

# Implementación de funciones booleanas usando un ALC



David Jesús Horat Flotats  
Jorge Luis Cañizales Díaz

# Índice:

<b>1. Problema a resolver.....</b>	<b>2</b>
<b>2. Estudio teórico de las redes ADALINE .....</b>	<b>3</b>
Estructura de la Red.....	3
Función de Aprendizaje.....	5
<b>3. Resolución .....</b>	<b>9</b>
Estudio teórico.....	9
Implementación en Mathematica.....	11
Implementación en Visual Basic .....	21
<b>4. Conclusiones.....</b>	<b>26</b>
<b>5. Bibliografía.....</b>	<b>27</b>
<b>Apéndice A.1: .....</b>	<b>28</b>
<b>Apéndice A.2: .....</b>	<b>30</b>
<b>Apéndice B: .....</b>	<b>32</b>
<b>Apéndice C: .....</b>	<b>34</b>

## **1. Problema a resolver**

El problema planteado en la práctica es la implementación de una neurona que simule la operación booleana XOR, cuya tabla de entradas y salidas se describe en la figura 1, a continuación de este párrafo. Para ello utilizaremos un ALC (Adaptive Linear Combiner). El objetivo de esta práctica es estudiar el método de aprendizaje del tipo corrección de error Widrow-Hoff o regla delta. Además debemos profundizar en el método base de los algoritmos de aprendizaje de arquitecturas mono y multicapa supervisadas.

<b>XOR</b>		
X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	1

Figura 1

## 2. Estudio teórico de las redes ADALINE

### *Estructura de la Red*

Las redes ADALINE (Adaptative Linear Element), fueron desarrolladas por Bernie Widrow en la Universidad de Stanford. Dicha red usa neuronas con función de transferencia escalón, y está limitada a una única neurona de salida. Las redes ADALINE están formadas por un elemento denominado combinador adaptativo lineal (ALC), que obtiene una salida lineal que puede ser aplicada a otro elemento de conmutación bipolar, de forma que si la salida del ALC es positiva, la salida de la red es +1 y -1 en el caso contrario. Utiliza la denominada regla Delta de Widrow-Hoff o regla del mínimo error cuadrado medio (LMS), basada en la búsqueda del mínimo de una expresión del error entre la salida deseada y la salida lineal obtenida antes de aplicarle la función de activación escalón. Estas redes pueden procesar información analógica, tanto de entrada como de salida, utilizando una función de activación lineal o sigmoideal.

En cuanto a su estructura, está formada por un elemento denominado combinador adaptativo lineal (ALC) que obtiene una salida lineal(s) que pueda ser aplicada a otro elemento de conmutación bipolar, de forma que si la salida del ALC es positiva, la salida de la red ADALINE es +1; si la salida es negativa, entonces la salida de la red ADALINE es -1.

En la figura 2 se muestra la red ADALINE, compuesta por un combinador adaptativo lineal y una función de salida bipolar.

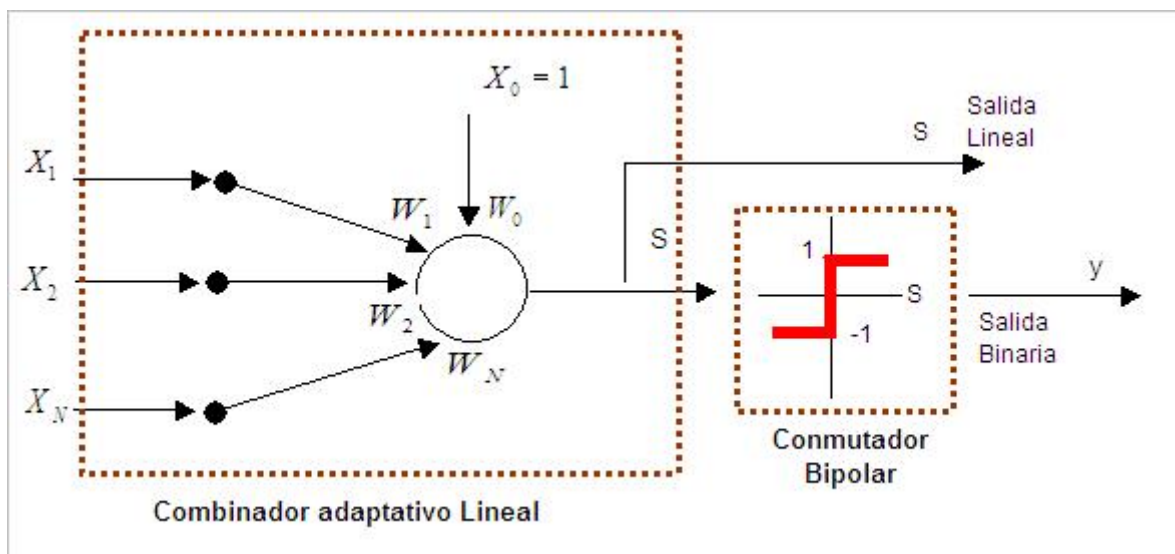


Figura 2

El ALC realiza el cálculo de la suma ponderada de las entradas:

$$S = \omega_0 + \sum_{j=1}^N \omega_j x_j$$

El umbral de la función de transferencia se representa a través de una conexión ficticia de peso  $\omega_0$ . Si tenemos en cuenta que para esta entrada se toma el valor de  $x_0 = 1$ , se puede escribir la anterior ecuación de la forma:

$$S = \sum_{j=0}^N \omega_j x_j = X \times W^T$$

Esta es la salida lineal que genera el ALC. La salida binaria correspondiente de la red ADALINE es, por tanto:

$$y(t+1) = \begin{cases} +1 & s > 0 \\ y(t) & s = 0 \\ -1 & s < 0 \end{cases}$$

La red ADALINE se puede utilizar para generar una salida analógica utilizando un conmutador sigmoidal, en lugar de binario; en tal caso, la salida  $y$  se obtendrá aplicando una función tipo sigmoidal, como la tangente hiperbólica ( $\tanh(s)$ ) o la exponencial ( $1/(1+e^{-s})$ ).

## ***Función de Aprendizaje***

---

La red ADALINE utiliza un aprendizaje OFF LINE con supervisión denominado LMS (*Least Mean Squared*) o regla del mínimo cuadrado medio. También se conoce como *regla delta* porque trata de minimizar una delta o diferencia entre el valor observado y el deseado en la salida de la red. La salida considerada es el valor previo a la aplicación de la función de activación de la neurona.

La regla aprendizaje de mínimos cuadrados es un método para hallar el vector de pesos  $W$  deseado, el cual deberá ser único y asociar con éxito cada vector del conjunto de vectores o patrones de entrada  $\{X^1, X^2, X^3, \dots, X^L\}$  con su correspondiente valor de salida correcto (o deseado)  $d_k$ ,  $k = 1, \dots, L$ . El problema hallar un conjunto de pesos  $W$  que para un único vector de entrada  $X$  dé lugar a un vector de salida correcto resulta sencillo, lo que no ocurre cuando se dispone de un conjunto de vectores de entrada, cada uno con su propio valor de salida asociado. El entrenamiento de la red consiste en adaptar los pesos a medida que se vayan presentando los patrones de entrenamiento y salidas deseadas para cada uno de ellos. Para cada combinación entrada-salida se realiza un proceso automático de pequeños ajustes en los valores de los pesos hasta que se obtienen las salidas correctas.

La primera cuestión a resolver es definir qué significa obtener el *mejor* vector de pesos obtenido a partir de unas parejas de valores *ejemplo*  $(X^i, d^i)$  de forma que, una vez encontrado, desearemos que al aplicar todos los vectores de entrada se obtenga como resultado el valor de salida correcto. Se trata de eliminar, o por lo menos, minimizar la diferencia entre la salida deseada y la real para todos los vectores de entrada.

La regla de aprendizaje LMS minimiza el error cuadrado medio, definido como:

$$\langle \epsilon_k^2 \rangle = \frac{1}{2L} \sum_{k=1}^L \epsilon_k^2$$

donde  $L$  es el número de vectores de entrada (patrones) que forman el *conjunto de entrenamiento*, y  $\epsilon_k$  la diferencia entre la salida deseada y la obtenida cuando se introduce el patrón  $k$ -ésimo, que, se expresa como  $\epsilon_k = (d_k - s_k)$ , siendo  $s_k$  la salida del ALC.

La función de error es pues una función matemática definida en el espacio de pesos multidimensional para un conjunto de patrones dados. Es una superficie que tendrá muchos mínimos (global y locales), y la regla de aprendizaje va a buscar el punto en el espacio de pesos donde se encuentra el mínimo global de esta superficie. Aunque la superficie de error es desconocida, el método de gradiente decreciente consigue obtener información local de dicha superficie a través del gradiente. Con esta información se decide qué dirección tomar para llegar hasta el mínimo global de dicha superficie.

Basándose en el método del gradiente decreciente, se obtiene una regla para modificar los pesos de tal manera que hallamos un nuevo punto en el espacio de pesos más próximo al punto mínimo. Es decir, las modificaciones en los pesos son proporcionales al gradiente decreciente de la función error  $\Delta\omega_j = -\alpha (\partial\epsilon_k / \partial\omega_j)$ . Por tanto, se deriva la función error con respecto a los pesos para ver cómo varía el error con el cambio de pesos.

Aplicamos la regla de la cadena para el cálculo de dicha derivada:

$$\Delta\omega_i = -\alpha \frac{\partial \langle \epsilon_k^2 \rangle}{\partial \omega_i} = -\alpha \frac{\partial \langle \epsilon_k^2 \rangle}{\partial s_k} \frac{\partial s_k}{\partial \omega_i}$$

$$s_k = X_k \times W^T = \sum_{j=0}^N \omega_j x_k$$

Se calcula la primera derivada:

$$\frac{\partial \langle \epsilon_k^2 \rangle}{\partial s_k} = \frac{\partial \left[ \frac{1}{2} (d_k - s_k)^2 \right]}{\partial s_k} =$$

$$\frac{1}{2} (2 (d_k - s_k) (-1)) = - (d_k - s_k) = -\epsilon_k$$

por tanto, queda:

$$\frac{\partial \langle \epsilon_k^2 \rangle}{\partial s_k} = -\epsilon_k$$

Teniendo en cuenta que  $S_k$  es la salida lineal:

$$s_k = \sum_{j=0}^N \omega_j x_k$$

calculamos la segunda derivada de la expresión de  $\Delta\omega_j$ :

$$\frac{\partial s_k}{\partial \omega_i} = \frac{\partial \left[ \sum_{j=0}^N \omega_j x_k \right]}{\partial \omega_i} = \frac{\partial (\omega_i x_k)}{\partial \omega_i} = x_k$$

Las modificaciones en los pesos son proporcionales al gradiente descendente de la función error:

$$\begin{aligned}\Delta\omega_i &= -\alpha (-\varepsilon_k X_k) = \alpha (\varepsilon_k X_k) = \alpha (d_k - s_k) X_k \\ \omega_i(t+1) &= \omega_i(t) + \alpha (d_k - s_k) X_k\end{aligned}$$

siendo  $\alpha$  la constante de proporcionalidad o tasa de aprendizaje.

En notación matricial, quedaría:

$$W(t+1) = W(t) + \alpha \varepsilon_k X_k = W(t) + \alpha (d_k - s_k) X_k$$

Esta expresión representa la modificación de pesos obtenida al aplicar el algoritmo LMS.  $\alpha$  es el parámetro que determina la estabilidad y la velocidad de convergencia del vector de pesos hacia el valor de error mínimo. Los cambios en dicho vector deben hacerse relativamente pequeños en cada iteración, sino podría ocurrir que no se encontrase nunca un mínimo, o se encontrase sólo por accidente, en lugar de ser el resultado de una convergencia sostenida hacia él.

La aplicación del proceso iterativo de aprendizaje consta de los siguientes pasos:

1. Se aplica un vector o patrón de entrada,  $X_k$ , en las entradas del ADALINE.

$$s_k = X_k \times W^T = \sum_{j=0}^N \omega_j x_k$$

2. Se obtiene la salida lineal  $s_k$  y se calcula la diferencia con respecto a la deseada  $\varepsilon_k = (d_k - s_k)$ .

3. Se actualizan los pesos:  $W(t+1) = W(t) + \alpha \varepsilon_k X_k$

4. Se repiten los pasos del 1 al 3 con todos los vectores de entrada ( $L$ ).

5. Si el error cuadrado medio:  $\langle \varepsilon_k^2 \rangle = \frac{1}{2L} \sum_{k=1}^L \varepsilon_k^2$  es un valor reducido aceptable, termina el proceso de aprendizaje; sino, se repite otra vez desde el paso 1 con todos los patrones.

Cuando se utiliza una red ADALINE para resolver un problema concreto, es necesario determinar una serie de aspectos prácticos, como el número de vectores de entrenamiento necesarios, hallar la forma de generar la salida deseada para cada vector de entrenamiento, o la dimensión óptima del vector de pesos, o cuales deberían ser los valores iniciales de los pesos, así como si es necesario o no un umbral  $\theta$ , o cuál debe ser el valor de  $\alpha$ , o cuándo se debe finalizar el entrenamiento, etc.

Respecto al número de componentes del vector de pesos, si el número de entradas está bien definido, entonces habrá un peso por cada entrada, con la opción de añadir o no un peso para la entrada del umbral.

La solución es diferente cuando sólo se dispone de una señal de entrada. En estos casos, la aplicación más común es el *filtro adaptativo* para, por ejemplo, eliminar ruido de la señal de entrada.

La dimensión del vector de pesos tiene una influencia directa en el tiempo necesario de entrenamiento, generalmente, se debe tomar un compromiso entre este aspecto y la aceptabilidad de la solución (normalmente se mejora el error aumentando el número de pesos).

El valor del parámetro  $\alpha$  tiene una gran influencia sobre el entrenamiento. Si  $\alpha$  es demasiado grande, la convergencia es posible que no se produzca, debido a que se *darán saltos* en torno al mínimo sin alcanzarlo. Si  $\alpha$  es demasiado pequeño, alcanzaremos la convergencia, pero la etapa de aprendizaje será más larga.

En cuanto al momento en el que debemos detener el entrenamiento, depende de los requerimientos del sistema. El entrenamiento se detiene cuando el error observado es menor que el admisible en la señal de salida de forma sostenida. Se suele tomar el error cuadrático medio como la magnitud que determina el instante en el que un sistema ha convergido.

### **3. Resolución**

Antes de empezar a tocar el ordenador, decidimos estudiar teóricamente los datos, acto seguido para conseguir nuestros objetivos, una vez sabiendo el tipo de red que vamos a utilizar, nos decantamos por hacerlo en Mathematica 4, cuyo interfaz nos da sencillez en la entrada de datos aunque hallamos tenido que aprender a utilizar su lenguaje propio, ayudándonos por supuesto de la ayuda que incorpora este programa.

Una vez resuelto el problema y vistos los resultados, quedaba pendiente conseguir que otras personas pudiesen comprobarlo igual de fácil por lo que nos decidimos a hacer un programa con un interfaz muy amigable e interactivo, que nos resuelva el planteamiento inicial y otros similares. Nos decantamos por usar Visual Basic como lenguaje de programación.

#### ***Estudio teórico***

---

La salida del ALC es el producto matricial de la matriz de pesos por el vector de entrada, así que encontrar un ALC que resuelva la función XOR equivale a resolver este sistema:

$$\begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} \omega_1 \\ \omega_2 \end{pmatrix}$$

La primera fila (el primer patrón) no aporta ninguna restricción, por lo que el sistema se reduce a:

$$\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \times \begin{pmatrix} \omega_1 \\ \omega_2 \end{pmatrix}$$

Que es claramente incompatible. Esto es así para todas las funciones booleanas interesantes. La forma de resolverlo es añadiendo un tercer término a la neurona, que haga a los tres últimos patrones de entrada linealmente independientes y, por tanto, linealmente separables. Por ejemplo se puede añadir el resultado del And de las dos entradas originales, quedando el sistema así:

$$\begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix}$$

Equivalente a:

$$\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix}$$

Cuya solución es:

$$\begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ -2 \end{pmatrix}$$

Surge un problema con las funciones cuya salida para las entradas 0 0 sea 1 (como por ejemplo la Equ). En este caso la primera fila no se puede suprimir del sistema, y es evidente que hay que añadir un tercer término con un 1 en la primera fila. Pero aún así siguen quedando 3 incógnitas para 4 ecuaciones, por lo que hemos de añadir una cuarta entrada, que puede ser por ejemplo un término bias, que esté siempre a 1. Para la Equ quedaría:

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{pmatrix}$$

Cuya solución es:

$$\begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{pmatrix} = \begin{pmatrix} -1 \\ -1 \\ 2 \\ 1 \end{pmatrix}$$

Todo este desarrollo se completa en el apéndice D, que no hemos tenido tiempo de maquetar debido a su complejidad matemática por lo que se entregará próximamente como trabajo adicional, y se comprueba con los resultados de la implementación, que se muestran en el siguiente apartado.

## ***Implementación en Mathematica***

---

La implementación de esta red en el lenguaje inventado por Stephen Wolfram es bastante sencilla. Además, al ser un lenguaje interpretado (en contraposición a compilado) y compartir el mismo documento para la entrada y la salida, se consigue una productividad bastante alta, ya que se puede modificar el código del programa ejecutado nada más detectar una salida incoherente.

Otra comodidad que ofrecen programas de desarrollo matemático como éste o el Matlab es una potente y desarrollada librería gráfica, que permite ver el resultado de la evolución directamente de manera visual.

En el apéndice A.1 se incluye el código del programa con cambio entre patrones de entrada/salida de forma cíclica, y en el apéndice A.2 el mismo pero con selección aleatoria de patrones. En el apéndice B se incluye de nuevo el mismo programa pero con la condición de parada añadida de haber alcanzado aproximadamente el cero de la función de coste (el error cuadrático medio).

---

El programa da como salida la gráfica de la evolución del error, del error cuadrático medio y los pesos finales.

Se invoca al programa con tres parámetros: El primero es la tasa de aprendizaje, que controla la velocidad a la que se sigue la dirección del gradiente hacia el mínimo. El segundo es el número de entradas de la neurona. Y el tercero es una matriz con los patrones de entrada/salida, dispuestos de esta manera:

$$\text{paresXor} = \begin{pmatrix} (0 \ 0) & (0) \\ (0 \ 1) & (1) \\ (1 \ 0) & (1) \\ (1 \ 1) & (0) \end{pmatrix}$$

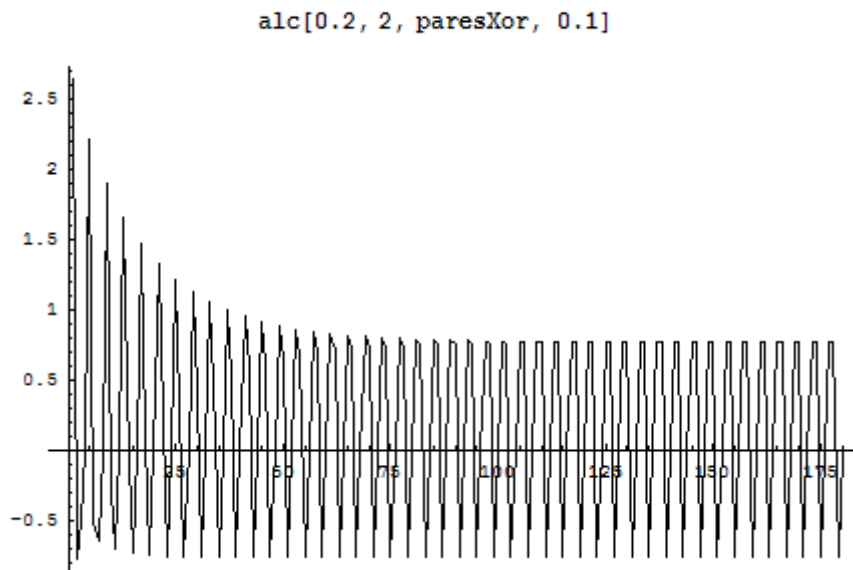
De modo que en cada fila se encuentran el vector de entrada (dispuesto como un vector fila) y la salida asociada.

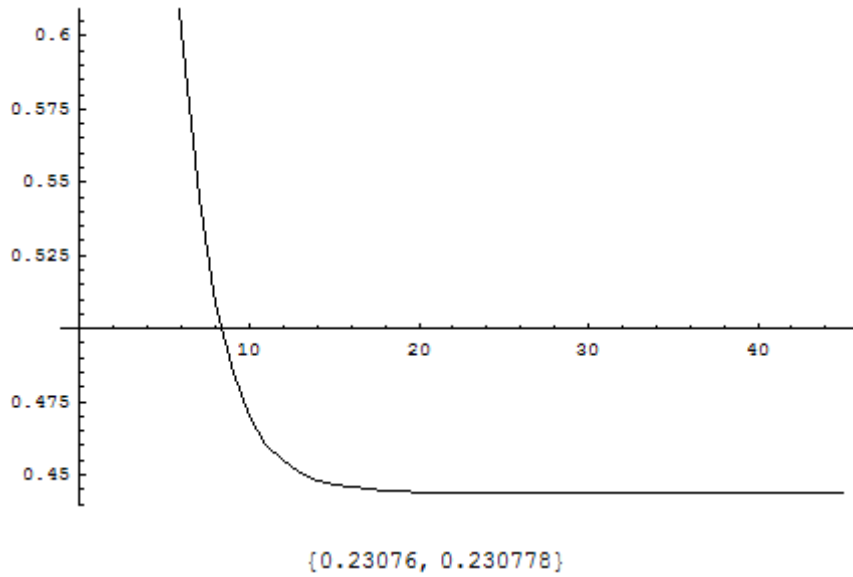
Cuando la condición de parada ha de ser el cero del error cuadrático medio, la tolerancia se indica como cuarto parámetro. Y hay siempre un último parámetro opcional que establece el número máximo de iteraciones.

---

Presentamos los resultados de la ejecución de los programas que terminan el aprendizaje de la red cuando ésta llega a cero del error cuadrático medio.

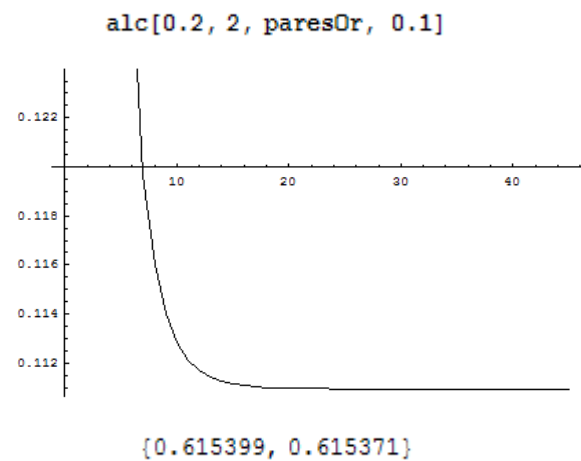
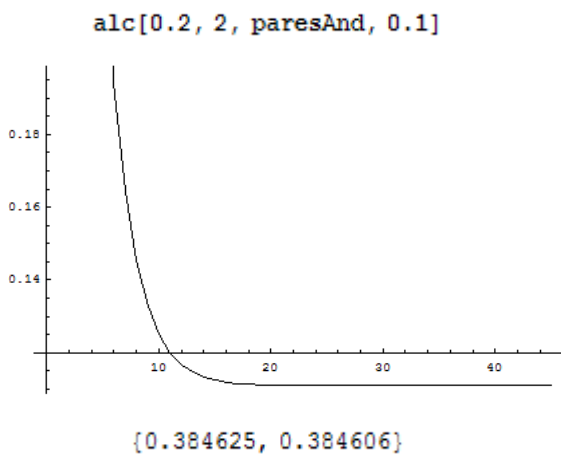
La ejecución del programa que escoge los patrones de forma cíclica, con los patrones de la función Xor da estos resultados:



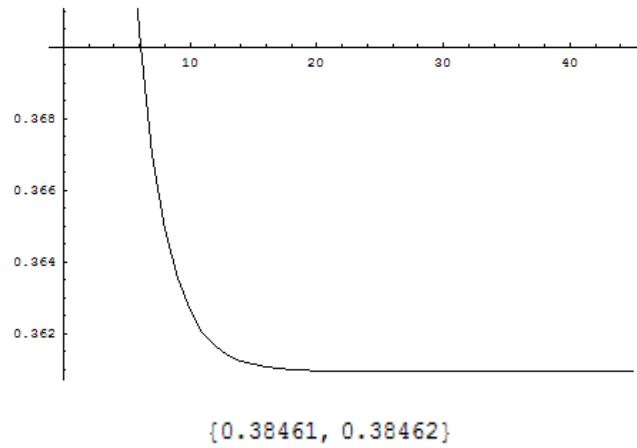


Se ve cómo la red no converge a una raíz de la función de coste, ya que ésta no tiene ceros. Se queda estable en el mínimo de ésta, que presenta un error medio apreciable. Para otras funciones lógicas el comportamiento es similar:

$$\text{paresAnd} = \begin{pmatrix} (0\ 0) & (0) \\ (0\ 1) & (0) \\ (1\ 0) & (0) \\ (1\ 1) & (1) \end{pmatrix} \quad \text{paresOr} = \begin{pmatrix} (0\ 0) & (0) \\ (0\ 1) & (1) \\ (1\ 0) & (1) \\ (1\ 1) & (1) \end{pmatrix} \quad \text{paresEqu} = \begin{pmatrix} (0\ 0) & (1) \\ (0\ 1) & (0) \\ (1\ 0) & (0) \\ (1\ 1) & (1) \end{pmatrix}$$



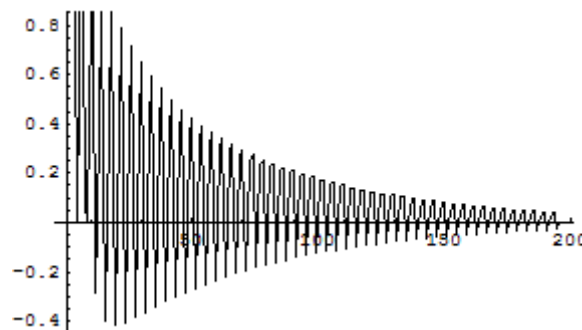
alc[0.2, 2, paresEqu, 0.1]

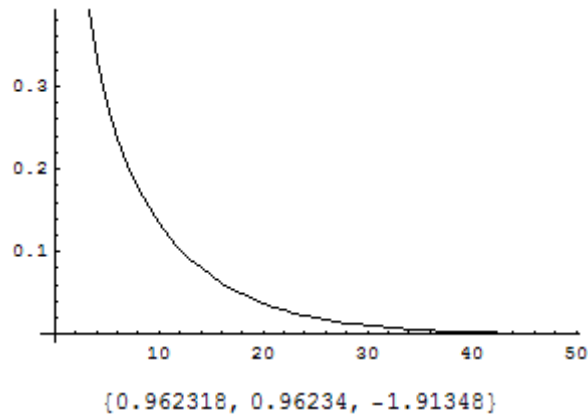


Comprobamos además un resultado del desarrollo teórico (apéndice D): en la red no está influyendo la salida de (0, 0), por lo que al suministrarle los patrones de la función Equ, convergerá de la misma forma que con una And.

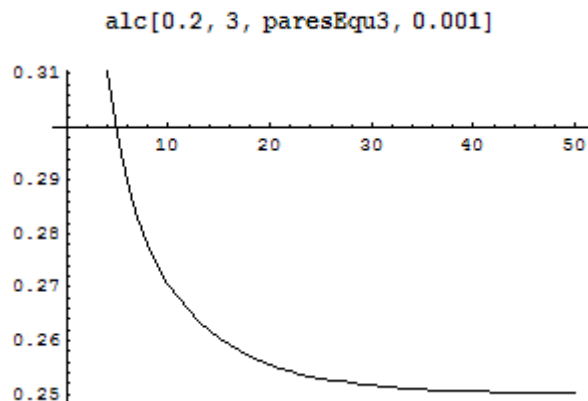
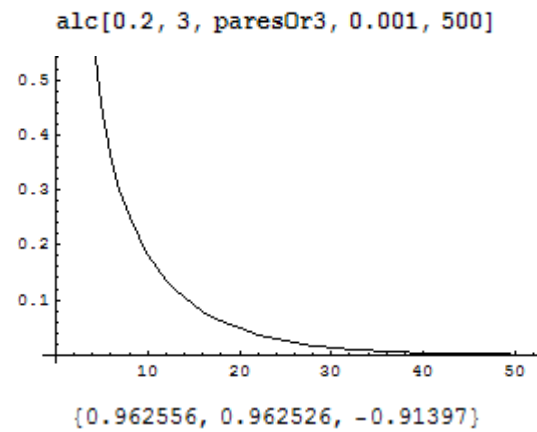
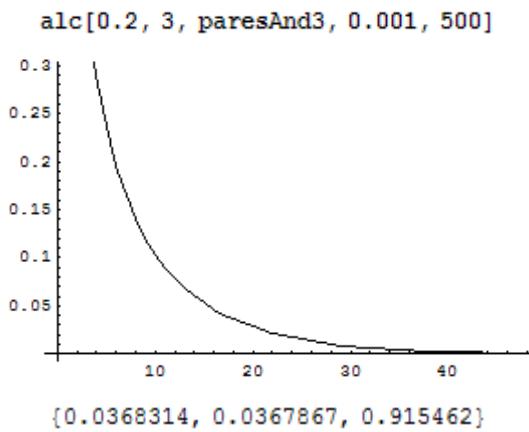
Si añadimos la capa de neuronas intermedia entre las entradas y el ALC, de modo que nuestra neurona reciba tres entradas: las dos originales y el resultado de su And, obtenemos estos resultados:

$$\begin{aligned}
 \text{paresAnd3} &= \begin{pmatrix} (0 & 0 & 0) & (0) \\ (0 & 1 & 0) & (0) \\ (1 & 0 & 0) & (0) \\ (1 & 1 & 1) & (1) \end{pmatrix} & \text{paresXor3} &= \begin{pmatrix} (0 & 0 & 0) & (0) \\ (0 & 1 & 0) & (1) \\ (1 & 0 & 0) & (1) \\ (1 & 1 & 1) & (0) \end{pmatrix} \\
 \text{paresOr3} &= \begin{pmatrix} (0 & 0 & 0) & (0) \\ (0 & 1 & 0) & (1) \\ (1 & 0 & 0) & (1) \\ (1 & 1 & 1) & (1) \end{pmatrix} & \text{paresEqu3} &= \begin{pmatrix} (0 & 0 & 0) & (1) \\ (0 & 1 & 0) & (0) \\ (1 & 0 & 0) & (0) \\ (1 & 1 & 1) & (1) \end{pmatrix} \\
 & & \text{alc}[0.2, 3, \text{paresXor3}, 0.001, 500] &
 \end{aligned}$$





Vemos como la red sí encuentra una solución de la función Xor como combinación lineal de  $X_1$ ,  $X_2$  y  $X_1 \cdot X_2$ . Específicamente nos devuelve:  $X_1 + X_2 - 2X_1 \cdot X_2$ . Con las demás funciones sucede igual, salvo con la Equ:

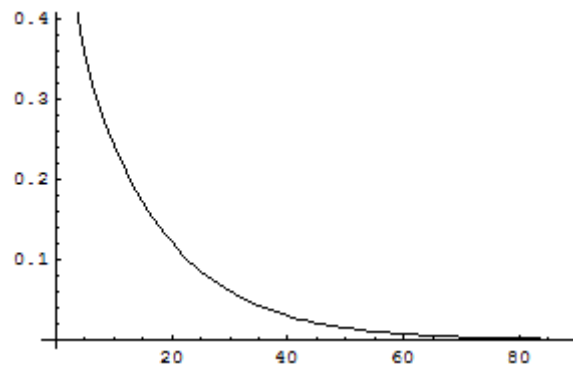


```
{-0.0134856, -0.0134968, 1.03098}
```

El motivo de que no se pueda obtener la función Equ es que esta tiene 1 como salida de (0, 0, 0), con lo que la neurona necesita el término de umbral de valor 1 que usará para desplazar su salida:

$$\text{paresEqu3B} = \begin{pmatrix} (0 & 0 & 0 & 1) & (1) \\ (0 & 1 & 0 & 1) & (0) \\ (1 & 0 & 0 & 1) & (0) \\ (1 & 1 & 1 & 1) & (1) \end{pmatrix}$$

```
alc[0.8, 4, paresEqu3B, 0.001, 500]
```



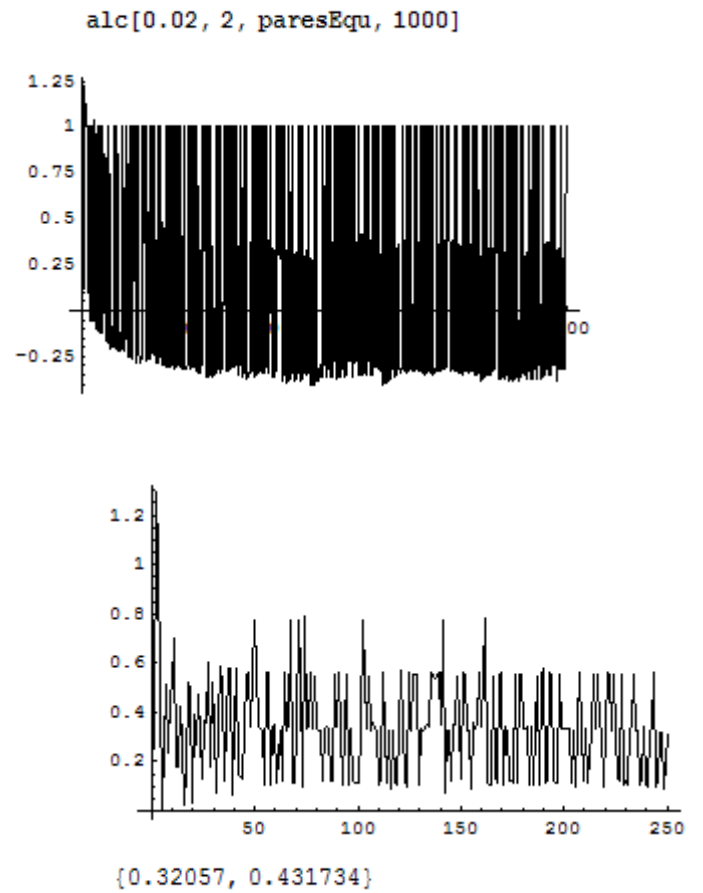
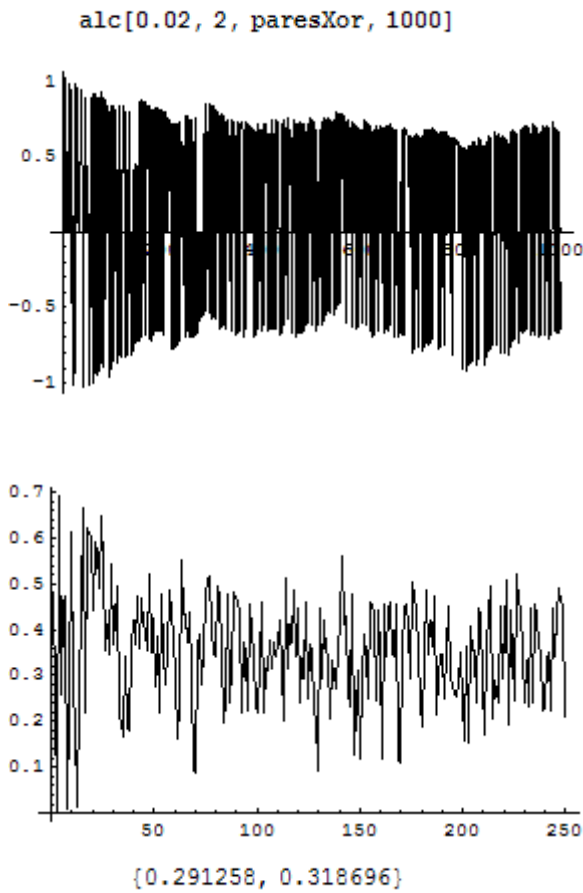
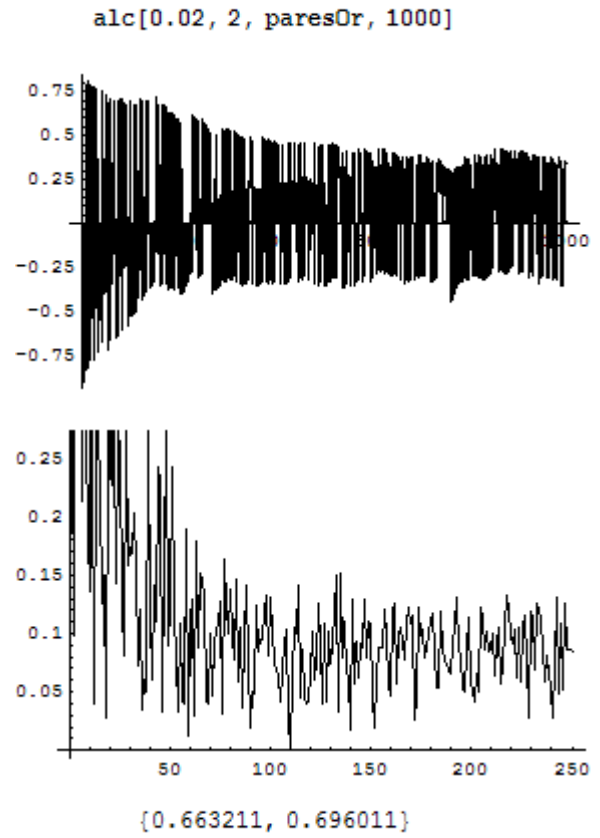
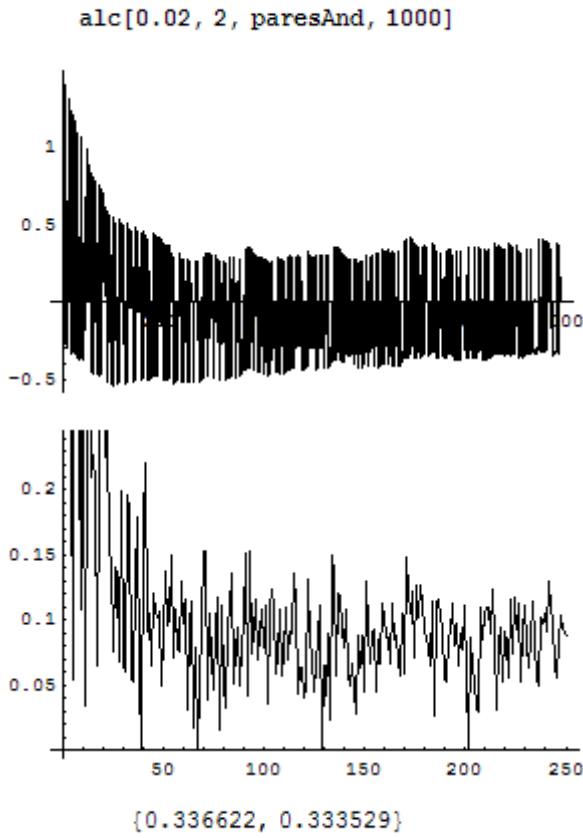
```
{-0.93052, -0.936859, 1.9103, 0.963113}
```

Los resultados que ofrece el programa que no detiene la red hasta haber alcanzado un número establecido de iteraciones son, obviamente, los mismos.

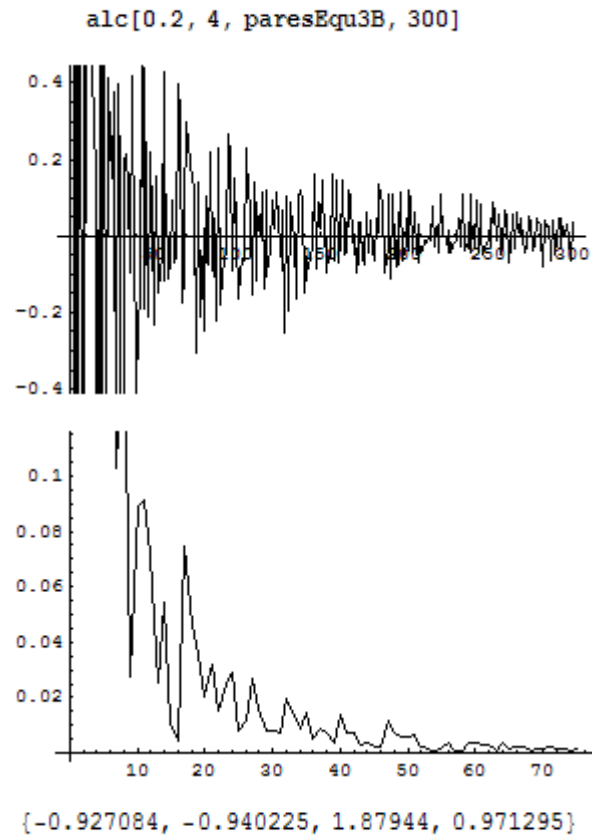
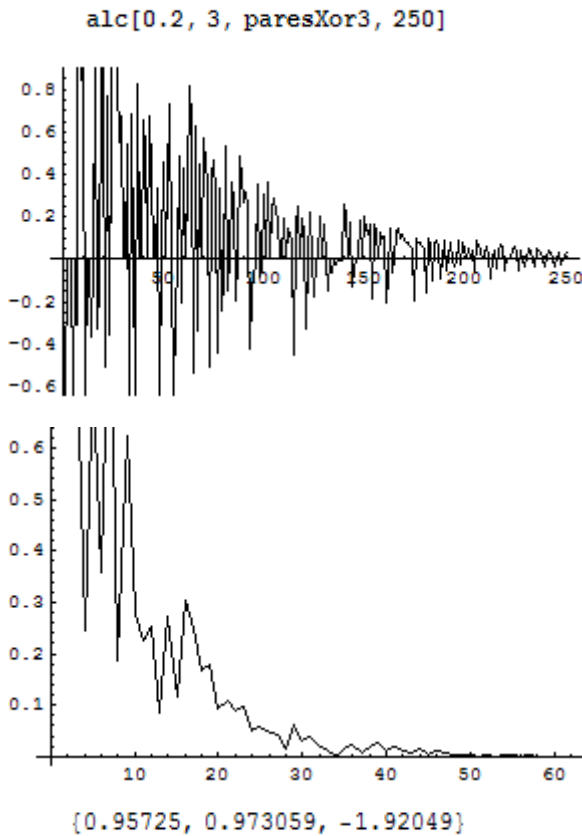
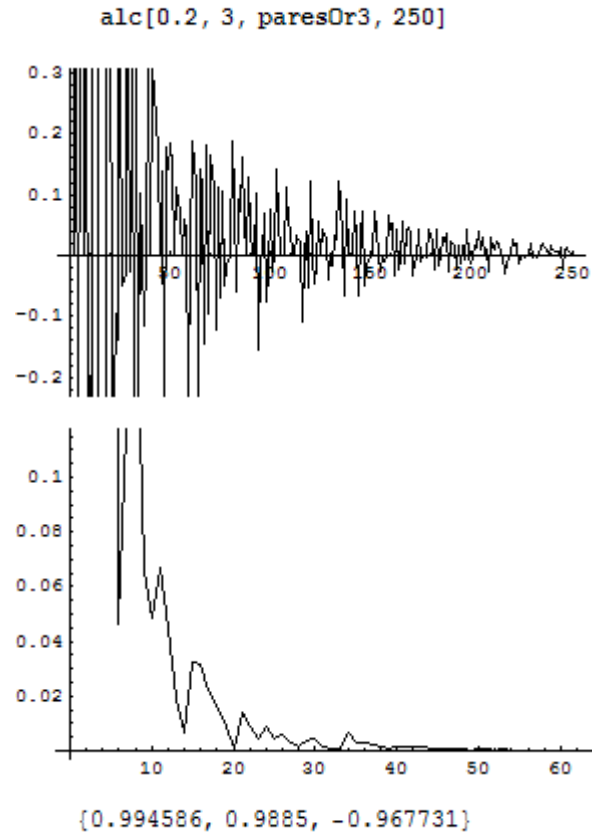
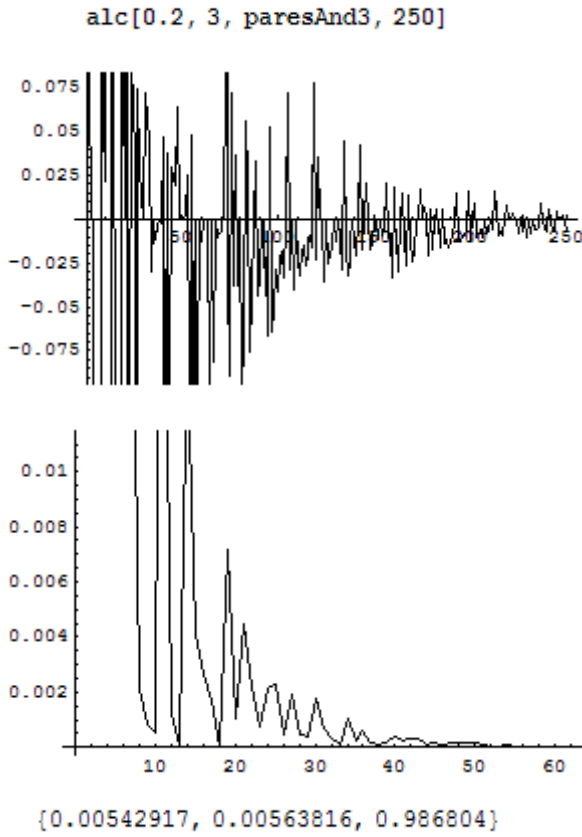
---

La versión del programa que en cada iteración escoge un patrón al azar falla al hacer parar el aprendizaje según el error cuadrático medio. Esto se debe a que desde que se repita tres veces (o a veces dos) el mismo patrón de entrada/salida, el coste caerá y se tomará la falsa decisión de que la red ha aprendido.

A continuación se muestran los resultados de la versión que en cada iteración escoge un patrón al azar con un número máximo de iteraciones fijado:

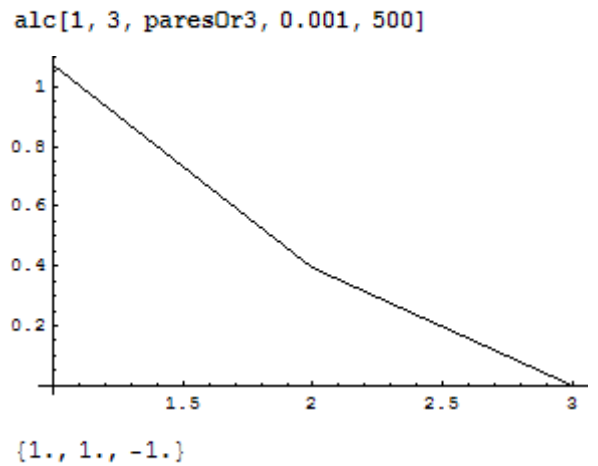
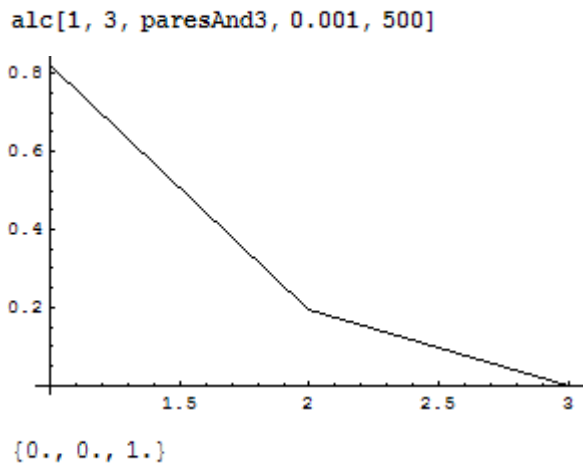
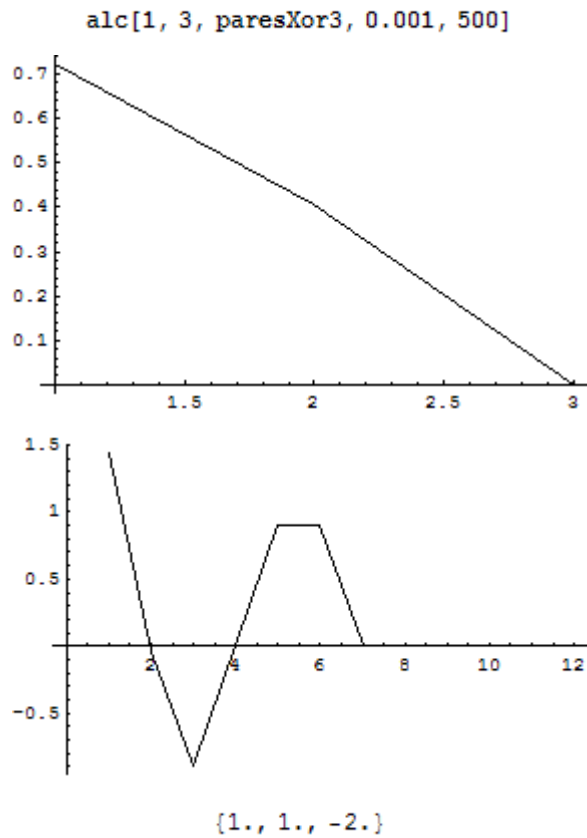


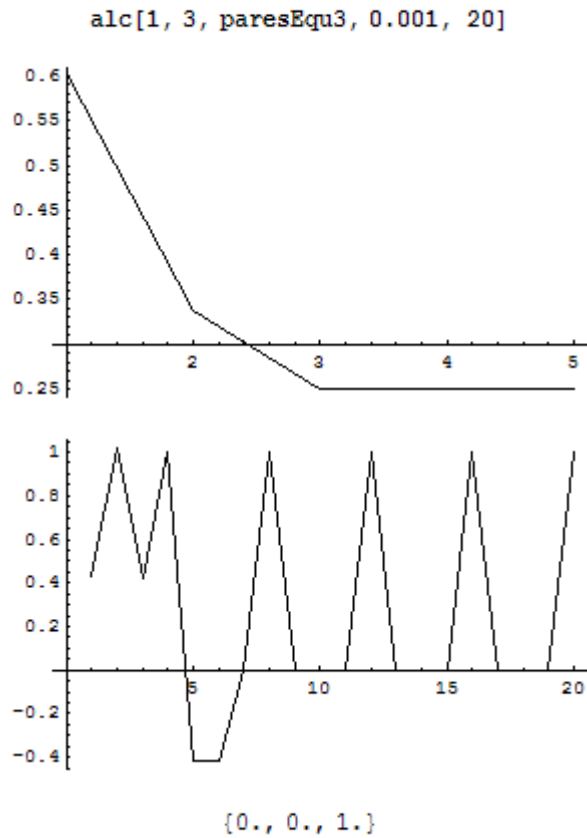
Añadiendo la capa oculta de neuronas y el término de umbral a la Equ:



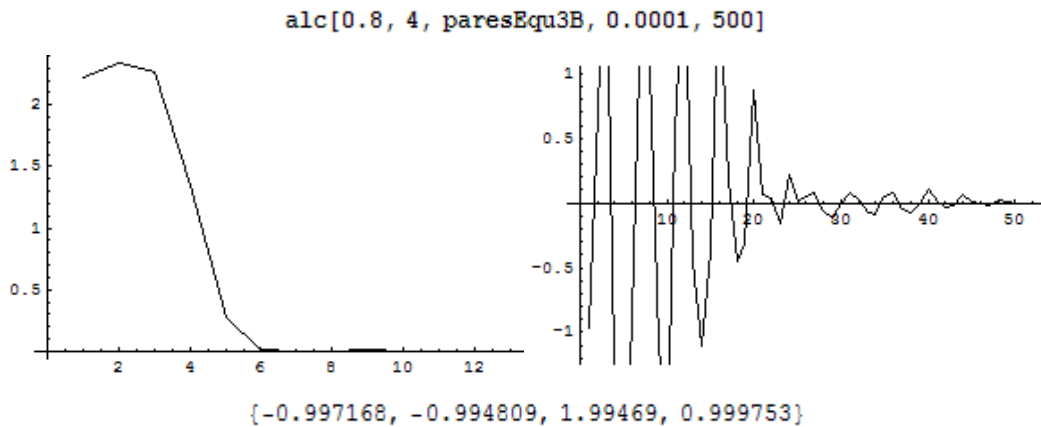
La importancia de una buena elección de la tasa de aprendizaje en la dinámica de la red es tremenda: Investigando en la práctica para confirmar los resultados que obtuvimos en el estudio teórico sobre la influencia de la tasa de aprendizaje (ver apéndice D) hemos hallado los siguientes resultados:

Para los conjuntos de patrones con tres entradas (sin término de umbral), la tasa de aprendizaje óptima es 1, alcanzando la convergencia con la ínfima cantidad de 12 iteraciones:





Para la función Equ con cuatro entradas, la tasa de aprendizaje que la hacía converger más rápidamente es 0,8:



## Implementación en Visual Basic

Para ilustrar los resultados anteriores, creamos un programa en Visual Basic de fácil manejo, y por lo tanto con una implementación relativamente compleja. A continuación se describe el interfaz del programa y en el apéndice C puede encontrar el código fuente comentado del algoritmo de iteración propuesto.

En la figura 3 encontramos la pantalla inicial del programa nada más iniciarlo, en donde se comenta cada uno de los objetos dispuestos para interactuar con el usuario.

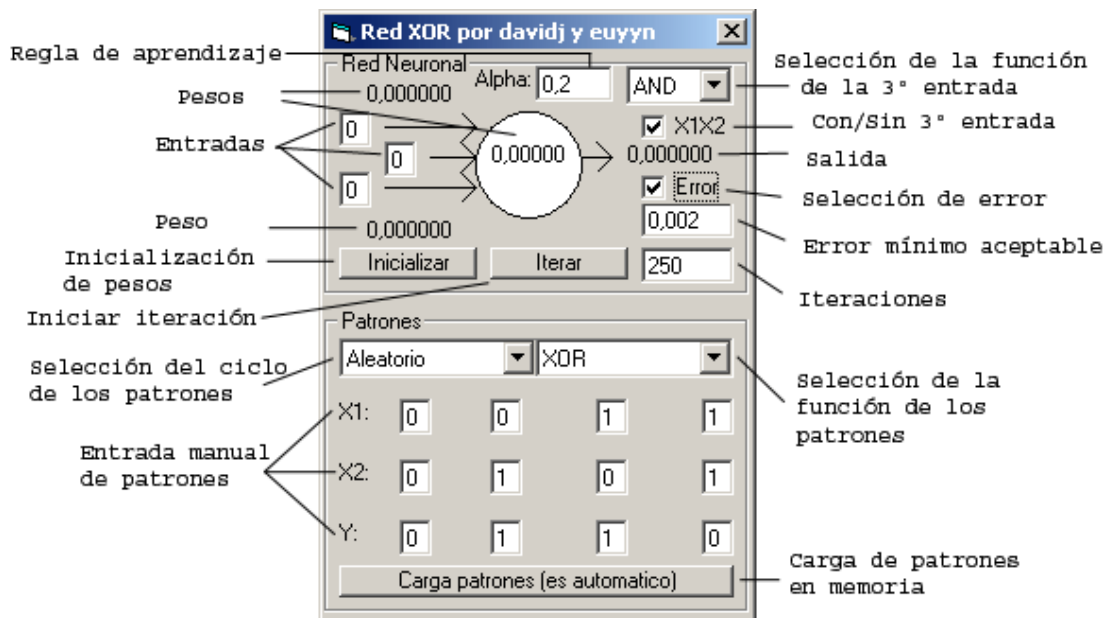


Figura 3

Cabe destacar que ante la desactivación del checkbox X1X2 que permite la activación de la 3ª entrada, desaparecería dicha entrada, su peso y la selección de su función, además de hacerse los cálculos tan sólo con 2 entradas. En tal caso, la pantalla inicial quedaría como se ve en la figura 4.



Figura 4

Para comprobar su funcionamiento haremos dos pruebas que muestran la resolución del planteamiento inicial de esta práctica.

Deseleccionamos el checkbox X1X2 y hacemos los cambios pertinentes para que las entradas queden como en la figura 4. El patrón de aprendizaje será una función XOR con orden aleatorio. El número de iteraciones será 250. Pulsamos en el botón “Inicializar” de tal forma que los pesos de las entradas se ponen de forma aleatoria. No es necesario que coincidan con estos números con valores concretos, simplemente son los que salieron al inicializar de forma aleatoria. A continuación pulsamos el botón “Iterar”, agrandándose inmediatamente el programa hacia la derecha y descubriendo dos cuadros en blanco en donde se dibujan en tiempo real las gráficas del error y del error cuadrático medio. El tiempo que tarda el ordenador en rellenar depende del número de iteraciones y de la capacidad del ordenador. Para 250 iteraciones, que es el valor por defecto, el tiempo suele estar por debajo de un segundo. En la figura 5 se muestra el resultado de nuestras operaciones.

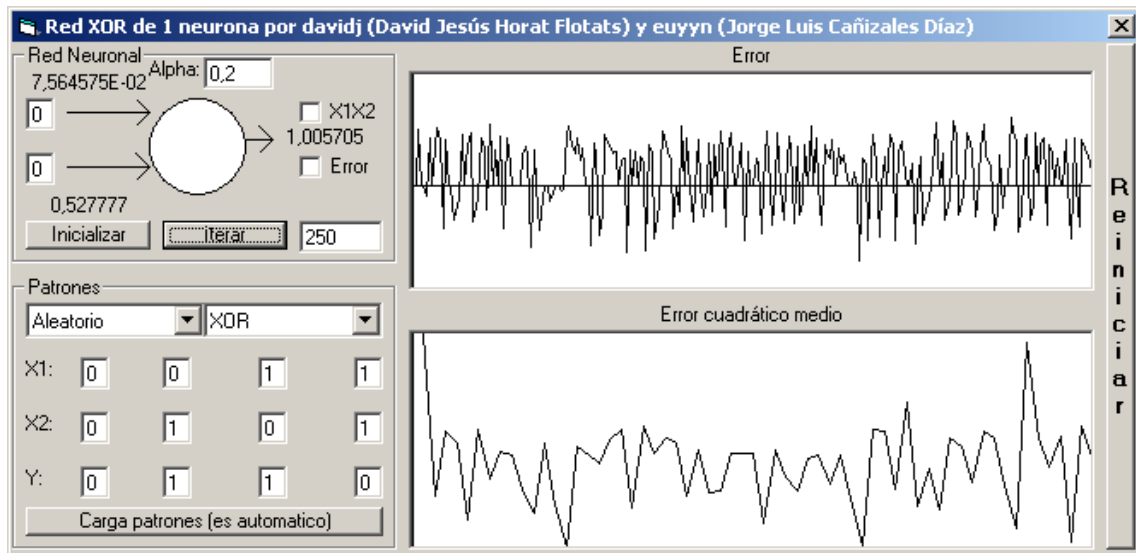


Figura 5

Las conclusiones de los resultados serán expuestas en el próximo punto, sin embargo podemos observar que el error no se estabiliza ni se reduce.

En la próxima prueba dejaremos los parámetros de entrada como en la figura 1, es decir con 3 entradas, siendo los patrones de tipo XOR y la función de la 3° entrada de tipo AND. Inicializamos los pesos picando en el botón inicializar y a continuación pulsamos el botón iterar. La figura 6 muestra uno de los resultados de esta operación.

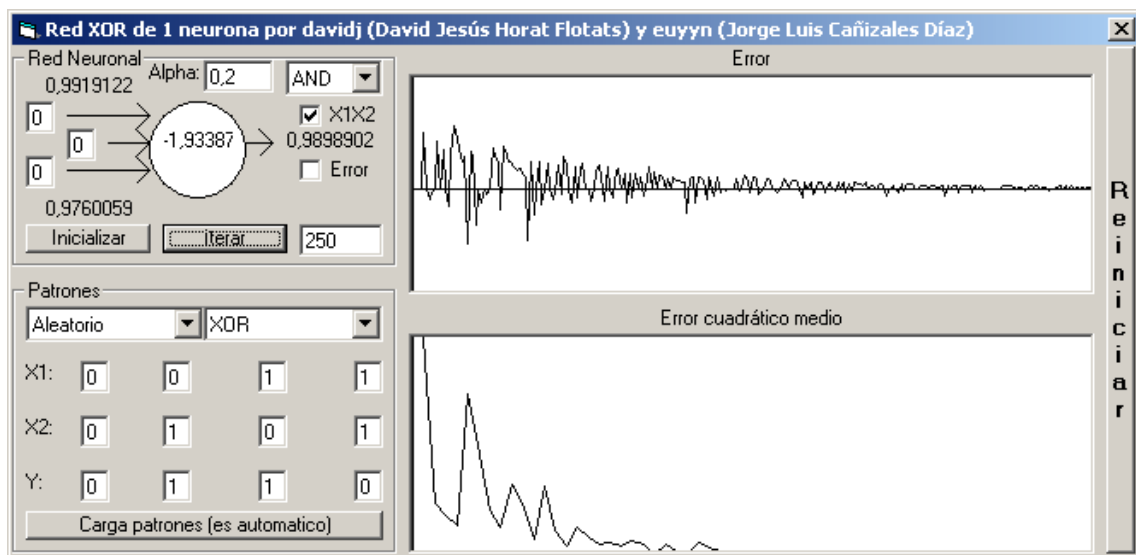


Figura 6

Comprobamos claramente como el error disminuye considerablemente y como nuestro programa muestra de forma clara y precisa los resultados de las operaciones seleccionadas por el usuario. Con dos entradas podemos simular  $2^3$  operaciones que simbolizan las posibilidades en los parámetros de los patrones. Sin embargo hemos preseleccionado las 6 operaciones básicas del álgebra booleana por ser las más representativas. En el caso de 3 entradas podemos representar  $8 * 6 = 48$  posibilidades distintas, ya que la 3º entrada solo puede ser calculada mediante las 6 operaciones booleanas básicas. En total 56 pruebas interactivas para que el usuario comprenda la resolución de las funciones booleanas usando un ALC.

Además, en esta segunda versión del programa hemos añadido la posibilidad de que el programa, en vez de iterar un número determinado de veces, también podamos tener un mínimo de error, que cuando se sobrepase el programa parará y nos dirá cuantas iteraciones llevamos en el cuadro de “iteraciones”. Además, también podemos variar el valor de la regla de aprendizaje, es decir, el alpha. Para probar estos cambios hemos decidido comprobar como aumentando el valor de la regla de aprendizaje y dejando el error mínimo a 0,002 el número de iteraciones necesarias para converger a dicho valor es menor. En la figura 7 podemos ver los resultados para un alpha de 0,2, en la figura 8 para un alpha de 0,5 y en la figura 9, el mínimo número de iteraciones que hemos encontrado, dado para un alpha 1.

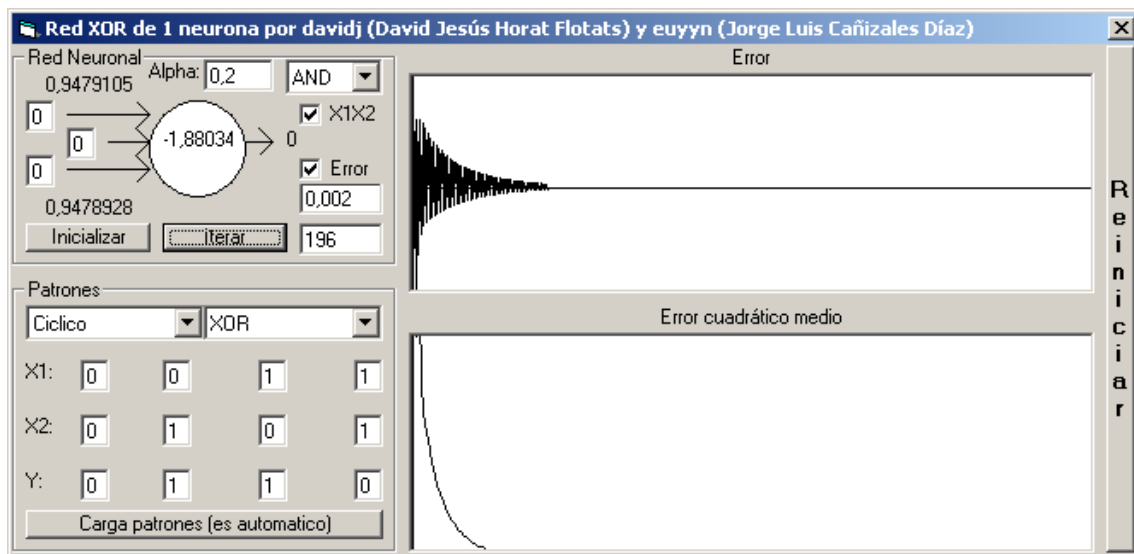


Figura 7

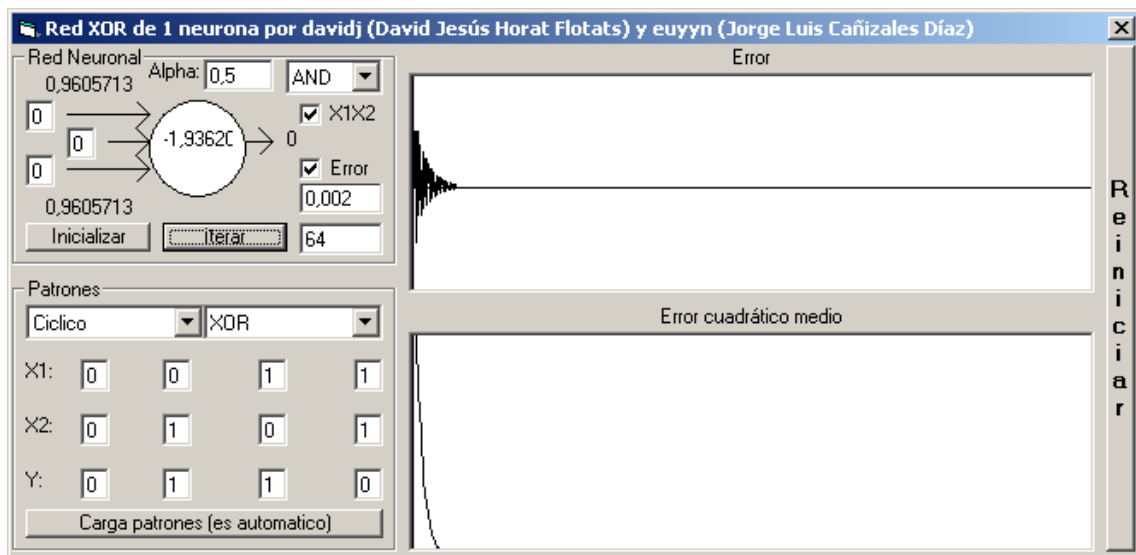


Figura 8

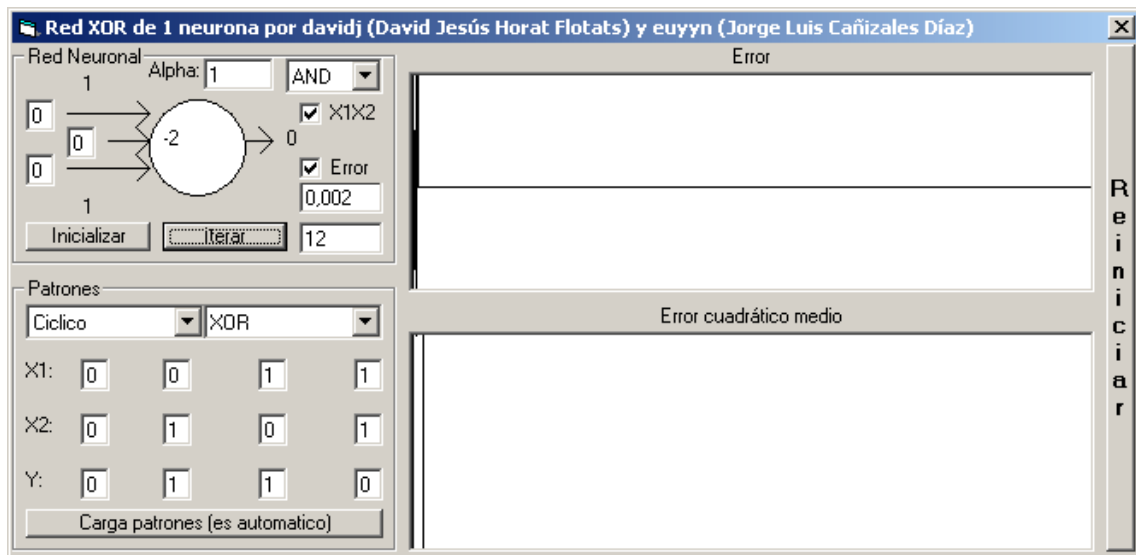


Figura 9

## **4. Conclusiones**

Hemos mostrado que el método de Widrow Hoff es válido, logrando hacer aprender a una red ALC diversas funciones booleanas.

También hemos mostrado la limitación de la linealidad del ALC, y cómo se puede resolver esa limitación formando una red multicapa con neuronas no lineales.

Se ha mostrado también el efecto de usar distintas tasas de aprendizaje: Cómo con una tasa más alta se alcanzaba más rápidamente la convergencia en situaciones en que la red no tuviera problemas de estabilidad. Y cómo con una tasa más baja la red iba convergiendo de manera más estable, pero más lentamente, con lo que había que multiplicar el tiempo necesario para que la red convergiera.

Hemos visto además el resultado de dos formas de organizar el entorno de la red: presentándole los patrones de entrada/salida de forma cíclica o al azar.

Presentando los patrones al azar la red es más inestable, por lo que se hace necesario disminuir la tasa de aprendizaje, de modo que el peso lo adquiera el promedio de los patrones presentados.

Presentando los patrones de forma cíclica la red es mucho más estable, con lo que se puede aumentar la tasa de aprendizaje para que la red converja antes sin problema.

Esperamos que esta práctica, tanto la memoria como los programas, ayuden a otras personas a entender el funcionamiento de las redes neuronales, especialmente el Adaline y las redes feed-forward en general.

## 5. Bibliografía

### Libros:

Referencia	[Hecht-Nielsen/90]	TÍTULO <b>Neurocomputing</b>		
AUTORES	Hecht-Nielsen, R			
EDITORIAL	Addison-Wesley Company		AÑO	1990

Referencia	[Hertz-91]	TÍTULO <b>Introduction to the theory of Neural Computation</b>		
AUTORES	J. Hertz, A. Krogh and R.G. Palmer			
EDITORIAL	Addison-Wesley		AÑO	1991

### Manuales:

Manual del programa **Mathematica 4** (Wolfram Research)

Biblioteca digital de programación de **MSDN** (Microsoft Developer Network)

### Páginas web:

<http://www.philbrierley.com/> -> Información y código sobre redes neuronales

<http://citeseer.nj.nec.com/abdi96widrowhoff.html> -> A Widrow-Hof learning rule for a generalization of the linear autoasociator

<http://thales.cica.es/rd/Recursos/rd98/TecInfo/07/capitulo4.html> -> Redes neuronales con conexiones hacia delante

<http://www.monografias.com/trabajos1/redneuro/redneuro.shtml> -> Introducción a la computación neuronal

<http://www.gc.ssr.upm.es/inves/neural/ann2/unsupmod/fixweigh/contamn.htm> -> Patrones de entrada contínuos

## Apéndice A.1:

Este es el programa que realiza las iteraciones de la red escrito en el lenguaje de Mathematica. Esta versión le va suministrando a la red los patrones de entrada/salida de forma cíclica. Para su fácil comprensión, incluye comentarios en el código.

```

Clear[alc]
alc[alpha_, nEntradas_, paresES_, nIters_:180] :=
Module[
  (* Variables locales : *)
  {entrada, w, esperada,
   error, listaErrores,
   error2m, listaErrores2m},
  (* Cuerpo de la función : *)
  error2m = 0;
  w = Table[4*Random[] - 2, {nEntradas}]; (*Pesos aleatorios*)
  listaErrores = Table[0, {nIters}];
  listaErrores2m = Table[0, { IntegerPart[nIters/4] }];
  Do[
    (* Escoge el siguiente patrón : *)
    {entrada, esperada} = paresES[[Mod[i, 4] + 1]];
    (* 4 = n° de patrones *)

    (* Modifica los pesos : *)
    error = esperada - w.Transpose[entrada];
    w += alpha error.entrada;

    (* Actualiza las gráficas de errores : *)
    listaErrores[[i]] = First[error];
    error2m += error.error/4;
    If[ Mod[i, 4] == 0, listaErrores2m[[i/4]] = error2m;
      error2m = 0],

    {i, nIters}];
  (* Representa los errores : *)
  ListPlot[listaErrores, PlotJoined -> True];
  ListPlot[listaErrores2m, PlotJoined -> True];
  (* Y devuelve los últimos pesos : *)

```

Return[**w**]

---

## Apéndice A.2:

Esta es la versión del programa que suministra a la red los patrones de entrada/salida de forma aleatoria:

```

Clear[alc]
alc[alpha_, nEntradas_, paresES_, nIters_:1000] :=
Module[
  (* Variables locales : *)
  {entrada, w, esperada,
   error, listaErrores,
   error2m, listaErrores2m},
  (* Cuerpo de la función : *)
  error2m = 0;
  w = Table[4*Random[] - 2, {nEntradas}]; (*Pesos aleatorios*)
  listaErrores = Table[0, {nIters}];
  listaErrores2m = Table[0, {IntegerPart[nIters/4] }];
  Do[
    (* Escoge un patrón al azar : *)
    {entrada, esperada} = paresES[[ Random[Integer, {1, 4}]]];
                                     (* 4 = nº de patrones *)

    (* Modifica los pesos : *)
    error = esperada - w.Transpose[entrada];
    w += alpha error.entrada;

    (* Actualiza las gráficas de errores : *)
    listaErrores[[i]] = First[error];
    error2m += error.error/4;
    If[ Mod[i, 4] == 0, listaErrores2m[[i/4]] = error2m;
      error2m = 0],
    {i, nIters}];

  (* Representa los errores : *)
  ListPlot[listaErrores, PlotJoined -> True];
  ListPlot[listaErrores2m, PlotJoined -> True];

  (* Y devuelve los últimos pesos : *)
  Return[w]]

```



## Apéndice B:

Ésta es la versión del programa que finaliza la evolución de la red cuando ésta alcanza un cero del error cuadrático medio. Los patrones son seleccionados de forma cíclica:

```

Clear[alc]
alc[alpha_, nEntradas_, paresES_, tolerancia_, maxIter_:200] :=
Module[
  (* Variables locales : *)
  {entrada, w, esperada,
   error, listaErrores,
   error2m, listaErrores2m,
   i},
  (* Cuerpo de la función : *)
  w = Table[4*Random[] - 2, {nEntradas}];
                                     (* Pesos iniciales aleatorios *)
  listaErrores = {};
  listaErrores2m = {};
  error2m = 0;
  For[i = 1,
    (i <= 4 || Last[listaErrores2m] > tolerancia) &&
    i <= maxIter,
    i++,
    (* Escoge el siguiente patrón : *)
    {entrada, esperada} = paresES[[Mod[i, 4] + 1]];
                                     (* 4 = n° de patrones *)
    (* Modifica los pesos : *)
    error = esperada - w.Transpose[entrada];
    w += alpha error.entrada;
    (* Actualiza las gráficas de errores : *)
    AppendTo[listaErrores, First[error]];
    error2m += error.error/4;
    If[ Mod[i, 4] == 0,
      AppendTo[listaErrores2m, error2m];
      error2m = 0;
  ]
]

```

```
];  
];  
(* Representa los errores : *)  
ListPlot[listaErrores2m, PlotJoined -> True];  
ListPlot[listaErrores, PlotJoined -> True];  
(* Y devuelve los últimos pesos : *)  
Return[w]
```

---

## Apéndice C:

A continuación listamos el código fuente de la función encargada de realizar las iteraciones de la red implementada en Visual Basic. Debido a la falta de espacio no publicamos el código fuente completo ya que serían más de 15 páginas, pero está disponible previa petición.

```
Private Sub cmdIterar_Click()  
  
    'Variables locales  
    Dim saLida As Single  
    Dim ErrorOr As Single  
    Dim ErrorCuaMed As Single  
    Dim ValAntE As Single  
    Dim valAntE2 As Single  
    Dim Iter As Long  
    Dim Parar As Boolean 'Condicion de parada  
    Dim NumMaxIter As Long  
    'Interrupcion de errores  
    On Error GoTo ErrorHandler  
    'Inicilizacion  
    Alpha = CSng(txtAlpha.Text)  
    Iter = 1  
    Parar = True  
    ErrorCuaMed = 0  
    ValAntE = 0  
    valAntE2 = 1  
    Randomize Timer  
    MakeBigger  
    floatW1 = CLng(lblX1.Caption)  
    floatW2 = CLng(lblX2.Caption)  
    floatW12 = CLng(lblPesoX1X2.Caption)  
    If chkError.Value Then  
        NumMaxIter = 1000  
    Else  
        NumMaxIter = CLng(txtIterar.Text)  
    End If  
    'Dibujar ejes error  
    picError.Line (0, 0)-(0, picError.Height)
```

```

    picError.Line (0, picError.Height / 2)-(picError.Width,
picError.Height / 2)
    picError2.Line (0, 0)-(0, picError.Height)
    picError2.Line (0, picError.Height)-(picError.Width,
picError.Height)
    'Iteramos
    While Parar
        'Seleccionamos el patron
        If cboPat.ListIndex = 0 Then
            j = Iter Mod 4 'Ciclico
        Else
            j = Fix(Rnd * 4) 'Aleatorio
        End If
        If chkX1X2.Value Then
            'Calculamos salida y error
            saLida = PatX1(j) * floatW1 + PatX2(j) * floatW2 +
PatX1X2(j) * floatW12
            Y.Caption = saLida
            ErrOr = PatY(j) - saLida
            ErrorCuaMed = ErrorCuaMed + ErrOr * ErrOr / 4
            'Actualizamos pesos
            floatW1 = floatW1 + Alpha * ErrOr * PatX1(j)
            lblX1 = floatW1
            floatW2 = floatW2 + Alpha * ErrOr * PatX2(j)
            lblX2 = floatW2
            floatW12 = floatW12 + Alpha * ErrOr * PatX1X2(j)
            lblPesoX1X2.Caption = floatW12
        Else
            'Calculamos salida y error
            saLida = PatX1(j) * floatW1 + PatX2(j) * floatW2
            Y.Caption = saLida
            ErrOr = PatY(j) - saLida
            ErrorCuaMed = ErrorCuaMed + ErrOr * ErrOr / 4
            'Actualizamos pesos
            floatW1 = floatW1 + Alpha * ErrOr * PatX1(j)
            lblX1 = floatW1
            floatW2 = floatW2 + Alpha * ErrOr * PatX2(j)
            lblX2 = floatW2
        End If
        'Dibujamos el error y el error cuadratico medio

```

```
        picError.Line (Iter * picError.Width / NumMaxIter, (2 -
ValAntE) / 4 * picError.Height)-((Iter + 1) * picError.Width /
NumMaxIter, (2 - Error) / 4 * picError.Height)
        DoEvents
        ValAntE = Error
        If (Iter Mod 4 = 0) Then
            picError2.Line (Iter * picError2.Width / NumMaxIter, (1 -
valAntE2) * picError2.Height)-((Iter + 4) * picError2.Width /
NumMaxIter, (1 - ErrorCuaMed) * picError2.Height)
            DoEvents
            valAntE2 = ErrorCuaMed
            ErrorCuaMed = 0
        End If
        If chkError.Value = 1 Then
            If CSng(txtIterar2.Text) >= valAntE2 Then
                txtIterar.Text = CStr(Iter)
                Parar = False
            End If
            If NumMaxIter < Iter Then
                txtIterar.Text = NumMaxIter
                Parar = False
            End If
        End If
        If chkError.Value = 0 Then
            If CLng(txtIterar.Text) = Iter Then
                'txtIterar2.Text = CStr(valAntE2)
                Parar = False
            End If
        End If
        Iter = Iter + 1
    Wend
Exit Sub

ErrorHandler:
    MsgBox "La funcion no converge", vbCritical
End Sub
```