno de los diferentes patrones sobre los que esté operando la red.

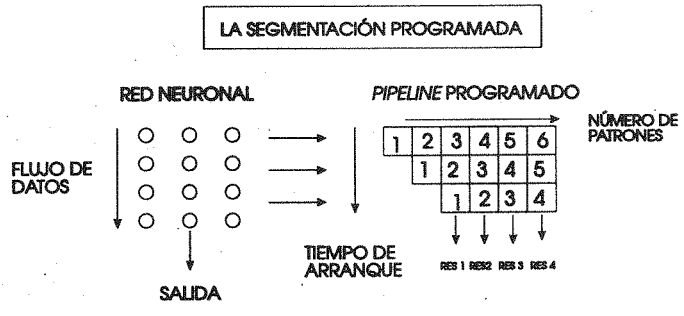


Figura 8. *Pipeline* programado.

Es fundamental entender que este *pipeline* creado a nivel de *software* es totalmente diferente al *pipeline* real de la máquina. Cuando hablamos de que el *pipeline* ideado con nuestra técnica consigue un cierto grado de paralelismo, nos estamos refiriendo al «paralelismo» de lazo que hemos definido en nuestro apartado de conceptos. Es decir, estamos consiguiendo realizar «procesos paralelos» porque hemos introducido en un mismo lazo las operaciones involucradas para las N capas de la red operando cada una sobre datos asociados a diferentes patrones de entrada. Esto es proyectable a nivel de programación en un único bloque con 3 lazos anidados para los índices PAT, I y NEURONA, cuyo mecanismo de operación se puede dividir en subbloques que se entenderían como sublazos que procesan cada capa con su patrón correspondiente. También se puede observar que los lazos más internos se corresponden con los índices NEURONA e I, cuestión evidente si queremos conseguir el «paralelismo» en planos paralelos al X-Y.

El mecanismo de operación de la implementación (figura 9) se inicia fijando el valor 1 para I (selección de la primera neurona de entrada para cada capa) y después recorriendo el índice NEURONA en el eje X.

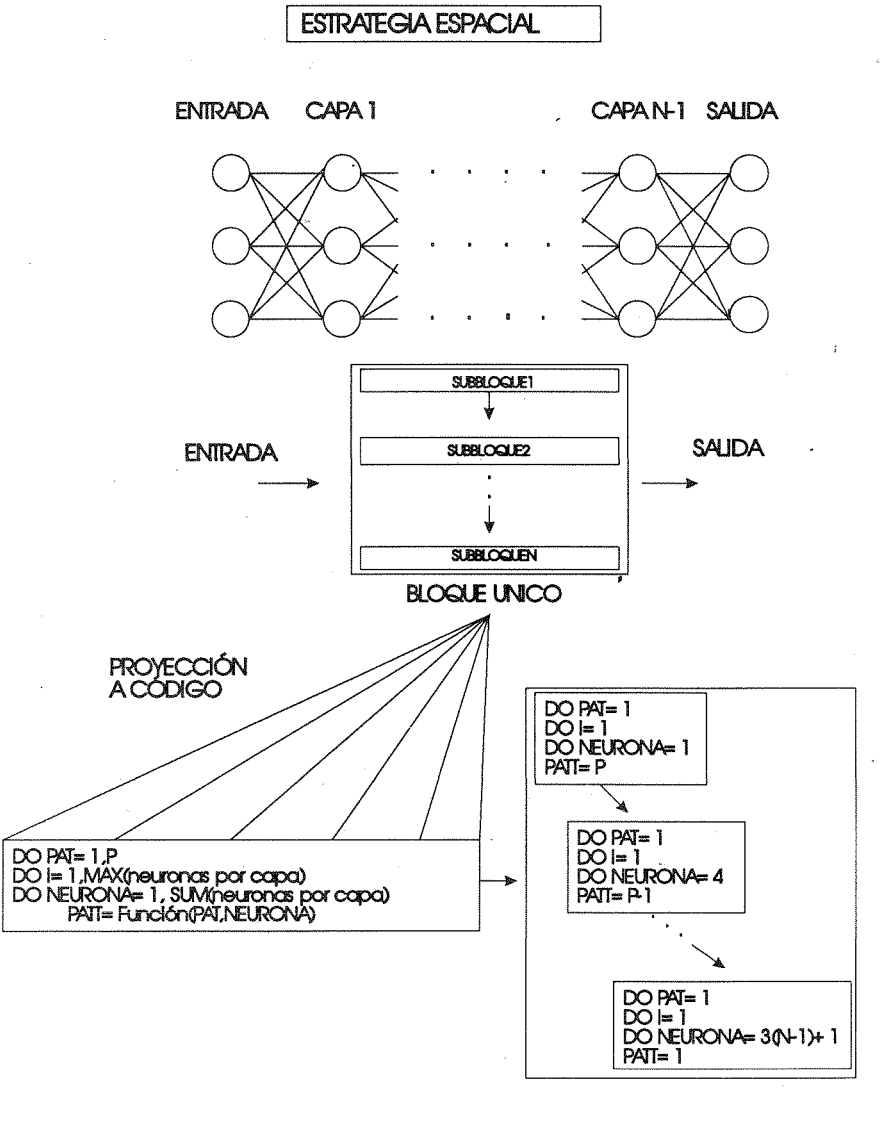


Figura 9. Estrategia espacial.

El índice NEURONA representa una neurona «excitada» para cada una de las capas de la red. Es decir, fijado el valor I=1 para la capa 1, el índice NEURONA recorrerá todas las neuronas de la capa 2 para calcular sus conexiones con la neurona I=1 de la capa 1. Cuando NEURONA termine con la capa 2 pasará a la capa 3 con

lo que automáticamente se fijará I=1 pero en este caso para la capa 2. Este proceso se repite propagándose hacia todas las capas y trasladando los índices NEURONA e I de una capa a otra para que recorran sus respectivas neuronas «excitadoras» y «excitadas».

Por último, representamos la estrategia espacial en función de los ejes de vectorización en la figura 10.

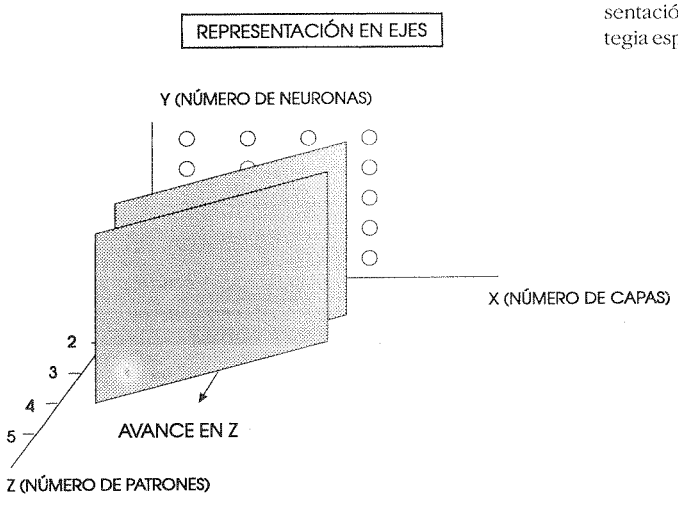


Figura 10. Representación de la estrategia espacial en ejes.

Supongamos de partida que hemos «rellenado la red» con diferentes patrones para cada capa (en un proceso similar al que se lleva a cabo en los *pipelines* de la máquina). Para el caso sencillo de dos capas, tal y como hemos expuesto en la figura 11, la orientación de los planos que avanzan sobre el eje Z de patrones no es totalmente paralela al plano X-Y. Esta orientación es debida a que para cada iteración a lo largo de X y de Y (relacionados con los índices NEURONA e I, respectivamente) el patrón que va a procesar cada capa será diferente. Así, una vez calculado el patrón 1 para la capa 1, la siguiente iteración se encargará del cálculo del patrón 1 para la capa 2 y del patrón 2 para la capa 1, después seguirá el patrón 2 para la capa 2 y el patrón 3 para la capa 1, y así sucesivamente. Aún así, podemos decir que estamos realizando «paralelismo» en el plano X-Y y que éste se propaga a lo largo del eje Z.

A nivel de vectorización, lo que estamos persiguiendo no es ni más ni menos que introducir los índices I y NEURONA como internos, extendiendo su recorrido a todas las neuronas de la red que puedan actuar como neuronas «excitadoras» (caso del índice I) o a aquellas que puedan actuar como neuronas «excitadas» (caso del índice NEURONA).

Nos queda comentar la dependencia de la estrategia espacial con la regularidad de la red. La eficiencia de este tipo de técnica va a depender en gran medida del grado de regularidad de la red. Como estamos vectorizando en el eje Y, el lazo que lo recorre tendrá que realizar tantas iteraciones como neuronas tengamos en la capa de mayor número. Esto significa que para los lazos asociados a aquellas capas donde el número de neuronas no coincida con el máximo, se estarán realizando operaciones sobre neuronas que realmente no existen (operaciones por cero). Evidentemente esto puede reducir considerablemente la eficiencia de la implementación. Así, en redes irregulares donde el número de operaciones desaprovechado puede ser muy elevado se desaconseja esta metodología de implementación.

4. LAS COMPONENTES DE LAS RNA Y SU FUNCIONAMIENTO: LAS OPERACIONES DE COMBINACIÓN Y DE ACTIVACIÓN

Ya hemos visto cómo podemos acomodar la implementación de la topología de las RNA a arquitecturas vectoriales. Ahora nos toca afrontar cómo proyectar las características de las componentes de una red a nivel de programación. Cuando hablamos de componentes nos referimos a las neuronas que forman parte de la estructura. A nivel funcional cada neurona realiza dos tareas: en primer lugar procesa los datos de entrada para obtener información del estado de otras neuronas; en segundo lugar procesa la información anterior para determinar su propia salida ante las otras neuronas. El cálculo asociado a la primera tarea lo realiza una función denominada de combinación, mientras que el cálculo asociado a la segunda corre a cargo de otra función denominada de activación. Para cada una de ellas nos podemos encontrar múltiples casos particulares, pero con las mismas características básicas. Estas son las que vamos a analizar para realizar una óptima implementación de dichas funciones. Para llevar a cabo esto, podemos hacer un tratamiento de las componentes independiente al tratamiento global de la estructura, lo cual puede ser interesante si sólo se persigue una vectorización rápida de la implementación. Alternativamente, podemos hacer un tratamiento acorde a las estrategias que comentamos en el apartado anterior, y que tienen en cuenta las propiedades topológicas de las RNA. Discutiremos ambos tratamientos en las siguientes secciones.

4.1. Las funciones de combinación

En general las funciones de combinación obedecen a expresiones del tipo:

$$net_j = f_j\ (x_0, x_1, \ldots, x_M) = f_j(X) \qquad (1)$$

donde x_0, ..., x_M son las entradas a la neurona.

La función denotada por f se encarga de realizar una ponderación de las diferentes entradas a través de los pesos de la red (conexiones). Esta ponderación puede realizarse a través de una multiplicación entre las entradas y los pesos (por ejemplo, la función de combinación lineal), o a través de cálculos más complejos (por ejemplo, la función de media varianza) [Regueiro y col., 1995]. En ambos casos, el funcionamiento es similar, ya que de lo que se trata es de sumar las distintas entradas a la neurona, eso sí, con diferentes ponderaciones. La forma más usual de realizar dicha suma en pseudocódigo FORTRAN, a través de un tratamiento individual de la función de combinación, es:

```
      DO 10 NEURONA=1,num_neuronas_capa_n
            NET = 0.0
            DO 20 I=1,num_neuronas_capa_n-1
                  NET = NET + (PONDERACIÓN DE x(I) POR
                              w(NEURONA,I))
20          CONTINUE
            Salida(NEURONA) = Función de activación(NET)
10    CONTINUE
```

En este código utilizamos NET para almacenar temporalmente los valores de la función de combinación para cada neurona. Tan pronto calculamos NET para una neurona se procesa la función de activación de dicha neurona y se deja que NET pueda ser utilizada en otra iteración. Uno de los inconvenientes que plantea esta solución es que los cálculos de las funciones de combinación y de activación no están debidamente separados. Esto se consigue con esta variante:

```
      DO 10 NEURONA=1,num_neuronas_capa_n
            SUM = 0.0
            DO 20 I=1,num_neuronas_capa_n-1
                  SUM = SUM + (PONDERACIÓN DE x(I) POR
                              w(NEURONA,I))
20          CONTINUE
            NET(NEURONA) = SUM
10    CONTINUE
```

En este caso utilizamos un vector NET(NEURONA) donde almacenamos los valores de la función de combinación de cada neurona. Esto nos permite diferenciar las fases de combinación y de activación. Pero aún cabría un ajuste más. Podríamos utilizar desde un principio un vector para cada capa que se encargara de almacenar todos los cálculos. Tendríamos entonces:

```
      DO 10 NEURONA=1,num_neuronas_capa_n
            NET(NEURONA)=0.0
10    CONTINUE
      DO 20 NEURONA=1,num_neuronas_capa_n
            DO 30 I=1,num_neuronas_capa_n-1
                  NET(NEURONA) = NET(NEURONA) + (PONDERACIÓN
                                    DE x(I) POR w(NEURONA,I))
30          CONTINUE
20    CONTINUE
```

A esta técnica se la conoce cómo expansión escalar, ya que estamos reemplazando una variable escalar por un vector. También sería interesante «paralelizar» el cálculo de varias neuronas de una capa. Esto se consigue repitiendo la instrucción donde procesamos la función de combinación para varias neuronas. A esto se le conoce con el nombre de *unrolling*:

```
      DO 10 NEURONA=1,num_neuronas_capa_n
            NET(NEURONA)=0.0
10    CONTINUE
      DO 20 NEURONA=1,(num_neuronas_capa_n)-1,2
            DO 30 I=1,num_neuronas_capa_n-1
                  NET(NEURONA) = NET(NEURONA) + (PONDERACIÓN
                                    DE x(I) POR w(NEURONA,I))
                  NET(NEURONA+1) = NET(NEURONA+1) +
                              (PONDERACION DE x(I) POR w(NEURONA,I))
30          CONTINUE
20    CONTINUE
```

En este caso el índice NEURONA recorre su dominio barriendo dos valores en cada ciclo. A continuación I fija una neurona de la capa anterior y se calcula NET para dos neuronas diferentes. La técnica de *unrolling* se aplica a NEURONA por ser éste el índice más externo y, por tanto, el menos susceptible a ser vectorizado.

Todo lo que acabamos de explicar es importante porque nos da una idea de cómo adecuamos la programación sobre arquitecturas vectoriales a las características de las componentes de las RNA. Pero si buscamos optimizar la implementación global de la red, tendremos que tener en cuenta qué implicaciones tiene en la proyección de las componentes la aplicación de las estrategias temporal y espacial. Un análisis global nos introduce la dimensión temporal especificada por los patrones.

Esto significa trabajar con un lazo más a nivel de programación. También supone la utilización de matrices para almacenar los valores de NET, ya que tenemos que poseer información sobre qué neurona estamos actuando y sobre qué patrón estamos procesando. Resumiendo, las restricciones de las que partimos son:

Estrategia temporal

```
DO 10 I=1,...
        DO 20 NEURONA=1,...
                DO 30 PAT=1,...

                NET(NEURONA,PAT) = NET(NEURONA,PAT) + ...
```

Estrategia espacial

```
DO 10 PAT=1,...
        DO 20 I=1,...
                DO 30 NEURONA=1,...
```

Para la proyección de ambas estrategias se van a utilizar básicamente las mismas técnicas de vectorización: el intercambio de índices y el *unrolling*. La primera consiste en leer las matrices más importantes del lazo por columnas, ya que el FORTRAN almacena de esta manera los elementos matriciales en memoria. La segunda ya la hemos comentado previamente. La implementación de ambas técnicas será diferente para las dos estrategias, porque el anidamiento de los lazos también lo es. Para el caso temporal tendremos:

```
        DO 10 NEURONA=1,(num_neuronas_capa_n)-1,2
                DO 20 PAT=1,P
                        NET(PAT,NEURONA)=0.0
                        NET(PAT,NEURONA+1)=0.0
20              CONTINUE
10      CONTINUE
        DO 30 I=1,num_neuronas_capa_n-1
                DO 40 NEURONA=1,(num_neuronas_capa_n)-1,2
                        DO 50 PAT=1,p
                            NET(PAT,NEURONA)=NET(PAT,NEURONA) +
                                (PONDERACIÓN DE x(I) POR w(NEURONA,I))
                            NET(PAT,NEURONA+1)=NET(PAT,NEURONA+1) +
                                (PONDERACIÓN DE x(I) POR w(NEURONA,I))
50                      CONTINUE
40              CONTINUE
30      CONTINUE
```

donde una vez fijada la columna de la matriz NET a través del índice NEURONA, se procede a recorrer sus filas a través de PAT (accediendo de esta manera a posiciones consecutivas de memoria). Esta es una buena forma de mejorar la eficiencia de la

implementación vía el acceso a los datos en memoria. También hemos aplicado el *unrolling* al índice NEURONA, por ser este el más externo y el que en principio no estará vectorizado.

Para el caso espacial el código se complica al tener que calcular qué patrón tiene que procesar cada neurona dependiendo de la capa con la que estemos trabajando. El código con las técnicas utilizadas sería:

```
        DO 10 PAT=1,P-1,2
              DO 20 NEURONA=1,SUM(neuronas por capa)
                    NET(NEURONA,PAT)=0.0
                    NET(NEURONA,PAT+1)=0.0
20            CONTINUE
10      CONTINUE
        DO 30 PAT=1,P
              DO 40 I=1,MAX(neuronas por capa)
                    DO 50 NEURONA=1,SUM(neuronas por capa)-1,2
                          PATT=Función(PAT,NEURONA)
                          NET(NEURONA,PATT)=NET(NEURONA,PATT) +
                                (PONDERACIÓN DE x(PATT,I) POR
                                w(NEURONA,I))
                          PATT=Función(PAT,NEURONA+1)
                          NET(NEURONA+1,PATT)=NET(NEURONA+1,PATT)+
                                (PONDERACIÓN DE x(PATT,I) POR
                                w(NEURONA+1,I))
50                  CONTINUE
40            CONTINUE
30      CONTINUE
```

Observamos que los índices para NET están cambiados respecto a la estrategia temporal. Esto es debido a que el anidamiento de los lazos es diferente y que, por tanto, el recorrido de los índices PAT y NEURONA se ha invertido. También hay que comentar que hemos realizado *unrolling* sobre PAT en el primer bloque de lazos y sobre NEURONA en el segundo bloque. Como hemos comentado previamente el *unrolling* se debe realizar sobre el índice más externo. Por tanto la utilización de esta técnica es correcta para el primer bloque. Entonces, ¿por qué no hemos hecho lo mismo para el segundo bloque?. Si hubiéramos realizado *unrolling* sobre PAT, significaría que no sólo tendríamos un patrón diferente para cada capa de la red sino dos, el propio y el siguiente, éste último ya precisándose en la capa anterior. Esto implica dependencias entre las variables (no podemos operar sobre el patrón p de

la capa n antes de que se procese en la capa n-1) rompiéndose el paralelismo para cada iteración del lazo interno. Realizando *unrolling* sobre NEURONA no tenemos este problema pero la eficiencia es inferior ya que estamos trabajando sobre el índice vectorizado. Otra posibilidad, quizás más interesante que la anterior, sería aplicar la técnica al lazo intermedio, vectorizando sobre I.

4.2. Funciones de activación

Asumiendo que, en general, las funciones de activación obedecen a expresiones del tipo:

$$x_j = g_j\ (net_j) \qquad (2)$$

donde la función denotada por g puede ser de muchos tipos: sigmoide, paso, lineal, rampa, etc., y su implementación en pseudo código, al margen de estrategias de implementación globales, será:

```
        DO 10 NEURONA=1,(num_neuronas_capa_n)-1,2
              Salida(NEURONA)=Función de activación(NET(NEURONA))
              Salida(NEURONA+1)=Función de activación(NET(NEURONA+1))
10      CONTINUE
```

donde ya hemos introducido el *unrolling*.

Pero, ¿qué pasa cuando tenemos en cuenta las restricciones planteadas por las implementaciones globales?. En primer lugar tendremos que incluir un lazo que recorra los diferentes patrones, lo que implica la utilización de matrices al igual que cuando hablábamos de las funciones de combinación. Para el caso de la técnica temporal, podemos expresar los dos tipos de funciones de activación que hemos considerado de la siguiente manera:

```
        DO 10 NEURONA=1,(num_neuronas_capa_n)-1,2
              DO 20 PAT=1,P
                    Salida(PAT,NEURONA)=Función de
                          activación(NET(PAT,NEURONA))
                    Salida(PAT,NEURONA+1)=Función de
                          activación(NET(PAT,NEURONA+1)
20            CONTINUE
10      CONTINUE
```

donde hemos realizado *unrolling* sobre NEURONA y situado los índices de las matrices acorde a lo establecido para las funciones de combinación. Si analizamos ahora cómo proyectaríamos las funciones de activación sobre la estrategia espacial:

```
        DO 10 PAT=1,P
              DO 20 NEURONA=1,SUM(neuronas por capa)-1,2
                    PATT=Función(PAT,NEURONA)
                    Salida(NEURONA,PATT)=Función de
                          activación(NET(NEURONA,PATT))
                    PATT=Función(PAT,NEURONA+1)
                    Salida(NEURONA+1,PATT)=Función de
                          activación(NET(NEURONA+1,PATT))
20            CONTINUE
10      CONTINUE
```

y nos encontramos ante el mismo problema que el que tuvimos que afrontar para la proyección de la función de combinación en la estrategia horizontal: la dependencia con la variable PAT si realizamos *unrolling* sobre ella. Por ello, planteamos nuevamente la alternativa de aplicar dicha técnica sobre NEURONA, con las desventajas ya comentadas.

Con esto terminamos nuestro estudio sobre las componentes de las RNA. Hemos visto qué técnicas son aplicables cuando las tratamos por separado y cuando las tratamos bajo estrategias globales como la temporal y la espacial. Con muestras de código genérico de funciones de combinación y de activación hemos razonado las conveniencias y la viabilidad de unas u otras. Nos hemos dado cuenta de que las características de cada una de las estrategias de implementación obligan a la modificación del código y a la selección de las técnicas empleadas.

Ya sólo nos queda el último apartado, la proyección del algoritmo *BackPropagation*, en su modalidad *Batch*, sobre una red *Perceptron* Multicapa. Aquí expondremos experimentalmente, con medidas de tiempos, la validez de las estrategias presentadas en este capítulo. También comprobaremos qué pasa con redes irregulares, y todo ello tomando como punto de referencia una implementación puramente secuencial basada en la vectorización automática del compilador vectorial, en la que no se ha atendido ni a paralelismos de la estructura ni a técnicas que exploten las características del vectorial.

Se hace necesario comentar que el trabajo de investigación no se termina aquí. Nuestro estudio completo, no expuesto en estas páginas por el enfoque editorial que persigue una publicación a la vez divulgativa y científica, continuaría con otros puntos que ya hemos estudiado parcialmente y que resultan de enorme interés: la

vectorización de otros tipos de funciones de activación, el análisis de las técnicas expuestas para redes recurrentes y no recurrentes en general, y el tratamiento del concepto de entrenamiento bajo la influencia de las estrategias globales aquí planteadas.

5. UN EJEMPLO DE IMPLEMENTACIÓN: LA RED *PERCEPTRON* MULTICAPA Y EL ALGORITMO *BACKPROPAGATION*

Toca el turno de exponer con un ejemplo concreto cómo realizar una implementación completa de una red. La red con la que vamos a trabajar será la más conocida: la *Perceptron* Multicapa. Va a estar constituida por tres capas: entrada, oculta y salida. Investigaremos dos tipos diferentes: una regular y otra irregular. Para el caso regular utilizaremos una 100-100-100 (100 neuronas en cada capa) y una 1000-1000-1000, y para el asimétrico una 10-1000-10 y una 100-10000-100. El tipo de entrenamiento, el más popular: el *BackPropagation* con *Batch*. Este algoritmo aplicado a esta red permite todo tipo de paralelismos con lo cual es un buen marco de estudio de nuestras estrategias y de las diferentes técnicas de vectorización. Las estrategias que vamos a utilizar son: la estrategia de referencia, basada en código secuencial y donde la vectorización corre a cargo enteramente por el compilador vectorial; la estrategia básica, basada en código vectorizado con las técnicas de vectorización generales a cualquier problema; la estrategia temporal y la estrategia espacial. Para calibrar la eficiencia de las diferentes implementaciones calcularemos los tiempos de CPU empleados por el supercomputador para cada caso. Utilizaremos como punto de comparación los tiempos asociados a la implementación de referencia. Al final procederemos a analizar los tiempos con una gráfica comparativa y a comentar los resultados finales.

5.1. El supercomputador vectorial FUJITSU VP-2400/10

Los ejemplos que mostramos a continuación han sido desarrollados sobre el supercomputador vectorial de FUJITSU VP-2400/10 instalado en el CEntro de Supercomputación de GAlicia (CESGA). Las características más relevantes que hay que tener en cuenta para evaluar la eficiencia de las implementaciones son:

a) Registros vectoriales: la serie VP-2400/10 de FUJITSU introduce un gran número de registros vectoriales con el fin de incrementar la velocidad del procesamiento de los datos. El tamaño de estos registros es de 1024 bytes.

b) Registros de máscara: se han implementado este tipo de registros para almacenar información condicional que pueda afectar el flujo de datos en los *pipelines* del vectorial. Con esto se permite la vectorización incluso cuando aparecen sentencias condicionales en el código.

c) Instrucciones *hardware* vectoriales: el VP contiene una unidad escalar y otra unidad vectorial. Para esta configuración disponemos de 240 instrucciones *hardware* escalares y 133 vectoriales. Esto permite reemplazar un gran número de sentencias escalares por sus respectivas vectoriales.

d) Múltiples *pipelines*: dos para multiplicación y suma, y uno para resta y división.

Con estas características el VP-2400/10 puede alcanzar un rendimiento teórico de 2.5 GFLOPS.

5.2. Estudio comparativo

En primer lugar vamos a presentar las tablas (tablas IV-VII) de los tiempos de ejecución para los bloques para las diferentes redes estudiadas. En estas tablas exponemos los datos para las diferentes estrategias de implementación con compilador escalar y vectorial. También incluimos los porcentajes de vectorización obtenidos cuando trabajamos con el compilador vectorial.

Tabla IV. Red 100-100-100.

RED 100-100-100

Tiempo total de ejecución (Fase *forward*+ Fase de actualización) para 1000 patrones		Referencia** Vectorial***	Referencia** Escalar***	Básica** Vectorial***	Básica** Escalar***
	UV	0.4*	0.0	0.2	0.0
	CPU	1.7	10.1	1.4	9.2
		Temporal** Vectorial***	Temporal** Escalar***	Espacial** Vectorial***	Espacial** Escalar***
	UV	0.1	0.0	0.5	0.0
	CPU	0.1	7.2	0.6	8.2
		Referencia** Vectorial***	Básica** Vectorial***	Temporal** Vectorial***	Espacial** Vectorial***
Porcentaje vectorización para cada estrategia		23.5	14.3	100	83.3

* Datos en segundos
** Tipo de estrategia
*** Tipo de compilador
CPU= UV+ US UV= Unidad vectorial US= Unidad escalar

Tabla V. Red 1000-1000-1000.

RED 1000-1000-1000

Tiempo total de ejecución (Fase *forward*+ Fase de actualización) para 1000 patrones		Referencia** Vectorial***	Referencia** Escalar***	Básica** Vectorial***	Básica** Escalar***
	UV	20.5*	0.0	12.5	0.0
	CPU	22.1	2013.2	13.9	909.8
		Temporal** Vectorial***	Temporal** Escalar***	Espacial** Vectorial***	Espacial** Escalar***
	UV	9.5	0.0	61.4	0.0
	CPU	9.5	832.9	61.4	> 10exp9
Porcentaje vectorización para cada estrategia		Referencia** Vectorial***	Básica** Vectorial***	Temporal** Vectorial***	Espacial** Vectorial***
		92.8	90.1	100	100

* Datos en segundos
** Tipo de estrategia
*** Tipo de compilador
CPU= UV+ US UV= Unidad vectorial US= Unidad escalar

Tabla VI. Red 10-1000-10.

RED 10-1000-10

Tiempo total de ejecución (Fase *forward*+ Fase de actualización) para 1000 patrones		Referencia** Vectorial***	Referencia** Escalar***	Básica** Vectorial***	Básica** Escalar***
	UV	1.5*	0.0	0.2	0.0
	CPU	2.9	9.4	1.4	9.6
		Temporal** Vectorial***	Temporal** Escalar***	Espacial** Vectorial***	Espacial** Escalar***
	UV	0.1	0.0	21.5	0.0
	CPU	0.1	9.9	21.6	815.2
Porcentaje vectorización para cada estrategia		Referencia** Vectorial***	Básica** Vectorial***	Temporal** Vectorial***	Espacial** Vectorial***
		51.7	14.3	100	99.5

* Datos en segundos
** Tipo de estrategia
*** Tipo de compilador
CPU= UV+ US UV= Unidad vectorial US= Unidad escalar

Tabla VII. Red 100-10000-100.

RED 100-10000-100

Tiempo total de ejecución (Fase *forward*+ Fase de actualización) para 1000 patrones		Referencia** Vectorial***	Referencia** Escalar***	Básica** Vectorial***	Básica** Escalar***
	UV CPU	37.3* 40.1	0.0 1872.5	19.2 20.6	0.0 300.7
		Temporal** Vectorial***	Temporal** Escalar***	Espacial** Vectorial***	Espacial** Escalar***
	UV CPU	11.2 11.3	0.0 1097.9	mem. mem.	mem. mem.
Porcentaje vectorización para cada estrategia		Referencia** Vectorial***	Básica** Vectorial***	Temporal** Vectorial***	Espacial** Vectorial***
		93.0	93.2	99.1	-

* Datos en segundos
** Tipo de estrategia
*** Tipo de compilador
CPU= UV+ US UV= Unidad vectorial US= Unidad escalar
mem= limitación de memoria en el administrdor de colas del VP

A modo de resumen de los datos obtenidos se presentan gráficas de las diferentes velocidades en *epoch*/min[2] obtenidas para cada red con las diferentes estrategias (figuras 11-12) y en la tabla VIII las eficiencias en MCUPS[3].

Ahora vamos a analizar los datos que poseemos, en dos partes: análisis comparativo de las diferentes estrategias y análisis global.

1) Análisis comparativo de las diferentes estrategias.

- En la estrategia de referencia los tiempos de ejecución de los subbloques de las fases *forward* y actualización (Salida capa oculta y Salida capa salida para la fase *forward*; Validación capa salida, Validación capa oculta, Modificación pesos salida y Modificación pesos oculta) consiguen una vectorización razonable (estos resultados no se exponen en el capítulo). En cambio cuando consideramos los tiempos globales para ambas fases (tiempos correspondientes a los bloques Fase *forward* y

2 Un *epoch* representa una iteración en la que se presentan a la red, actualizandose ésta en función de los resultados, todos los patrones que conforman el conjunto de entrenamiento.

3 MCUPS significa millones de conexiones actualizadas por segundo.

Fase actualización) observamos un gran desajuste entre los tiempos de CPU y UV. Esto es debido a que en la implementación estamos utilizando el lazo que recorre el índice asociado al número de patrones como lazo externo (procesamos patrón a patrón ambas fases sin atender a paralelismos). Éste lazo externo no lo vectoriza el VP y es el causante de los bajos porcentajes de vectorización al final de cada fase y por tanto de unos tiempos elevados.

- En la estrategia básica, hemos optimizado cada bloque aplicando las técnicas ya descritas. Vemos que el porcentaje de vectorización aumenta para cada subbloque con lo que forzamos la disminución del tiempo de ejecución total. Aún así, no podemos evitar la situación de un lazo externo no vectorizado para recorrer los patrones que heredamos de nuestra implementación de referencia. Esto también es la causa de que, si bien mejoramos los tiempos respecto a la estrategia anterior, los resultados sean susceptibles de mejora.

- Y llegamos a la estrategia temporal, producto de enfocar la implementación a partir del estudio de nuestro problema en cuestión. Los tiempos conseguidos con esta metodología son muy buenos y más teniendo en cuenta que los comparamos con un código, el de la estrategia básica, ya vectorizado. ¿Que está pasando?. A tenor de los datos es sencillo de explicar. Para ello tendremos en cuenta dos factores: el proceso de optimización de los subbloques de ambas fases y la optimización de los tiempos globales de cada una de ellas. La primera se consigue básicamente por la reestructuración de los lazos impuesta por la estrategia temporal. Con ello conseguimos leer correctamente (por columnas) casi todas las matrices de los lazos con lo que el tiempo de acceso a memoria disminuye. La segunda es producto de introducir el lazo que recorre los patrones en cada uno de los subbloques. Esto implica que cuando se termina de procesar el último subbloque de cada fase ya hemos terminado con la fase en sí misma. Observamos que ahora no hay lazos totalmente escalares y, por tanto, el porcentaje de vectorización se dispara y el tiempo total de ejecución disminuye. Si sumamos ahora los tiempos de CPU y UV de los subbloques de una fase obtenemos prácticamente el tiempo de ejecución asociado a dicha fase. Esto no ocurría en las estrategias de referencia y básica.

- La estrategia espacial también implica una mejora en la eficiencia tal y como habíamos apostado. Ésta no es tan acentuada como en el caso temporal por la introducción de código extra en los bloques de *pipeline* y modificación de pesos necesario para especificar qué patrón es necesario procesar en cada momento. Es difícil realizar una comparación de cada subbloque con las estrategias anteriores porque la filosofía de la implementación es muy diferente. Por eso, sólo hemos expresado los tiempos de los bloques que sí son comparables. Es interesante comentar que los porcentajes de vectorización conseguidos son muy altos lo que indica, junto a la disminución del tiempo de ejecución, que hemos desarrollado una implementación eficiente.

Figura 11. Gráfica comparativa de las velocidades de las diferentes estrategias en redes regulares.

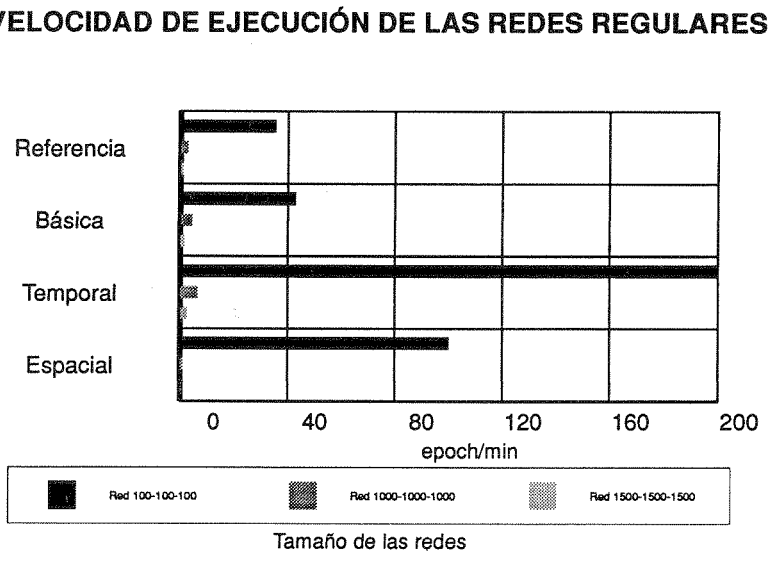


Figura 12. Gráfica comparativa de las velocidades de las diferentes estrategias en redes irregulares.

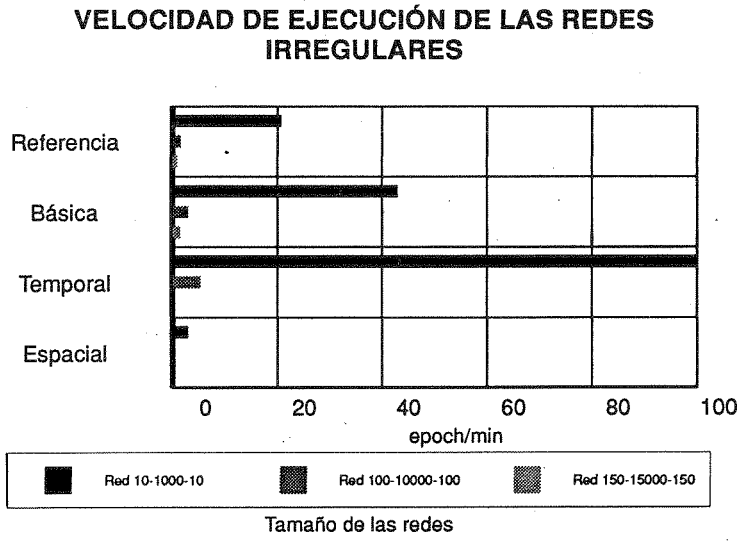


Tabla VIII. Tabla de eficiencias en MCUPS.

MCUPS PARA LAS IMPLEMENTACIONES				
	Referencia*	Básica*	Temporal*	Espacial*
RED 100-100-100	11.6	14.6	142,8	33.9
RED 1000-1000-1000	90.5	143.9	211.2	32,6
RED 10-1000-10	6.9	14.0	141.8	0.92
RED 100-10000-100	49.8	97.1	177.1	-

* Tipo de estrategia

2) Análisis global.

- Salta a la vista la mejora de todos los tiempos de cada estrategia cuando utilizamos el compilador vectorial frente al escalar. Éste no hace uso de las unidades vectoriales con lo que los tiempos de ejecución se disparan.

- Analizando redes de distinto tamaño observamos un rendimiento máximo para las redes 1000-1000-1000. Una posible explicación radica en la configuración dinámica que el VP hace de sus registros. Si el tamaño de los vectores con los que trabajamos implica el semillenado de dichos registros, la eficiencia de la tarea puede disminuir. Este caso es particularmente posible para el caso de redes superiores a la 1000-1000-1000 teniendo en cuenta que el tamaño máximo de los registros es de 1024 bytes.

- La estrategia espacial supone también una mejora en algunos casos respecto a la estrategia básica. Cuando las redes son irregulares los tiempos de ejecución tal y como habíamos postulado aumentan. Para redes regulares no demasiado grandes observamos que también disminuye el rendimiento. Esto es debido al tamaño de las matrices que estamos utilizando (estas matrices contienen la información de toda la red) que puede sobrepasar el tamaño de los registros disminuyendo el rendimiento de estos tal y como hemos explicado en el anterior punto.

- Otro punto importante que no se ve reflejado en las tablas pero que hay que tener en cuenta es el relacionado con el coste de memoria de cada estrategia. Un estudio que hemos desarrollado nos indica que las estrategias temporal y espacial implican un mayor consumo de memoria que la de referencia y la básica. Este

aumento es razonable para redes regulares (de 0.5 Megas para la red 100-100-100 y de 4.0 Megas para la 1000-1000-1000) elevándose en redes irregulares (de 4-10 Megas para redes 10-100-10 hasta 40-800 para las 100-100-100). En general, e independientemente de las estrategia utilizada, las redes irregulares consumen mucha más memoria que las regulares por la disposición inicial de las matrices de datos.

6. CONCLUSIÓN

En estas páginas hemos explicado brevemente el funcionamiento de los supercomputadores vectoriales y su valor como herramientas de simulación de RNA. También hemos expuesto una línea de trabajo basada en la conexión entre los paralelismos reales de las RNA y los «paralelismos» temporales que aparecen en los *pipelines* de un ordenador vectorial. Hemos comentado cómo sería la proyección a pseudocódigo de las funciones de combinación y activación de redes genéricas, comentando las técnicas más recomendables para cada caso. Por último, hemos puesto en práctica las hipótesis de trabajo implementando una RNA completa. Los resultados confirman las expectativas, sobre todo en lo referente a la estrategia temporal, y aumentan nuestro interés por profundizar en el tema.

AGRADECIMIENTOS

Al CEntro de Supercomputación de GAlicia (CESGA), por la utilización del supercomputador vectorial VP-2400/10 de FUJITSU.

REFERENCIAS

Cabestany, J., Moreno, J.M. y Castillo, F., «Implementación VLSI de Redes Neuronales Artificiales», *en este mismo libro,* 1995.

DARPA, *Neural Network study*, AFCEA International Press, 1988.

Dasgupta, Subrata, *Computer architecture: a modern synthesis. Vol. 2, Advanced Topics*, John Wiley & Sons, cop., New York, 1989.

Humpert, B., «A comparative study of neural network architectures», *Computer Physics Communications*, N. 58, (1990), 223-256.

Hush, Don R. y Horne, B., «An overview of neural networks. Part I: static networks. Part II: dynamic networks», *Informática y automática*, Vol. 25, N. 1 y 2, (1992).

Jai-Hoon Chung, Hyunsoo Yoon y Seung Ryoul Maeng, «A systolic array exploiting the inherent parallelisms of artificial neural networks», *Microprocessing and Microprogramming*, N. 33, (1991/92), 145-159.

Lazou, Christopher, *Supercomputers and their use*, Ed. Rev. Oxford, Clarendon, 1988.

Lippmann, Richard P., «An introduction to computing with Neural Nets», *IEEE ASSP Magazine*, abril, (1987), 4-22.

Millán, José del R. y Bofill, Pau, «Learning by back-propagation: a systolic algorithm and its transputer implementation», *Neural Networks*, Vol. 1, N.3, julio, (1989).

Parallel Computing, Especial Redes neuronales, Vol. 14, N. 3, agosto 1990.

Regueiro, C.V., Barro, S., Sánchez, E. y Fernández-Delgado, M., «Modelos básicos de redes neuronales artificiales», *en este mismo libro,* 1995.

Sánchez, E., Barro, S. y Regueiro, C.V. «Artificial Neural Networks implementation on vectorial supercomputers», *IEEE International Conference on Neural Networks,* Orlando, Florida, (1994), 3938-3943.

Sánchez, E., Barro, S. y Regueiro, C.V., «Supercomputación y Redes Neuronales Artificiales», *de próxima publicación*, 1995.

Thomborson, Clark D., «Does your workstation computation belong on a vector supercomputer?», *Communications of the ACM*, Vol. 36, N. 11, noviembre, (1993).

Valderrama, E. y Carrabina, J., «Chips neuronales», *en este mismo libro,* 1995.

Zima, Hans, *Supercomputers for parallel and vector computers*, ACM, cop., New York, 1991.