



2020

Redes Neuronales. Segunda Edición

Rafael Alberto Moreno Parra

Contenido

Otros libros del autor 2

Página web del autor y canal en Youtube..... 2

Sitio en GitHub 2

Licencia de este libro..... 3

Licencia del software 3

Marcas registradas..... 3

Introducción 4

Iniciando..... 5

Perceptrón simple..... 7

Fórmula de Frank Rosenblatt..... 12

Perceptrón simple: Aprendiendo la tabla del OR 14

Límites del Perceptrón Simple 15

Encontrando el mínimo en una ecuación 17

Descenso del gradiente..... 21

Mínimos locales y globales 23

Búsqueda de mínimos y redes neuronales 24

Perceptrón Multicapa 25

Las neuronas 26

Pesos y como nombrarlos..... 29

La función de activación de la neurona 31

Introducción al algoritmo “Backpropagation” (backward propagation of errors) 35

Nombrando las entradas, pesos, umbrales, salidas y capas..... 36

Regla de la cadena 39

Derivadas parciales 40

Las derivadas en el algoritmo de propagación hacia atrás..... 41

Tratamiento del error en el algoritmo de propagación hacia atrás 55

Variando los pesos y umbrales con el algoritmo de propagación hacia atrás..... 61

Implementación en C# del perceptrón multicapa 62

Algoritmo de retro propagación en C# 71

Código completo del perceptrón en C# 77

Reconocimiento de números de un reloj digital..... 84

Detección de patrones en series de tiempo 91

El sobreajuste..... 101

Bibliografía y webgrafía 108

Otros libros del autor

Libro 13: "Algoritmos Genéticos". En Colombia 2020. Libro y código fuente descargable en <https://github.com/ramsoftware/LibroAlgoritmoGenetico2020>

Libro 12: "Redes Neuronales. Segunda Edición". En Colombia 2020. Libro y código fuente descargable en: <https://github.com/ramsoftware/LibroRedNeuronal2020>

Libro 11: "Capacitándose en JavaScript". En Colombia 2019. En Colombia 2020. Libro y código fuente descargable en: <https://github.com/ramsoftware/javascript>

Libro 10: "Desarrollo de aplicaciones para Android usando MIT App Inventor 2". En: Colombia 2016. Págs. 102. Ubicado en: <https://openlibra.com/es/book/desarrollo-de-aplicaciones-para-android-usando-mit-app-inventor-2>

Libro 9: "Redes Neuronales. Parte 1.". En Colombia 2016. Libro descargable en: <https://openlibra.com/es/book/redes-neuronales-parte-1>

Libro 8: "Segunda parte de uso de algoritmos genéticos para la búsqueda de patrones". En Colombia 2015. Págs. 303. En publicación por la Universidad Libre – Cali.

Libro 7: "Desarrollo de un evaluador de expresiones algebraicas. **Versión 2.0.** C++, C#, Visual Basic .NET, Java, PHP, JavaScript y Object Pascal (Delphi)". En: Colombia 2013. Págs. 308. Ubicado en: <https://openlibra.com/es/book/evaluador-de-expresiones-algebraicas-ii>

Libro 6: "Un uso de algoritmos genéticos para la búsqueda de patrones". En Colombia 2013. En publicación por la Universidad Libre – Cali.

Libro 5: Desarrollo fácil y paso a paso de aplicaciones para Android usando MIT App Inventor. En Colombia 2013. Págs. 104. Estado: Obsoleto (No hay enlace).

Libro 4: "Desarrollo de un evaluador de expresiones algebraicas. C++, C#, Visual Basic .NET, Java, PHP, JavaScript y Object Pascal (Delphi)". En: Colombia 2012. Págs. 308. Ubicado en: <https://openlibra.com/es/book/evaluador-de-expresiones-algebraicas>

Libro 3: "Simulación: Conceptos y Programación" En: Colombia 2012. Págs. 81. Ubicado en: <https://openlibra.com/es/book/simulacion-conceptos-y-programacion>

Libro 2: "Desarrollo de videojuegos en 2D con Java y Microsoft XNA". En: Colombia 2011. Págs. 260. Ubicado en: <https://openlibra.com/es/book/desarrollo-de-juegos-en-2d-usando-java-y-microsoft-xna> . ISBN: 978-958-8630-45-8

Libro 1: "Desarrollo de gráficos para PC, Web y dispositivos móviles" En: Colombia 2009. ed.: Artes Gráficas Del Valle Editores Impresores Ltda. ISBN: 978-958-8308-95-1 v. 1 págs. 317

Artículo: "Programación Genética: La regresión simbólica".
Entramado ISSN: 1900-3803 ed.: Universidad Libre Seccional Cali
v.3 fasc.1 p.76 - 85, 2007

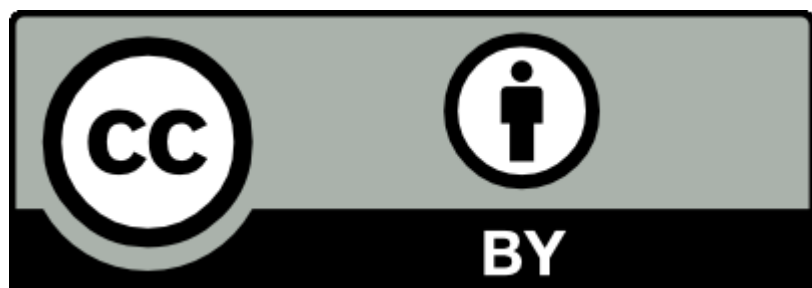
Página web del autor y canal en Youtube

Investigación sobre Vida Artificial: <http://darwin.50webs.com>

Canal en Youtube: <http://www.youtube.com/user/RafaelMorenoP> (dedicado al desarrollo en C#)

Sitio en GitHub

El código fuente se puede descargar en <https://github.com/ramsoftware/>



Todo el software desarrollado aquí tiene licencia LGPL “Lesser General Public License”



En este libro se hace uso de las siguientes tecnologías registradas:

Microsoft ® Windows ® Enlace: <http://windows.microsoft.com/en-US/windows/home>

Microsoft ® Visual Studio 2019 ® Enlace: <http://windows.microsoft.com/en-US/windows/home>

Introducción

Escribí un libro titulado "Redes Neuronales. Parte 1" en el año 2016. La "Parte 2" no llegó, en su momento, pensaba escribir para la segunda parte, el uso de librerías o "frameworks" de terceros. Sin embargo, me di cuenta de que hay una gran cantidad de muy buenos libros, tutoriales y clases en línea que enseñan a usar esas librerías. Y por mi lado, lo que me atrae de este tema es cómo están hechas por dentro las redes neuronales, las matemáticas, los algoritmos usados y su implementación directa en un lenguaje de programación.

También me he dado cuenta de algunos fallos pedagógicos del libro anterior que pueden dificultar entender este tema. Ahora se tendrá en cuenta a futuro como será la implementación en diversos lenguajes de programación.

Los conocimientos requeridos para entender este libro es saber programar en algún lenguaje de programación orientado a objetos como C#. Se recomienda el manejo de conceptos matemáticos como el uso de derivadas e integrales.

El código fuente se puede descargar en <https://github.com/ramsoftware/LibroRedNeuronal2020>

ramsoftware Add files via upload		Latest commit 26aff26 3 minutes ago
Neuronal1	Create Program.cs	6 minutes ago
Neuronal2	Create Program.cs	5 minutes ago
Neuronal3	Create Program.cs	4 minutes ago
Neuronal4	Add files via upload	3 minutes ago
algoritmo01	Create Program.cs	7 minutes ago
algoritmo02	Create Program.cs	7 minutes ago

Ilustración 1: Código fuente en GitHub

Código y en qué página es usado:

algoritmo01	Página 10
algoritmo02	Página 12
Neuronal1	Página 77
Neuronal2	Página 87
Neuronal3	Página 95
Neuronal4	Página 102

Iniciando

Las redes neuronales son como una caja negra en la cual hay unas entradas, la caja en sí y unas salidas.

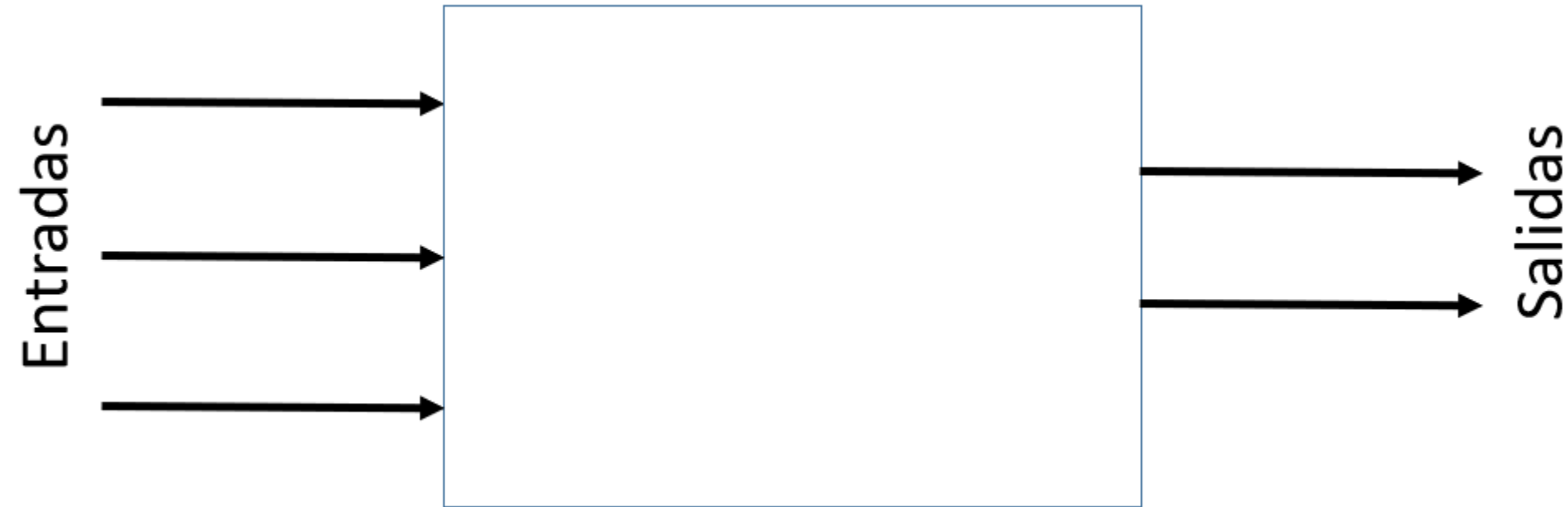


Ilustración 2: Caja negra, entradas y salidas

Lo particular es que hay unos pesos que dependiendo de su valor (más arriba o más abajo) afectan el valor de las salidas.

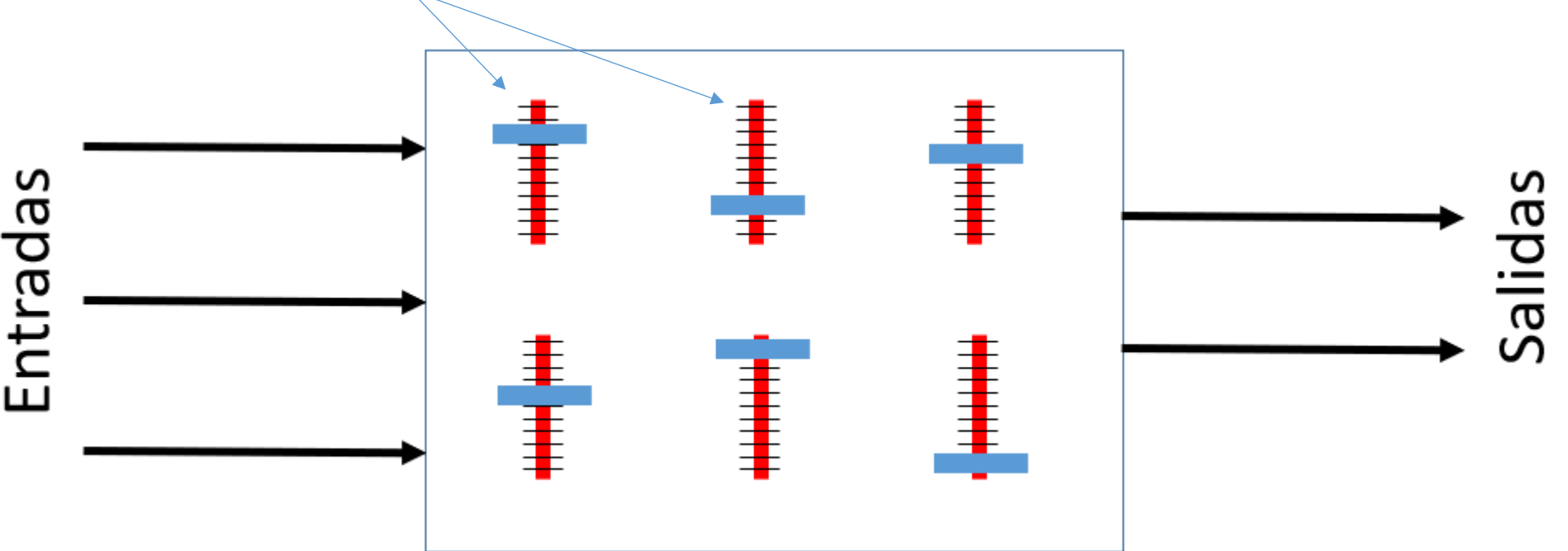


Ilustración 3: Pesos al interior de la caja

En el ejemplo, tenemos las siguientes entradas y salidas

	Entrada 1	Entrada 2	Entrada 3	Entrada 4	Salida deseada 1	Salida deseada 2
Ejemplo A	1	4	7	10	25	58
Ejemplo B	2	5	8	11	36	64
Ejemplo C	3	6	9	12	47	70

Significa que, si entran los números 1, 4, 7, 10, (ejemplo A), deberían salir 25 y 58. Luego hay que ajustar pesos (moviéndolos arriba o a abajo) hasta obtener esa salida deseada.

Luego se prueban esos pesos con el nuevo conjunto de datos (ejemplo B). Se ingresa 2, 5, 8, 11 y debería salir 36 y 64. ¿Qué pasaría si eso no sucede? Que hay que cambiar los pesos con nuevos valores y probar desde el inicio (con el ejemplo A). ¿Cuándo termina? Cuando los valores de los pesos encontrados funcionen para los tres ejemplos (A, B y C).

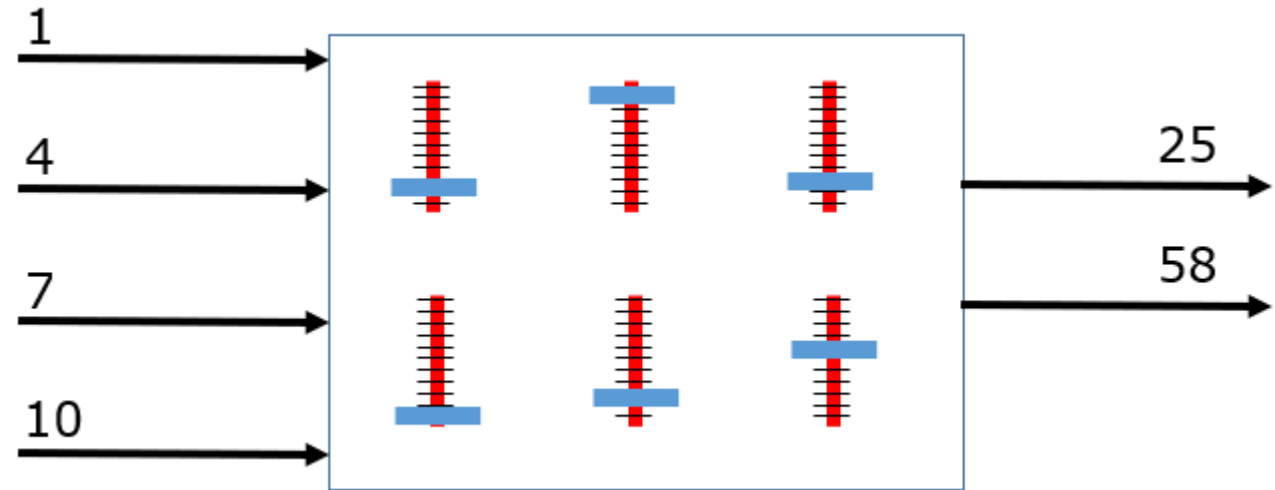


Ilustración 4: Esos pesos ya operan con el ejemplo A

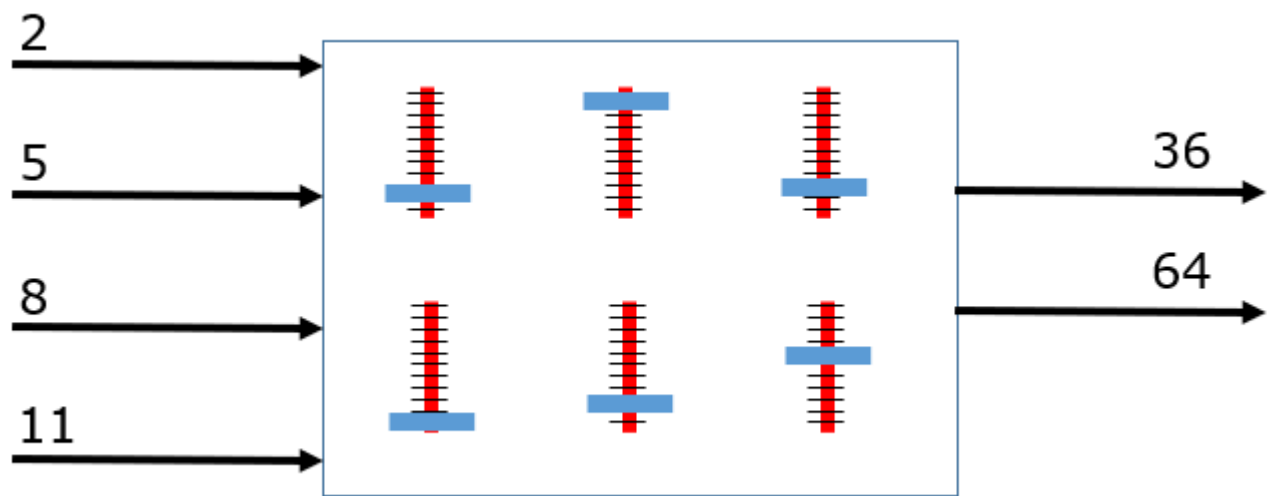


Ilustración 5: Los mismos pesos funcionan para el ejemplo B

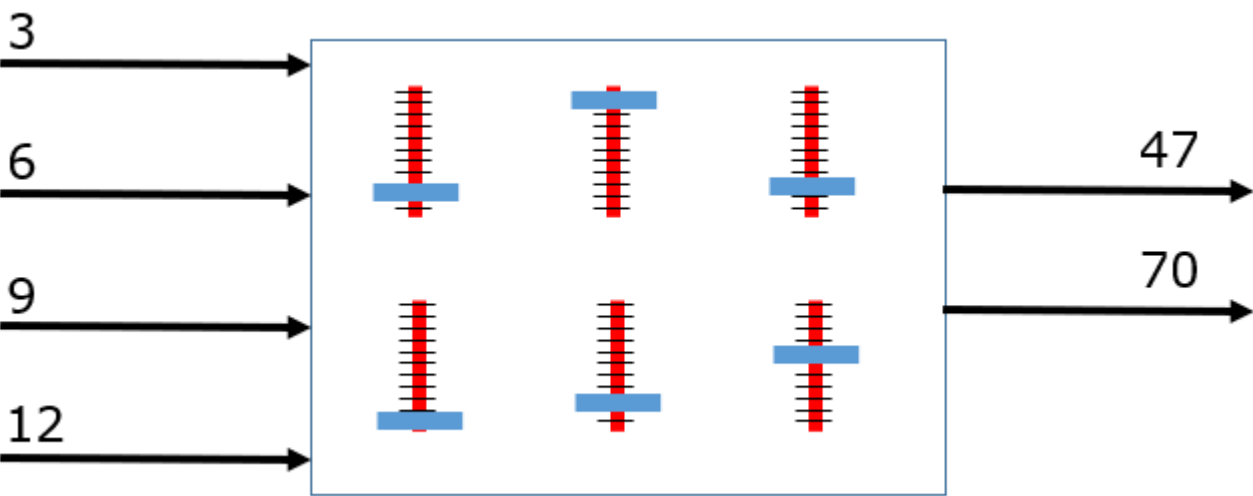
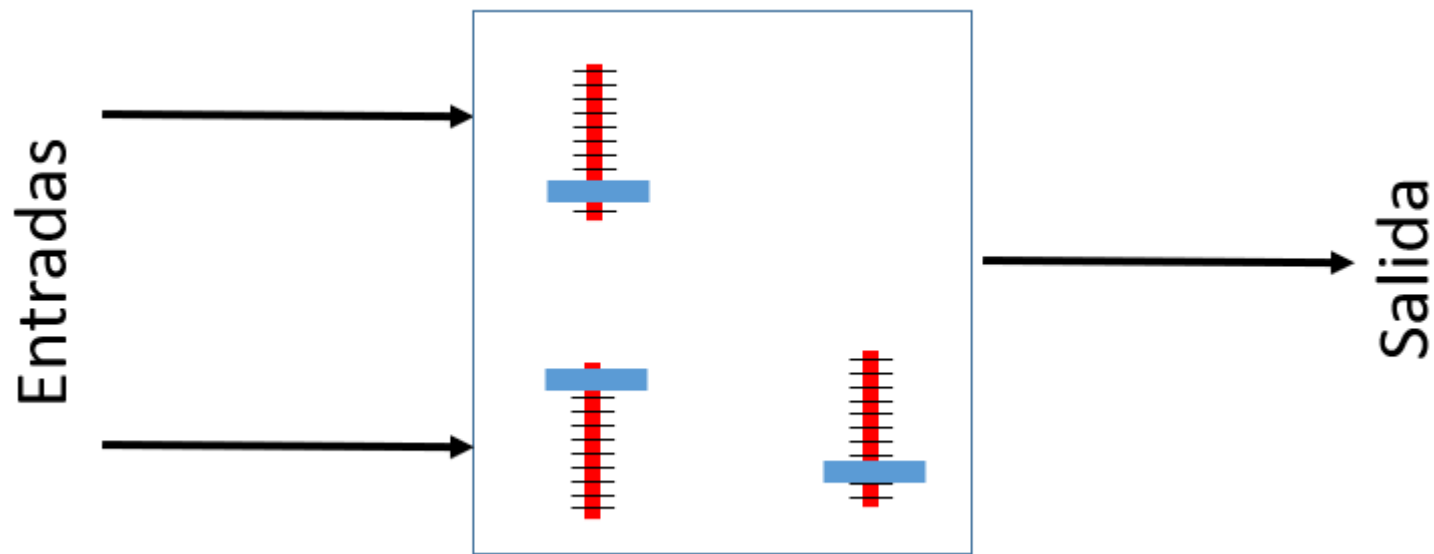


Ilustración 6: Y esos mismos pesos funcionan para el ejemplo C

¿Cómo es el proceso? Al iniciar, esos pesos tienen valores al azar y poco a poco se van ajustando. Existen técnicas matemáticas que colaboran en encontrar esos pesos rápidamente porque de lo contrario, sería un ajuste al azar continuamente hasta que por suerte se encuentren los valores correctos.

Perceptr3n simple

Se inicia con una neurona. Esta es su representaci3n:



Ilustraci3n 7: Perceptr3n simple

Dos entradas, una salida y tres pesos. Se demostrar3 que esta neurona puede "aprender" como opera la tabla del AND:

A	B	Resultado (A AND B)
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Falso
Falso	Verdadero	Falso
Falso	Falso	Falso

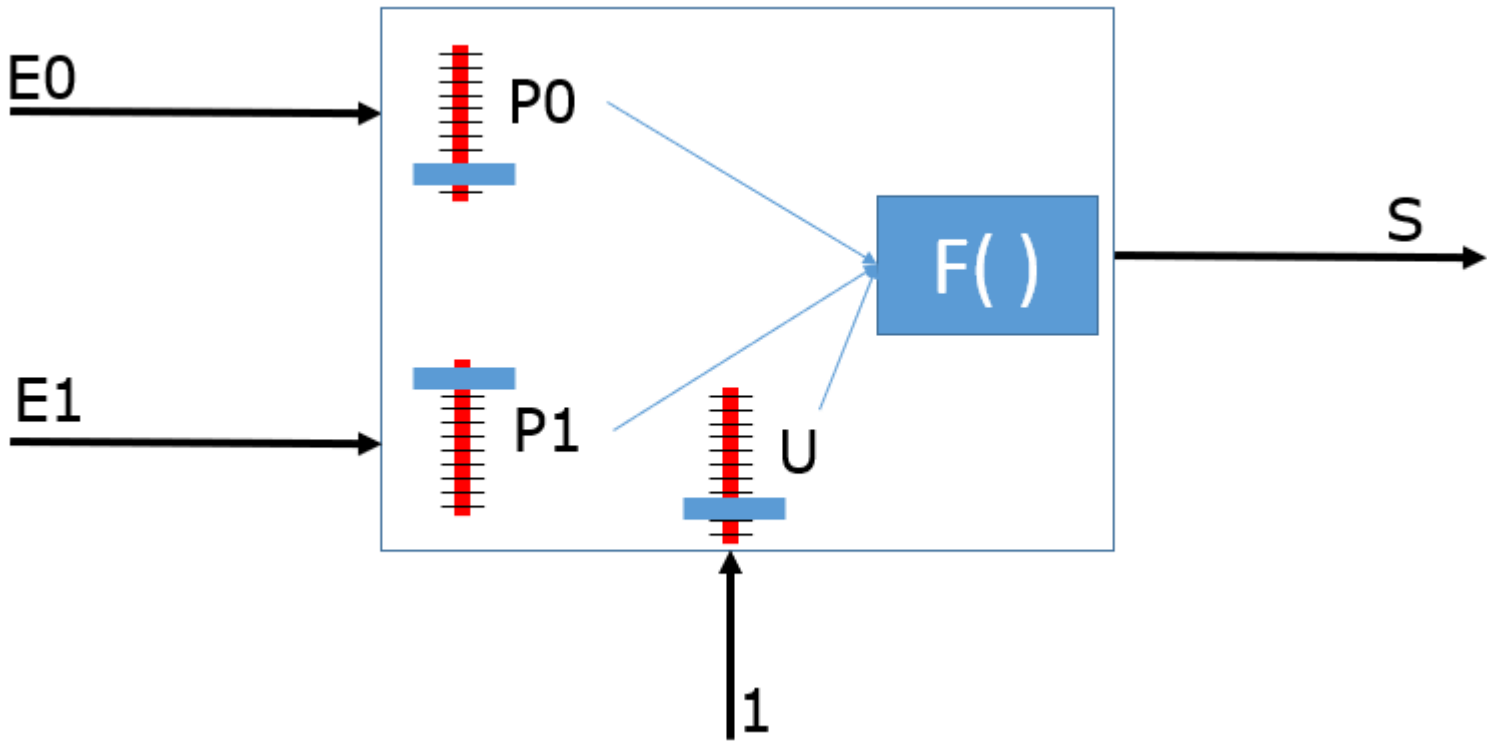
Esa neurona se le conoce con el nombre de Perceptr3n Simple.

Paso 1: Hacerlo cuantitativo (1 es verdadero, 0 es falso)

A	B	Resultado (A AND B)
1	1	1
1	0	0
0	1	0
0	0	0

La raz3n de este cambio es que se requieren valores cuantitativos para ser usados dentro de f3rmulas matem3ticas.

Paso 2: Dise1ando la neurona:



Ilustraci3n 8: Funcionamiento del perceptr3n simple

Un peso por cada entrada y se le adiciona una entrada interna que se llama umbral y tiene el valor de 1 con su propio peso.

E0 y E1 son las entradas

P0, P1 son los pesos de las entradas

U es el peso del umbral
S es la salida
F() es la función que le da el valor a S

Paso 3: Haciendo los cálculos:

La salida S se calcula con la siguiente fórmula matemática

$$S = F (E0 * P0 + E1 * P1 + 1 * U)$$

Se inicia con la primera regla de la tabla AND (verdadero y verdadero, da verdadero), en este caso se ingresa 1 y 1, la salida debería ser 1.

E0 = 1 (verdadero)
E1 = 1 (verdadero)
P0 = 0.6172 (un valor al azar)
P1 = 0.4501 (un valor real al azar)
U = 0.3789 (un valor real al azar)

Entonces la salida sería:

$$S = F (E0 * P0 + E1 * P1 + 1 * U)$$

$$S = F (1 * 0.6172 + 1 * 0.4501 + 1 * 0.3789)$$

$$S = F (1.4462)$$

¿Y que es F()? una función que podría ser matemática o un algoritmo. Así:

```
Función F(valor)
Inicio
    Si valor > 0.7 entonces
        retorne 1
    de lo contrario
        retorne 0
    fin si
Fin
```

Continuando con el ejemplo entonces

$$S = F (1.4462)$$
$$S = 1$$

Y ese es el valor esperado. Los pesos funcionan para esas entradas.

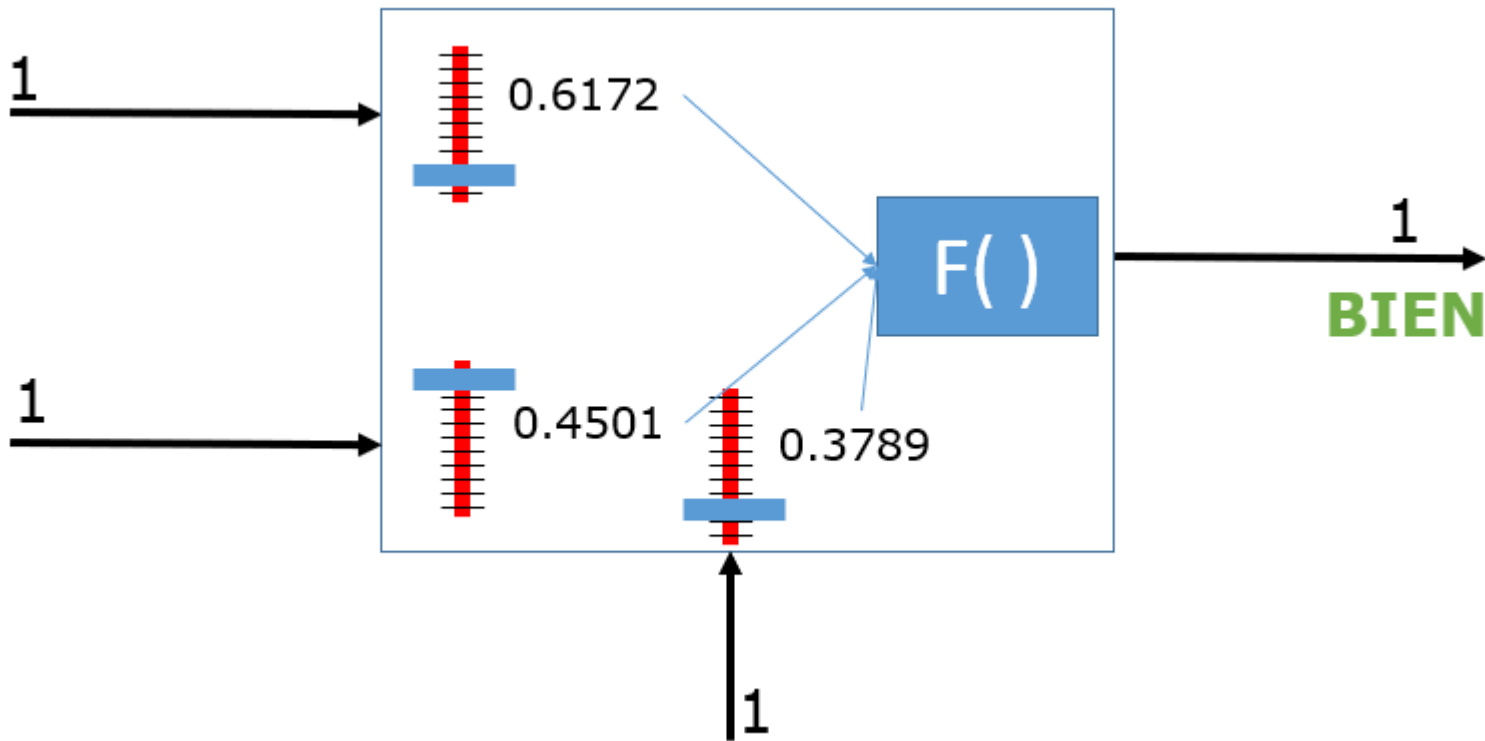


Ilustración 9: Los pesos funcionan para esa regla

¿Funcionarán esos pesos para las otras reglas de la tabla del AND? Se prueba entonces Verdadero y Falso, debe dar Falso

E0 = 1 (verdadero)

E1 = 0 (falso)

S = F (E0 * P0 + E1 * P1 + 1 * U)

S = F (1 * 0.6172 + 0 * 0.4501 + 1 * 0.3789)

S = F (0.9961)

S = 1

No, no funcionó, debería haber dado cero

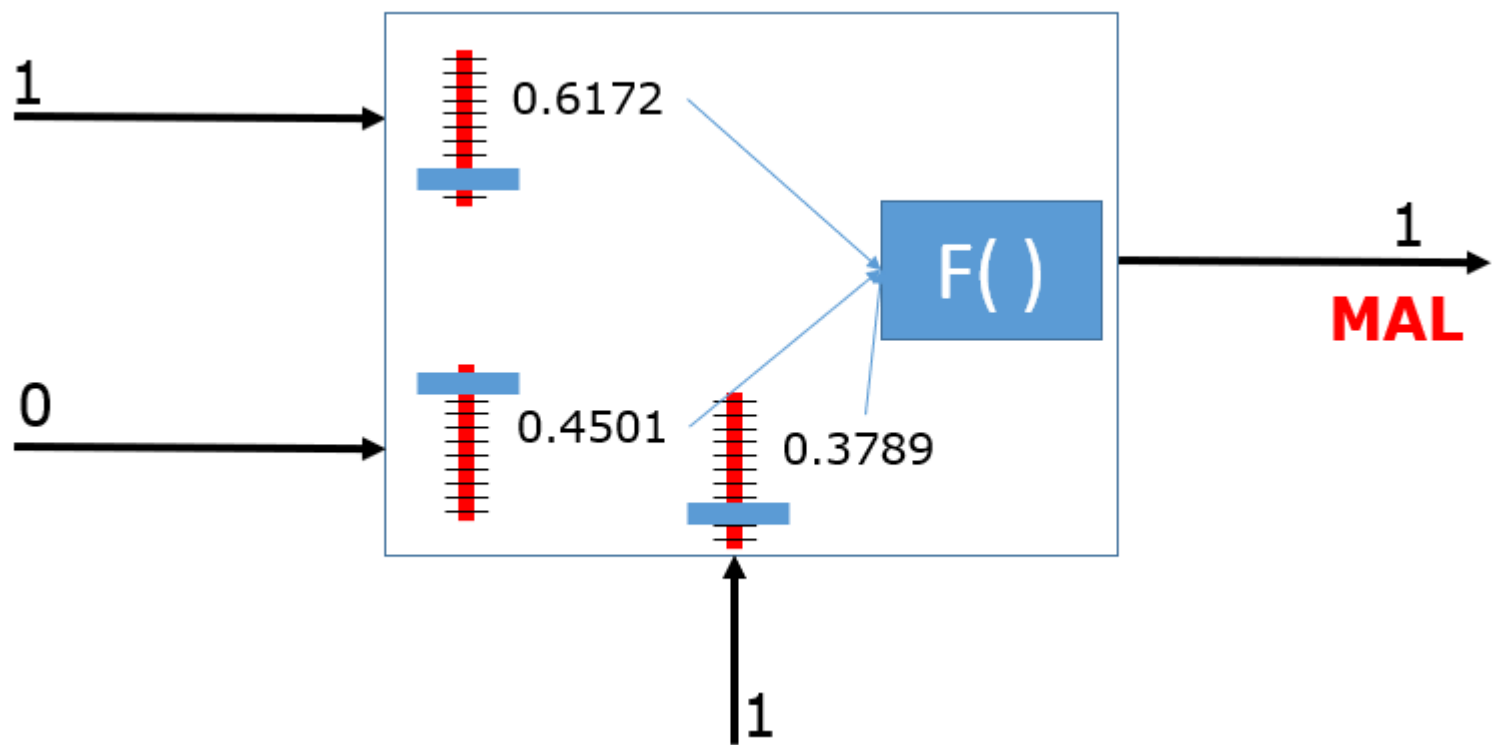


Ilustración 10: Esos pesos fallan con la segunda regla

¿Y entonces? Habrá que utilizar otros valores para los pesos. Una forma es darle otros valores al azar. Ejecutar de nuevo el proceso, probar con todas las reglas hasta que finalmente de las salidas esperadas.

Este sería la implementación en C# (algoritmo01)

```
using System;

namespace Neuronal {
    class Program {
        static void Main(string[] args) {
            //Único generador de números aleatorios
            Random azar = new Random();

            //Tabla de verdad AND
            int[,] entradas = { { 1, 1 }, { 1, 0 }, { 0, 1 }, { 0, 0 } };
            int[] salidas = { 1, 0, 0, 0 };

            //Inicializa los pesos al azar
            double P0 = azar.NextDouble();
            double P1 = azar.NextDouble();
            double U = azar.NextDouble();

            //Variable que mantiene el proceso activo de buscar los pesos
            bool proceso = true;

            //Cuenta el número de iteraciones
            int iteracion = 0;

            //Hasta que aprenda la tabla AND
            while (proceso) {
                iteracion++;

                //Optimista, en esta iteración se encuentra los pesos
                proceso = false;

                //Va por todas las reglas de la tabla AND
                for (int cont = 0; cont <= 3; cont++) {

                    //Calcula el valor de entrada a la función
                    double operacion = entradas[cont, 0] * P0 + entradas[cont, 1] * P1 + U;

                    //La función de activación
                    int salidaEntera;
                    if (operacion > 0.7)
                        salidaEntera = 1;
                    else
                        salidaEntera = 0;

                    //Si la salida no coincide con lo esperado, cambia los pesos al azar
                    if (salidaEntera != salidas[cont]) {
                        P0 = azar.NextDouble();
                        P1 = azar.NextDouble();
                        U = azar.NextDouble();
                        proceso = true; //Y sigue buscando
                    }
                }
            }

            //Muestra que el perceptrón simple aprendió.
            for (int cont = 0; cont <= 3; cont++) {
                double operacion = entradas[cont, 0] * P0 + entradas[cont, 1] * P1 + U;

                //La función de activación
                int salidaEntera;
                if (operacion > 0.7)
                    salidaEntera = 1;
                else
                    salidaEntera = 0;

                //Imprime
                Console.WriteLine("Entradas: " + entradas[cont, 0].ToString() + " y " + entradas[cont, 1].ToString() + " = " +
                    salidas[cont].ToString() + " perceptron: " + salidaEntera.ToString());
            }

            Console.WriteLine("Pesos encontrados P0= " + P0.ToString() + " P1= " + P1.ToString() + " U= " + U.ToString());
            Console.WriteLine("Iteraciones requeridas: " + iteracion.ToString());
            Console.ReadKey();
        }
    }
}
```

Ejecución 1:

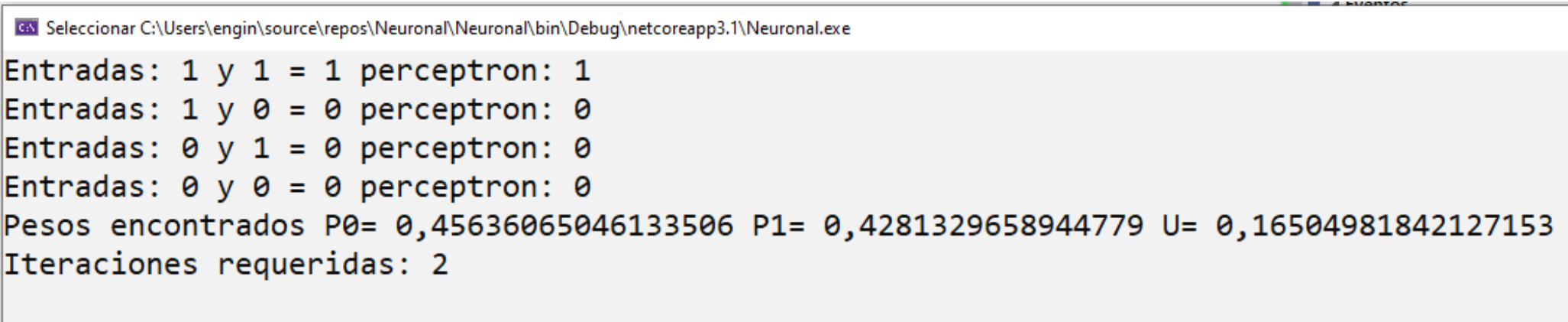


Ilustración 11: Encuentra los pesos (con buena suerte) en dos iteraciones

Ejecución 2:

```
C:\Users\engin\source\repos\Neuronal\Neuronal\bin\Debug\netcoreapp3.1\Neuronal.exe

Entradas: 1 y 1 = 1 perceptron: 1
Entradas: 1 y 0 = 0 perceptron: 0
Entradas: 0 y 1 = 0 perceptron: 0
Entradas: 0 y 0 = 0 perceptron: 0
Pesos encontrados P0= 0,2412350425688713 P1= 0,4598892966564229 U= 0,02857327416007094
Iteraciones requeridas: 14
```

Ilustración 12: Encontró los pesos, pero esta vez necesitó más iteraciones

Los valores de los pesos no es una respuesta única, pueden ser distintos y son números reales (en C# se implementaron de tipo double). También observamos que en una ejecución requirió sólo 2 iteraciones y en la siguiente ejecución requirió 14 iteraciones. El cambio de pesos sucede en estas líneas:

```
P0 = azar.NextDouble();
P1 = azar.NextDouble();
U = azar.NextDouble();
```

En caso de que no funcionasen los pesos, el programa simplemente los cambiaba al azar en un valor que oscila entre 0 y 1. Eso puede ser muy ineficiente y riesgoso porque limita los valores a estar entre 0 y 1 ¿Y si los pesos requieren valores mucho más altos o bajos?

Afortunadamente, hay un método matemático que minimiza el uso del azar y puede dar con valores de los pesos en cualquier rango. ¿Cómo funciona? Al principio los pesos tienen un valor al azar, pero de allí en adelante el cálculo de esos pesos se basa en comparar la salida esperada con la salida obtenida, si difieren, ese error sirve para ir cuadrando poco a poco los pesos.

Fórmula de Frank Rosenblatt

Los pesos cambian haciendo uso de una fórmula matemática:

```
Error = SalidaEsperada - SalidaReal

Si Error != cero entonces

    P0 = P0 + tasaAprende * Error * E0

    P1 = P1 + tasaAprende * Error * E1

    U = U + tasaAprende * Error * 1

Fin Si
```

tasaAprende es un valor constante de tipo real y de valor entre 0 y 1 (sin tomar el 0, ni el 1)

Este es el código en C# (Algoritmo02)

```
using System;

namespace Neuronal {
    class Program {
        static void Main(string[] args) {
            //Único generador de números aleatorios
            Random azar = new Random();

            //Tabla de verdad AND
            int[,] entradas = { { 1, 1 }, { 1, 0 }, { 0, 1 }, { 0, 0 } };
            int[] salidas = { 1, 0, 0, 0 };

            //Inicializa los pesos al azar
            double P0 = azar.NextDouble();
            double P1 = azar.NextDouble();
            double U = azar.NextDouble();

            //Variable que mantiene el proceso activo de buscar los pesos
            bool proceso = true;

            //Cuenta el número de iteraciones
            int iteracion = 0;

            //Tasa de aprendizaje
            double tasaAprende = 0.3;

            //Hasta que aprenda la tabla AND
            while (proceso) {
                iteracion++;

                //Optimista, en esta iteración se encuentra los pesos
                proceso = false;

                //Va por todas las reglas de la tabla AND
                for (int cont = 0; cont <= 3; cont++) {

                    //Calcula el valor de entrada a la función
                    double operacion = entradas[cont, 0] * P0 + entradas[cont, 1] * P1 + U;

                    //La función de activación
                    int salidaEntera;
                    if (operacion > 0.7)
                        salidaEntera = 1;
                    else
                        salidaEntera = 0;

                    //El error
                    int error = salidas[cont] - salidaEntera;

                    //Si la salida no coincide con lo esperado, cambia los pesos con la ecuación
                    if (error != 0) {
                        P0 += tasaAprende * error * entradas[cont, 0];
                        P1 += tasaAprende * error * entradas[cont, 1];
                        U += tasaAprende * error * 1;
                        proceso = true; //Y sigue buscando
                    }
                }
            }

            //Muestra que el perceptrón simple aprendió.
            for (int cont = 0; cont <= 3; cont++) {
                double operacion = entradas[cont, 0] * P0 + entradas[cont, 1] * P1 + U;

                //La función de activación
                int salidaEntera;
                if (operacion > 0.7)
                    salidaEntera = 1;
                else
                    salidaEntera = 0;

                //Imprime
                Console.WriteLine("Entradas: " + entradas[cont, 0].ToString() + " y " + entradas[cont, 1].ToString() + " = " +
```

```
        salidas[cont].ToString() + " perceptron: " + salidaEntera.ToString());
    }

    Console.WriteLine("Pesos encontrados P0= " + P0.ToString() + " P1= " + P1.ToString() + " U= " + U.ToString());
    Console.WriteLine("Iteraciones requeridas: " + iteracion.ToString());
    Console.ReadKey();
}
}
```

Ejecución 1:

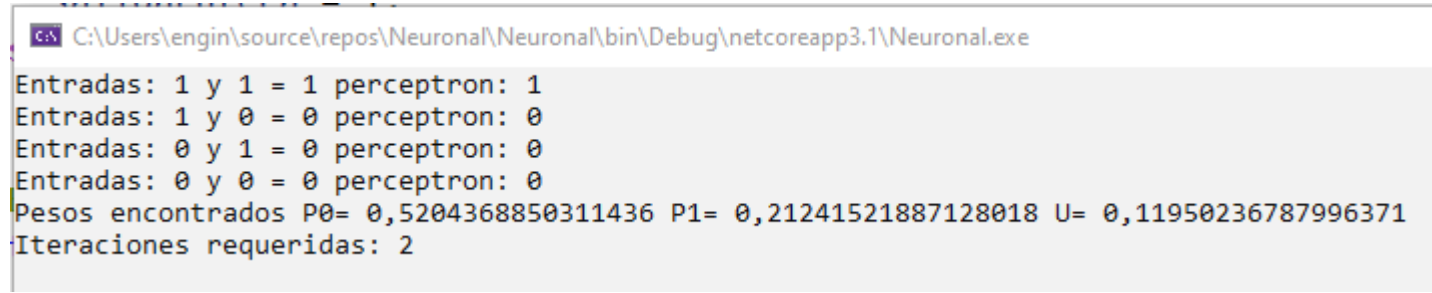


Ilustración 13: Encuentra los pesos con pocas iteraciones

Ejecución 2:

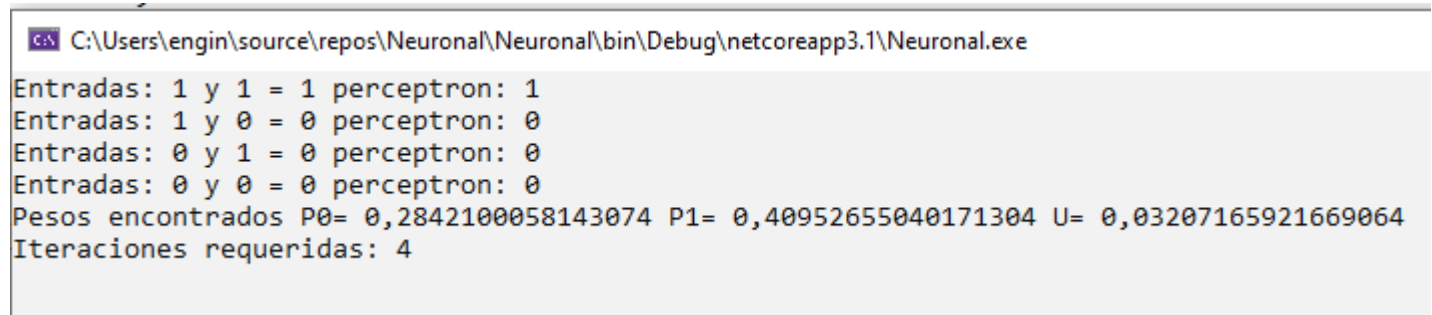


Ilustración 14: Encuentra los pesos con pocas iteraciones

Como se puede observar, se necesitan menos iteraciones en promedio usando la fórmula para hallar los pesos.

El ejemplo anterior el perceptr3n simple aprenda la tabla AND, 3y con la OR?

A	B	Resultado (A OR B)
Verdadero	Verdadero	Verdadero
Verdadero	Falso	Verdadero
Falso	Verdadero	Verdadero
Falso	Falso	Falso

Paso 1: Volver cuantitativa esa tabla (1 es verdadero, 0 es falso)

A	B	Resultado (A OR B)
1	1	1
1	0	1
0	1	1
0	0	0

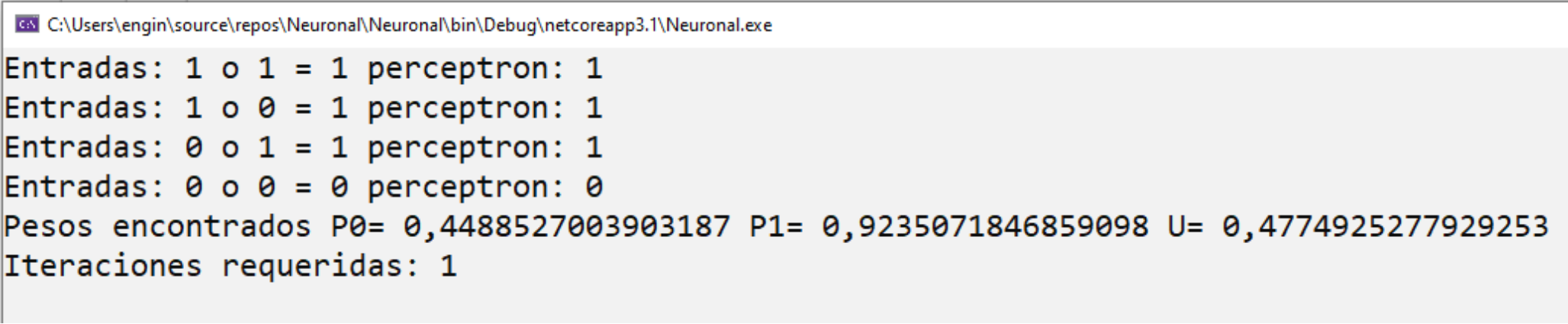
Es s3lo cambiar estas l3neas del programa

```
//Tabla de verdad AND
int[,] entradas = { { 1, 1 }, { 1, 0 }, { 0, 1 }, { 0, 0 } };
int[] salidas = { 1, 0, 0, 0 };
```

por estas

```
//Tabla de verdad OR
int[,] entradas = { { 1, 1 }, { 1, 0 }, { 0, 1 }, { 0, 0 } };
int[] salidas = { 1, 1, 1, 0 };
```

Y volver a ejecutar la aplicaci3n



Ilustraci3n 15: Encuentra los pesos. El perceptr3n simple ha aprendido la tabla del OR

Límites del Perceptrón Simple

El perceptrón simple tiene un límite: que sólo sirve cuando la solución se puede separar con **una** recta. Se explica a continuación

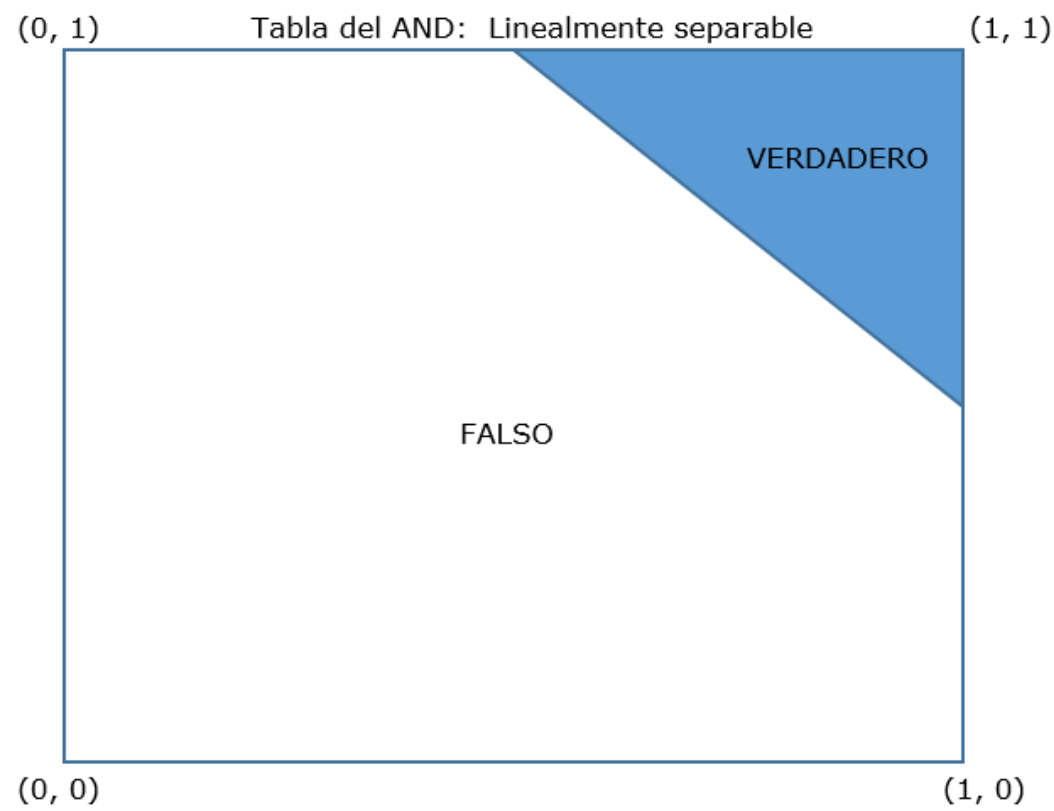


Ilustración 16: Tabla del AND

En cambio, si se quiere abordar un problema que requiera dos separaciones, no lo podría hacer el perceptrón simple. El ejemplo clásico es la tabla XOR:

A	B	Resultado (A XOR B)
Verdadero	Verdadero	Falso
Verdadero	Falso	Verdadero
Falso	Verdadero	Verdadero
Falso	Falso	Falso

Volviendo cuantitativa esa tabla:

A	B	Resultado (A XOR B)
1	1	0
1	0	1
0	1	1
0	0	0

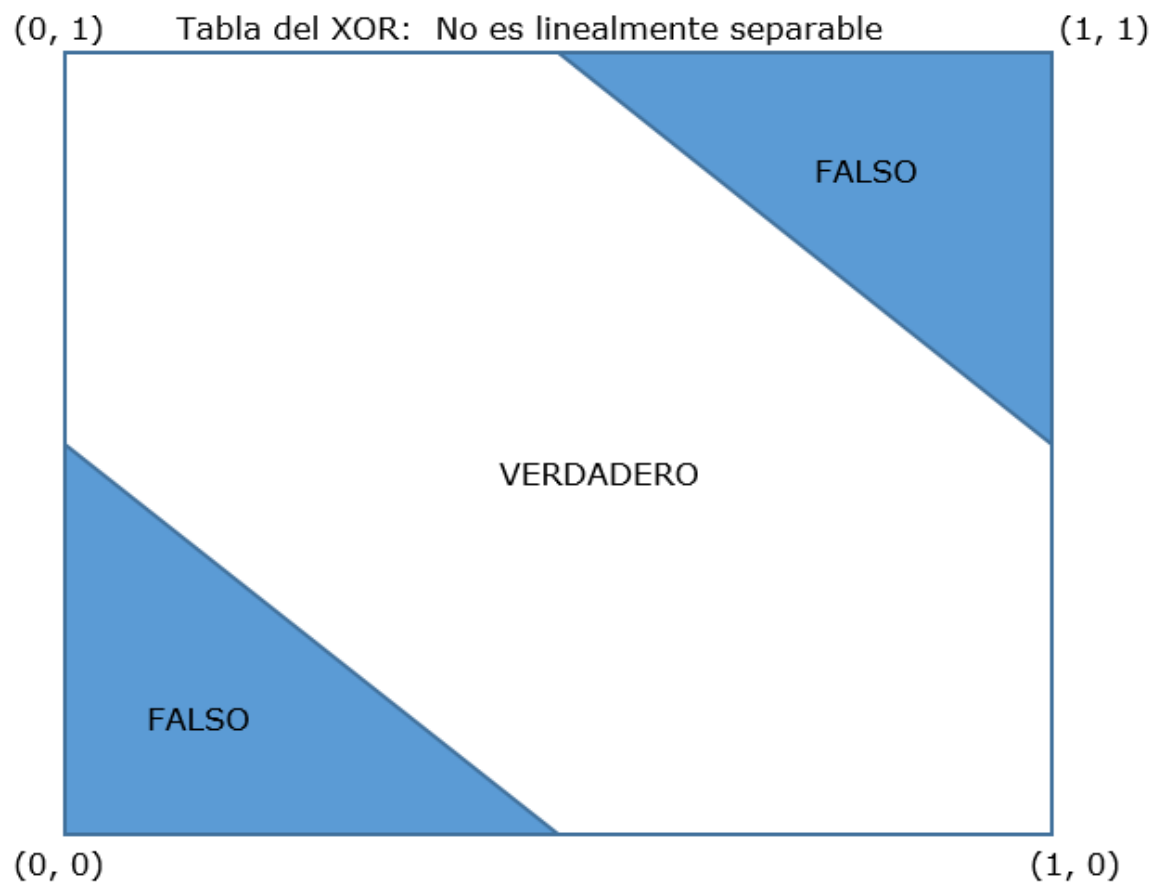


Ilustración 17: Tabla del XOR

Para solucionar ese problema es necesario usar más neuronas puestas en varias capas. Por ejemplo:

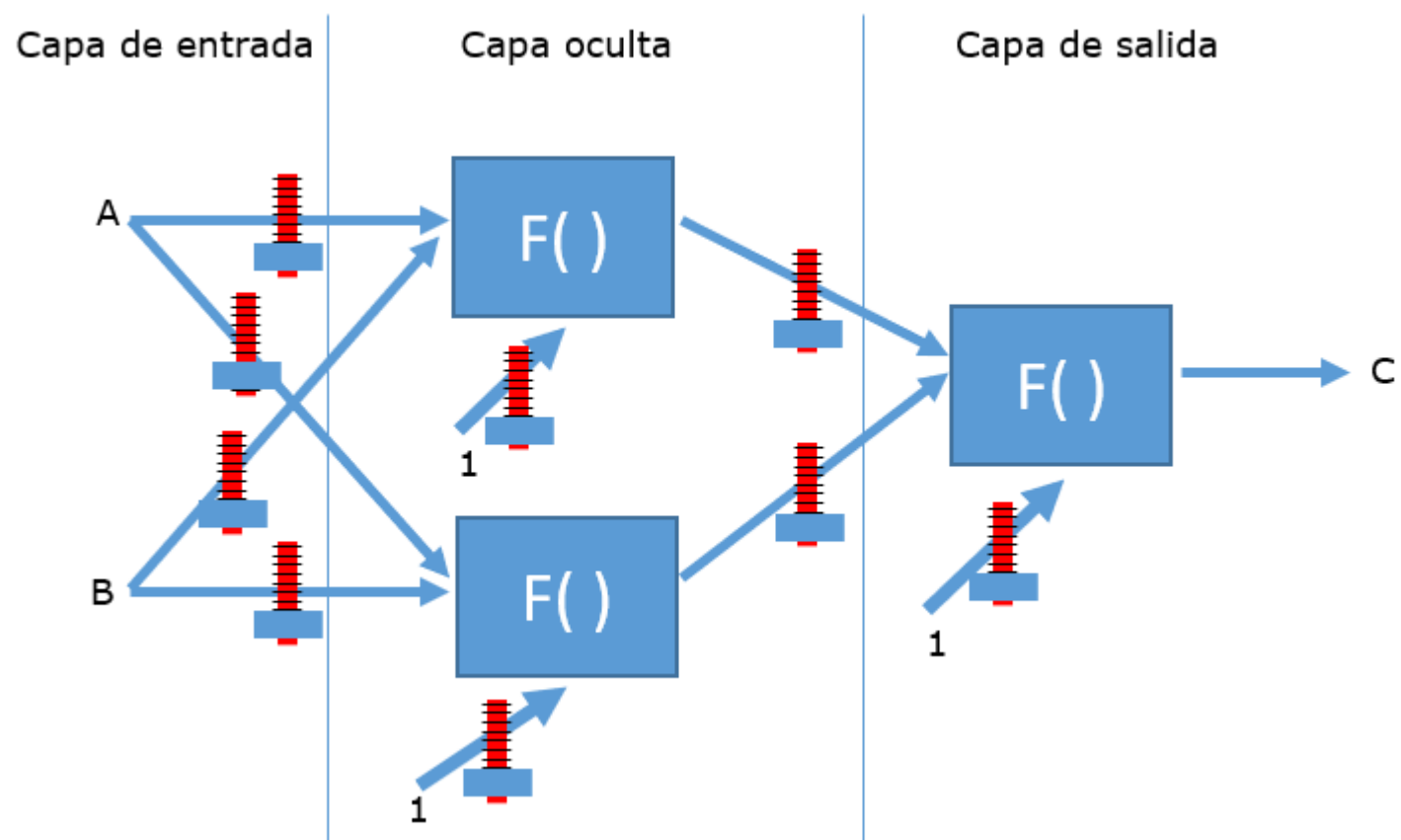


Ilustración 18: Red neuronal con 3 neuronas

O así

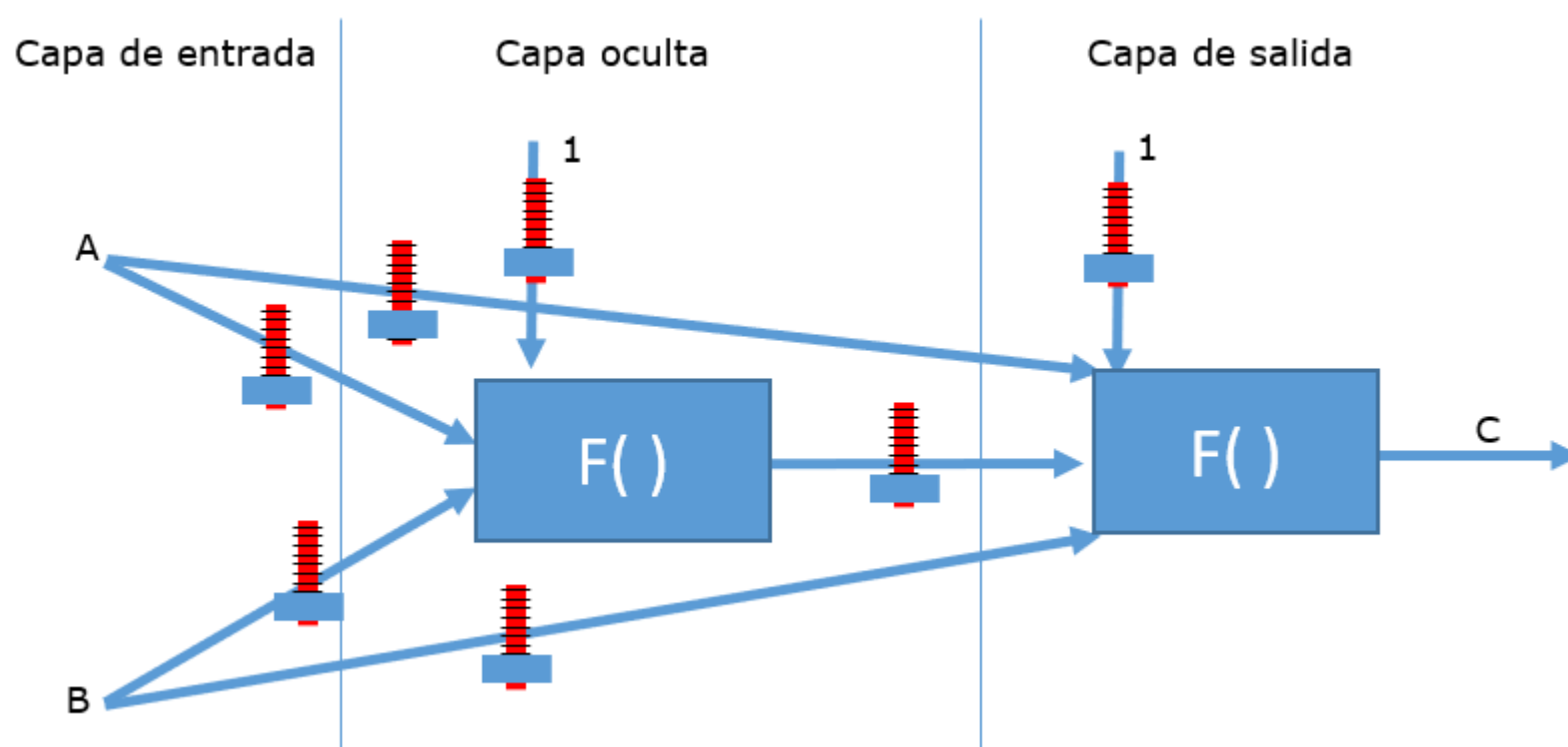


Ilustración 19: Red neuronal con otro tipo de conexiones

Muchos más pesos, luego el reto es cómo dar con cada peso para que se cumplan las salidas. Hay entonces un modelo matemático para lograr esto.

Encontrando el mínimo en una ecuación

A continuación, se explica la matemática que ayudará a deducir los pesos en una red neuronal.
Para dar con el mínimo de una ecuación se hace uso de las derivadas. Por ejemplo, dada la ecuación

$$y = 5 * x^2 - 7 * x - 13$$

Tabla de datos y gráfico

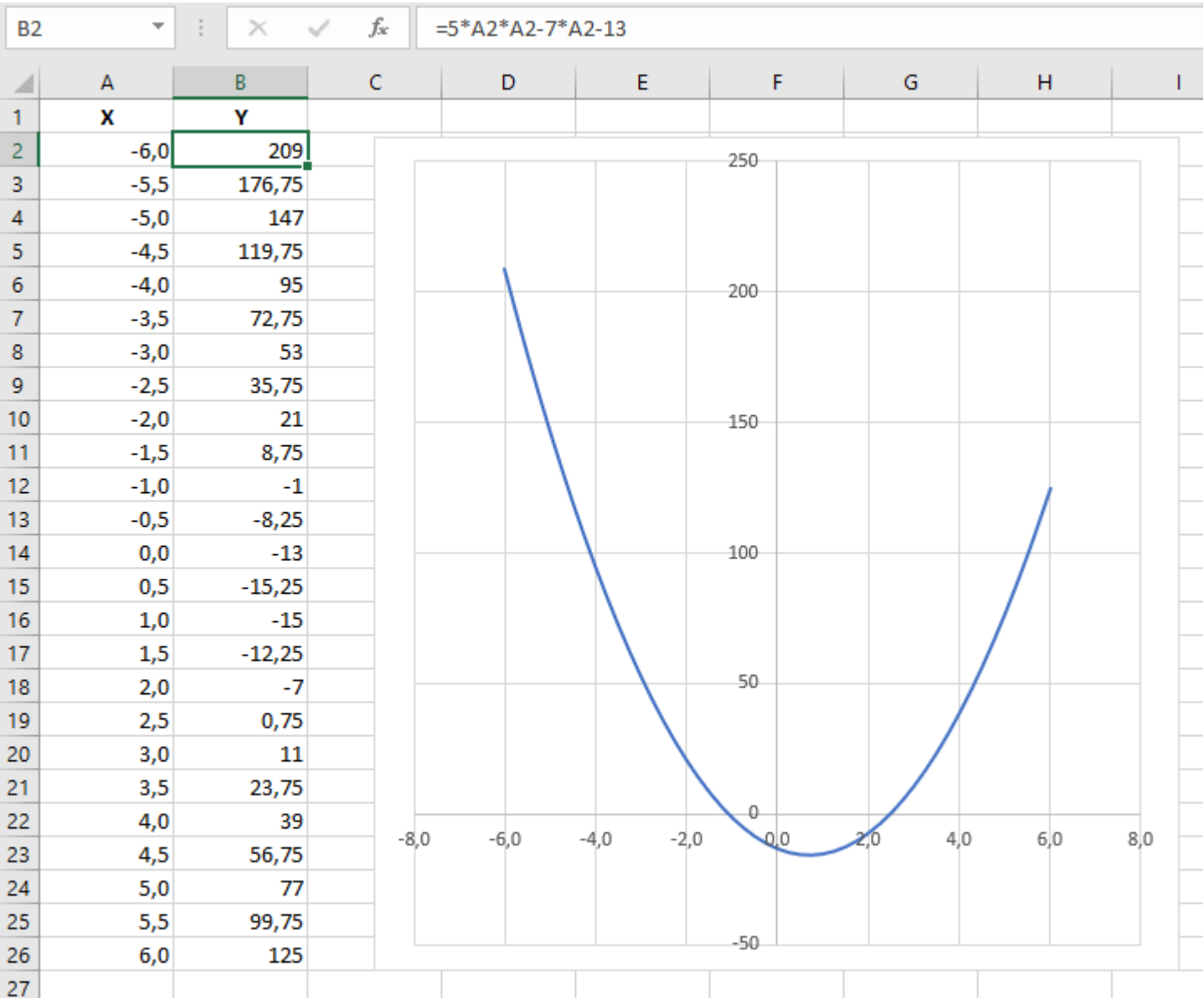


Ilustración 20: Tabla y gráfico de la ecuación hechos con Microsoft Excel

Si queremos dar con el valor de x para que y sea el mínimo valor, el primer paso es derivar

$$y' = 10 * x - 7$$



derivate 5*x^2-7*x-13

Extended Keyboard

Upload

Examples

Random

Derivative:

☒ Step-by-step solution

$$\frac{d}{dx}(5x^2 - 7x - 13) = 10x - 7$$

Ilustración 21: Derivada usando en WolframAlpha

Luego esa derivada se iguala a cero

$$0 = 10 * x - 7$$

Se resuelve el valor de x

$$x = 7/10$$

$$x = 0.7$$

Y tenemos el valor de x con el que se obtiene el mínimo valor de y

$$y = 5 * x^2 - 7 * x - 13$$

$$y = 5 * 0.7^2 - 7 * 0.7 - 13$$

$$y = -15.45$$

En este caso fue fácil dar con la derivada, porque fue un polinomio de grado 2, el problema sucede cuando la ecuación es compleja, derivarla se torna un desafío y despejar x sea muy complicado.

Otra forma de dar con el mínimo es iniciar con algún punto x al azar, por ejemplo, $x = 1.0$

Valor de X	$y = 5 * x^2 - 7 * x - 13$
1.0	-15

Luego un desplazamiento tanto a la izquierda como a la derecha de 0.5 en 0.5, es decir, $x=0.5$ y $x=1.5$

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.5	-15.25
1.0	-15
1.5	-12.25

Se obtiene un nuevo valor de X más prometedor que es 0.5, luego se repite el procedimiento, izquierda y derecha, es decir, $x=0.0$ y $x=1.0$

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.0	-13
0.5	-15.25
1.0	-15

El valor de 0.5 se mantiene como el mejor, luego se hace izquierda y derecha a un paso menor de 0.25

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.25	-14.4375
0.50	-15.25
0.75	-15.4375

El valor de $x=0.75$ es el que muestra mejor comportamiento, luego se hace izquierda y derecha a un paso de 0.25

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.50	-15.25
0.75	-15.4375
1.00	-15

Sigue $x=0.75$ como mejor valor, luego se prueba a izquierda y derecha, pero en una variación menor de 0.125

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.625	-15.421875
0.75	-15.4375
0.875	-15.296875

Sigue $x=0.75$ como mejor valor, luego se prueba a izquierda y derecha, pero en una variación menor de 0.0625

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.6875	-15.4492188
0.75	-15.4375
0.8125	-15.3867188

Ahora es $x=0.6875$ como mejor valor, luego se prueba a izquierda y derecha en una variación de 0.0625

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.625	-15.421875

0.6875	-15.4492188
0.75	-15.4375

Sigue x=0.6875 como mejor valor, luego se prueba a izquierda y derecha, pero en una variación menor de 0.03125

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.65625	-15.4404297
0.6875	-15.4492188
0.71875	-15.4482422

Sigue x=0.6875 como mejor valor, luego se prueba a izquierda y derecha, pero en una variación menor de 0.015625

Valor de X	$y = 5 * x^2 - 7 * x - 13$
0.671875	-15.4460449
0.6875	-15.4492188
0.703125	-15.4499512

Ahora es x=0,703125 como mejor valor. Este método poco a poco se aproxima a x=0.7 que es el resultado que se dedujo con las derivadas.

Implementado en C#:

```
using System;

namespace Minimo {
    class Program {
        static void Main(string[] args) {
            double x = 1; //valor inicial
            double Yini = Ecuacion(x);
            double variacion = 1;

            while (Math.Abs(variacion) > 0.00001) {
                double Ysigue = Ecuacion(x + variacion);
                if (Ysigue > Yini) { //Si no disminuye, cambia de dirección a un paso menor
                    variacion *= -1;
                    variacion /= 10;
                }
                else {
                    Yini = Ysigue; //Disminuye
                    x += variacion;
                    Console.WriteLine("x: " + x.ToString() + " Yini:" + Yini.ToString());
                }
            }
            Console.WriteLine("Respuesta: " + x.ToString());
            Console.ReadKey();
        }

        static double Ecuacion(double x) {
            return 5 * x * x - 7 * x - 13;
        }
    }
}
```

Y así ejecuta

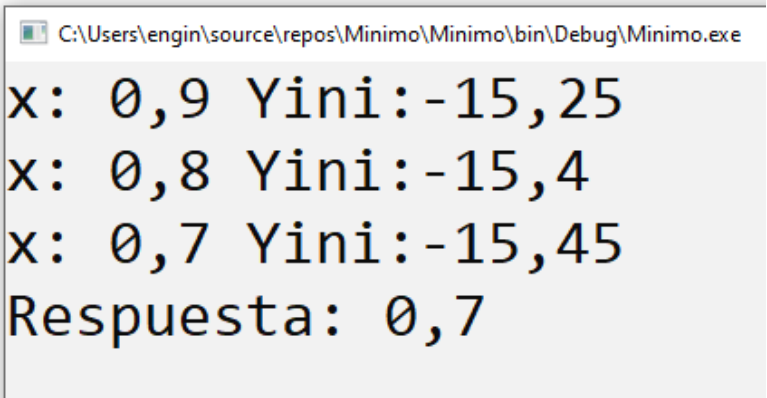


Ilustración 22: Buscando el mínimo de una ecuación

Y cambiando el valor inicial de x a un valor x=1.13 por ejemplo, esto pasaría:

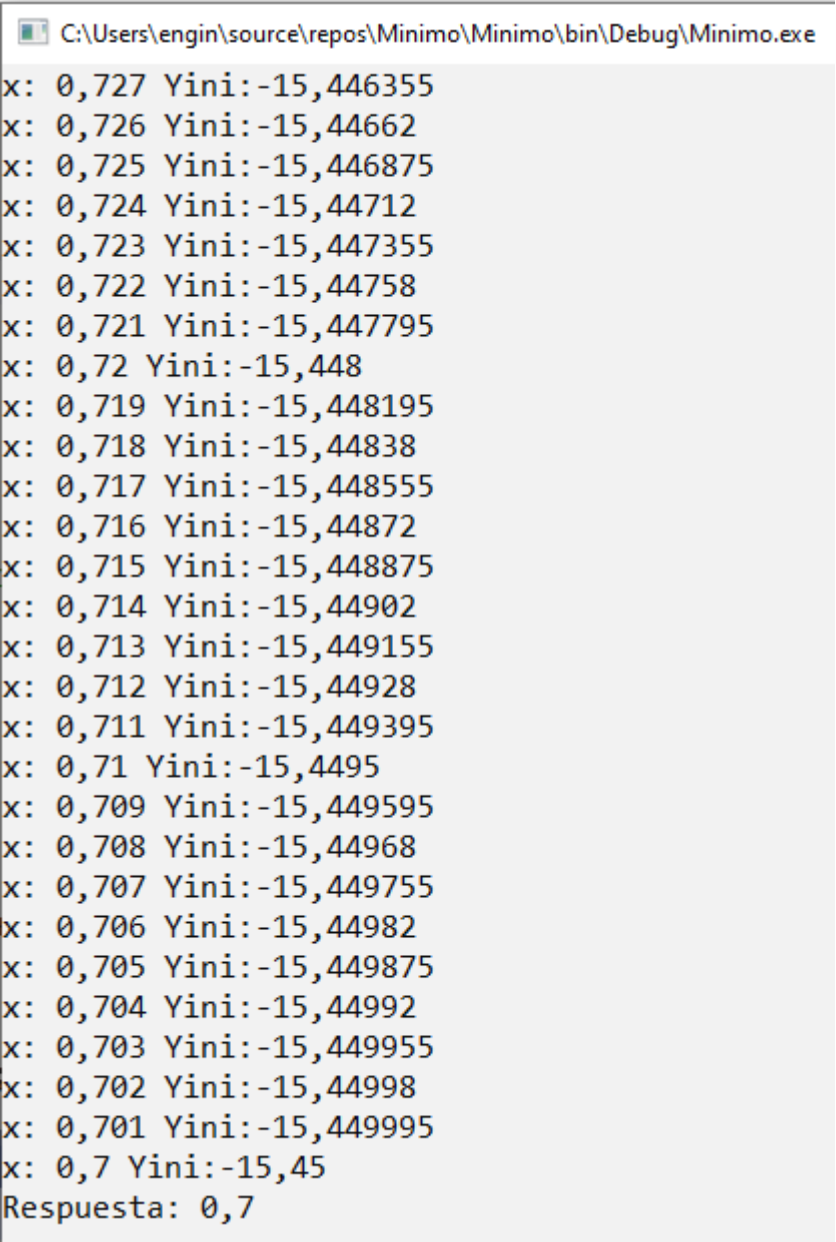


Ilustración 23: Buscando el mínimo de una ecuación

Descenso del gradiente

Anteriormente se mostró, con las aproximaciones, como buscar el mínimo valor de Y modificando el valor de X, ya sea yendo por la izquierda (disminuyendo) o por la derecha (aumentando). Matemáticamente para saber en qué dirección ir, es con esta expresión:

$$\Delta x = -y'$$

¿Qué significa? Que x debe modificarse en contra de la derivada de la ecuación.

¿Por qué? La derivada muestra la tangente que pasa por el punto que se seleccionó al azar al inicio. Esa tangente es una línea recta y como toda línea recta tiene una pendiente. Si la pendiente es positiva entonces X se debe ir hacia la izquierda (el valor de X debe disminuir), en cambio, si la pendiente es negativa entonces X debe ir hacia la derecha (el valor de X debe aumentar). Con esa indicación ya se sabe por dónde ir para dar con el valor de X que obtiene el mínimo Y.

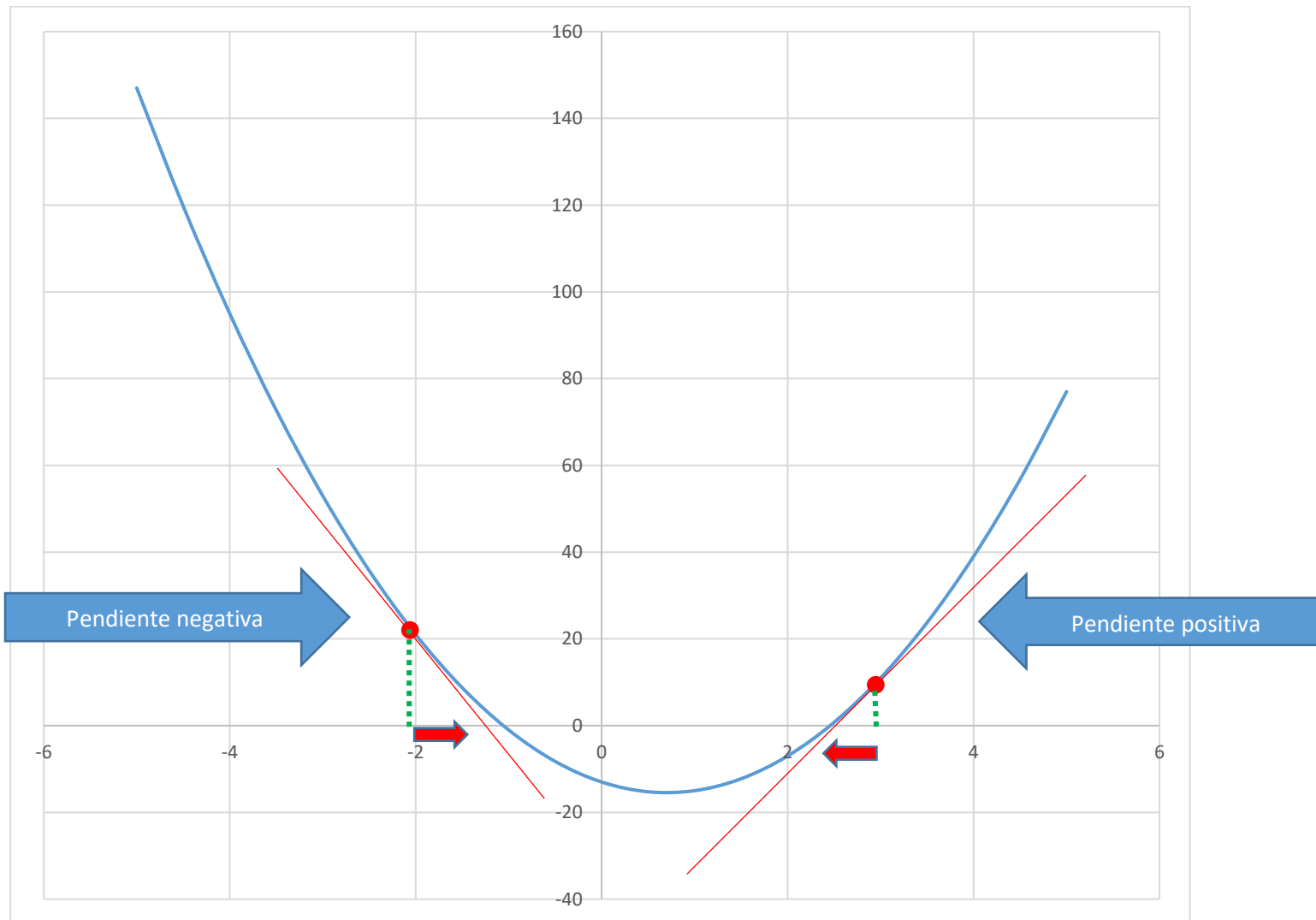


Ilustración 24: Pendientes

Para dar con el nuevo valor de X esta sería la expresión:

$$x_{nuevo} = x_{anterior} + \Delta x$$

Reemplazando

$$x_{nuevo} = x_{anterior} - y'$$

EJEMPLO

Con la ecuación anterior

$$y = 5 * x^2 - 7 * x - 13$$

$$y' = 10 * x - 7$$

$$x_{nuevo} = x_{anterior} - y'$$

$$x_{nuevo} = x_{anterior} - (10 * x - 7)$$

X inicia en 0.4 por ejemplo, luego

$$x_{nuevo} = 0.4 - (10 * 0.4 - 7)$$

$$x_{nuevo} = -3.4$$

Ahora hay un nuevo valor para X que es -2. En la siguiente tabla se muestra como progresa X

Xanterior	Xnuevo	Y
0.4	3.4	21
3.4	-23.6	2937
-23.6	219.4	239133
219.4	-1967.6	19371009
-1967.6	17715.4	1569052965
17715.4	-159431.6	1.27093E+11
-159431.6	1434891.4	1.02946E+13
1434891.4	-12914015.6	8.33859E+14
-12914015.6	116226147.4	6.75426E+16
116226147.4	-1046035320	5.47095E+18
-1046035320	9414317883	4.43147E+20

El valor de X se dispara, se vuelve extremo hacía la izquierda o derecha. Se debe arreglar agregando una constante a la ecuación:

$$x_{nuevo} = x_{anterior} - \alpha * y'$$

Se agrega entonces un α que es una constante muy pequeña, por ejemplo $\alpha=0.05$ y esto es lo que sucede

$$x_{nuevo} = x_{anterior} - 0.05 * (10 * x_{anterior} - 7)$$

Xanterior	Xnuevo	Y
0.4	0.55	-15.3375
0.55	0.625	-15.421875
0.625	0.6625	-15.4429688
0.6625	0.68125	-15.4482422
0.68125	0.690625	-15.4495605
0.690625	0.6953125	-15.4498901
0.6953125	0.69765625	-15.4499725
0.69765625	0.698828125	-15.4499931
0.698828125	0.699414063	-15.4499983
0.699414063	0.699707031	-15.4499996
0.699707031	0.699853516	-15.4499999

Tiene más sentido y se acerca a X=0.7 que es la respuesta correcta.

Este método se le conoce como el descenso del gradiente que se expresa así

$$x_{nuevo} = x_{anterior} - \alpha * f'(x_{anterior})$$

En formato matemático

$$x_{n+1} = x_n - \alpha * f'(x_n)$$

La siguiente curva es generada por el siguiente polinomio

$$y = 0.1 * x^6 + 0.6 * x^5 - 0.7 * x^4 - 6 * x^3 + 2 * x^2 + 2 * x + 1$$

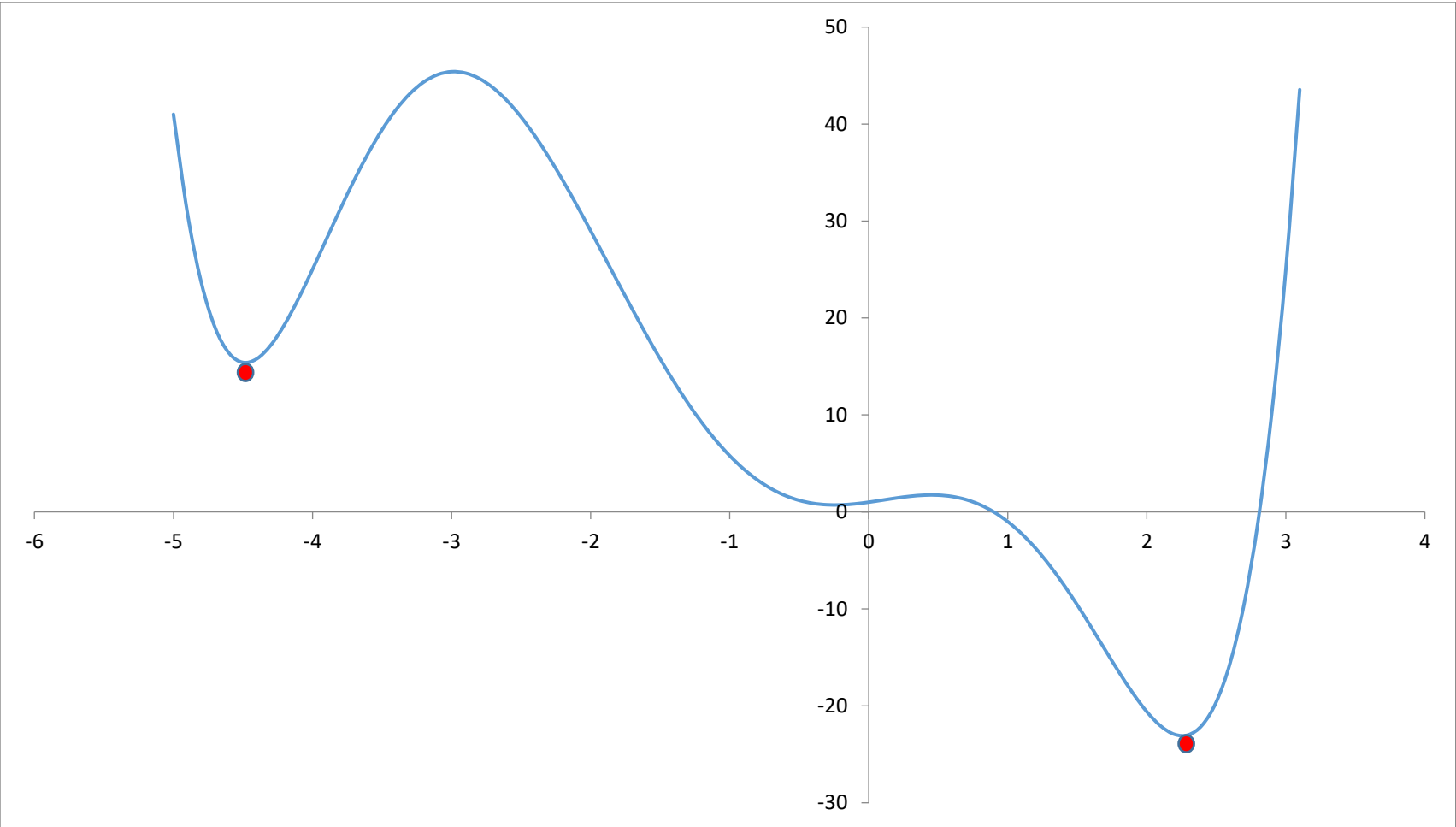


Ilustración 25: Gráfico de un polinomio de quinto grado

Se aprecian dos puntos donde claramente la curva desciende y vuelve a ascender (se han marcado con puntos en rojo), por supuesto, el segundo a la derecha es el mínimo real, pero ¿Qué pasaría si se hubiese hecho una búsqueda iniciando en x=-4? La respuesta es que el algoritmo se hubiese decantado por el mínimo de la izquierda:

$$x_{nuevo} = x_{anterior} - \alpha * y'$$

$$x_{nuevo} = x_{anterior} - 0.01 * (0.6 * x^5 + 3 * x^4 - 2.8 * x^3 - 18 * x^2 + 4 * x + 2)$$

Xanterior	Xnuevo	Y
-4	-4.308	25
-4.308	-4.4838	17.0485
-4.4838	-4.4815	15.3935
-4.4815	-4.4822	15.3933
-4.4822	-4.482	15.3933
-4.482	-4.482	15.3933

Este problema se le conoce como caer en mínimo local y también lo sufren los algoritmos genéticos. Así que se deben probar otros valores de X para iniciar, si fuese X=2 observamos que si acierta con el mínimo real:

Xanterior	Xnuevo	Y
2	2.172	-20.6
2.172	2.24349	-22.777
2.24349	2.25489	-23.08
2.25489	2.25559	-23.087
2.25559	2.25562	-23.087
2.25562	2.25563	-23.087
2.25563	2.25563	-23.087

Fue fácil darse cuenta donde está el mínimo real viendo la gráfica, pero el problema estará vigente cuando no sea fácil generar el gráfico o peor aún, cuando no sea una sola variable independiente f(x) sino varias, como funciones del tipo f(a,b,c,d,e)

En la figura, hay dos entradas: A y B, y una salida: C, todo eso son constantes porque son los datos de entrenamiento, no tenemos control sobre estos. Lo que, si podemos variar, son los pesos. Si queremos saber que tanto debe ajustar cada peso, el procedimiento matemático de obtener mínimos, se enfoca solamente en esos pesos.

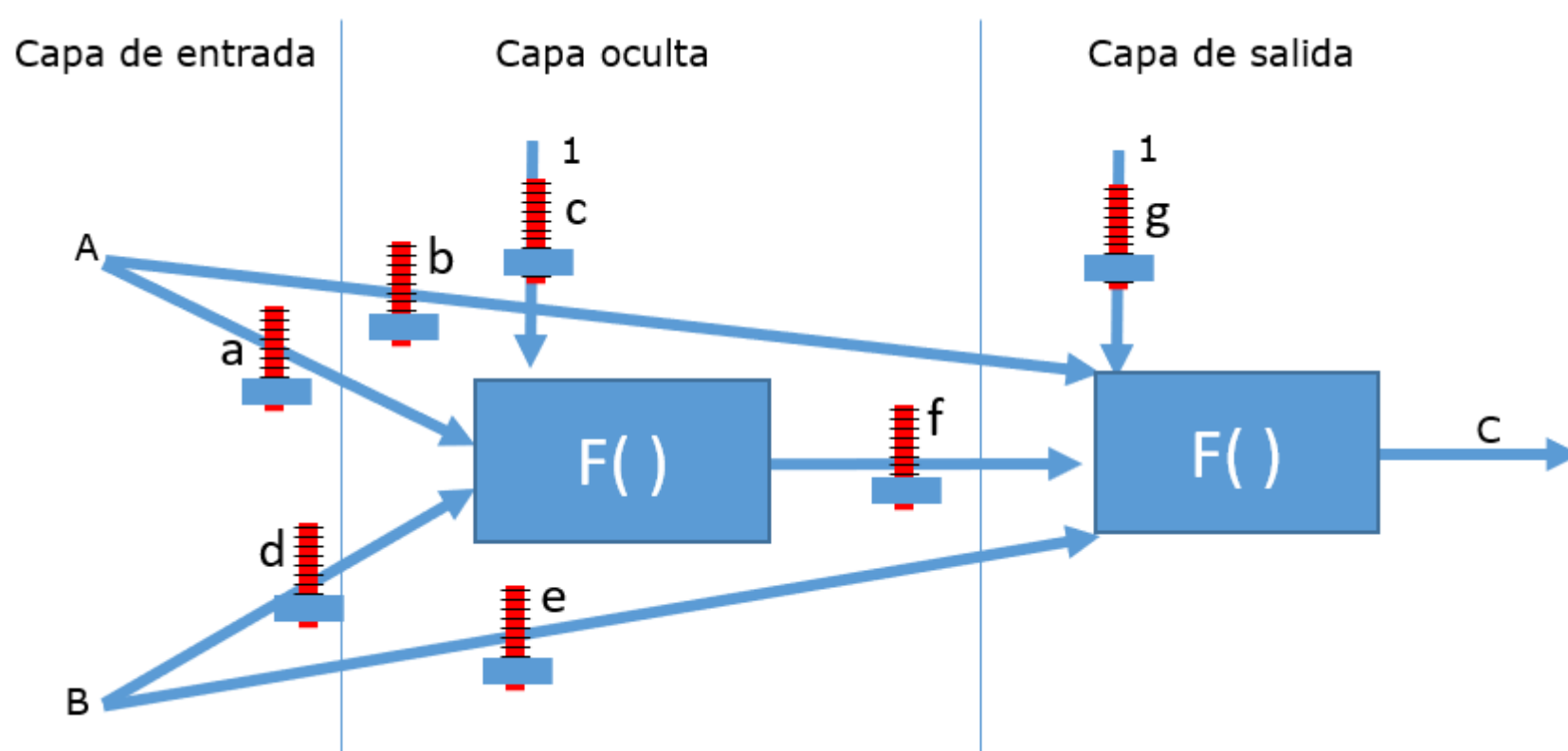


Ilustración 26: Red neuronal con varias conexiones distintas

En la figura se aprecian 7 pesos: a,b,c,d,e,f,g. ¿Cómo obtener un mínimo? En ese caso se utilizan derivadas parciales, es decir, se deriva por 'a' dejando el resto como constantes, luego por 'b' dejando el resto constantes y así sucesivamente. Esos mínimos servirán para ir ajustando los pesos.

Perceptrón Multicapa

Es un tipo de red neuronal en donde hay varias capas:

1. Capa de entrada
2. Capas ocultas
3. Capa de salida

En la siguiente figura se muestra un ejemplo de perceptrón multicapa, los círculos representan las neuronas. Tiene dos capas ocultas. Las capas ocultas donde cada una tiene 3 neuronas y la capa de salida con 2 neuronas.

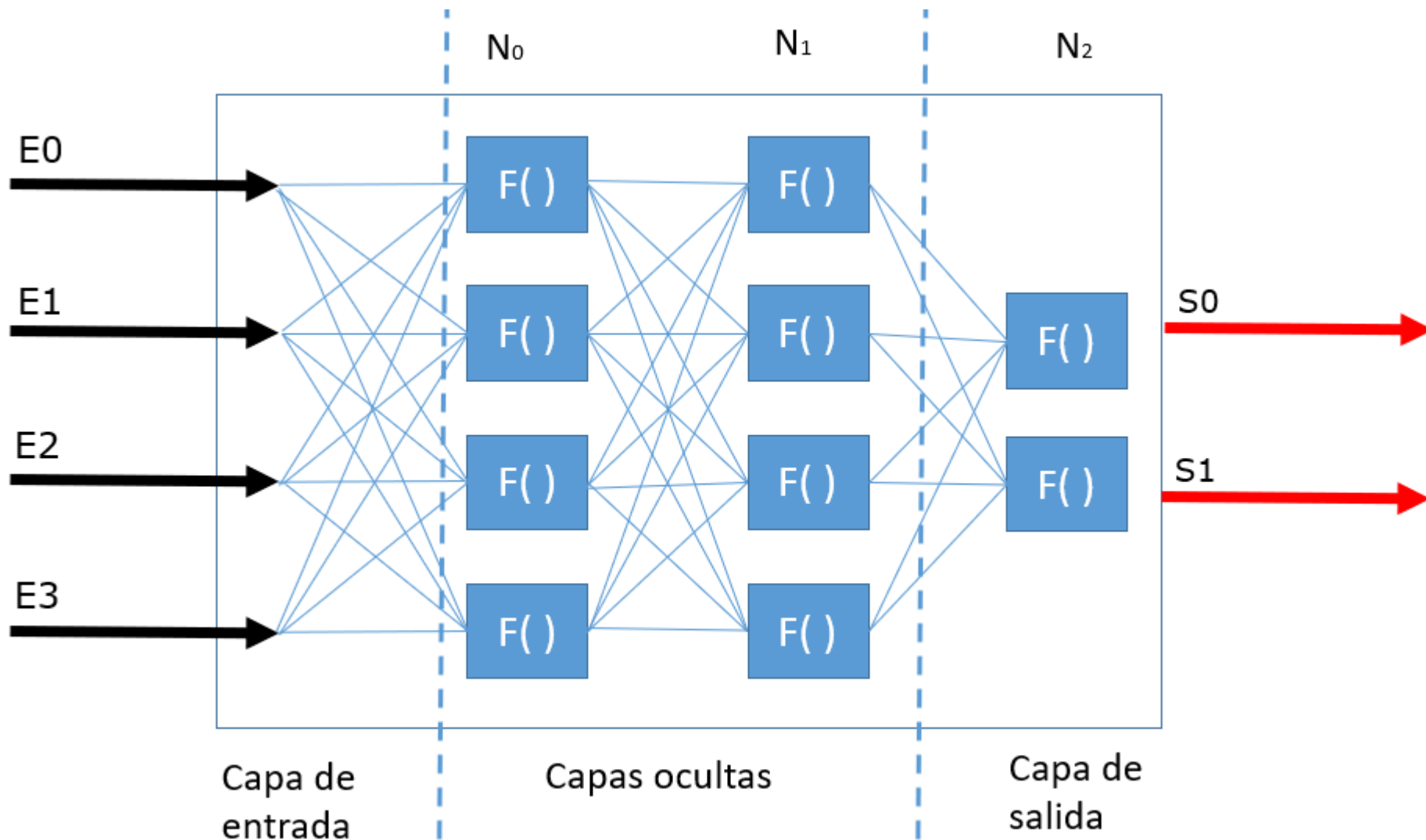


Ilustración 27: Perceptrón multicapa

Las capas se denotarán con la letra 'N', luego

$N_0=4$ (capa 0, que es oculta, tiene 4 neuronas)

$N_1=4$ (capa 1, que es oculta, tiene 4 neuronas)

$N_2=2$ (capa 2, que es la de salida, tiene 2 neuronas)

La capa de entrada no hace ningún proceso, sólo recibe los datos de entrada.

En el perceptrón multicapa, las neuronas de la capa 0 se conectan con las neuronas de la capa 1, las neuronas de la capa 1 con las neuronas de la capa 2. No está permitido conectar neuronas de la capa 0 con las neuronas de la capa 2 por ejemplo, ese salto podrá suceder en otro tipo de redes neuronales, pero no en el perceptrón multicapa.

Las neuronas

De nuevo se muestra un esquema de cómo es una neurona con dos entradas externas y su salida.

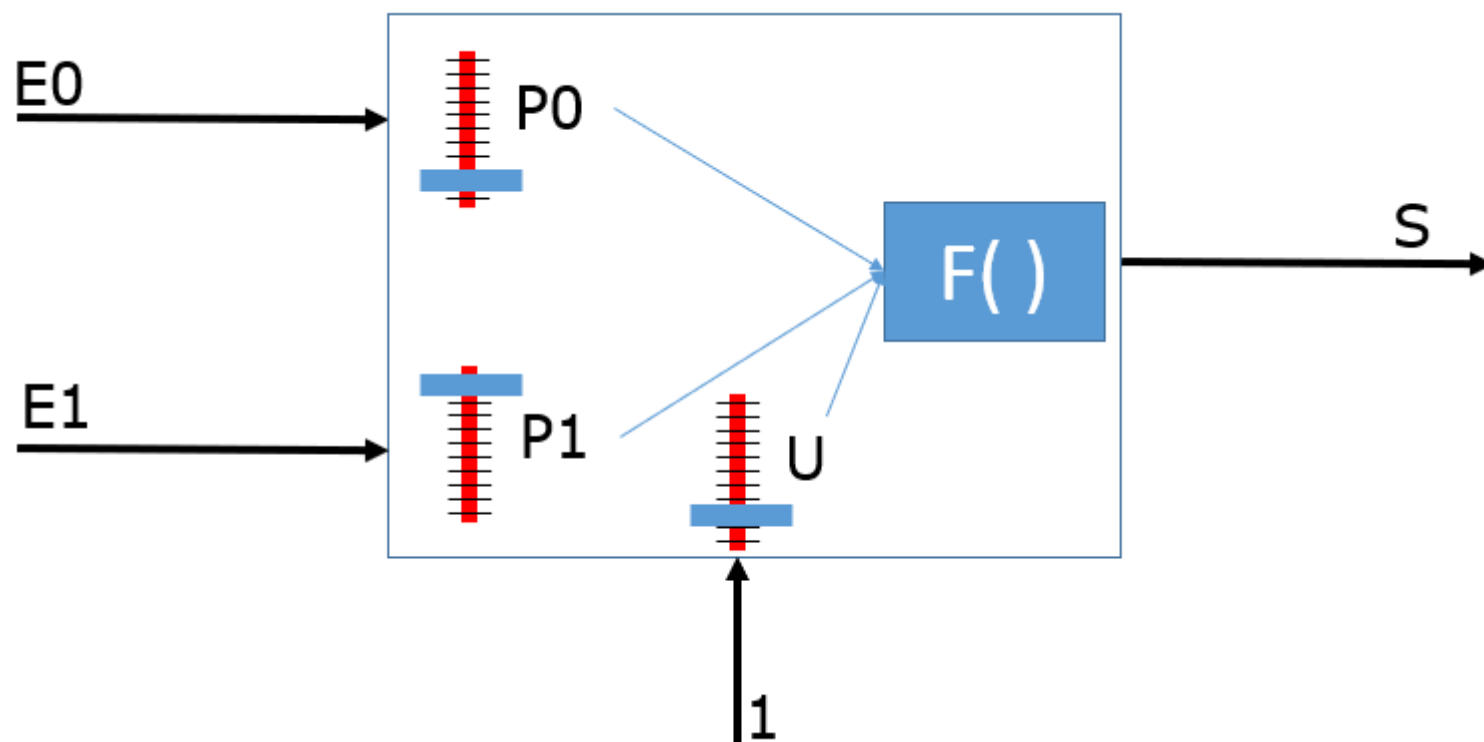


Ilustración 28: Esquema de una neurona

Mostrado como una clase en C#, esta sería su implementación:

```
namespace RedesNeuronales {
    class Neurona {
        public double calculaSalida(double E0, double E1) {
            double S;
            //Se hace una operación aquí

            return S;
        }
    }

    class Program {
        static void Main(string[] args) {
            Neurona algunaCapasOcultas = new Neurona();
            Neurona algunaCapaSalida = new Neurona();
        }
    }
}
```

En cada entrada hay un peso P_0 y P_1 . Para la entrada interna, que siempre es 1, el peso se llama U

```
namespace RedesNeuronales {
    class Neurona {
        //Pesos para cada entrada P0 y P1; y el peso de la entrada interna U
        private double P0;
        private double P1;
        private double U;

        public double calculaSalida(double E0, double E1) {
            double S;

            //Se hace una operación aquí

            return S;
        }
    }

    class Program {
        static void Main(string[] args) {
            Neurona algunaCapasOcultas = new Neurona();
            Neurona algunaCapaSalida = new Neurona();
        }
    }
}
```

Los pesos se inicializan con un valor al azar y un buen sitio es hacerlo en el constructor. En el ejemplo se hace uso de la clase Random y luego NextDouble() que retorna un número real al azar entre 0 y 1.

```
namespace RedesNeuronales {
    class Neurona {
        //Pesos para cada entrada P0 y P1; y el peso de la entrada interna U
        private double P0;
        private double P1;
        private double U;

        public Neurona() { //Constructor
            Random azar = new Random();
            P0 = azar.NextDouble();
            P1 = azar.NextDouble();
            U = azar.NextDouble();
        }

        public double calculaSalida(double E0, double E1) {
            double S;

            //Se hace una operación aquí

            return S;
        }
    }

    class Program {
        static void Main(string[] args) {
            Neurona algunaCapasOcultas = new Neurona();
            Neurona algunaCapaSalida = new Neurona();
        }
    }
}
```

Hay que tener especial cuidado con los generadores de números aleatorios, no es bueno crearlos constantemente porque se corre el riesgo que inicien con una misma semilla (el reloj de la máquina) generando la misma colección de números aleatorios. A continuación, se modifica un poco el código para tener un solo generador de números aleatorios y evitar el riesgo de repetir números.

```

namespace RedesNeuronales {
    class Neurona {
        //Pesos para cada entrada P0 y P1; y el peso de la entrada interna U
        private double P0;
        private double P1;
        private double U;

        public Neurona(Random azar) { //Constructor
            P0 = azar.NextDouble();
            P1 = azar.NextDouble();
            U = azar.NextDouble();
        }

        public double calculaSalida(double E0, double E1) {
            double S;

            //Se hace una operación aquí

            return S;
        }
    }

    class Program {
        static void Main(string[] args) {
            Random azar = new Random(); //Un solo generador
            Neurona algunaCapasOcultas = new Neurona(azar);
            Neurona algunaCapaSalida = new Neurona(azar);
        }
    }
}

```

Pesos y como nombrarlos

En el gráfico se dibujan algunos pesos y como se podrá dilucidar, el número de estos pesos crece rápidamente a medida que se agregan capas y neuronas.

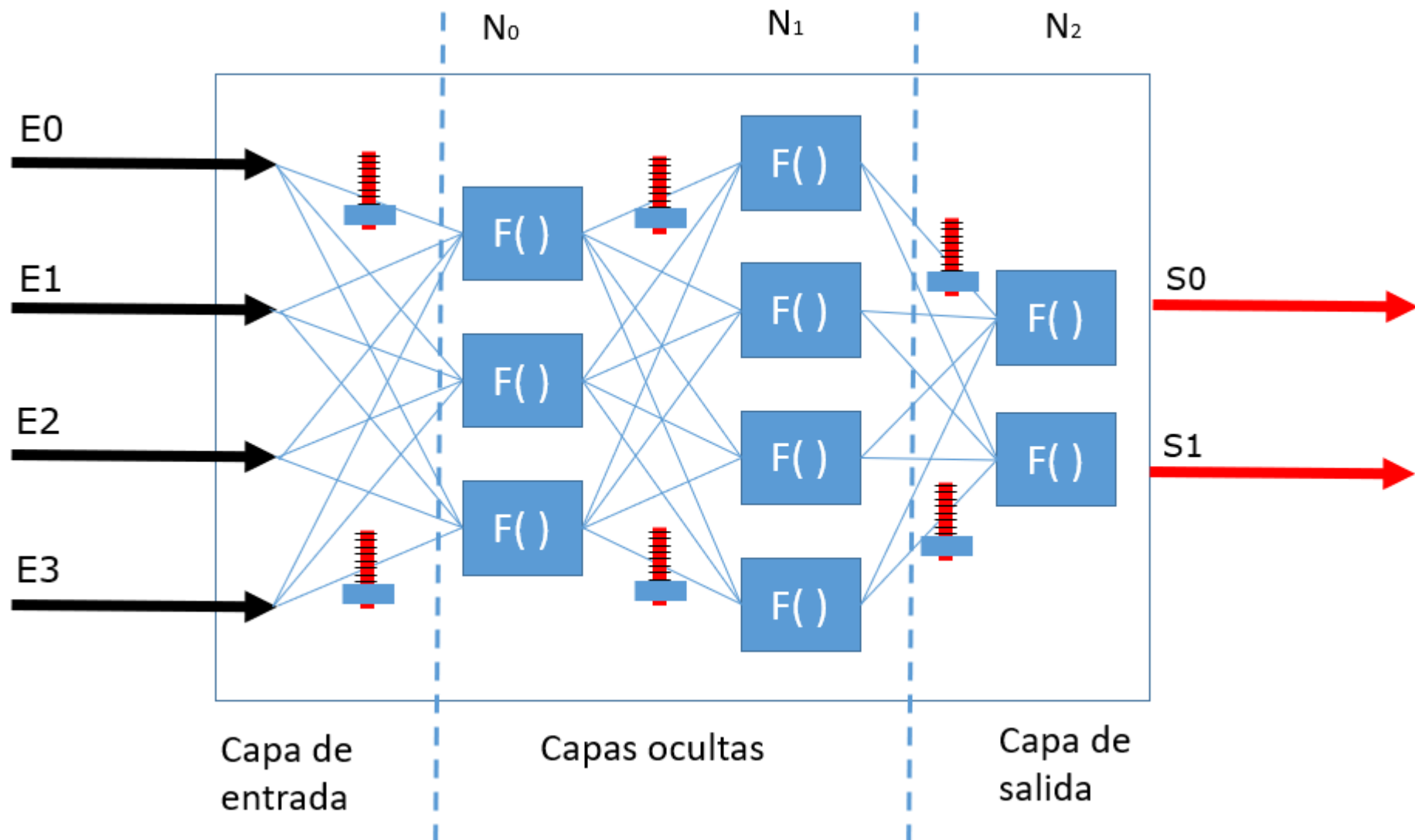


Ilustración 29: Esquema de un perceptrón multicapa

Un ejemplo: Capa 0 tiene 3 neuronas, capa 1 tiene 4 neuronas, luego el total de conexiones entre Capa 0 y Capa 1 son $3 \times 4 = 12$ conexiones, luego son 12 pesos. Si se usaran más neuronas por capa, habría tantos pesos que nombrarlos con una sola letra no sería conveniente. Por tal motivo, hay otra forma de nombrarlos y es el siguiente:

$$w_{\text{neurona inicial, neurona final}}^{(\text{capa a donde llega la conexión})}$$

W es la letra inicial de la palabra peso en inglés: Weight.

(Capa de donde sale la conexión) Las capas se enumeran desde 0 que sería en este caso la primera capa oculta. ¿Cómo nombrar los pesos iniciales? Esos pesos de la capa de entrada tendrán como "capa de donde sale la conexión" la letra E.

Neurona inicial, de donde parte la conexión. Se enumeran desde 0 de arriba abajo en la capa.

Neurona final, a donde llega la conexión. Se enumeran desde 0 de arriba abajo en la capa.

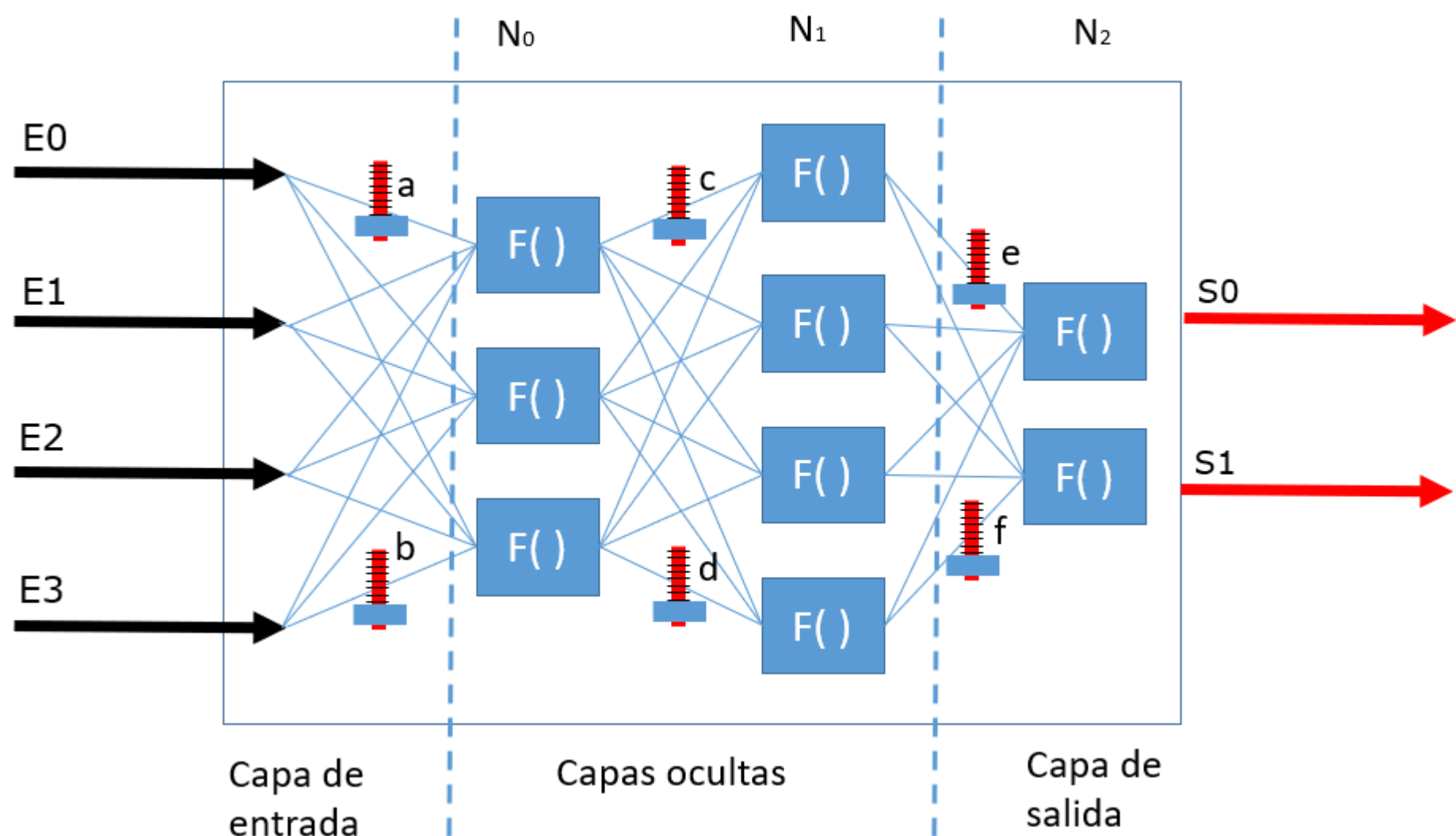


Ilustración 30: Nombrando los pesos con una letra

Para nombrar el peso mostrado con la letra 'a' sería entonces

$$w_{\text{neurona inicial, neurona final}}^{(\text{capa a donde llega la conexión})}$$

$$w_{0,0}^{(0)}$$

En esta tabla se muestra como se nombrarían los pesos que se han puesto en la gráfica

Peso	Se nombra
a	$w_{0,0}^{(0)}$
b	$w_{3,2}^{(0)}$
c	$w_{0,0}^{(1)}$
d	$w_{2,3}^{(1)}$
e	$w_{0,0}^{(2)}$
f	$w_{3,1}^{(2)}$

Anteriormente se había mostrado que la función de activación era esta:

```
Función F(valor)
Inicio
    Si valor > 0.7 entonces
        retorne 1
    de lo contrario
        retorne 0
    fin si
Fin
```

En otros problemas, por lo general, esa función es la sigmoide que tiene la siguiente ecuación:

$$y = \frac{1}{1 + e^{-x}}$$

Esta sería una tabla de valores generados con esa función

x	y
-10	4.5E-05
-9	0.00012
-8	0.00034
-7	0.00091
-6	0.00247
-5	0.00669
-4	0.01799
-3	0.04743
-2	0.1192
-1	0.26894
0	0.5
1	0.73106
2	0.8808
3	0.95257
4	0.98201
5	0.99331
6	0.99753
7	0.99909
8	0.99966
9	0.99988
10	0.99995

Y esta sería su gráfica

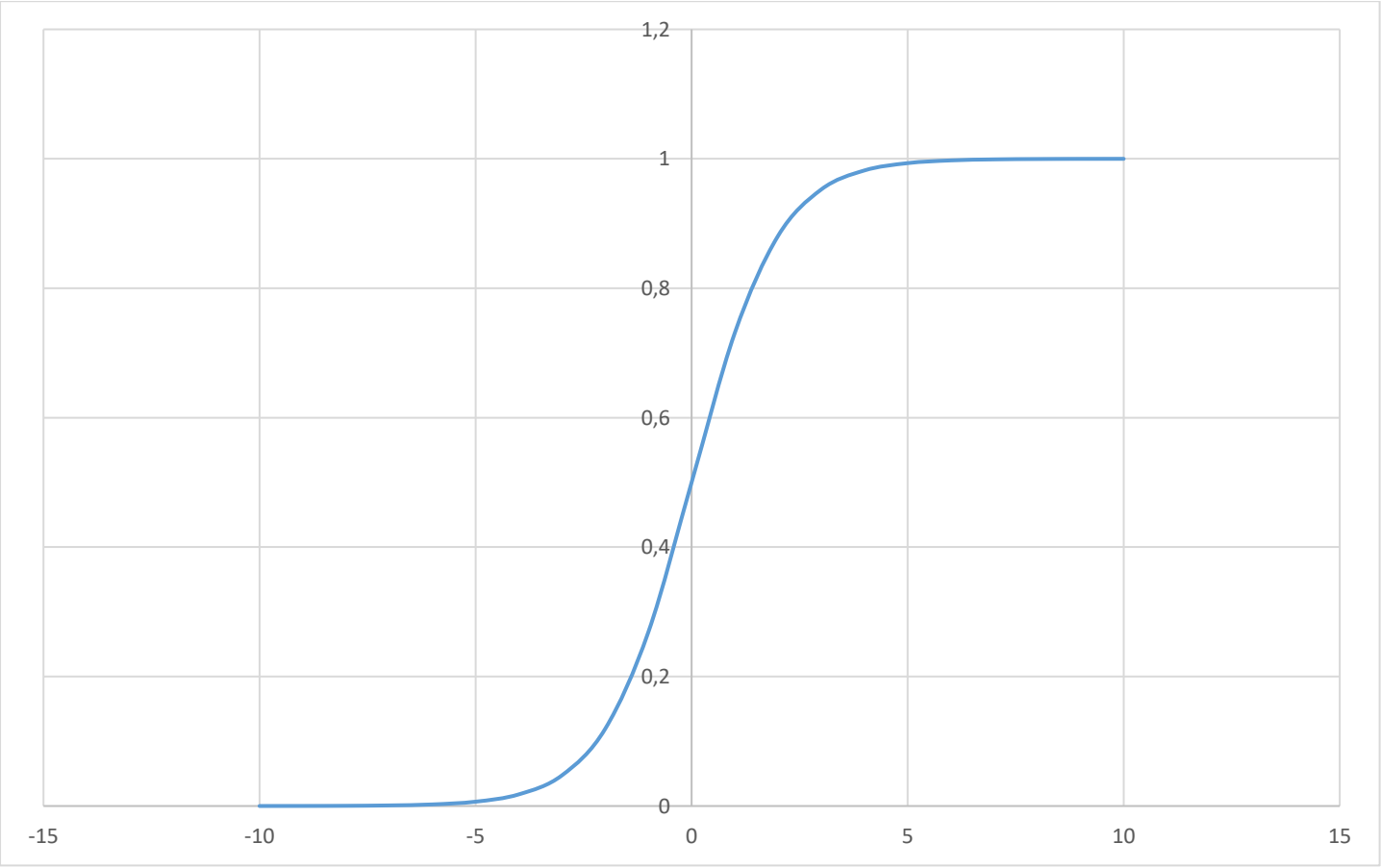


Ilustración 31: Gráfico de una función sigmoide

Al moverse a la izquierda el valor que toma es 0 y al moverse a la derecha toma el valor de 1. Hay una transición pronunciada de 0 a 1 en el rango [-5 y 5].

¿Qué tiene de especial esta función sigmoide? Su derivada.

Ecuación original:

$$y = \frac{1}{1 + e^{-x}}$$

Derivada no negativa de esa ecuación:

$$\partial y = \frac{e^{-x}}{(1 + e^{-x})^2}$$

Que equivale a esto:

$$\partial y = y * (1 - y)$$

Demostración de la equivalencia:

$$\begin{aligned}\partial y &= \frac{1}{1 + e^{-x}} * \left[1 - \frac{1}{1 + e^{-x}} \right] \\ \partial y &= \frac{1}{1 + e^{-x}} - \frac{1}{1 + e^{-x}} * \frac{1}{1 + e^{-x}} \\ \partial y &= \frac{1}{1 + e^{-x}} - \frac{1}{(1 + e^{-x})^2} \\ \partial y &= \frac{1 + e^{-x} - 1}{(1 + e^{-x})^2} \\ \partial y &= \frac{e^{-x}}{(1 + e^{-x})^2}\end{aligned}$$

Nota: Si hace uso de WolframAlpha y deriva la sigmoidea, sucede esto:

derive y=1/(1+e^-x)

Extended Keyboard

Upload

Input interpretation:

differentiate

$y = \frac{1}{1 + e^{-x}}$

with respect to

x

Result:

$y'(x) = \frac{e^x}{(e^x + 1)^2}$

Ilustración 32: Derivada de la función sigmoide

A primera vista parece diferente a la derivada mostrada anteriormente, pero si se compara

e^x/(e^x+1)^2 = e^-x/(1+e^-x)^2

Extended Keyboard

Upload

Input:

$\frac{e^x}{(e^x + 1)^2} = \frac{e^{-x}}{(1 + e^{-x})^2}$

Alternate form:

True

Ilustración 33: Comparativa de derivadas

El código del perceptrón multicapa progresa así:

```
using System;

namespace RedesNeuronales {
    class Neurona {
        //Pesos para cada entrada P0 y P1; y el peso de la entrada interna U
        private double P0;
        private double P1;
        private double U;

        public Neurona(Random azar) {
            P0 = azar.NextDouble();
            P1 = azar.NextDouble();
            U = azar.NextDouble();
        }

        public double calculaSalida(double E0, double E1) {
            double valor, S;

            valor = E0 * P0 + E1 * P1 + 1 * U;
            S = 1 / (1 + Math.Exp(-valor));

            return S;
        }
    }

    class Program {
        static void Main(string[] args) {
            Random azar = new Random(); //Un solo generador de números al azar
            Neurona algunaCapasOcultas = new Neurona(azar);
            Neurona algunaCapaSalida = new Neurona(azar);
        }
    }
}
```

El método calculaSalida implementa el procesamiento de la neurona. Tiene como parámetros las entradas, en este caso, dos entradas E0 y E1. En el interior cada entrada se multiplica con su peso respectivo, se suman, incluyendo la entrada interna (umbral). Una vez con ese valor, se calcula la salida con la sigmoide.

Introducción al algoritmo “Backpropagation” (backward propagation of errors)

En el siguiente ejemplo vemos una conexión entre tres neuronas

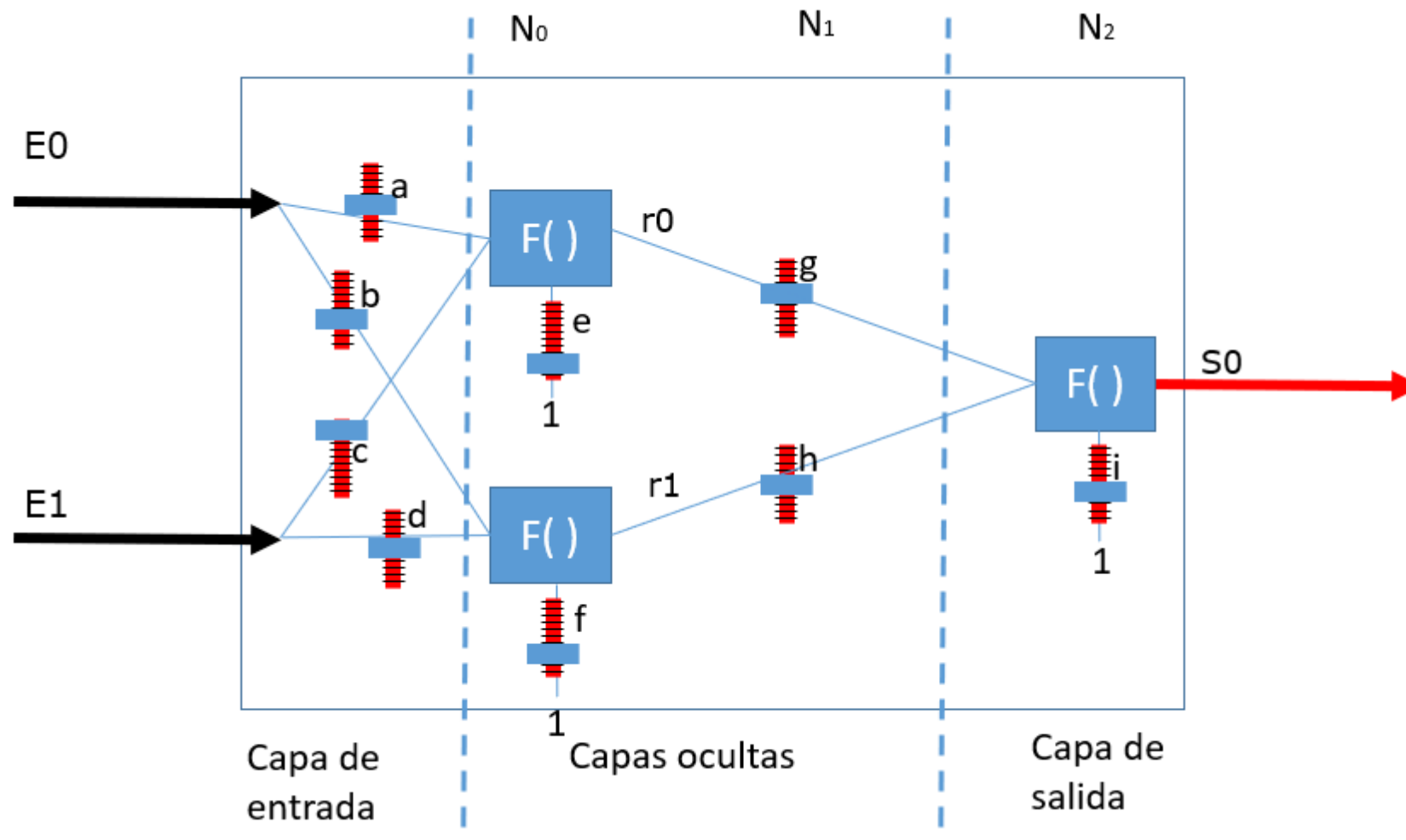


Ilustración 34: Esquema de un perceptrón multicapa

E₀ y E₁ son las entradas externas. Se observa que E₀ entra con el peso 'a' en la neurona de arriba y con peso 'b' en la neurona de abajo. Sucede lo mismo con la entrada E₁.

Lo interesante viene después, porque el resultado de la neurona de arriba 'r₀' y el resultado de la neurona de abajo 'r₁' se convierten en entradas para la neurona de la derecha y esas entradas a su vez tienen sus propios pesos. Al final el sistema genera una salida S₀ que es la respuesta final de la red neuronal.

Obsérvese que cada neurona tiene la entrada 1 y su peso llamado umbral.

¿Qué importancia tiene eso? Que, si queremos ajustar S₀ al resultado que esperamos, entonces retrocedemos a las entradas de esa neurona de la derecha ajustando sus pesos respectivos y por supuesto, ese ajuste nos hace retroceder más aún hasta mirar los pesos de las neuronas de arriba y abajo. Eso se conocerá como el algoritmo “Backpropagation”.

Luego:

$$r_0 = F(E_0 * a + E_1 * c + 1 * e)$$

$$r_1 = F(E_0 * b + E_1 * d + 1 * f)$$

$$S_0 = F(r_0 * g + r_1 * h + 1 * i)$$

Se concluye entonces que

$$S_0 = F(F(E_0 * a + E_1 * c + 1 * e) * g + F(E_0 * b + E_1 * d + 1 * f) * h + 1 * i)$$

Volviendo al perceptrón multicapa. Es momento de ponerle nombres a las diversas partes de la red neuronal:

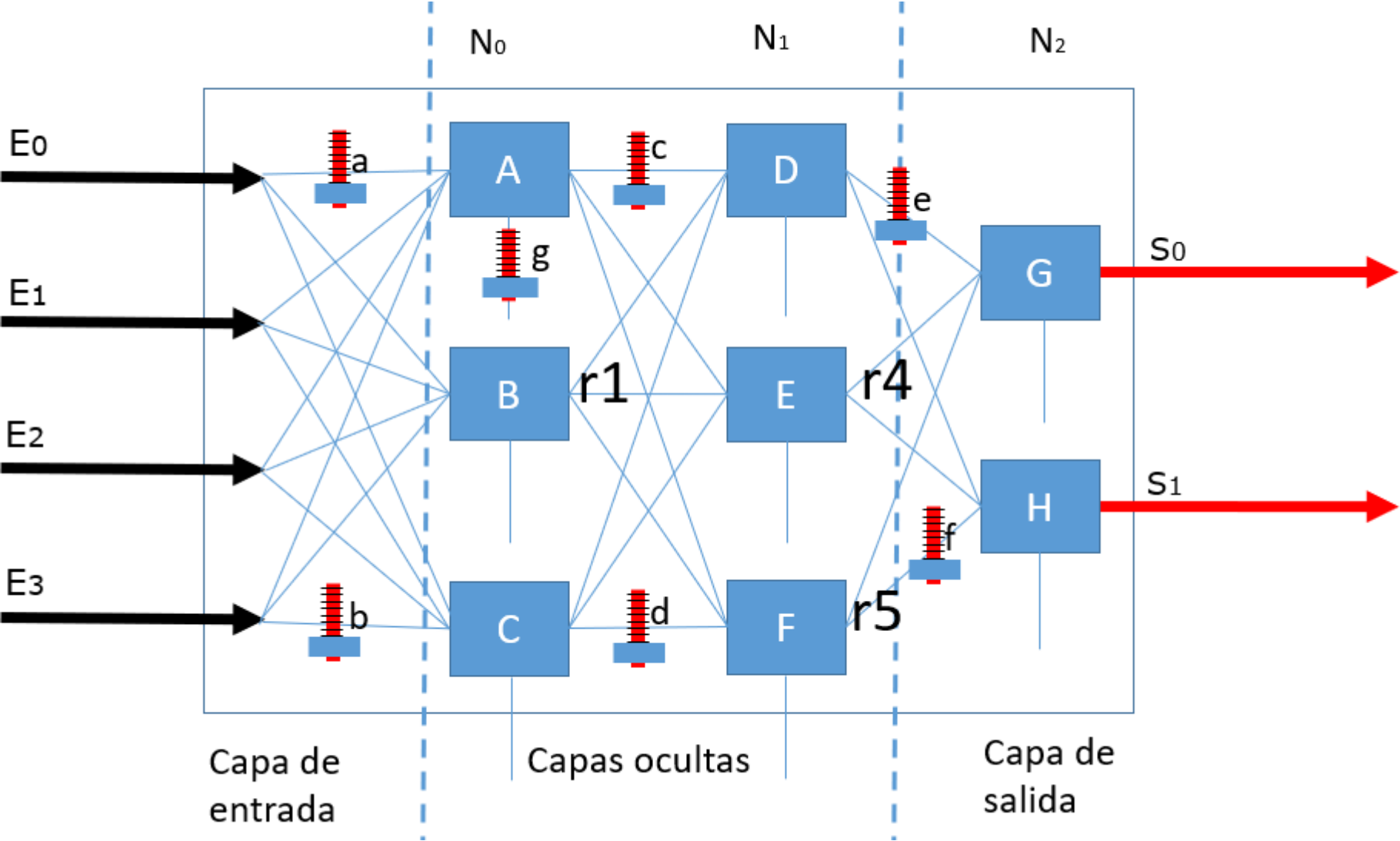


Ilustración 35: Partes de un perceptrón multicapa

Las entradas:

$E_{\text{número de la entrada}}$

Los pesos:

$w_{\text{neurona inicial,neurona final}}^{(\text{capa a donde llega la conexión})}$

Los umbrales:

$u_{\text{neurona que tiene esa entrada interna}}^{(\text{capa de la neurona que tiene esa entrada interna})}$

Las salidas internas de cada neurona:

$a_{\text{neurona de esa salida}}^{(\text{capa de la neurona de esa salida})}$

Las capas:

$n_{\text{número de la capa}}$

Entonces, viendo el gráfico:

Como se nombró antes	Nueva nomenclatura
E_0	E_0
E_1	E_1
E_2	E_2
E_3	E_3
a	$w_{0,0}^{(0)}$
b	$w_{3,3}^{(0)}$

c	$w_{0,0}^{(1)}$
d	$w_{2,2}^{(1)}$
e	$w_{0,0}^{(2)}$
f	$w_{2,1}^{(2)}$
g	$u_0^{(0)}$
r1	$a_1^{(0)}$
r4	$a_1^{(1)}$
r5	$a_2^{(1)}$
S ₀	$a_0^{(2)}$
S ₁	$a_1^{(2)}$

Luego

$$a_0^{(2)} = F \left(a_0^{(1)} * w_{0,0}^{(2)} + a_1^{(1)} * w_{1,0}^{(2)} + a_2^{(1)} * w_{2,0}^{(2)} + u_0^{(2)} \right)$$

$$a_1^{(2)} = F \left(a_0^{(1)} * w_{0,1}^{(2)} + a_1^{(1)} * w_{1,1}^{(2)} + a_2^{(1)} * w_{2,1}^{(2)} + u_1^{(2)} \right)$$

$$a_0^{(1)} = F \left(a_0^{(0)} * w_{0,0}^{(1)} + a_1^{(0)} * w_{1,0}^{(1)} + a_2^{(0)} * w_{2,0}^{(1)} + u_0^{(1)} \right)$$

$$a_1^{(1)} = F \left(a_0^{(0)} * w_{0,1}^{(1)} + a_1^{(0)} * w_{1,1}^{(1)} + a_2^{(0)} * w_{2,1}^{(1)} + u_1^{(1)} \right)$$

$$a_2^{(1)} = F \left(a_0^{(0)} * w_{0,2}^{(1)} + a_1^{(0)} * w_{1,2}^{(1)} + a_2^{(0)} * w_{2,2}^{(1)} + u_2^{(1)} \right)$$

$$a_0^{(0)} = F \left(E_0 * w_{0,0}^{(0)} + E_1 * w_{1,0}^{(0)} + E_2 * w_{2,0}^{(0)} + E_3 * w_{3,0}^{(0)} + u_0^{(0)} \right)$$

$$a_1^{(0)} = F \left(E_0 * w_{0,1}^{(0)} + E_1 * w_{1,1}^{(0)} + E_2 * w_{2,1}^{(0)} + E_3 * w_{3,1}^{(0)} + u_1^{(0)} \right)$$

$$a_2^{(0)} = F \left(E_0 * w_{0,2}^{(0)} + E_1 * w_{1,2}^{(0)} + E_2 * w_{2,2}^{(0)} + E_3 * w_{3,2}^{(0)} + u_2^{(0)} \right)$$

¿Qué hay de E₀, E₁, E₂ y E₃? Podrían tomarse como salidas de las neuronas de la capa E (Entrada). Cabe recordar que en esa capa no hay procesamiento. Luego:

Como se nombró antes	Nueva nomenclatura
E ₀	$a_0^{(E)}$
E ₁	$a_1^{(E)}$
E ₂	$a_2^{(E)}$

E ₃	$a_3^{(E)}$
----------------	-------------

Luego las tres últimas ecuaciones quedan así:

$$\begin{aligned}
 a_0^{(0)} &= F \left(a_0^{(E)} * w_{0,0}^{(0)} + a_1^{(E)} * w_{1,0}^{(0)} + a_2^{(E)} * w_{2,0}^{(0)} + a_3^{(E)} * w_{3,0}^{(0)} + u_0^{(0)} \right) \\
 a_1^{(0)} &= F \left(a_0^{(E)} * w_{0,1}^{(0)} + a_1^{(E)} * w_{1,1}^{(0)} + a_2^{(E)} * w_{2,1}^{(0)} + a_3^{(E)} * w_{3,1}^{(0)} + u_1^{(0)} \right) \\
 a_2^{(0)} &= F \left(a_0^{(E)} * w_{0,2}^{(0)} + a_1^{(E)} * w_{1,2}^{(0)} + a_2^{(E)} * w_{2,2}^{(0)} + a_3^{(E)} * w_{3,2}^{(0)} + u_2^{(0)} \right)
 \end{aligned}$$

¡OJO! Las capas tendrían este orden E, 0, 1, 2

Y generalizando se puede decir que

$$a_i^{(k)} = F \left(a_j^{(k-1)} * w_{j,i}^{(k)} + a_{j+1}^{(k-1)} * w_{j+1,i}^{(k)} + a_{j+2}^{(k-1)} * w_{j+2,i}^{(k)} + a_{j+3}^{(k-1)} * w_{j+3,i}^{(k)} + 1 * u_i^{(k)} \right)$$

Es decir que si k-1 < 0 entonces se está refiriendo a la capa E la de entrada.

Se puede simplificar más:

$$a_i^{(k)} = f \left(u_i^{(k)} + \sum_{j=0}^{n_k-1} a_j^{(k-1)} * w_{j,i}^{(k)} \right)$$

Donde k inicia en 0 y termina en el número de la última capa.

Regla de la cadena

Para continuar con el algoritmo de “Backpropagation”, cabe recordar esta regla matemática llamada regla de la cadena.

$$\partial\{F[g(x)]\} = \partial F[g(x)] * \partial g(x)$$

Un ejemplo, hay dos funciones:

$$g(x) = 3 * x^2$$

$$F[p] = 7 - p^3$$

Luego

$$F[g(x)] = 7 - (3 * x^2)^3 = 7 - 27 * x^6$$

Reemplazando “p” con lo que tiene g(x)

Derivando

$$\partial\{F[g(x)]\} = -162 * x^5$$

Usando la regla de la cadena

$$\partial\{F[g(x)]\} = \partial F[g(x)] * \partial g(x) = \partial(7 - p^3) * \partial(3 * x^2) =$$

$$(0 - 3 * p^2) * (6 * x) = (0 - 3 * (3 * x^2)^2) * (6 * x) =$$

$$(0 - 3 * (9 * x^4)) * (6 * x) = -162 * x^5$$

La regla de la cadena funciona.

Dada una ecuación que tenga dos o más variables independientes, es posible derivar por una variable considerando las demás constantes, eso es conocido como derivada parcial.

$$q = a^2 + b^3 + c^4$$
$$\frac{\partial q}{\partial a}(a, b, c) = 2 * a + 0 + 0$$

$$\frac{\partial q}{\partial a}(a, b, c) = 2 * a$$

$$\frac{\partial q}{\partial b}(a, b, c) = 0 + 3 * b^2 + 0$$

$$\frac{\partial q}{\partial b}(a, b, c) = 3 * b^2$$

$$\frac{\partial q}{\partial c}(a, b, c) = 0 + 0 + 4 * c^3$$

$$\frac{\partial q}{\partial c}(a, b, c) = 4 * c^3$$


$$\frac{d}{da} a^2 + b^3 + c^4$$

Σ_θ^π Extended Keyboard

 Upload

Examp

Derivative:

☒ St

$$\frac{\partial}{\partial a}(a^2 + b^3 + c^4) = 2a$$

Ilustración 36: Derivada parcial

Observamos el siguiente gráfico muy sencillo de un perceptrón multicapa. Hay que recordar que la capa de entrada no hace proceso.

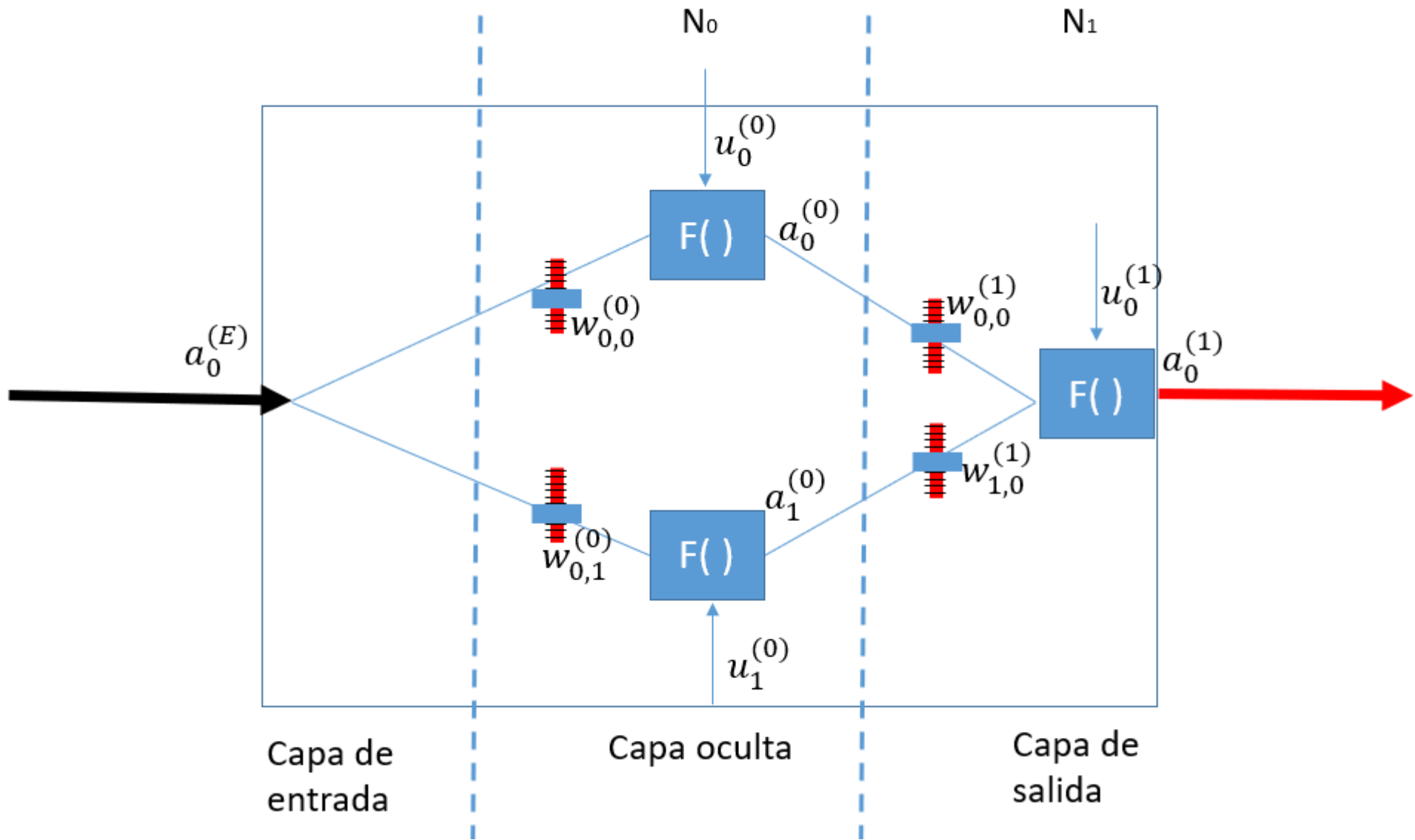


Ilustración 37: Nombramiento de pesos, umbrales y salidas

$$a_0^{(1)} = F(a_0^{(0)} * w_{0,0}^{(1)} + a_1^{(0)} * w_{1,0}^{(1)} + 1 * u_0^{(1)})$$

$$a_0^{(0)} = F(a_0^{(E)} * w_{0,0}^{(0)} + 1 * u_0^{(0)})$$

$$a_1^{(0)} = F(a_0^{(E)} * w_{0,1}^{(0)} + 1 * u_1^{(0)})$$

Luego reemplazando

$$a_0^{(1)} = F(F(a_0^{(E)} * w_{0,0}^{(0)} + 1 * u_0^{(0)}) * w_{0,0}^{(1)} + F(a_0^{(E)} * w_{0,1}^{(0)} + 1 * u_1^{(0)}) * w_{1,0}^{(1)} + 1 * u_0^{(1)})$$

Simplificando

$$a_0^{(1)} = F(F(a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)}) * w_{0,0}^{(1)} + F(a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)}) * w_{1,0}^{(1)} + u_0^{(1)})$$

Como la función $F[]$ es una sigmoidea y la derivada de las sigmoideas es así:

$$\partial y = y * (1 - y)$$

Luego:

$$\partial F[r] = F[r] * (1 - F[r])$$

¿Qué sucedería si r es a su vez una función sigmoidea?

$$r = g(k)$$

Tener en cuenta que $g()$ es una sigmoidea. Y además de eso, k es una función polinómica.

Recordando la regla de la cadena, esto sucedería:

$$\partial\{F[g(k)]\} = \partial F[g(k)] * \partial g(k)$$

Y como $F()$ es sigmoidea, entonces al derivar

$$\partial\{F[g(k)]\} = F(g(k)) * (1 - F(g(k))) * \partial g(k)$$

$g(k)$ es una función sigmoidea y k es una función polinómica, luego la derivada de $g(k)$ aplicando la regla de la cadena sería:

$$\partial g(k) = \partial g(k) * \partial k = g(k) * (1 - g(k)) * \partial k$$

La derivada queda así

$$\partial\{F[g(k)]\} = F(g(k)) * (1 - F(g(k))) * g(k) * (1 - g(k)) * \partial k$$

Simplificando

$$\partial\{F[g(k)]\} = F(r) * (1 - F(r)) * g(k) * (1 - g(k)) * \partial k$$

Esta es la ecuación (la salida total de esa red neuronal) que se va a derivar parcialmente con respecto a un peso en particular

$$a_0^{(1)} = F(F(a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)}) * w_{0,0}^{(1)} + F(a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)}) * w_{1,0}^{(1)} + u_0^{(1)})$$

En el ejemplo, se deriva con respecto a $w_{0,0}^{(0)}$ (una derivada parcial). En rojo se pone que ecuación interna es derivable con respecto a $w_{0,0}^{(0)}$

$$\frac{\partial a_0^{(1)}}{\partial w_{0,0}^{(0)}} = \partial F \left(\partial \left[F \left(a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)} \right) * w_{0,0}^{(1)} \right] + 0 + 0 \right)$$

Para derivar entonces se deriva la F externa (que está en negro y es F(r)), luego la F interna (que está en rojo y es g(k) y que la multiplica la constante $w_{0,0}^{(1)}$) y por último el polinomio (que es k) que está en verde porque allí está $w_{0,0}^{(0)}$. Hay tres derivaciones.

Entonces, se sabe que:

$$F(r) = a_0^{(1)}$$

$$g(k) = a_0^{(0)}$$

$$k = a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)}$$

$$\partial k = a_0^{(E)}$$

$$\frac{\partial a_0^{(1)}}{\partial w_{0,0}^{(E)}} = \partial \{F[g(k)]\}$$

$$\partial \{F[g(k)]\} = F(r) * (1 - F(r)) * g(k) * (1 - g(k)) * \partial k$$

Entonces:

$$\frac{\partial a_0^{(1)}}{\partial w_{0,0}^{(0)}} = a_0^{(1)} * (1 - a_0^{(1)}) * a_0^{(0)} * (1 - a_0^{(0)}) * a_0^{(E)} * w_{0,0}^{(1)}$$

Esta es la ecuación (la salida total de esa red neuronal) que se va a derivar parcialmente con respecto a un peso en particular

$$a_0^{(1)} = F(F(a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)}) * w_{0,0}^{(1)} + F(a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)}) * w_{1,0}^{(1)} + u_0^{(1)})$$

En el ejemplo, se deriva con respecto a $w_{0,1}^{(0)}$ (una derivada parcial). En azul se pone que ecuación interna es derivable con respecto a $w_{0,1}^{(0)}$

$$\frac{\partial a_0^{(1)}}{\partial w_{0,0}^{(0)}} = \partial F \left(0 + \partial [F(a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)}) * w_{1,0}^{(1)}] + 0 \right)$$

Para derivar entonces se deriva la F externa (que está en negro y es F(r)), luego la F interna (que está en azul y es g(k) y que la multiplica la constante $w_{1,0}^{(1)}$) y por último el polinomio (que es k) que está en verde porque allí está $w_{0,0}^{(0)}$. Hay tres derivaciones.

Entonces, se sabe que:

$$F(r) = a_0^{(1)}$$

$$g(k) = a_1^{(0)}$$

$$k = a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)}$$

$$\partial k = a_0^{(E)}$$

$$\frac{\partial a_0^{(1)}}{\partial w_{0,1}^{(0)}} = \partial \{F[g(k)]\}$$

$$\partial \{F[g(k)]\} = F(r) * (1 - F(r)) * g(k) * (1 - g(k)) * \partial k$$

Entonces:

$$\frac{\partial a_0^{(1)}}{\partial w_{0,1}^{(0)}} = a_0^{(1)} * (1 - a_0^{(1)}) * a_1^{(0)} * (1 - a_1^{(0)}) * a_0^{(E)} * w_{1,0}^{(1)}$$

Generalizando

$$\frac{\partial a_0^{(1)}}{\partial w_{0,j}^{(0)}} = a_0^{(1)} * (1 - a_0^{(1)}) * a_j^{(0)} * (1 - a_j^{(0)}) * a_0^{(E)} * w_{j,0}^{(1)}$$

Donde j puede ser 0 o 1. Esa sería la generalización para los pesos $w_{0,0}^{(0)}$ y $w_{0,1}^{(0)}$

Para el peso $W_{0,0}^{(1)}$ este sería el tratamiento:

$$a_0^{(1)} = F(F(a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)}) * w_{0,0}^{(1)} + F(a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)}) * w_{1,0}^{(1)} + u_0^{(1)})$$

En el ejemplo, se deriva con respecto a $W_{0,0}^{(1)}$ (una derivada parcial). Lo que está en rojo es lo que “sobrevive” de esa derivada parcial (se comporta como una constante porque se deriva parcialmente por $W_{0,0}^{(1)}$), quedando así:

$$\frac{\partial a_0^{(1)}}{\partial w_{0,0}^{(1)}} = \partial F \left(F(a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)}) + 0 + 0 \right)$$

Entonces

$$\frac{\partial a_0^{(1)}}{\partial w_{0,0}^{(1)}} = a_0^{(1)} * (1 - a_0^{(1)}) * F(a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)})$$

Y como

$$a_0^{(0)} = F(a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)})$$

Entonces

$$\frac{\partial a_0^{(1)}}{\partial w_{0,0}^{(1)}} = a_0^{(1)} * (1 - a_0^{(1)}) * a_0^{(0)}$$

Para el peso $w_{1,0}^{(1)}$ este sería el tratamiento:

$$a_0^{(1)} = F(F(a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)}) * w_{0,0}^{(1)} + F(a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)}) * w_{1,0}^{(1)} + u_0^{(1)})$$

En el ejemplo, se deriva con respecto a $w_{1,0}^{(1)}$ (una derivada parcial). Lo que está en azul es lo que “sobrevive” de esa derivada parcial (se comporta como una constante porque se deriva parcialmente por $w_{1,0}^{(1)}$), quedando así:

$$\frac{\partial a_0^{(1)}}{\partial w_{1,0}^{(1)}} = \partial F \left(0 + F(a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)}) + 0 \right)$$

Se comporta como una constante

Entonces

$$\frac{\partial a_0^{(1)}}{\partial w_{1,0}^{(1)}} = a_0^{(1)} * (1 - a_0^{(1)}) * F(a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)})$$

Y como

$$a_1^{(0)} = F(a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)})$$

Entonces

$$\frac{\partial a_0^{(1)}}{\partial w_{1,0}^{(1)}} = a_0^{(1)} * (1 - a_0^{(1)}) * a_1^{(0)}$$

Generalizando

$$\frac{\partial a_0^{(1)}}{\partial w_{j,0}^{(1)}} = a_0^{(1)} * (1 - a_0^{(1)}) * a_j^{(0)}$$

Donde j=0 o 1

Para el umbral $u_0^{(0)}$ este sería el tratamiento:

$$a_0^{(1)} = F(F(a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)}) * w_{0,0}^{(1)} + F(a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)}) * w_{1,0}^{(1)} + u_0^{(1)})$$

En el ejemplo, se deriva con respecto a $u_0^{(0)}$ (una derivada parcial). En rojo se pone que ecuación interna es derivable con respecto a $u_0^{(0)}$

$$\frac{\partial a_0^{(1)}}{\partial u_0^{(0)}} = \partial F \left(\partial F(a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)}) * w_{0,0}^{(1)} + 0 + 0 \right)$$

Para derivar entonces se deriva la F externa (que está en negro y es F(r)), luego la F interna (que está en rojo y es g(k) y que la multiplica la constante $w_{0,0}^{(1)}$) y por último el polinomio (que es k) que está en verde porque allí está $u_0^{(0)}$. Hay tres derivaciones.

Entonces, se sabe que:

$$F(r) = a_0^{(1)}$$

$$g(k) = a_0^{(0)}$$

$$k = a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)}$$

$$\partial k = 1$$

$$\frac{\partial a_0^{(1)}}{\partial u_0^{(0)}} = \partial \{F[g(k)]\}$$

$$\partial \{F[g(k)]\} = F(r) * (1 - F(r)) * g(k) * (1 - g(k)) * \partial k$$

Entonces:

$$\frac{\partial a_0^{(1)}}{\partial u_0^{(0)}} = a_0^{(1)} * (1 - a_0^{(1)}) * a_0^{(0)} * (1 - a_0^{(0)}) * 1 * w_{0,0}^{(1)}$$

Para el umbral $u_1^{(0)}$ este sería el tratamiento:

$$a_0^{(1)} = F(F(a_0^{(E)} * w_{0,0}^{(0)} + u_0^{(0)}) * w_{0,0}^{(1)} + F(a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)}) * w_{1,0}^{(1)} + u_0^{(1)})$$

En el ejemplo, se deriva con respecto a $u_1^{(0)}$ (una derivada parcial). En azul se pone que ecuación interna es derivable con respecto a $u_1^{(0)}$

$$\frac{\partial a_0^{(1)}}{\partial u_1^{(0)}} = \partial F \left(0 + F(a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)}) * w_{1,0}^{(1)} + 0 \right)$$

Para derivar entonces se deriva la F externa (que está en negro y es F(r)), luego la F interna (que está en azul y es g(k) y que la multiplica la constante $w_{1,0}^{(1)}$) y por último el polinomio (que es k) que está en verde porque allí está $u_1^{(0)}$. Hay tres derivaciones.

Entonces, se sabe que:

$$F(r) = a_0^{(1)}$$

$$g(k) = a_1^{(0)}$$

$$k = a_0^{(E)} * w_{0,1}^{(0)} + u_1^{(0)}$$

$$\partial k = 1$$

$$\frac{\partial a_0^{(1)}}{\partial u_1^{(0)}} = \partial \{F[g(k)]\}$$

$$\partial \{F[g(k)]\} = F(r) * (1 - F(r)) * g(k) * (1 - g(k)) * \partial k$$

Entonces:

$$\frac{\partial a_0^{(1)}}{\partial u_1^{(0)}} = a_0^{(1)} * (1 - a_0^{(1)}) * a_1^{(0)} * (1 - a_1^{(0)}) * 1 * w_{1,0}^{(1)}$$

Con un ejemplo más complejo en el que la capa oculta tiene dos capas de neuronas y cada capa tiene dos neuronas.

Luego $N_0 = 2, N_1 = 2, N_2 = 1$

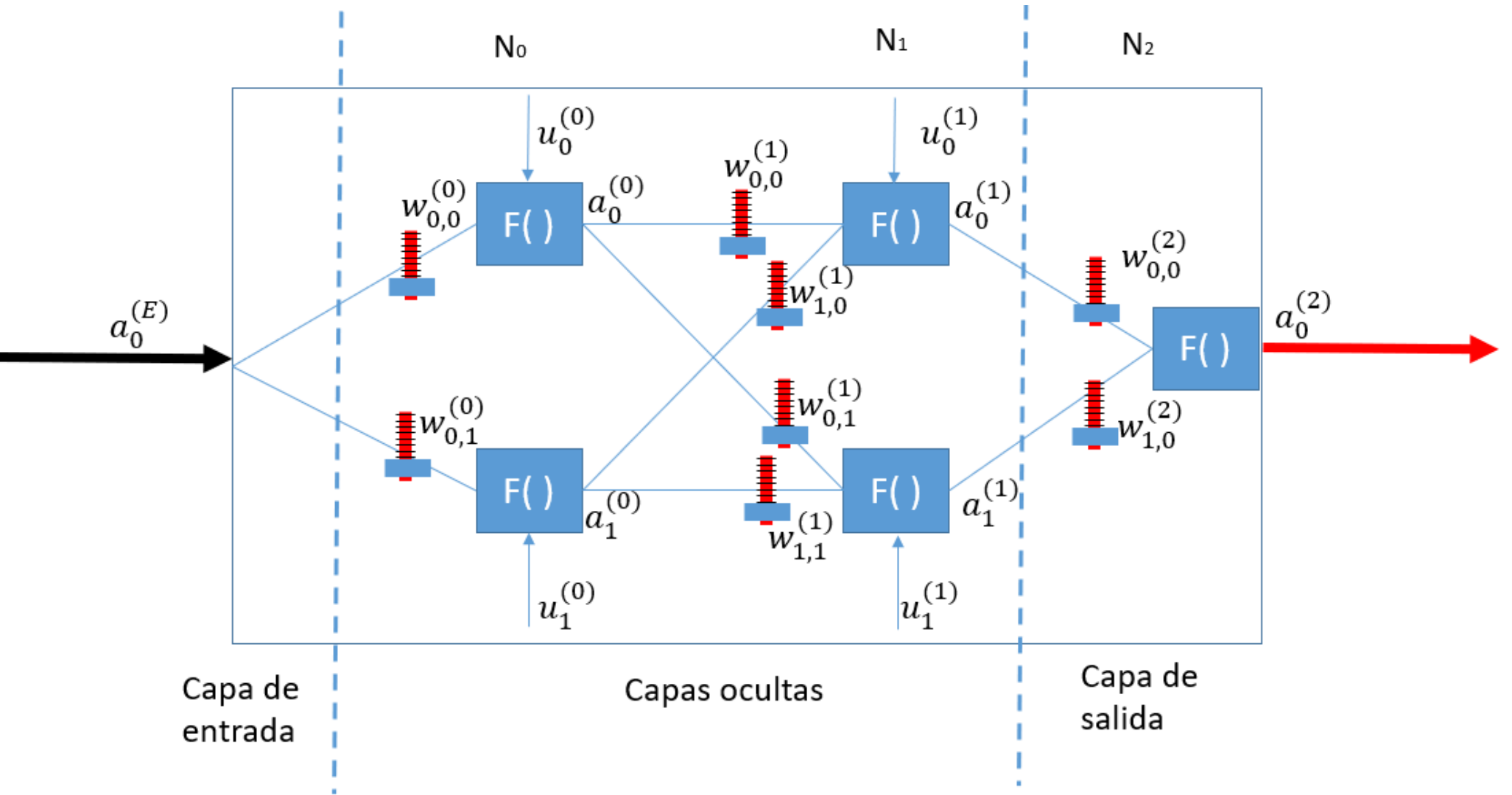


Ilustración 38: Esquema de un perceptrón multicapa

Se buscan los caminos para $w_{0,0}^{(0)}$, entonces hay dos marcados en rojo

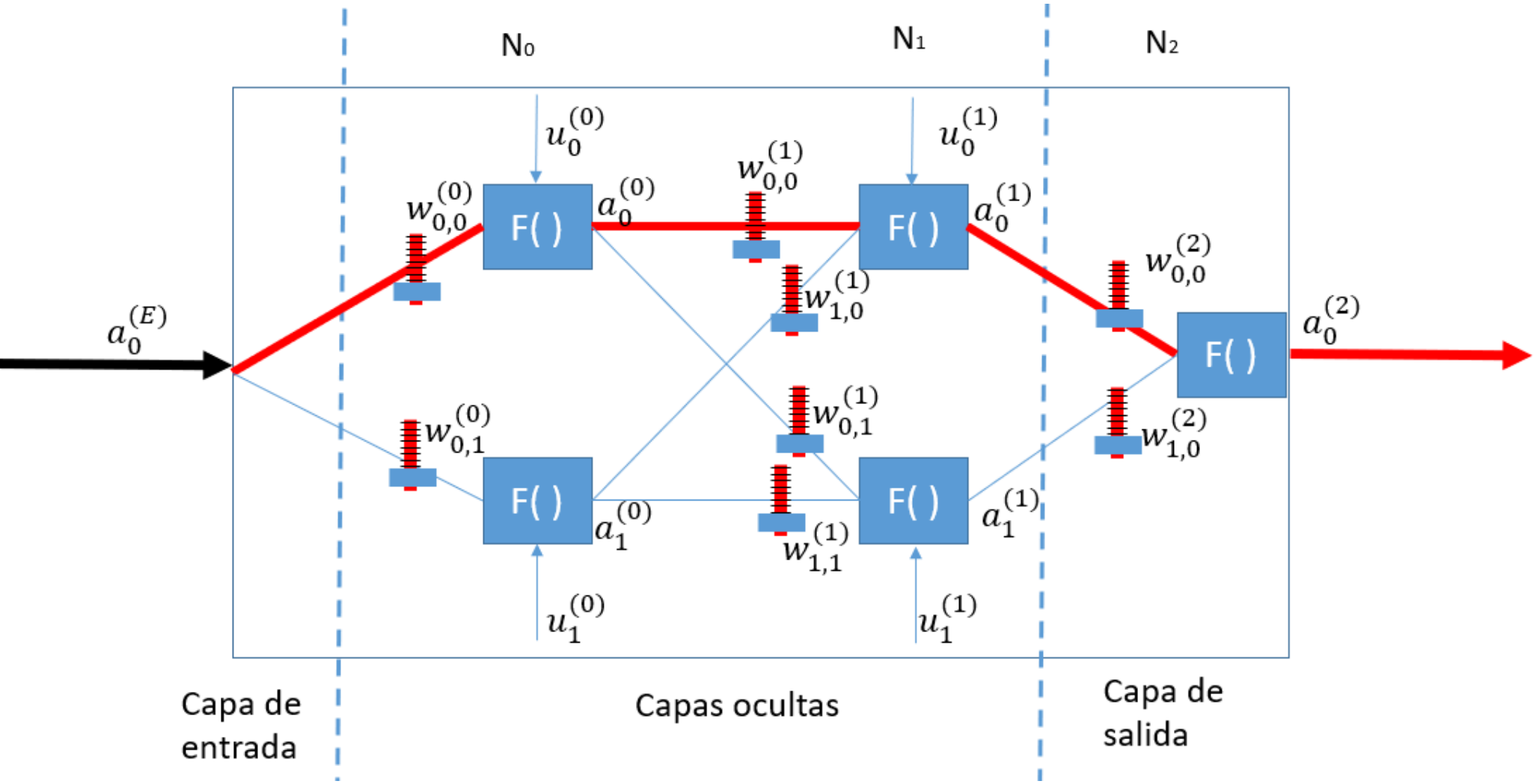


Ilustración 39: Primer camino para ese peso

Y

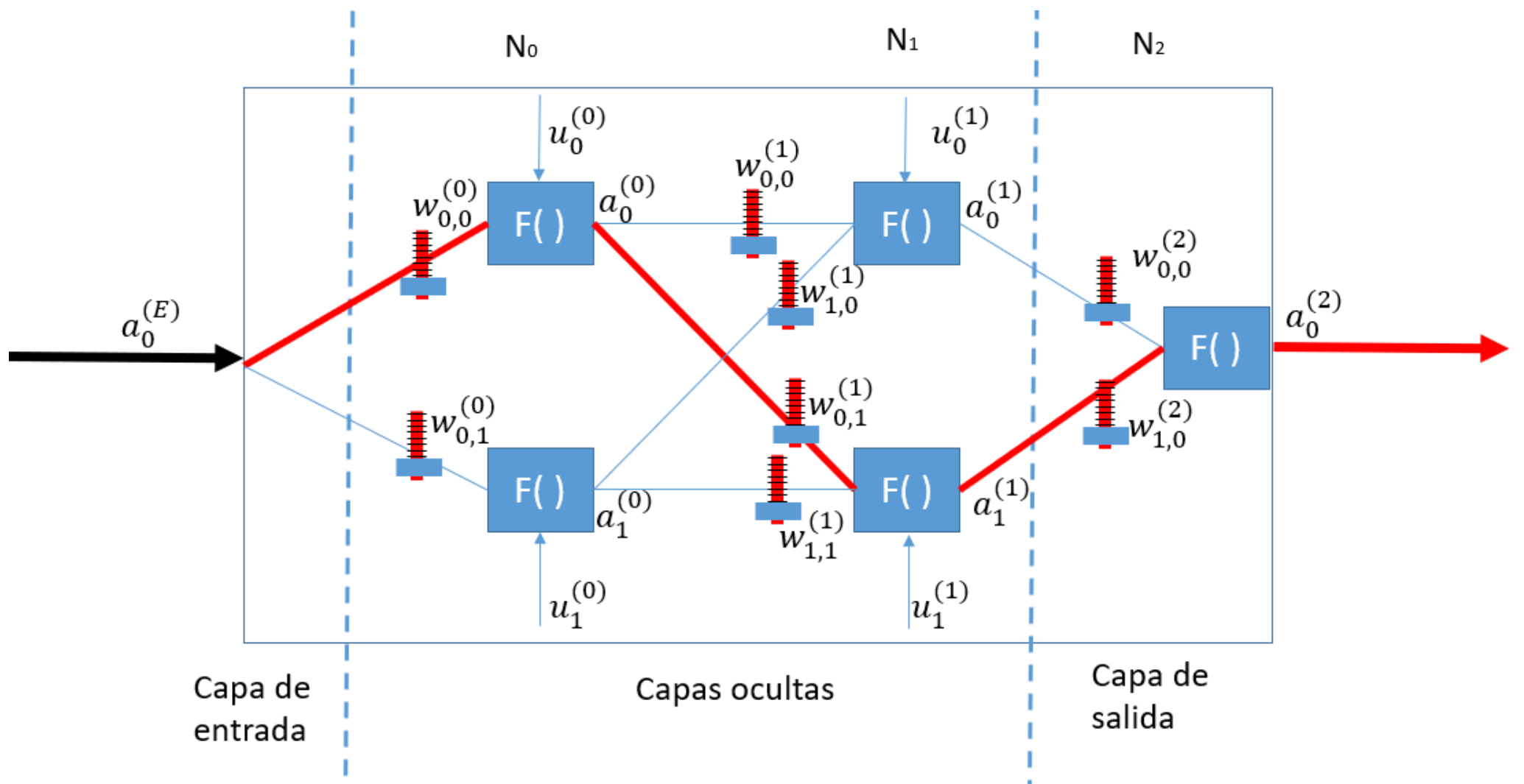


Ilustración 40: Segundo camino para ese peso

Luego la siguiente expresión para la derivada parcial con respecto a $w_{0,0}^{(0)}$ es seguir los dos caminos, sumando ambos

$$\frac{\partial a_0^{(2)}}{\partial w_{0,0}^{(0)}} = a_0^{(2)} * (1 - a_0^{(2)}) * w_{0,0}^{(2)} * a_0^{(1)} * (1 - a_0^{(1)}) * w_{0,0}^{(1)} * a_0^{(0)} * (1 - a_0^{(0)}) * a_0^{(E)} +$$

$$a_0^{(2)} * (1 - a_0^{(2)}) * w_{1,0}^{(2)} * a_1^{(1)} * (1 - a_1^{(1)}) * w_{0,1}^{(1)} * a_0^{(0)} * (1 - a_0^{(0)}) * a_0^{(E)}$$

Recomendado ir de la entrada a la salida para ver cómo se incrementa el nivel de las capas

$$\frac{\partial a_0^{(2)}}{\partial w_{0,0}^{(0)}} = a_0^{(E)} * a_0^{(0)} * (1 - a_0^{(0)}) * w_{0,0}^{(1)} * a_0^{(1)} * (1 - a_0^{(1)}) * w_{0,0}^{(2)} * a_0^{(2)} * (1 - a_0^{(2)}) +$$

$$a_0^{(E)} * a_0^{(0)} * (1 - a_0^{(0)}) * w_{0,1}^{(1)} * a_1^{(1)} * (1 - a_1^{(1)}) * w_{1,0}^{(2)} * a_0^{(2)} * (1 - a_0^{(2)})$$

Y así poder generalizar

$$\frac{\partial a_0^{(2)}}{\partial w_{0,0}^{(0)}} = a_0^{(E)} * a_0^{(0)} * (1 - a_0^{(0)}) * \left[\sum_{j=0}^{N_2-1} w_{0,j}^{(1)} * a_j^{(1)} * (1 - a_j^{(1)}) * w_{j,0}^{(2)} \right] * a_0^{(2)} * (1 - a_0^{(2)})$$

La ventaja es que, si las capas ocultas tienen más neuronas, sería cambiar el límite máximo en la sumatoria.

Renombrando la entrada y salida del perceptrón así:

$$a_0^{(E)} = x_0$$

$$a_0^{(2)} = y_0$$

Entonces

$$\frac{\partial y_0}{\partial w_{0,0}^{(0)}} = x_0 * a_0^{(0)} * (1 - a_0^{(0)}) * \left[\sum_{j=0}^{N_2-1} w_{0,j}^{(1)} * a_j^{(1)} * (1 - a_j^{(1)}) * w_{j,0}^{(2)} \right] * y_0 * (1 - y_0)$$

Con un perceptrón con más entradas y salidas, $N_0 = 4$, $N_1 = 4$, $N_2 = 2$, y si se desea dar con $\frac{\partial y_0}{\partial w_{0,0}^{(0)}}$, hay que considerar los diferentes caminos:

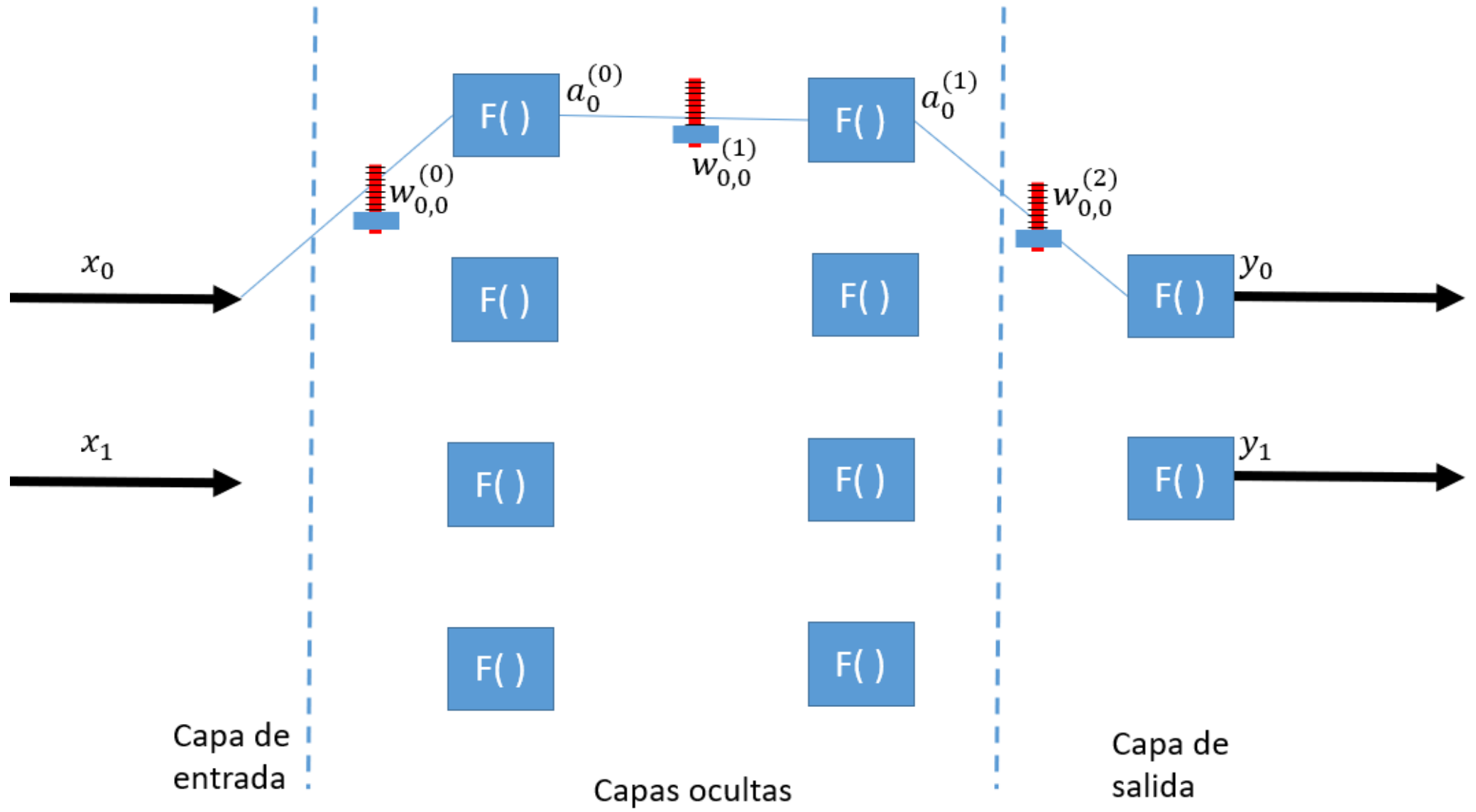


Ilustración 41: Primer camino

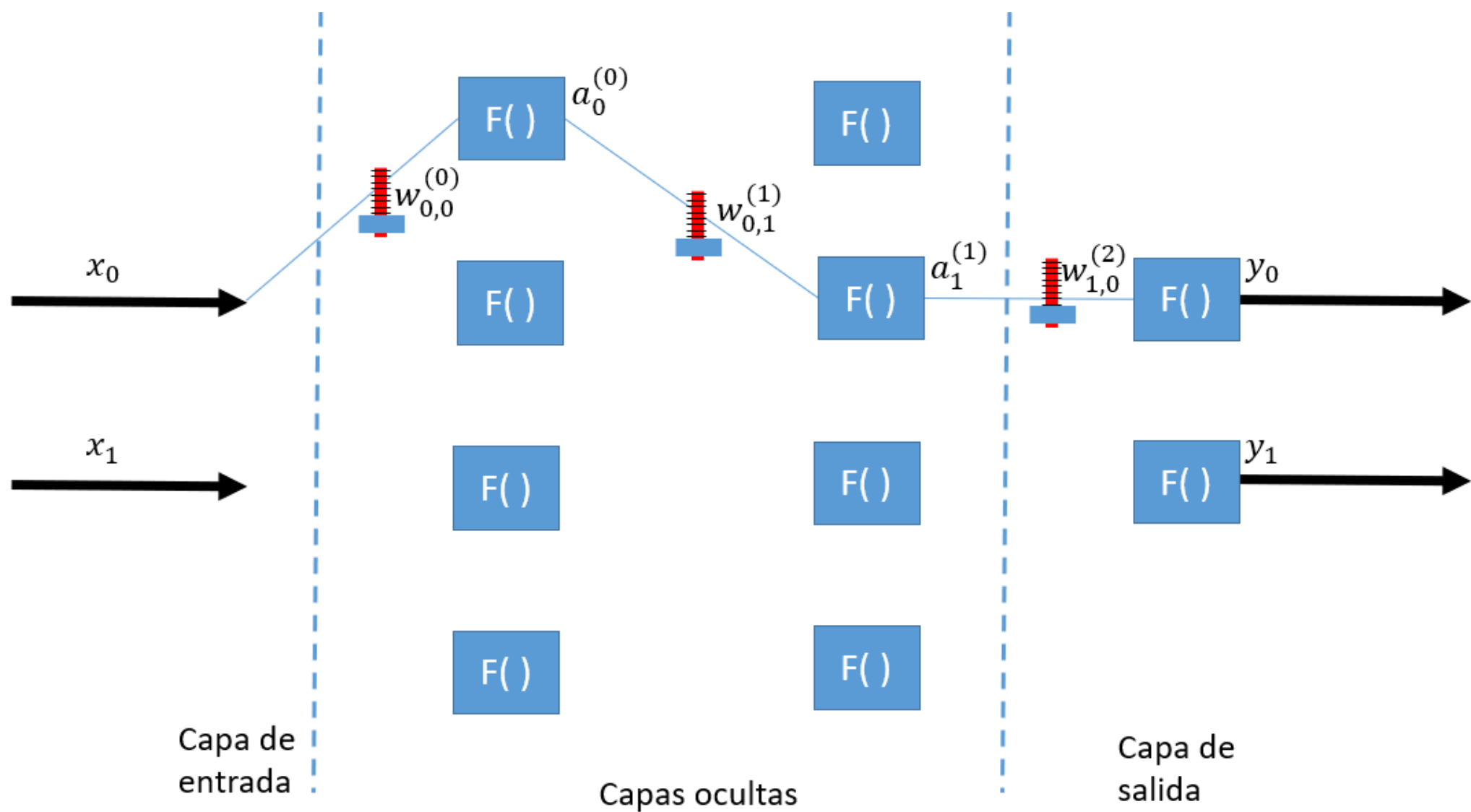


Ilustración 42: Segundo camino

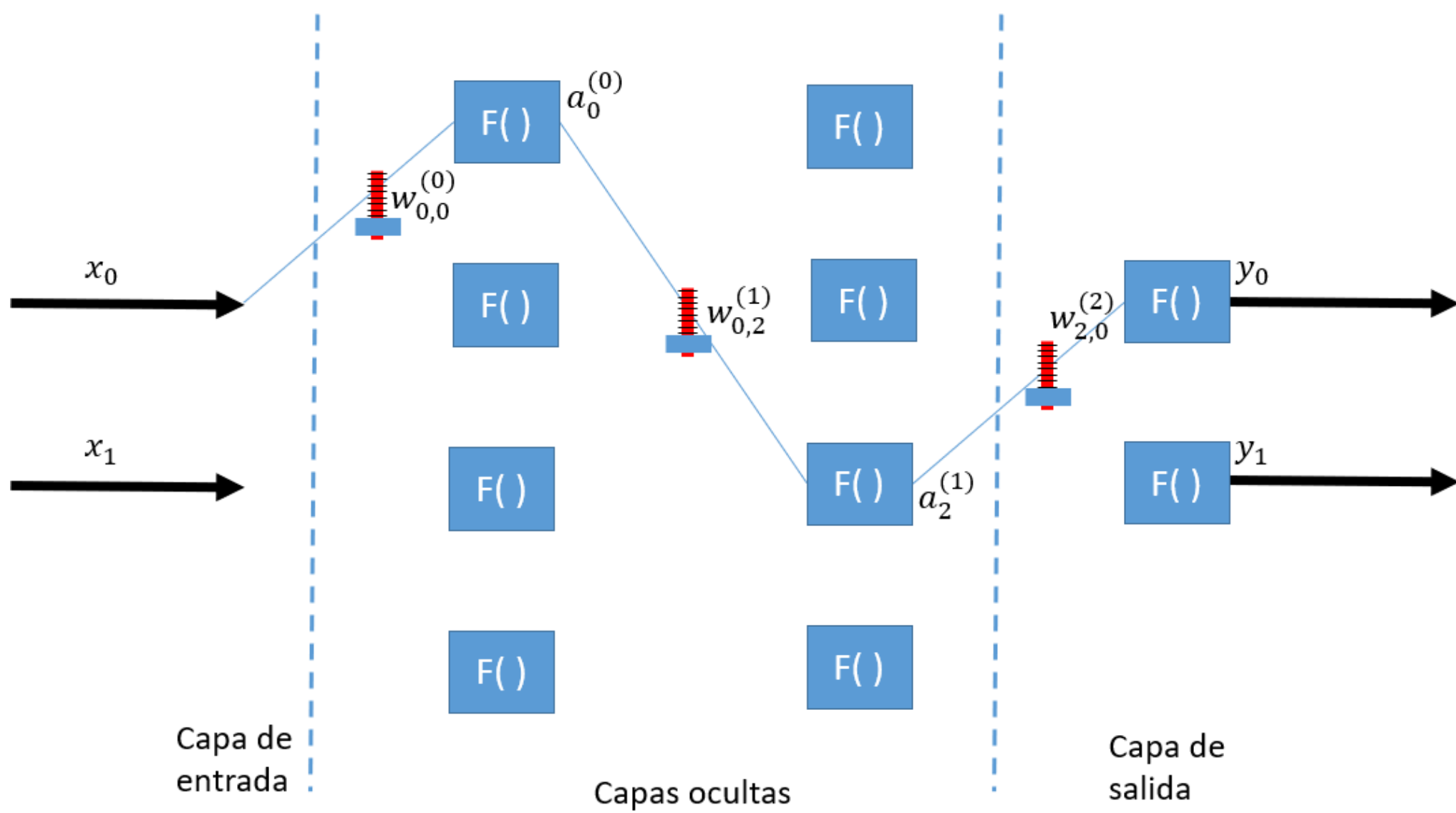


Ilustración 43: Tercer camino

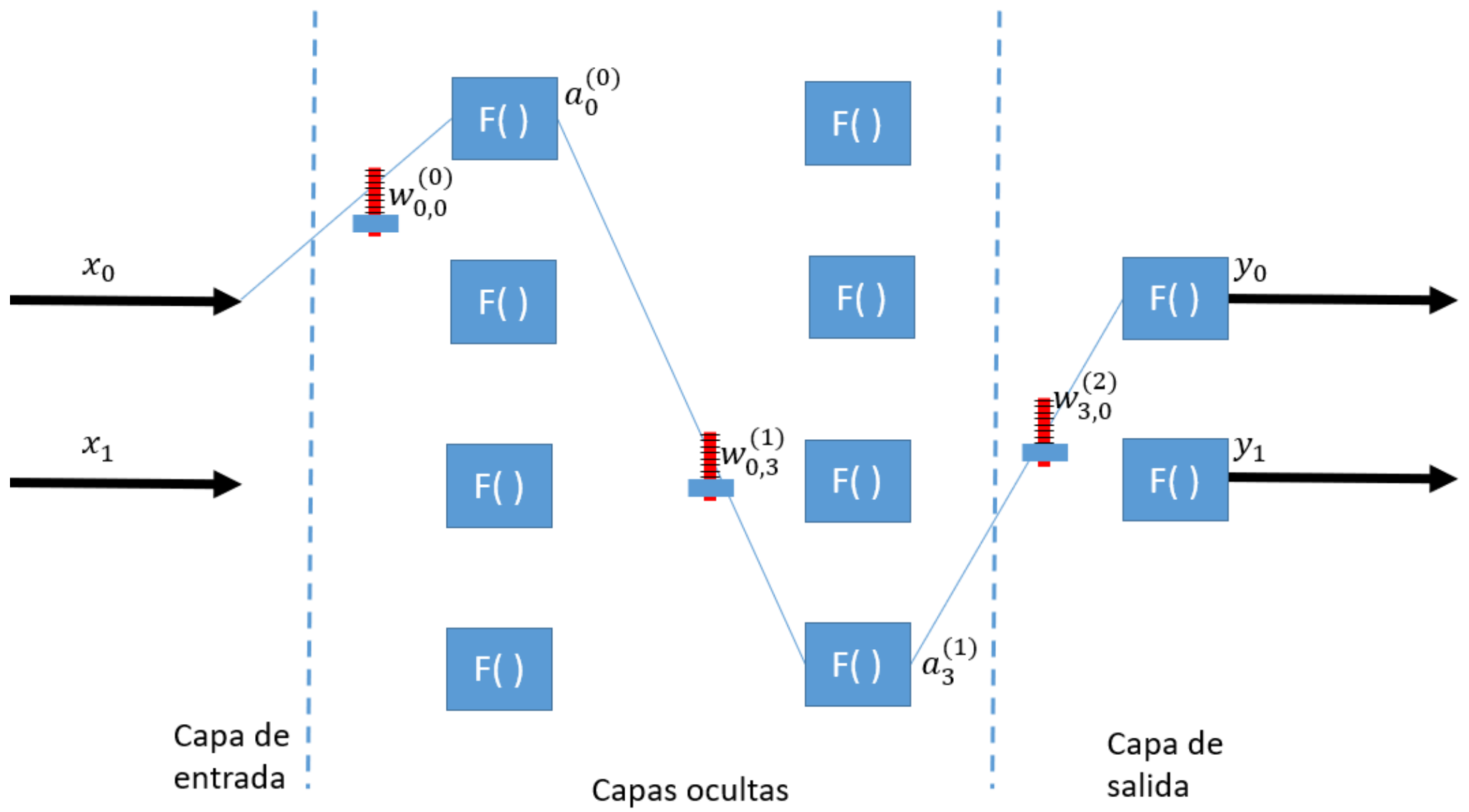


Ilustración 44: Cuarto camino

Luego

$$\frac{\partial y_0}{\partial w_{0,0}^{(0)}} = x_0 * a_0^{(0)} * (1 - a_0^{(0)}) * \left[\sum_{p=0}^{N_1-1} w_{0,p}^{(1)} * a_p^{(1)} * (1 - a_p^{(1)}) * w_{p,0}^{(2)} \right] * y_0 * (1 - y_0)$$

Generalizando

$$\frac{\partial y_i}{\partial w_{j,k}^{(0)}} = x_j * a_k^{(0)} * \left(1 - a_k^{(0)}\right) * \left[\sum_{p=0}^{N_1-1} w_{k,p}^{(1)} * a_p^{(1)} * \left(1 - a_p^{(1)}\right) * w_{p,i}^{(2)}\right] * y_i * (1 - y_i)$$

Donde i=0 a 1, j=0 a 3, k=0 a 3

¿Y para los $w^{(1)}$?

$$\frac{\partial y_i}{\partial w_{j,k}^{(1)}} = a_j^{(0)} * a_k^{(1)} * \left(1 - a_k^{(1)}\right) * w_{k,i}^{(2)} * y_i * (1 - y_i)$$

¿Y para los $w^{(2)}$?

$$\frac{\partial y_i}{\partial w_{j,i}^{(2)}} = a_j^{(1)} * y_i * (1 - y_i)$$

¿Y los umbrales $u^{(0)}$?

$$\frac{\partial y_i}{\partial u_k^{(0)}} = 1 * a_k^{(0)} * \left(1 - a_k^{(0)}\right) * \left[\sum_{p=0}^{N_1-1} w_{k,p}^{(1)} * a_p^{(1)} * \left(1 - a_p^{(1)}\right) * w_{p,i}^{(2)}\right] * y_i * (1 - y_i)$$

Donde i=0 a 1, k=0 a 3

¿Y los umbrales $u^{(1)}$?

$$\frac{\partial y_i}{\partial u_k^{(1)}} = 1 * a_k^{(1)} * \left(1 - a_k^{(1)}\right) * w_{k,i}^{(2)} * y_i * (1 - y_i)$$

¿Y los umbrales $u^{(2)}$?

$$\frac{\partial y_i}{\partial u_i^{(2)}} = 1 * y_i * (1 - y_i)$$

Tenemos la siguiente tabla

Entrada X_0	Entrada X_1	Valor esperado de salida S_0	Valor esperado de salida S_1
1	0	0	1
0	0	1	1
0	1	0	0

Pero en realidad estamos obteniendo con el perceptrón estas salidas

Entrada X_0	Entrada X_1	Salida real Y_0	Salida real Y_1
1	0	1	1
0	0	1	0
0	1	0	1

Hay un error evidente con las salidas porque no coinciden con lo esperado. ¿Qué hacer? Ajustar los pesos y los umbrales.

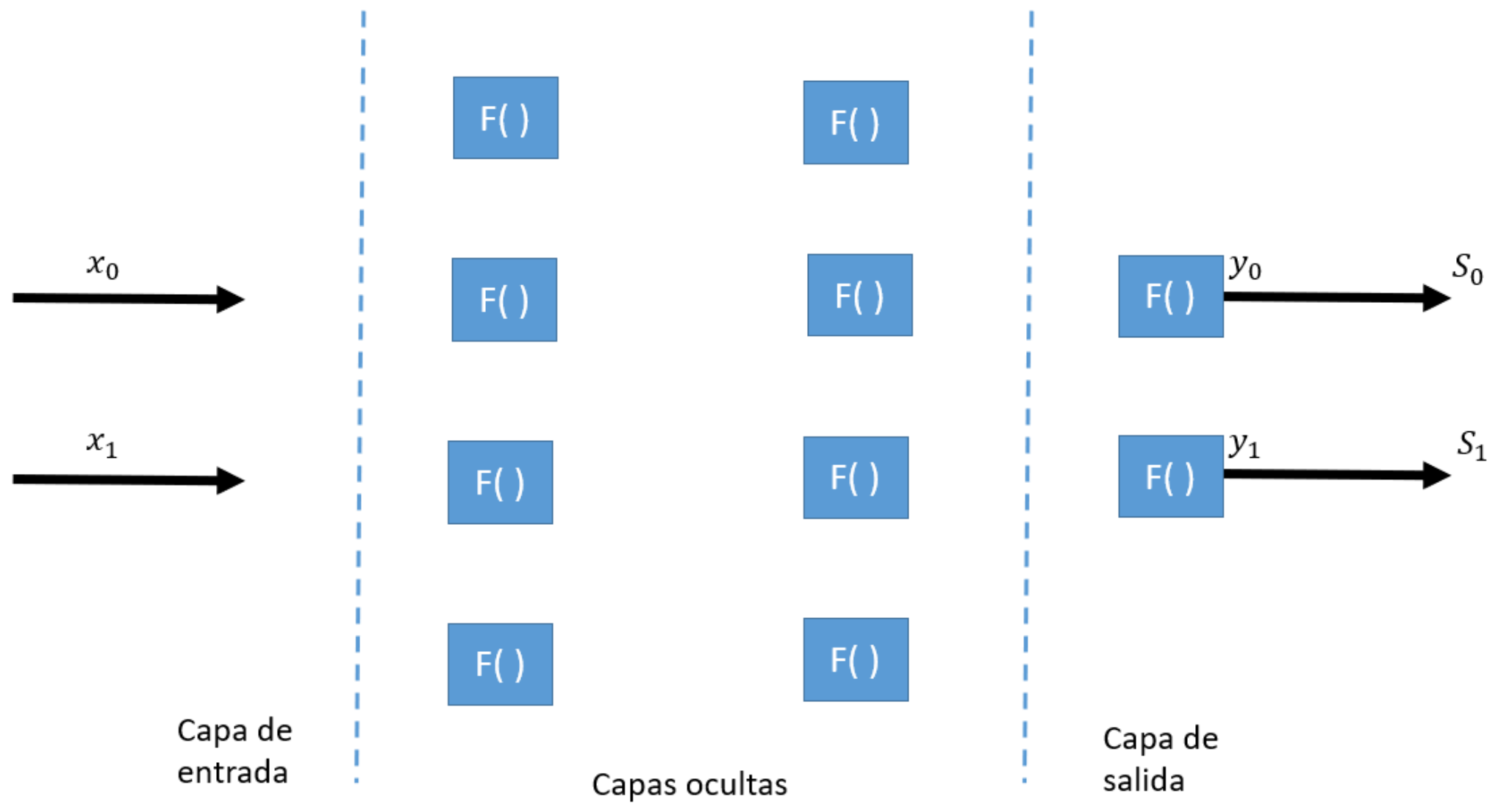


Ilustración 45: Dos entradas y dos salidas

Si tomásemos las salidas y_0 y y_1 como coordenadas e igualmente S_0 y S_1 , tendríamos lo siguiente:

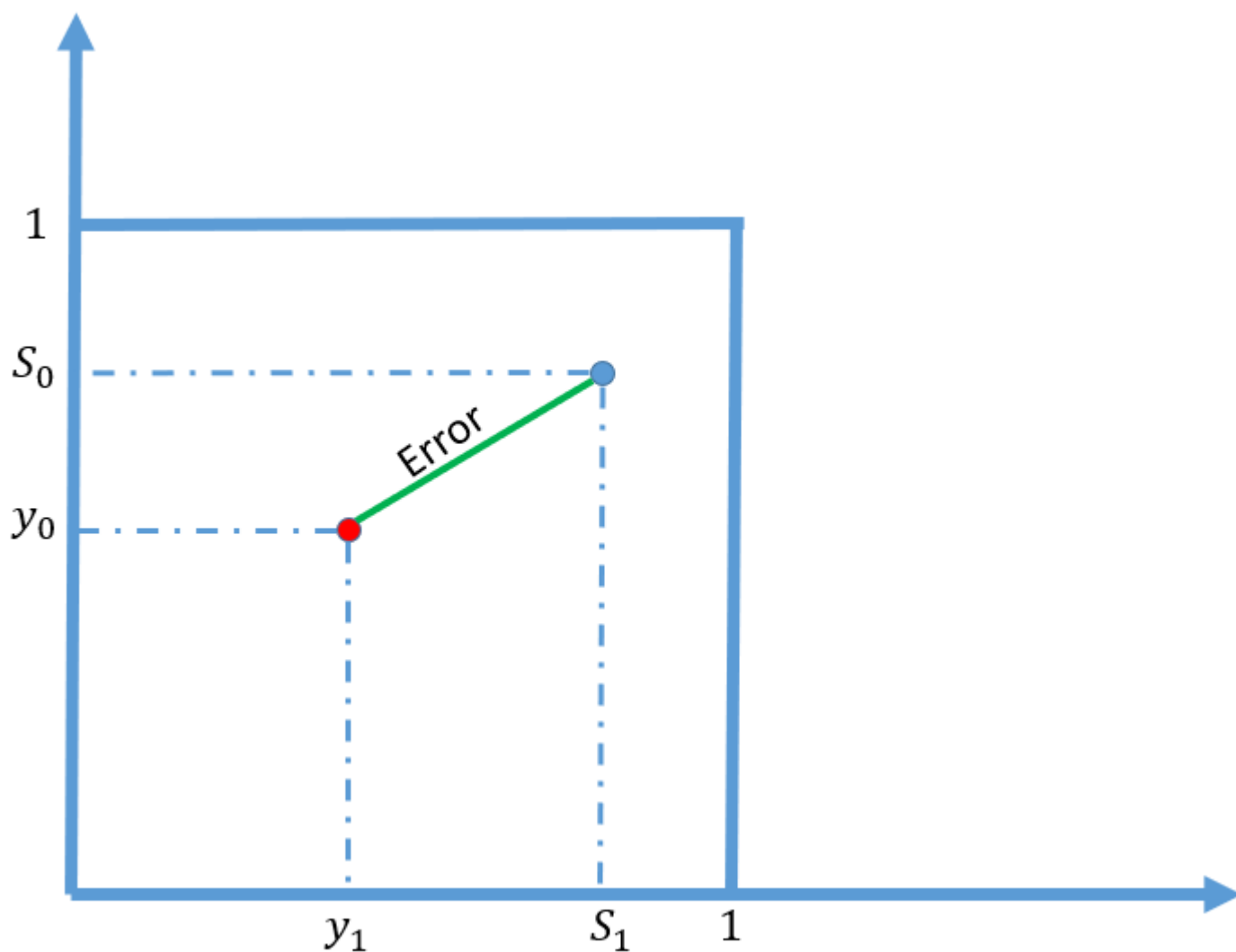


Ilustración 46: Representación del error

Como la función de salida de las neuronas es la sigmoidea, la salida está entre 0 y 1.

En el gráfico de color verde está el error y para calcularlo es usar la fórmula de distancia entre dos puntos en un plano:

$$Error = \sqrt{(S_1 - y_1)^2 + (S_0 - y_0)^2}$$

Para minimizar ese error hay que considerar que dado:

$$F(x) = \sqrt{g(x)}$$

Al derivar:

$$\partial F(x) = \frac{\partial g(x)}{2 * \sqrt{g(x)}}$$

Y como hay que minimizar se iguala esa derivada a cero

$$\partial F(x) = \frac{\partial g(x)}{2 * \sqrt{g(x)}} = 0$$

Luego

$$\partial g(x) = 0$$

En otras palabras, la raíz cuadrada de $F(x)$, es irrelevante cuando se busca minimizar, porque lo importante es minimizar el interior. Luego la ecuación del error pasa a ser:

$$Error = (S_1 - y_1)^2 + (S_0 - y_0)^2$$

Que es más sencilla de evaluar. El siguiente paso es multiplicarla por unas constantes quedando así:

$$Error = \frac{1}{2} (S_1 - y_1)^2 + \frac{1}{2} (S_0 - y_0)^2$$

¿Y por qué se hizo eso? Para hacer que la derivada de Error sea más sencilla. Y no hay que preocuparse porque afecte los resultados: como se busca minimizar, esas constantes no afectan el procedimiento.

¡OJO! Hay que recordar que y_0, y_1 , varían, en cambio, S_0, S_1 son constantes porque son los valores esperados.

Hay que considerar esta regla matemática: Si P es una función con varias variables independientes, es decir: P(m,n) y Q también es otra función con esas mismas variables independientes, es decir: Q(m,n) y hay una *superfunción* que hace uso de P y Q, es decir: K(P,Q), entonces para derivar a K por una de las variables independientes, tenemos:

$$\frac{\partial K}{\partial m} = \frac{\partial K}{\partial P} * \frac{\partial P}{\partial m} + \frac{\partial K}{\partial Q} * \frac{\partial Q}{\partial m}$$

o

$$\frac{\partial K}{\partial n} = \frac{\partial K}{\partial P} * \frac{\partial P}{\partial n} + \frac{\partial K}{\partial Q} * \frac{\partial Q}{\partial n}$$

Luego

$$\frac{\partial Error}{\partial \blacksquare} = \frac{\partial Error}{\partial y_0} * \frac{\partial y_0}{\partial \blacksquare} + \frac{\partial Error}{\partial y_1} * \frac{\partial y_1}{\partial \blacksquare}$$

¿Y qué es ese cuadro relleno negro? Puede ser algún peso o algún umbral. Generalizando:

$$\frac{\partial Error}{\partial \blacksquare} = \sum_{i=0}^{N_2-1} \left(\frac{\partial Error}{\partial y_i} * \frac{\partial y_i}{\partial \blacksquare} \right)$$

Sabiendo que

$$Error = \frac{1}{2} (S_1 - y_1)^2 + \frac{1}{2} (S_0 - y_0)^2$$

Entonces la derivada de Error con respecto a y_0 es:

$$\frac{\partial Error}{\partial y_0} = y_0 - S_0$$

Generalizando

$$\frac{\partial Error}{\partial y_i} = y_i - S_i$$

En el ejemplo, hay dos salidas y_0 y y_1 que están en la capa de salida N_2

Luego, la sumatoria sería de 0 a 1

$$\frac{\partial Error}{\partial \blacksquare} = \sum_{i=0}^{N_2-1} \left((y_i - S_i) * \frac{\partial y_i}{\partial \blacksquare} \right)$$

Queda entonces el cuadro relleno negro que como se mencionó anteriormente puede ser un peso o un umbral. Entonces si tenemos por ejemplo que:

$$\blacksquare = w_{j,i}^{(2)}$$

Entonces como hay una **i** en particular, la sumatoria se retira luego.

$$\frac{\partial Error}{\partial w_{j,i}^{(2)}} = (y_i - S_i) * \frac{\partial y_i}{\partial w_{j,i}^{(2)}}$$

Y como

$$\frac{\partial y_i}{\partial w_{j,i}^{(2)}} = a_j^{(1)} * y_i * (1 - y_i)$$

Luego

$$\frac{\partial Error}{\partial w_{j,i}^{(2)}} = (y_i - S_i) * a_j^{(1)} * y_i * (1 - y_i)$$

Ordenando

$$\frac{\partial Error}{\partial w_{j,i}^{(2)}} = a_j^{(1)} * (y_i - S_i) * y_i * (1 - y_i)$$

De nuevo la derivada del error

$$\frac{\partial Error}{\partial \blacksquare} = \sum_{i=0}^{N_2-1} \left((y_i - S_i) * \frac{\partial y_i}{\partial \blacksquare} \right)$$

Suponiendo que

$$\blacksquare = w_{j,k}^{(1)}$$

Entonces

$$\frac{\partial Error}{\partial w_{j,k}^{(1)}} = \sum_{i=1}^{N_2-1} \left((y_i - S_i) * \frac{\partial y_i}{\partial w_{j,k}^{(1)}} \right)$$

Y como

$$\frac{\partial y_i}{\partial w_{j,k}^{(1)}} = a_j^{(0)} * a_k^{(1)} * (1 - a_k^{(1)}) * w_{k,i}^{(2)} * y_i * (1 - y_i)$$

Entonces

$$\frac{\partial Error}{\partial w_{j,k}^{(1)}} = \sum_{i=0}^{N_2-1} \left((y_i - S_i) * a_j^{(0)} * a_k^{(1)} * (1 - a_k^{(1)}) * w_{k,i}^{(2)} * y_i * (1 - y_i) \right)$$

Simplificando

$$\frac{\partial Error}{\partial w_{j,k}^{(1)}} = a_j^{(0)} * a_k^{(1)} * (1 - a_k^{(1)}) * \sum_{i=0}^{N_2-1} \left((y_i - S_i) * w_{k,i}^{(2)} * y_i * (1 - y_i) \right)$$

De nuevo la derivada del error

$$\frac{\partial Error}{\partial \blacksquare} = \sum_{i=0}^{N_2-1} \left((y_i - S_i) * \frac{\partial y_i}{\partial \blacksquare} \right)$$

Suponiendo que

$$\blacksquare = w_{j,k}^{(0)}$$

Luego la derivada del error es:

$$\frac{\partial Error}{\partial w_{j,k}^{(0)}} = \sum_{i=0}^{N_2-1} \left((y_i - S_i) * \frac{\partial y_i}{\partial w_{j,k}^{(0)}} \right)$$

Y como se vio anteriormente que

$$\frac{\partial y_i}{\partial w_{j,k}^{(0)}} = x_j * a_k^{(0)} * (1 - a_k^{(0)}) * \left[\sum_{p=0}^{N_1-1} w_{k,p}^{(1)} * a_p^{(1)} * (1 - a_p^{(1)}) * w_{p,i}^{(2)} \right] * y_i * (1 - y_i)$$

Luego reemplazando en la expresión se obtiene:

$$\frac{\partial Error}{\partial w_{j,k}^{(0)}} = \sum_{i=0}^{N_2-1} \left((y_i - S_i) * x_j * a_k^{(0)} * (1 - a_k^{(0)}) * \left[\sum_{p=0}^{N_1-1} w_{k,p}^{(1)} * a_p^{(1)} * (1 - a_p^{(1)}) * w_{p,i}^{(2)} \right] * y_i * (1 - y_i) \right)$$

Simplificando

$$\frac{\partial Error}{\partial w_{j,k}^{(0)}} = x_j * a_k^{(0)} * (1 - a_k^{(0)}) * \sum_{i=0}^{N_2-1} \left((y_i - S_i) * \left[\sum_{p=0}^{N_1-1} w_{k,p}^{(1)} * a_p^{(1)} * (1 - a_p^{(1)}) * w_{p,i}^{(2)} \right] * y_i * (1 - y_i) \right)$$

$$\frac{\partial Error}{\partial w_{j,k}^{(0)}} = x_j * a_k^{(0)} * (1 - a_k^{(0)}) * \sum_{p=0}^{N_1-1} \left[w_{k,p}^{(1)} * a_p^{(1)} * (1 - a_p^{(1)}) * \sum_{i=0}^{N_2-1} \left(w_{p,i}^{(2)} * (y_i - S_i) * y_i * (1 - y_i) \right) \right]$$

En limpio las fórmulas para los pesos son:

$$\frac{\partial Error}{\partial w_{j,i}^{(2)}} = a_j^{(1)} * (y_i - S_i) * y_i * (1 - y_i)$$

$$\frac{\partial Error}{\partial w_{j,k}^{(1)}} = a_j^{(0)} * a_k^{(1)} * (1 - a_k^{(1)}) * \sum_{i=0}^{N_2-1} \left(w_{k,i}^{(2)} * (y_i - S_i) * y_i * (1 - y_i) \right)$$

$$\frac{\partial Error}{\partial w_{j,k}^{(0)}} = x_j * a_k^{(0)} * (1 - a_k^{(0)}) * \sum_{p=0}^{N_1-1} \left[w_{k,p}^{(1)} * a_p^{(1)} * (1 - a_p^{(1)}) * \sum_{i=0}^{N_2-1} \left(w_{p,i}^{(2)} * (y_i - S_i) * y_i * (1 - y_i) \right) \right]$$

Y para los umbrales sería:

$$\frac{\partial Error}{\partial u_i^{(2)}} = (y_i - S_i) * y_i * (1 - y_i)$$

$$\frac{\partial Error}{\partial u_k^{(1)}} = a_k^{(1)} * (1 - a_k^{(1)}) * \sum_{i=0}^{N_2-1} \left(w_{k,i}^{(2)} * (y_i - S_i) * y_i * (1 - y_i) \right)$$

$$\frac{\partial Error}{\partial u_k^{(0)}} = a_k^{(0)} * (1 - a_k^{(0)}) * \sum_{p=0}^{N_1-1} \left[w_{k,p}^{(1)} * a_p^{(1)} * (1 - a_p^{(1)}) * \sum_{i=0}^{N_2-1} \left(w_{p,i}^{(2)} * (y_i - S_i) * y_i * (1 - y_i) \right) \right]$$

La fórmula de variación de los pesos y umbrales es:

$$w_{j,i}^{(2)} \leftarrow w_{j,i}^{(2)} - \alpha * \frac{\partial Error}{\partial w_{j,i}^{(2)}}$$

$$w_{j,k}^{(1)} \leftarrow w_{j,k}^{(1)} - \alpha * \frac{\partial Error}{\partial w_{j,k}^{(1)}}$$

$$w_{j,k}^{(0)} \leftarrow w_{j,k}^{(0)} - \alpha * \frac{\partial Error}{\partial w_{j,k}^{(0)}}$$

$$u_i^{(2)} \leftarrow u_i^{(2)} - \alpha * \frac{\partial Error}{\partial u_i^{(2)}}$$

$$u_k^{(1)} \leftarrow u_k^{(1)} - \alpha * \frac{\partial Error}{\partial u_k^{(1)}}$$

$$u_k^{(0)} \leftarrow u_k^{(0)} - \alpha * \frac{\partial Error}{\partial u_k^{(0)}}$$

Donde α es el factor de aprendizaje con un valor pequeño entre 0.1 y 0.9

Implementación en C# del perceptrón multicapa

El siguiente modelo entidad-relación muestra cómo se compone un perceptrón multicapa

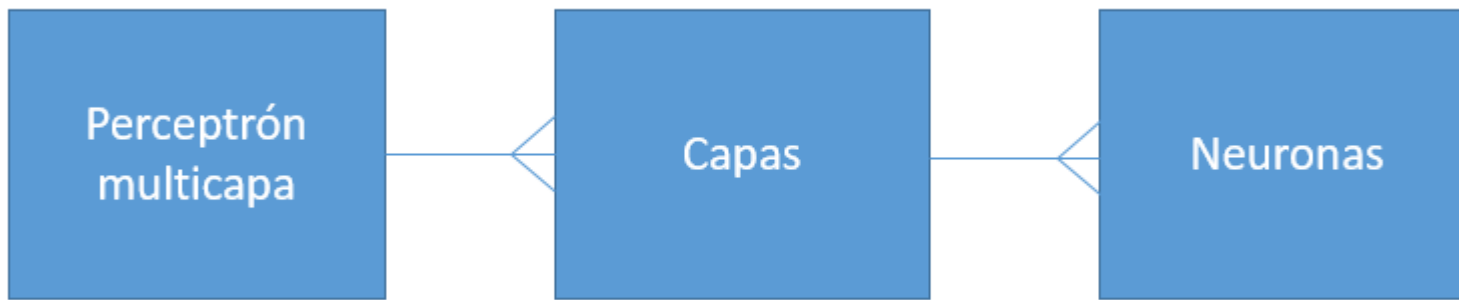


Ilustración 47: Modelo del perceptrón

Un perceptrón tiene dos o más capas (mínimo una oculta y la de salida). Una capa tiene uno o más neuronas.

Para implementarlo se hace uso de clases y listas.

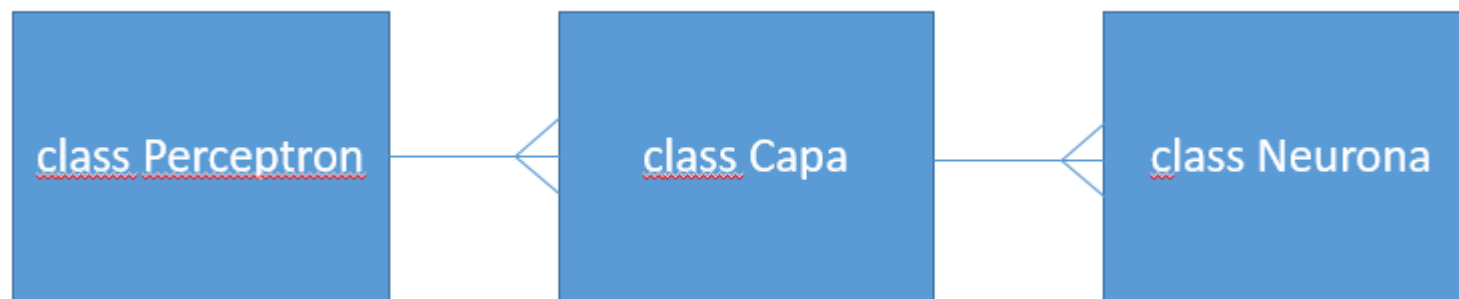


Ilustración 48: Modelo del perceptrón

Esta sería la plantilla del programa

```
using System.Collections.Generic;

namespace AplicacionConsola {
    class Program {
        static void Main(string[] args) {
        }
    }

    class Perceptron {
        List<Capa> capas;
    }

    class Capa {
        List<Neurona> neuronas;
    }

    class Neurona {
    }
}
```

Cada neurona tiene los pesos de entrada y el umbral. En el constructor se inicializan los pesos al azar. Así quedaría el código:

```
using System;
using System.Collections.Generic;

namespace AplicacionConsola {
    class Program {
        static void Main(string[] args) {
        }
    }

    class Perceptron {
        List<Capa> capas;
    }

    class Capa {
        List<Neurona> neuronas;
    }

    class Neurona {
        private List<double> pesos; //Los pesos para cada entrada
        double umbral; //El peso del umbral

        //Inicializa los pesos y umbral con valores al azar
        public Neurona(Random azar, int totalEntradas) {
            pesos = new List<double>();
            for (int cont=0; cont< totalEntradas; cont++) {
                pesos.Add(azar.NextDouble());
            }
            umbral = azar.NextDouble();
        }
    }
}
```

El objeto azar es enviado al constructor de esta clase porque es necesario mantener sólo un generador de números pseudo-aleatorios.

Se añade a la clase neurona el método que calcula la salida que tiene como parámetro un arreglo unidimensional con los datos de entrada.

```
using System;
using System.Collections.Generic;

namespace AplicacionConsola {
    class Program {
        static void Main(string[] args) {
        }
    }

    class Perceptron {
        List<Capa> capas;
    }

    class Capa {
        List<Neurona> neuronas;
    }

    class Neurona {
        private List<double> pesos; //Los pesos para cada entrada
        double umbral; //El peso del umbral

        //Inicializa los pesos y umbral con un valor al azar
        public Neurona(Random azar, int totalEntradas) {
            pesos = new List<double>();
            for (int cont=0; cont< totalEntradas; cont++) {
                pesos.Add(azar.NextDouble());
            }
            umbral = azar.NextDouble();
        }

        //Calcula la salida de la neurona dependiendo de las entradas
        public double calculaSalida(List<double> entradas) {
            double valor = 0;
            for (int cont = 0; cont < pesos.Count; cont++) {
                valor += entradas[cont] * pesos[cont];
            }
            valor += umbral;
            return 1 / (1 + Math.Exp(-valor));
        }
    }
}
```

La clase capa crea las neuronas en el constructor y almacena la salida de cada neurona de esa capa en la lista salidas para facilitar los cálculos más adelante. Se añade el método en que calcula la salida de cada neurona y guarda ese resultado en el listado de "salidas". Así queda el código:

```
using System;
using System.Collections.Generic;

namespace AplicacionConsola {
    class Program {
        static void Main(string[] args) {
        }
    }

    class Perceptron {
        List<Capa> capas;
    }

    class Capa {
        List<Neurona> neuronas; //Las neuronas que tendrá la capa
        List<double> salidas; //Almacena las salidas de cada neurona

        public Capa(Random azar, int totalNeuronas, int totalEntradas) {
            neuronas = new List<Neurona>();
            salidas = new List<double>();
            //Genera las neuronas e inicializa sus salidas
            for (int cont = 0; cont < totalNeuronas; cont++) {
                neuronas.Add(new Neurona(azar, totalEntradas));
                salidas.Add(0);
            }
        }

        //Calcula las salidas de cada neurona de la capa
        public void CalculaCapa(List<double> entradas) {
            for (int cont = 0; cont < neuronas.Count; cont++) {
                salidas[cont] = neuronas[cont].calculaSalida(entradas);
            }
        }
    }

    class Neurona {
        private List<double> pesos; //Los pesos para cada entrada
        double umbral; //El peso del umbral

        //Inicializa los pesos y umbral con un valor al azar
        public Neurona(Random azar, int totalEntradas) {
            pesos = new List<double>();
            for (int cont=0; cont< totalEntradas; cont++) {
                pesos.Add(azar.NextDouble());
            }
            umbral = azar.NextDouble();
        }

        //Calcula la salida de la neurona dependiendo de las entradas
        public double calculaSalida(List<double> entradas) {
            double valor = 0;
            for (int cont = 0; cont < pesos.Count; cont++) {
                valor += entradas[cont] * pesos[cont];
            }
            valor += umbral;
            return 1 / (1 + Math.Exp(-valor));
        }
    }
}
```

La clase Perceptron crea las capas

```
using System;
using System.Collections.Generic;

namespace AplicacionConsola {
    class Program {
        static void Main(string[] args) {
        }
    }

    class Perceptron {
        List<Capa> capas;

        //Crea las diversas capas
        public void creaCapas(Random azar, int numEntrada, int capa0, int capa1, int capa2) {
            capas = new List<Capa>();
            //Crea la capa 0
            capas.Add(new Capa(azar, capa0, numEntrada));

            //Crea la capa 1 (el número de entradas es el número de neuronas de la capa anterior)
            capas.Add(new Capa(azar, capa1, capa0));

            //Crea la capa 2 (el número de entradas es el número de neuronas de la capa anterior)
            capas.Add(new Capa(azar, capa2, capa1));
        }
    }

    class Capa {
        List<Neurona> neuronas; //Las neuronas que tendrá la capa
        List<double> salidas; //Almacena las salidas de cada neurona

        public Capa(Random azar, int totalNeuronas, int totalEntradas) {
            neuronas = new List<Neurona>();
            salidas = new List<double>();
            //Genera las neuronas e inicializa sus salidas
            for (int cont = 0; cont < totalNeuronas; cont++) {
                neuronas.Add(new Neurona(azar, totalEntradas));
                salidas.Add(0);
            }
        }

        //Calcula las salidas de cada neurona de la capa
        public void CalculaCapa(List<double> entradas) {
            for (int cont = 0; cont < neuronas.Count; cont++) {
                salidas[cont] = neuronas[cont].calculaSalida(entradas);
            }
        }
    }

    class Neurona {
        private List<double> pesos; //Los pesos para cada entrada
        double umbral; //El peso del umbral

        //Inicializa los pesos y umbral con un valor al azar
        public Neurona(Random azar, int totalEntradas) {
            pesos = new List<double>();
            for (int cont=0; cont< totalEntradas; cont++) {
                pesos.Add(azar.NextDouble());
            }
            umbral = azar.NextDouble();
        }

        //Calcula la salida de la neurona dependiendo de las entradas
        public double calculaSalida(List<double> entradas) {
            double valor = 0;
            for (int cont = 0; cont < pesos.Count; cont++) {
                valor += entradas[cont] * pesos[cont];
            }
            valor += umbral;
            return 1 / (1 + Math.Exp(-valor));
        }
    }
}
```

En Perceptron se hace el cálculo de cada capa. Cabe recordar que la salida de la capa 0 es la entrada de la capa 1, y la salida de la capa 1 es la entrada de la capa 2. Así quedaría el código:

```
using System;
using System.Collections.Generic;

namespace AplicacionConsola {
    class Program {
        static void Main(string[] args) {
        }
    }

    class Perceptron {
        List<Capa> capas;

        //Crea las diversas capas
        public void creaCapas(Random azar, int numEntrada, int capa0, int capa1, int capa2) {
            capas = new List<Capa>();
            capas.Add(new Capa(azar, capa0, numEntrada)); //Crea la capa 0
            capas.Add(new Capa(azar, capa1, capa0)); //Crea la capa 1
            capas.Add(new Capa(azar, capa2, capa1)); //Crea la capa 2
        }

        public void calculaSalida(List<double> entradas) {
            capas[0].CalculaCapa(entradas);
            //Las salidas de la capa anterior son las entradas de la siguiente capa
            capas[1].CalculaCapa(capas[0].salidas);
            capas[2].CalculaCapa(capas[1].salidas);
        }
    }

    class Capa {
        List<Neurona> neuronas; //Las neuronas que tendrá la capa
        public List<double> salidas; //Almacena las salidas de cada neurona

        public Capa(Random azar, int totalNeuronas, int totalEntradas) {
            neuronas = new List<Neurona>();
            salidas = new List<double>();
            //Genera las neuronas e inicializa sus salidas
            for (int cont = 0; cont < totalNeuronas; cont++) {
                neuronas.Add(new Neurona(azar, totalEntradas));
                salidas.Add(0);
            }
        }

        //Calcula las salidas de cada neurona de la capa
        public void CalculaCapa(List<double> entradas) {
            for (int cont = 0; cont < neuronas.Count; cont++) {
                salidas[cont] = neuronas[cont].calculaSalida(entradas);
            }
        }
    }

    class Neurona {
        private List<double> pesos; //Los pesos para cada entrada
        double umbral; //El peso del umbral

        //Inicializa los pesos y umbral con un valor al azar
        public Neurona(Random azar, int totalEntradas) {
            pesos = new List<double>();
            for (int cont=0; cont< totalEntradas; cont++) {
                pesos.Add(azar.NextDouble());
            }
            umbral = azar.NextDouble();
        }

        //Calcula la salida de la neurona dependiendo de las entradas
        public double calculaSalida(List<double> entradas) {
            double valor = 0;
            for (int cont = 0; cont < pesos.Count; cont++) {
                valor += entradas[cont] * pesos[cont];
            }
            valor += umbral;
            return 1 / (1 + Math.Exp(-valor));
        }
    }
}
```

Ejemplo de uso de la clase Perceptron para generar este diseño en particular:

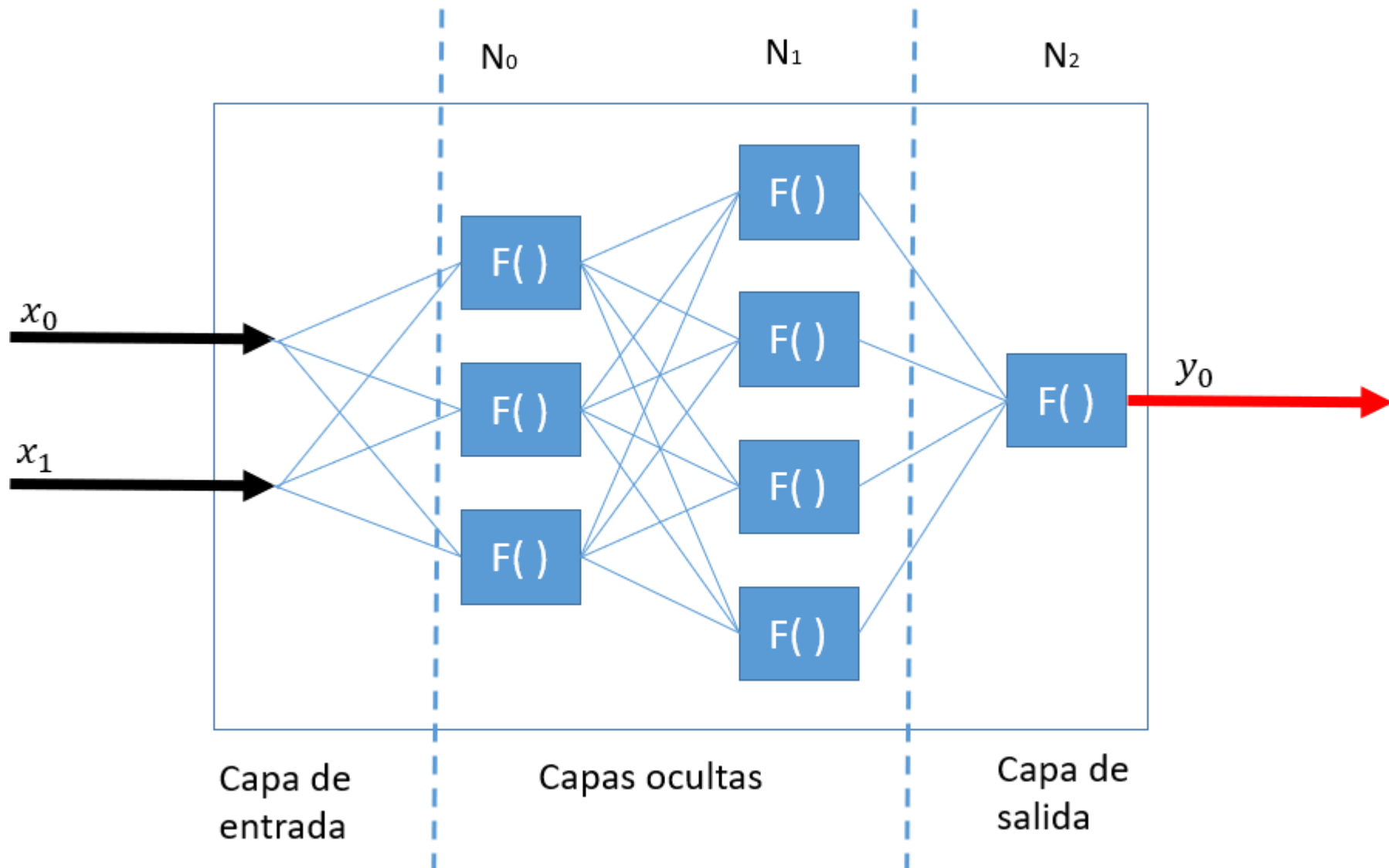


Ilustración 49: Un perceptrón multicapa

```
class Program {
    static void Main(string[] args) {
        Random azar = new Random(); //Un solo generador de números pseudo-aleatorios
        Perceptron perceptron = new Perceptron();

        //Entradas=2, capa0=4, capa1=3, capa2=2
        int numEntradas = 2;
        int capa0 = 3;
        int capa1 = 4;
        int capa2 = 1;
        perceptron.creaCapas(azar, numEntradas, capa0, capa1, capa2);

        //Estas serán las entradas externas al perceptrón
        List<double> entradas = new List<double>();
        entradas.Add(1);
        entradas.Add(0);

        //Se hace el cálculo
        perceptron.calculaSalida(entradas);

        Console.ReadKey();
    }
}
```

Es en el programa principal que se crea el objeto que genera los números pseudo-aleatorios y se le envía al perceptrón.

Código en C# para que muestre paso a paso como hace los cálculos por capa y por neurona:

```
using System;
using System.Collections.Generic;

namespace AplicacionConsola {
    class Program {
        static void Main(string[] args) {
            Random azar = new Random(); //Un solo generador de números pseudo-aleatorios
            Perceptron perceptron = new Perceptron();

            int numEntradas = 2; //Número de entradas
            int capa0 = 3; //Total neuronas en la capa 0
            int capa1 = 4; //Total neuronas en la capa 1
            int capa2 = 1; //Total neuronas en la capa 2
            perceptron.creaCapas(azar, numEntradas, capa0, capa1, capa2);

            //Estas serán las entradas externas al perceptrón
            List<double> entradas = new List<double>();
            entradas.Add(1);
            entradas.Add(0);

            //Se hace el cálculo
            perceptron.calculaSalida(entradas);
            Console.ReadKey();
        }
    }

    class Perceptron {
        List<Capa> capas;

        //Crea las diversas capas
        public void creaCapas(Random azar, int numEntrada, int capa0, int capa1, int capa2) {
            capas = new List<Capa>();
            Console.WriteLine("Genera Capa 0");
            capas.Add(new Capa(azar, capa0, numEntrada)); //Crea la capa 0
            Console.WriteLine("Genera Capa 1");
            capas.Add(new Capa(azar, capa1, capa0)); //Crea la capa 1
            Console.WriteLine("Genera Capa 2");
            capas.Add(new Capa(azar, capa2, capa1)); //Crea la capa 2
        }

        public void calculaSalida(List<double> entradas) {
            Console.WriteLine("\nCálculos capa 0");
            capas[0].CalculaCapa(entradas);
            Console.WriteLine("\nCálculos capa 1");
            capas[1].CalculaCapa(capas[0].salidas);
            Console.WriteLine("\nCálculos capa 2");
            capas[2].CalculaCapa(capas[1].salidas);
        }
    }

    class Capa {
        List<Neurona> neuronas; //Las neuronas que tendrá la capa
        public List<double> salidas; //Almacena las salidas de cada neurona

        public Capa(Random azar, int totalNeuronas, int totalEntradas) {
            neuronas = new List<Neurona>();
            salidas = new List<double>();
            //Genera las neuronas e inicializa sus salidas
            for (int cont = 0; cont < totalNeuronas; cont++) {
                neuronas.Add(new Neurona(azar, totalEntradas));
                salidas.Add(0);
            }
        }

        //Calcula las salidas de cada neurona de la capa
        public void CalculaCapa(List<double> entradas) {
            for (int cont = 0; cont < entradas.Count; cont++) {
                Console.Write(" Entra " + cont.ToString() + ": ");
                Console.Write("{0:F4};", entradas[cont]);
            }
            Console.WriteLine(" ");

            for (int cont = 0; cont < neuronas.Count; cont++) {
                salidas[cont] = neuronas[cont].calculaSalida(entradas);
                Console.Write(" Salir " + cont.ToString() + ": ");
                Console.Write("{0:F4};", salidas[cont]);
            }
        }
    }
}
```

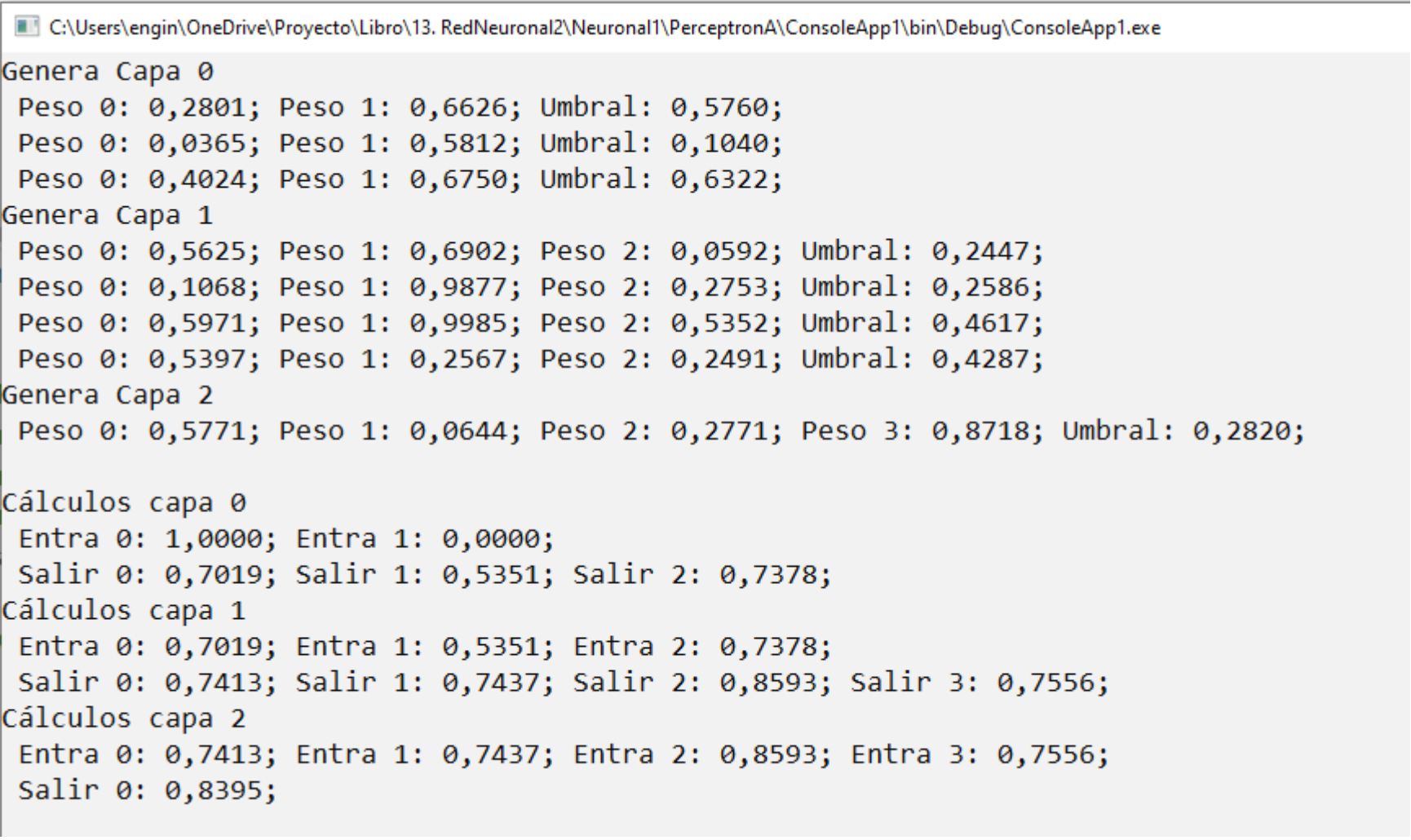
```

class Neurona {
    private List<double> pesos; //Los pesos para cada entrada
    double umbral; //El peso del umbral

    //Inicializa los pesos y umbral con un valor al azar
    public Neurona(Random azar, int totalEntradas) {
        pesos = new List<double>();
        for (int cont=0; cont< totalEntradas; cont++) {
            pesos.Add(azar.NextDouble());
            Console.Write(" Peso " + cont.ToString() + ": ");
            Console.Write("{0:F4};", pesos[cont]);
        }
        umbral = azar.NextDouble();
        Console.Write(" Umbral: ");
        Console.WriteLine("{0:F4};", umbral);
    }

    //Calcula la salida de la neurona dependiendo de las entradas
    public double calculaSalida(List<double> entradas) {
        double valor = 0;
        for (int cont = 0; cont < pesos.Count; cont++) {
            valor += entradas[cont] * pesos[cont];
        }
        valor += umbral;
        return 1 / (1 + Math.Exp(-valor));
    }
}

```



```

C:\Users\engin\OneDrive\Proyecto\Libro\13. RedNeuronal2\Neuronal1\PerceptronA\ConsoleApp1\bin\Debug\ConsoleApp1.exe
Genera Capa 0
Peso 0: 0,2801; Peso 1: 0,6626; Umbral: 0,5760;
Peso 0: 0,0365; Peso 1: 0,5812; Umbral: 0,1040;
Peso 0: 0,4024; Peso 1: 0,6750; Umbral: 0,6322;
Genera Capa 1
Peso 0: 0,5625; Peso 1: 0,6902; Peso 2: 0,0592; Umbral: 0,2447;
Peso 0: 0,1068; Peso 1: 0,9877; Peso 2: 0,2753; Umbral: 0,2586;
Peso 0: 0,5971; Peso 1: 0,9985; Peso 2: 0,5352; Umbral: 0,4617;
Peso 0: 0,5397; Peso 1: 0,2567; Peso 2: 0,2491; Umbral: 0,4287;
Genera Capa 2
Peso 0: 0,5771; Peso 1: 0,0644; Peso 2: 0,2771; Peso 3: 0,8718; Umbral: 0,2820;

Cálculos capa 0
Entra 0: 1,0000; Entra 1: 0,0000;
Salir 0: 0,7019; Salir 1: 0,5351; Salir 2: 0,7378;
Cálculos capa 1
Entra 0: 0,7019; Entra 1: 0,5351; Entra 2: 0,7378;
Salir 0: 0,7413; Salir 1: 0,7437; Salir 2: 0,8593; Salir 3: 0,7556;
Cálculos capa 2
Entra 0: 0,7413; Entra 1: 0,7437; Entra 2: 0,8593; Entra 3: 0,7556;
Salir 0: 0,8395;

```

Ilustración 50: Ejecución del programa

Nota: en el programa en C# los valores están formateados para mostrar 4 decimales, pero internamente el cálculo se hace con todos los decimales.

Las fórmulas del algoritmo de retro propagación para la capa 2:

$$\frac{\partial Error}{\partial w_{j,i}^{(2)}} = a_j^{(1)} * (y_i - S_i) * y_i * (1 - y_i)$$

Y

$$w_{j,i}^{(2)} \leftarrow w_{j,i}^{(2)} - \alpha * \frac{\partial Error}{\partial w_{j,i}^{(2)}}$$

Se implementa así:

```
//Procesa pesos capa 2
for (int j = 0; j < capa1; j++) //Va de neurona en neurona de la capa 1
    for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
        double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
        double Si = salidas[i]; //Salida esperada
        double alj = capas[1].salidas[j]; //Salida de la capa 1
        double dE2 = alj * (Yi - Si) * Yi * (1 - Yi); //La fórmula del error
        Console.WriteLine("dError/dW(2)" + j.ToString() + "," + i.ToString() + " = " + dE2.ToString());
        double nuevoPeso = capas[2].neuronas[i].pesos[j] - alpha * dE2; //Ajusta el nuevo peso
        Console.WriteLine("W(2)" + j.ToString() + "," + i.ToString() + " = " + nuevoPeso.ToString());
    }
```


Para la capa 1:

$$\frac{\partial Error}{\partial w_{j,k}^{(1)}} = a_j^{(0)} * a_k^{(1)} * \left(1 - a_k^{(1)}\right) * \sum_{i=0}^{N_2-1} \left(w_{k,i}^{(2)} * (y_i - S_i) * y_i * (1 - y_i)\right)$$

Y

$$w_{j,k}^{(1)} \leftarrow w_{j,k}^{(1)} - \alpha * \frac{\partial Error}{\partial w_{j,k}^{(1)}}$$

Se implementa así:

```
//Procesa pesos capa 1
for (int j = 0; j < capa0; j++) //Va de neurona en neurona de la capa 0
    for (int k = 0; k < capa1; k++) { //Va de neurona en neurona de la capa 1
        double acum = 0;
        for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
            double Yi = capas[2].salidas[i]; //Salida de la capa 2
            double Si = salidas[i]; //Salida esperada
            double W2ki = capas[2].neuronas[i].pesos[k];
            acum += W2ki * (Yi - Si) * Yi * (1 - Yi); //Sumatoria
        }
        double a0j = capas[0].salidas[j];
        double alk = capas[1].salidas[k];
        double dE1 = a0j * alk * (1 - alk) * acum;
        Console.WriteLine("dError/dW(1)" + j.ToString() + "," + k.ToString() + " = " + dE1.ToString());
        double nuevoPeso = capas[1].neuronas[k].pesos[j] - alpha * dE1;
        Console.WriteLine("W(1)" + j.ToString() + "," + k.ToString() + " = " + nuevoPeso.ToString());
    }
```

Para la capa 0:

$$\frac{\partial Error}{\partial w_{j,k}^{(0)}} = x_j * a_k^{(0)} * \left(1 - a_k^{(0)}\right) * \sum_{p=0}^{N_1-1} \left[w_{k,p}^{(1)} * a_p^{(1)} * \left(1 - a_p^{(1)}\right) * \sum_{i=0}^{N_2-1} \left(w_{p,i}^{(2)} * (y_i - S_i) * y_i * (1 - y_i) \right) \right]$$

Y

$$w_{j,k}^{(0)} \leftarrow w_{j,k}^{(0)} - \alpha * \frac{\partial Error}{\partial w_{j,k}^{(0)}}$$

Se implementa así:

```
//Procesa pesos capa 0
for (int j = 0; j < entradas.Count; j++) //Va de entrada en entrada
    for (int k = 0; k < capa0; k++) { //Va de neurona en neurona de la capa 0
        double acumular = 0;
        for (int p = 0; p < capa1; p++) { //Va de neurona en neurona de la capa 1
            double acum = 0;
            for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                double Yi = capas[2].salidas[i];
                double Si = salidas[i]; //Salida esperada
                double W2pi = capas[2].neuronas[i].pesos[p];
                acum += W2pi * (Yi - Si) * Yi * (1 - Yi); //Sumatoria interna
            }
            double W1kp = capas[1].neuronas[p].pesos[k];
            double alp = capas[1].salidas[p];
            acumular += W1kp * alp * (1 - alp) * acum; //Sumatoria externa
        }
        double xj = entradas[j];
        double a0k = capas[0].salidas[k];
        double dE0 = xj * a0k * (1 - a0k) * acumular;
        Console.WriteLine("dError/dW(0)" + j.ToString() + "," + k.ToString() + " = " + dE0.ToString());
        double W0jk = capas[0].neuronas[k].pesos[j];
        double nuevoPeso = W0jk - alpha * dE0;
        Console.WriteLine("W(1)" + j.ToString() + "," + k.ToString() + " = " + nuevoPeso.ToString());
    }
}
```

Para los umbrales de la capa 2

$$\frac{\partial Error}{\partial u_i^{(2)}} = (y_i - S_i) * y_i * (1 - y_i)$$

$$u_i^{(2)} \leftarrow u_i^{(2)} - \alpha * \frac{\partial Error}{\partial u_i^{(2)}}$$

```
//Procesa umbrales capa 2
for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
    double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
    double Si = salidas[i]; //Salida esperada
    double dE2 = (Yi - Si) * Yi * (1 - Yi);
    Console.WriteLine("dError/dU(2)" + i.ToString() + " = " + dE2.ToString());
    double nuevoUmbral = capas[2].neuronas[i].umbral - alpha * dE2;
    Console.WriteLine("U(2)" + i.ToString() + " = " + nuevoUmbral.ToString());
}
```

Para los umbrales de la capa 1

$$\frac{\partial Error}{\partial u_k^{(1)}} = a_k^{(1)} * \left(1 - a_k^{(1)}\right) * \sum_{i=0}^{N_2-1} \left(w_{k,i}^{(2)} * (y_i - S_i) * y_i * (1 - y_i)\right)$$

$$u_k^{(1)} \leftarrow u_k^{(1)} - \alpha * \frac{\partial Error}{\partial u_k^{(1)}}$$

```
//Procesa umbrales capa 1
for (int k = 0; k < capa1; k++) { //Va de neurona en neurona de la capa 1
    double acum = 0;
    for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
        double Yi = capas[2].salidas[i]; //Salida de la capa 2
        double Si = salidas[i];
        double W2ki = capas[2].neuronas[i].pesos[k];
        acum += W2ki * (Yi - Si) * Yi * (1 - Yi);
    }
    double alk = capas[1].salidas[k];
    double dE1 = alk * (1 - alk) * acum;
    Console.WriteLine("dError/dU(1)" + k.ToString() + " = " + dE1.ToString());
    double nuevoUmbral = capas[1].neuronas[k].umbral - alpha * dE1;
    Console.WriteLine("U(1)" + k.ToString() + " = " + nuevoUmbral.ToString());
}
```

Para los umbrales de la capa 0

$$\frac{\partial Error}{\partial u_k^{(0)}} = a_k^{(0)} * \left(1 - a_k^{(0)}\right) * \sum_{p=0}^{N_1-1} \left[w_{k,p}^{(1)} * a_p^{(1)} * \left(1 - a_p^{(1)}\right) * \sum_{i=0}^{N_2-1} \left(w_{p,i}^{(2)} * (y_i - S_i) * y_i * (1 - y_i) \right) \right]$$
$$u_k^{(0)} \leftarrow u_k^{(0)} - \alpha * \frac{\partial Error}{\partial u_k^{(0)}}$$

```
//Procesa umbrales capa 0
for (int k = 0; k < capa0; k++) {
    double acumular = 0;
    for (int p = 0; p < capal; p++) {
        double acum = 0;
        for (int i = 0; i < capa2; i++) {
            double Yi = capas[2].salidas[i];
            double Si = salidas[i];
            double W2pi = capas[2].neuronas[i].pesos[p];
            acum += W2pi * (Yi - Si) * Yi * (1 - Yi); //Sumatoria interna
        }
        double W1kp = capas[1].neuronas[p].pesos[k];
        double alp = capas[1].salidas[p];
        acumular += W1kp * alp * (1 - alp) * acum; //Sumatoria externa
    }
    double a0k = capas[0].salidas[k];
    double dE0 = a0k * (1 - a0k) * acumular;
    Console.WriteLine("dError/dU(0)" + k.ToString() + " = " + dE0.ToString());
    double nuevoUmbral = capas[0].neuronas[k].umbral - alpha * dE0;
    Console.WriteLine("U(0)" + k.ToString() + " = " + nuevoUmbral.ToString());
}
```

Como se puede ver en los códigos, se deducen nuevos pesos y umbrales. Esos nuevos valores posteriormente deben reemplazar los viejos. Luego hay que hacer un cambio a la clase Neurona para que almacene temporalmente los nuevos valores para que cuando se termine de calcularlos, entonces reemplaza los viejos.

El código del encabezado sería así:

```
class Neurona {
    public List<double> pesos; //Los pesos para cada entrada
    public List<double> nuevospesos; //Nuevos pesos dados por el algoritmo de "backpropagation"
    public double umbral; //El peso del umbral
    public double nuevoumbral; //Nuevo umbral dado por el algoritmo de "backpropagation"
}
```

Y tendría un nuevo método que actualizaría los pesos con los nuevos valores, **después** de ejecutar el algoritmo de “backpropagation”

```
//Reemplaza viejos pesos por nuevos
public void actualiza(){
    for (int cont = 0; cont < pesos.Count; cont++){
        pesos[cont] = nuevospesos[cont];
    }
    umbral = nuevoumbral;
}
```

Significa que en la clase Capa debe haber un método que llama la actualización de las neuronas

```
//Actualiza los pesos y umbrales de las neuronas
public void actualiza(){
    for (int cont = 0; cont < neuronas.Count; cont++) {
        neuronas[cont].actualiza();
    }
}
```

¿Y cómo se almacenan los nuevos pesos? En el algoritmo de “backpropagation”, por ejemplo, este es código de ajuste de pesos de la capa 2:

```
//Procesa pesos capa 2
for (int j = 0; j < capal; j++) //Va de neurona en neurona de la capa 1
    for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
        double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
        double Si = salidaEsperada[i]; //Salida esperada
        double alj = capas[1].salidas[j]; //Salida de la capa 1
        double dE2 = alj * (Yi - Si) * Yi * (1 - Yi); //La fórmula del error
        capas[2].neuronas[i].nuevospesos[j] = capas[2].neuronas[i].pesos[j] - alpha * dE2; //Ajusta el nuevo peso
    }
```

Código completo del perceptrón en C#

A continuación, se muestra el código completo (ubicado en la carpeta **RedNeuronal1** en GitHub) en el que se ha creado una clase que implementa el perceptrón (creación, proceso, entrenamiento) y en la clase principal se pone como datos de prueba la tabla del XOR que el perceptrón debe aprender.

```
using System;
using System.Collections.Generic;

namespace AplicacionConsola {
    class Program {
        static void Main(string[] args) {
            //Tabla de verdad XOR
            int[,] xorentra = { { 1, 1 }, { 1, 0 }, { 0, 1 }, { 0, 0 } };
            int[] xorsale = { 0, 1, 1, 0 };

            int numEntradas = 2; //Número de entradas
            int capa0 = 3; //Total neuronas en la capa 0
            int capa1 = 2; //Total neuronas en la capa 1
            int capa2 = 1; //Total neuronas en la capa 2
            Perceptron perceptron = new Perceptron(numEntradas, capa0, capa1, capa2);

            //Estas serán las entradas externas al perceptrón
            List<double> entradas = new List<double>();
            entradas.Add(0);
            entradas.Add(0);

            //Estas serán las salidas esperadas externas al perceptrón
            List<double> salidaEsperada = new List<double>();
            salidaEsperada.Add(0);

            //Ciclo que entrena la red neuronal
            int totalCiclos = 9000; //Ciclos de entrenamiento
            for (int ciclo = 1; ciclo <= totalCiclos; ciclo++) {
                //Por cada ciclo, se entrena el perceptrón con toda la tabla de XOR
                for (int conjunto = 0; conjunto < xorsale.Length; conjunto++) {
                    //Entradas y salidas esperadas
                    entradas[0] = xorentra[conjunto, 0];
                    entradas[1] = xorentra[conjunto, 1];
                    salidaEsperada[0] = xorsale[conjunto];

                    //Primero calcula la salida del perceptrón con esas entradas
                    perceptron.calculaSalida(entradas);

                    //Luego entrena el perceptrón para ajustar los pesos y umbrales
                    perceptron.Entrena(entradas, salidaEsperada);
                }
            }

            //Después de entrenar el perceptrón, imprime las salidas que produce
            for (int conjunto = 0; conjunto < xorsale.Length; conjunto++) {
                entradas[0] = xorentra[conjunto, 0];
                entradas[1] = xorentra[conjunto, 1];

                //Calcula la salida del perceptrón con esas entradas
                perceptron.calculaSalida(entradas);

                //Imprime la salida del perceptrón
                Console.WriteLine(perceptron.capas[2].salidas[0].ToString());
            }

            Console.ReadKey();
        }
    }

    class Perceptron {
        public List<Capa> capas;

        //Crea las diversas capas
        public Perceptron(int numEntrada, int capa0, int capa1, int capa2) {
            Random azar = new Random();
            capas = new List<Capa>();
            capas.Add(new Capa(azar, capa0, numEntrada)); //Crea la capa 0
            capas.Add(new Capa(azar, capa1, capa0)); //Crea la capa 1
            capas.Add(new Capa(azar, capa2, capa1)); //Crea la capa 2
        }

        //Dada las entradas al perceptrón, se calcula la salida de cada capa.
        //Con eso se sabrá que salidas se obtienen con los pesos y umbrales actuales.
        //Esas salidas son requeridas para el algoritmo de entrenamiento.
        public void calculaSalida(List<double> entradas) {
            capas[0].CalculaCapa(entradas);
            capas[1].CalculaCapa(capas[0].salidas);
            capas[2].CalculaCapa(capas[1].salidas);
        }

        //Con las salidas previamente calculadas con unas determinadas entradas
        //se ejecuta el algoritmo de entrenamiento "Backpropagation"
        public void Entrena(List<double> entradas, List<double> salidaEsperada) {
            int capa0 = capas[0].neuronas.Count;
            int capa1 = capas[1].neuronas.Count;
            int capa2 = capas[2].neuronas.Count;
```

```

//Factor de aprendizaje
double alpha = 0.4;

//Procesa pesos capa 2
for (int j = 0; j < capa1; j++) //Va de neurona en neurona de la capa 1
    for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
        double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
        double Si = salidaEsperada[i]; //Salida esperada
        double alj = capas[1].salidas[j]; //Salida de la capa 1
        double dE2 = alj * (Yi - Si) * Yi * (1 - Yi); //La fórmula del error
        capas[2].neuronas[i].nuevospesos[j] = capas[2].neuronas[i].pesos[j] - alpha * dE2; //Ajusta el nuevo peso
    }

//Procesa pesos capa 1
for (int j = 0; j < capa0; j++) //Va de neurona en neurona de la capa 0
    for (int k = 0; k < capa1; k++) { //Va de neurona en neurona de la capa 1
        double acum = 0;
        for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
            double Yi = capas[2].salidas[i]; //Salida de la capa 2
            double Si = salidaEsperada[i]; //Salida esperada
            double W2ki = capas[2].neuronas[i].pesos[k];
            acum += W2ki * (Yi - Si) * Yi * (1 - Yi); //Sumatoria
        }
        double a0j = capas[0].salidas[j];
        double alk = capas[1].salidas[k];
        double dE1 = a0j * alk * (1 - alk) * acum;
        capas[1].neuronas[k].nuevospesos[j] = capas[1].neuronas[k].pesos[j] - alpha * dE1;
    }

//Procesa pesos capa 0
for (int j = 0; j < entradas.Count; j++) //Va de entrada en entrada
    for (int k = 0; k < capa0; k++) { //Va de neurona en neurona de la capa 0
        double acumular = 0;
        for (int p = 0; p < capa1; p++) { //Va de neurona en neurona de la capa 1
            double acum = 0;
            for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                double Yi = capas[2].salidas[i];
                double Si = salidaEsperada[i]; //Salida esperada
                double W2pi = capas[2].neuronas[i].pesos[p];
                acum += W2pi * (Yi - Si) * Yi * (1 - Yi); //Sumatoria interna
            }
            double W1kp = capas[1].neuronas[p].pesos[k];
            double alp = capas[1].salidas[p];
            acumular += W1kp * alp * (1 - alp) * acum; //Sumatoria externa
        }
        double xj = entradas[j];
        double a0k = capas[0].salidas[k];
        double dE0 = xj * a0k * (1 - a0k) * acumular;
        double W0jk = capas[0].neuronas[k].pesos[j];
        capas[0].neuronas[k].nuevospesos[j] = W0jk - alpha * dE0;
    }

//Procesa umbrales capa 2
for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
    double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
    double Si = salidaEsperada[i]; //Salida esperada
    double dE2 = (Yi - Si) * Yi * (1 - Yi);
    capas[2].neuronas[i].nuevoumbrales = capas[2].neuronas[i].umbrales - alpha * dE2;
}

//Procesa umbrales capa 1
for (int k = 0; k < capa1; k++) { //Va de neurona en neurona de la capa 1
    double acum = 0;
    for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
        double Yi = capas[2].salidas[i]; //Salida de la capa 2
        double Si = salidaEsperada[i];
        double W2ki = capas[2].neuronas[i].pesos[k];
        acum += W2ki * (Yi - Si) * Yi * (1 - Yi);
    }
    double alk = capas[1].salidas[k];
    double dE1 = alk * (1 - alk) * acum;
    capas[1].neuronas[k].nuevoumbrales = capas[1].neuronas[k].umbrales - alpha * dE1;
}

//Procesa umbrales capa 0
for (int k = 0; k < capa0; k++) { //Va de neurona en neurona de la capa 0
    double acumular = 0;
    for (int p = 0; p < capa1; p++) { //Va de neurona en neurona de la capa 1
        double acum = 0;
        for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
            double Yi = capas[2].salidas[i];
            double Si = salidaEsperada[i];
            double W2pi = capas[2].neuronas[i].pesos[p];
            acum += W2pi * (Yi - Si) * Yi * (1 - Yi);
        }
        double W1kp = capas[1].neuronas[p].pesos[k];
        double alp = capas[1].salidas[p];
        acumular += W1kp * alp * (1 - alp) * acum;
    }
    double a0k = capas[0].salidas[k];
    double dE0 = a0k * (1 - a0k) * acumular;
    capas[0].neuronas[k].nuevoumbrales = capas[0].neuronas[k].umbrales - alpha * dE0;
}

//Actualiza los pesos

```

```
        capas[0].actualiza();
        capas[1].actualiza();
        capas[2].actualiza();
    }
}

class Capa {
    public List<Neurona> neuronas; //Las neuronas que tendrá la capa
    public List<double> salidas; //Almacena las salidas de cada neurona

    public Capa(Random azar, int totalNeuronas, int totalEntradas) {
        neuronas = new List<Neurona>();
        salidas = new List<double>();
        //Genera las neuronas
        for (int cont = 0; cont < totalNeuronas; cont++) {
            neuronas.Add(new Neurona(azar, totalEntradas));
            salidas.Add(0);
        }
    }

    //Calcula las salidas de cada neurona de la capa
    public void CalculaCapa(List<double> entradas) {
        for (int cont = 0; cont < neuronas.Count; cont++) {
            salidas[cont] = neuronas[cont].calculaSalida(entradas);
        }
    }

    //Actualiza los pesos y umbrales de las neuronas
    public void actualiza(){
        for (int cont = 0; cont < neuronas.Count; cont++) {
            neuronas[cont].actualiza();
        }
    }
}

class Neurona {
    public List<double> pesos; //Los pesos para cada entrada
    public List<double> nuevospesos; //Nuevos pesos dados por el algoritmo de "backpropagation"
    public double umbral; //El peso del umbral
    public double nuevoumbral; //Nuevo umbral dado por el algoritmo de "backpropagation"

    //Inicializa los pesos y umbral con un valor al azar
    public Neurona(Random azar, int totalEntradas) {
        pesos = new List<double>();
        nuevospesos = new List<double>();
        for (int cont = 0; cont < totalEntradas; cont++) {
            pesos.Add(azar.NextDouble());
            nuevospesos.Add(0);
        }
        umbral = azar.NextDouble();
        nuevoumbral = 0;
    }

    //Calcula la salida de la neurona dependiendo de las entradas
    public double calculaSalida(List<double> entradas) {
        double valor = 0;
        for (int cont = 0; cont < pesos.Count; cont++) {
            valor += entradas[cont] * pesos[cont];
        }
        valor += umbral;
        return 1 / (1 + Math.Exp(-valor));
    }

    //Reemplaza viejos pesos por nuevos
    public void actualiza(){
        for (int cont = 0; cont < pesos.Count; cont++){
            pesos[cont] = nuevospesos[cont];
        }
        umbral = nuevoumbral;
    }
}
}
```

Varios ejemplos de su ejecución

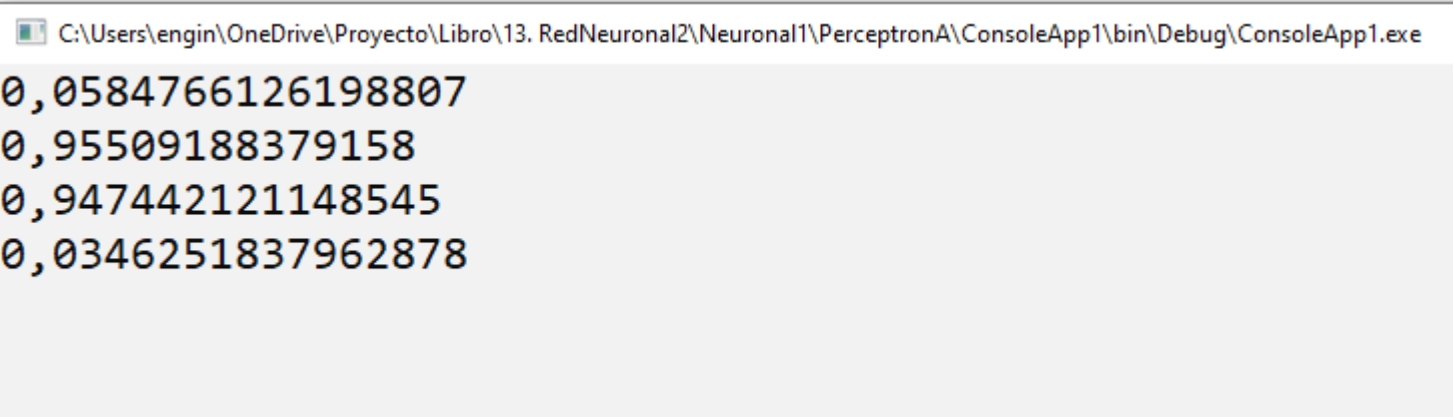


Ilustración 51: Ejecución del programa con el algoritmo de bakcpropagation


```
C:\Users\engin\OneDrive\Proyecto\Libro\13. RedNeuronal2\Neuronal1\PerceptronA\ConsoleApp1\bin\Debug\ConsoleApp1.exe
0,0486682625327801
0,957288852531577
0,95162011860572
0,038575804248038
```

Ilustración 52: Ejecución del programa con el algoritmo de bakcpropagation

El programa debe aprender la tabla XOR

Valor	Valor	Resultado esperado
1	1	0
1	0	1
0	1	1
0	0	0

En los resultados, se muestran las salidas de la última capa (última ejecución)

Valor	Valor	Resultado esperado	Salida del perceptrón
1	1	0	0,0486682625327801
1	0	1	0,957288852531577
0	1	1	0,95162011860572
0	0	0	0,038575804248038

Luego, se debe modificar el programa para que si el resultado está por encima de 0,5 imprima 1 de lo contrario imprima cero. Eso se haría en la clase Perceptrón agregando un nuevo método

```
public void SalidaPerceptron(List<double> entradas, double salidaesperada) {
    for (int cont = 0; cont < entradas.Count; cont++) {
        Console.Write(entradas[cont].ToString() + " | ");
    }
    Console.Write(" Esperada: " + salidaesperada.ToString() + " Calculada: ");
    for (int cont = 0; cont < capas[2].salidas.Count; cont++) {
        if (capas[2].salidas[cont] >= 0.5)
            Console.Write(" 1 | ");
        else
            Console.Write(" 0 | ");
        Console.Write(capas[2].salidas[cont].ToString() + " | ");
    }
    Console.WriteLine(" ");
}
```

Se modifica el programa principal para que muestre el progreso del entrenamiento cada 200 ciclos. Este sería el programa completo:

```
using System;
using System.Collections.Generic;

namespace AplicacionConsola {
    class Program {
        static void Main(string[] args) {
            //Tabla de verdad XOR
            int[,] xorentra = { { 1, 1 }, { 1, 0 }, { 0, 1 }, { 0, 0 } };
            int[] xorsale = { 0, 1, 1, 0 };

            int numEntradas = 2; //Número de entradas
            int capa0 = 4; //Total neuronas en la capa 0
            int capa1 = 4; //Total neuronas en la capa 1
            int capa2 = 1; //Total neuronas en la capa 2
            Perceptron perceptron = new Perceptron(numEntradas, capa0, capa1, capa2);

            //Estas serán las entradas externas al perceptrón
            List<double> entradas = new List<double>();
            entradas.Add(0);
            entradas.Add(0);

            //Estas serán las salidas esperadas externas al perceptrón
            List<double> salidaEsperada = new List<double>();
            salidaEsperada.Add(0);
        }
    }
}
```

```

//Ciclo que entrena la red neuronal
int totalCiclos = 8000; //Ciclos de entrenamiento
for (int ciclo = 1; ciclo <= totalCiclos; ciclo++) {
    //Por cada ciclo, se entrena el perceptrón con toda la tabla de XOR

    //Cada 200 ciclos muestra como progresa el entrenamiento
    if (ciclo % 200 == 0) Console.WriteLine("Ciclo: " + ciclo.ToString());

    for (int conjunto = 0; conjunto < xorsale.Length; conjunto++) {
        //Entradas y salidas esperadas
        entradas[0] = xorentra[conjunto, 0];
        entradas[1] = xorentra[conjunto, 1];
        salidaEsperada[0] = xorsale[conjunto];

        //Primero calcula la salida del perceptrón con esas entradas
        perceptron.calculaSalida(entradas);

        //Luego entrena el perceptrón para ajustar los pesos y umbrales
        perceptron.Entrena(entradas, salidaEsperada);

        //Cada 200 ciclos muestra como progresa el entrenamiento
        if (ciclo % 200 == 0) perceptron.SalidaPerceptron(entradas, salidaEsperada[0]);
    }
}

Console.WriteLine("Finaliza");
Console.ReadKey();
}

class Perceptron {
    public List<Capa> capas;

    //Imprime los datos de las diferentes capas
    public void SalidaPerceptron(List<double> entradas, double salidaesperada) {
        for (int cont = 0; cont < entradas.Count; cont++) {
            Console.Write(entradas[cont].ToString() + " | ");
        }
        Console.Write(" Esperada: " + salidaesperada.ToString() + " Calculada: ");
        for (int cont = 0; cont < capas[2].salidas.Count; cont++) {
            if (capas[2].salidas[cont] >= 0.5)
                Console.Write(" 1 | ");
            else
                Console.Write(" 0 | ");
            Console.Write(capas[2].salidas[cont].ToString() + " | ");
        }
        Console.WriteLine(" ");
    }

    //Crea las diversas capas
    public Perceptron(int numEntrada, int capa0, int capa1, int capa2) {
        Random azar = new Random();
        capas = new List<Capa>();
        capas.Add(new Capa(azar, capa0, numEntrada)); //Crea la capa 0
        capas.Add(new Capa(azar, capa1, capa0)); //Crea la capa 1
        capas.Add(new Capa(azar, capa2, capa1)); //Crea la capa 2
    }

    //Dada las entradas al perceptrón, se calcula la salida de cada capa.
    //Con eso se sabrá que salidas se obtienen con los pesos y umbrales actuales.
    //Esas salidas son requeridas para el algoritmo de entrenamiento.
    public void calculaSalida(List<double> entradas) {
        capas[0].CalculaCapa(entradas);
        capas[1].CalculaCapa(capas[0].salidas);
        capas[2].CalculaCapa(capas[1].salidas);
    }

    //Con las salidas previamente calculadas con unas determinadas entradas
    //se ejecuta el algoritmo de entrenamiento "Backpropagation"
    public void Entrena(List<double> entradas, List<double> salidaEsperada) {
        int capa0 = capas[0].neuronas.Count;
        int capa1 = capas[1].neuronas.Count;
        int capa2 = capas[2].neuronas.Count;

        //Factor de aprendizaje
        double alpha = 0.4;

        //Procesa pesos capa 2
        for (int j = 0; j < capa1; j++) //Va de neurona en neurona de la capa 1
            for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
                double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
                double Si = salidaEsperada[i]; //Salida esperada
                double a1j = capas[1].salidas[j]; //Salida de la capa 1
                double dE2 = a1j * (Yi - Si) * Yi * (1 - Yi); //La fórmula del error
                capas[2].neuronas[i].nuevospesos[j] = capas[2].neuronas[i].pesos[j] - alpha * dE2; //Ajusta el nuevo peso
            }

        //Procesa pesos capa 1
        for (int j = 0; j < capa0; j++) //Va de neurona en neurona de la capa 0
            for (int k = 0; k < capa1; k++) { //Va de neurona en neurona de la capa 1
                double acum = 0;
                for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                    double Yi = capas[2].salidas[i]; //Salida de la capa 2
                    double Si = salidaEsperada[i]; //Salida esperada
                    double W2ki = capas[2].neuronas[i].pesos[k];
                    acum += W2ki * (Yi - Si) * Yi * (1 - Yi); //Sumatoria
                }
                double a0j = capas[0].salidas[j];
            }
    }
}

```

```

        double a1k = capas[1].salidas[k];
        double dE1 = a0j * a1k * (1 - a1k) * acum;
        capas[1].neuronas[k].nuevospesos[j] = capas[1].neuronas[k].pesos[j] - alpha * dE1;
    }

    //Procesa pesos capa 0
    for (int j = 0; j < entradas.Count; j++) //Va de entrada en entrada
        for (int k = 0; k < capa0; k++) { //Va de neurona en neurona de la capa 0
            double acumular = 0;
            for (int p = 0; p < capa1; p++) { //Va de neurona en neurona de la capa 1
                double acum = 0;
                for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                    double Yi = capas[2].salidas[i];
                    double Si = salidaEsperada[i]; //Salida esperada
                    double W2pi = capas[2].neuronas[i].pesos[p];
                    acum += W2pi * (Yi - Si) * Yi * (1 - Yi); //Sumatoria interna
                }
                double W1kp = capas[1].neuronas[p].pesos[k];
                double a1p = capas[1].salidas[p];
                acumular += W1kp * a1p * (1 - a1p) * acum; //Sumatoria externa
            }
            double xj = entradas[j];
            double a0k = capas[0].salidas[k];
            double dE0 = xj * a0k * (1 - a0k) * acumular;
            double W0jk = capas[0].neuronas[k].pesos[j];
            capas[0].neuronas[k].nuevospesos[j] = W0jk - alpha * dE0;
        }

    //Procesa umbrales capa 2
    for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
        double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
        double Si = salidaEsperada[i]; //Salida esperada
        double dE2 = (Yi - Si) * Yi * (1 - Yi);
        capas[2].neuronas[i].nuevoumbrales = capas[2].neuronas[i].umbral - alpha * dE2;
    }

    //Procesa umbrales capa 1
    for (int k = 0; k < capa1; k++) { //Va de neurona en neurona de la capa 1
        double acum = 0;
        for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
            double Yi = capas[2].salidas[i]; //Salida de la capa 2
            double Si = salidaEsperada[i];
            double W2ki = capas[2].neuronas[i].pesos[k];
            acum += W2ki * (Yi - Si) * Yi * (1 - Yi);
        }
        double a1k = capas[1].salidas[k];
        double dE1 = a1k * (1 - a1k) * acum;
        capas[1].neuronas[k].nuevoumbrales = capas[1].neuronas[k].umbral - alpha * dE1;
    }

    //Procesa umbrales capa 0
    for (int k = 0; k < capa0; k++) { //Va de neurona en neurona de la capa 0
        double acumular = 0;
        for (int p = 0; p < capa1; p++) { //Va de neurona en neurona de la capa 1
            double acum = 0;
            for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                double Yi = capas[2].salidas[i];
                double Si = salidaEsperada[i];
                double W2pi = capas[2].neuronas[i].pesos[p];
                acum += W2pi * (Yi - Si) * Yi * (1 - Yi);
            }
            double W1kp = capas[1].neuronas[p].pesos[k];
            double a1p = capas[1].salidas[p];
            acumular += W1kp * a1p * (1 - a1p) * acum;
        }
        double a0k = capas[0].salidas[k];
        double dE0 = a0k * (1 - a0k) * acumular;
        capas[0].neuronas[k].nuevoumbrales = capas[0].neuronas[k].umbral - alpha * dE0;
    }

    //Actualiza los pesos
    capas[0].actualiza();
    capas[1].actualiza();
    capas[2].actualiza();
}

class Capa {
    public List<Neurona> neuronas; //Las neuronas que tendrá la capa
    public List<double> salidas; //Almacena las salidas de cada neurona

    public Capa(Random azar, int totalNeuronas, int totalEntradas) {
        neuronas = new List<Neurona>();
        salidas = new List<double>();
        //Genera las neuronas
        for (int cont = 0; cont < totalNeuronas; cont++) {
            neuronas.Add(new Neurona(azar, totalEntradas));
            salidas.Add(0);
        }
    }

    //Calcula las salidas de cada neurona de la capa
    public void CalculaCapa(List<double> entradas) {
        for (int cont = 0; cont < neuronas.Count; cont++) {
            salidas[cont] = neuronas[cont].calculaSalida(entradas);
        }
    }
}

```

```

        //Actualiza los pesos y umbrales de las neuronas
        public void actualiza(){
            for (int cont = 0; cont < neuronas.Count; cont++) {
                neuronas[cont].actualiza();
            }
        }
    }

    class Neuron {
        public List<double> pesos; //Los pesos para cada entrada
        public List<double> nuevospesos; //Nuevos pesos dados por el algoritmo de "backpropagation"
        public double umbral; //El peso del umbral
        public double nuevoumbral; //Nuevo umbral dado por el algoritmo de "backpropagation"

        //Inicializa los pesos y umbral con un valor al azar
        public Neuron(Random azar, int totalEntradas) {
            pesos = new List<double>();
            nuevospesos = new List<double>();
            for (int cont = 0; cont < totalEntradas; cont++) {
                pesos.Add(azar.NextDouble());
                nuevospesos.Add(0);
            }
            umbral = azar.NextDouble();
            nuevoumbral = 0;
        }

        //Calcula la salida de la neurona dependiendo de las entradas
        public double calculaSalida(List<double> entradas) {
            double valor = 0;
            for (int cont = 0; cont < pesos.Count; cont++) {
                valor += entradas[cont] * pesos[cont];
            }
            valor += umbral;
            return 1 / (1 + Math.Exp(-valor));
        }

        //Reemplaza viejos pesos por nuevos
        public void actualiza(){
            for (int cont = 0; cont < pesos.Count; cont++){
                pesos[cont] = nuevospesos[cont];
            }
            umbral = nuevoumbral;
        }
    }
}

```

Esta sería una ejecución de ejemplo:

```

C:\Users\engin\OneDrive\Proyecto\Libro\13. RedNeuronal2\Neuronal1\PerceptronA\ConsoleApp1\bin\Debug\ConsoleApp1.exe
1 | 0 | Esperada: 1 Calculada: 1 | 0,985928521767663 |
0 | 1 | Esperada: 1 Calculada: 1 | 0,985620491017832 |
0 | 0 | Esperada: 0 Calculada: 0 | 0,0165018966914197 |
Ciclo: 7200
1 | 1 | Esperada: 0 Calculada: 0 | 0,0161206031347767 |
1 | 0 | Esperada: 1 Calculada: 1 | 0,986287040427381 |
0 | 1 | Esperada: 1 Calculada: 1 | 0,985988250609527 |
0 | 0 | Esperada: 0 Calculada: 0 | 0,0161010795361778 |
Ciclo: 7400
1 | 1 | Esperada: 0 Calculada: 0 | 0,0157199460190425 |
1 | 0 | Esperada: 1 Calculada: 1 | 0,986621411746609 |
0 | 1 | Esperada: 1 Calculada: 1 | 0,986331204571934 |
0 | 0 | Esperada: 0 Calculada: 0 | 0,015726577325571 |
Ciclo: 7600
1 | 1 | Esperada: 0 Calculada: 0 | 0,0153454657741085 |
1 | 0 | Esperada: 1 Calculada: 1 | 0,986934172777499 |
0 | 1 | Esperada: 1 Calculada: 1 | 0,986651961540037 |
0 | 0 | Esperada: 0 Calculada: 0 | 0,0153756770841316 |
Ciclo: 7800
1 | 1 | Esperada: 0 Calculada: 0 | 0,01499449639415 |
1 | 0 | Esperada: 1 Calculada: 1 | 0,987227506377492 |
0 | 1 | Esperada: 1 Calculada: 1 | 0,986952765704332 |
0 | 0 | Esperada: 0 Calculada: 0 | 0,0150460387630933 |
Ciclo: 8000
1 | 1 | Esperada: 0 Calculada: 0 | 0,0146647327483302 |
1 | 0 | Esperada: 1 Calculada: 1 | 0,98750330194041 |
0 | 1 | Esperada: 1 Calculada: 1 | 0,987235559335537 |
0 | 0 | Esperada: 0 Calculada: 0 | 0,014735632146513 |
Finaliza

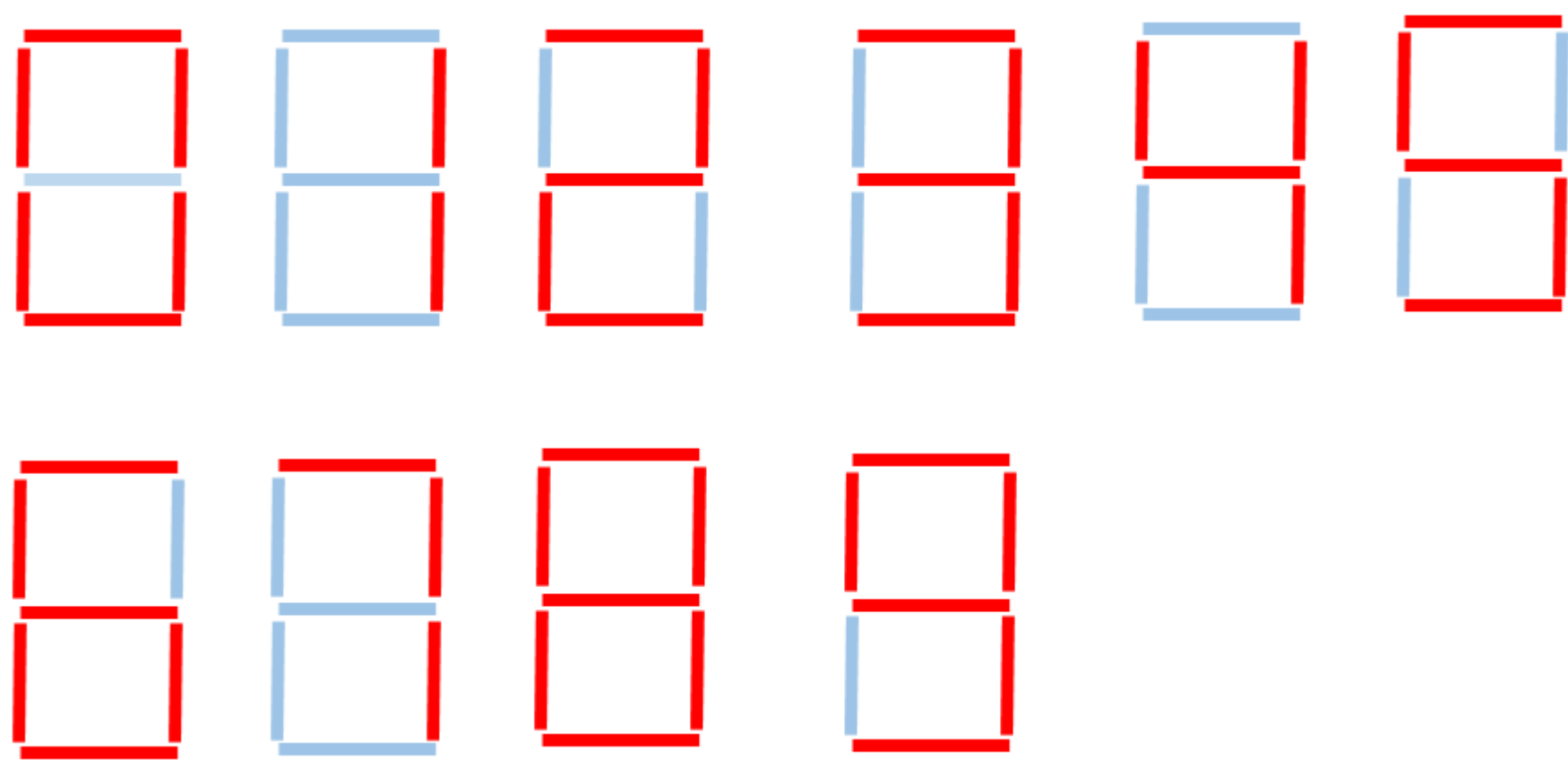
```

Ilustración 53: Aprendiendo la tabla XOR

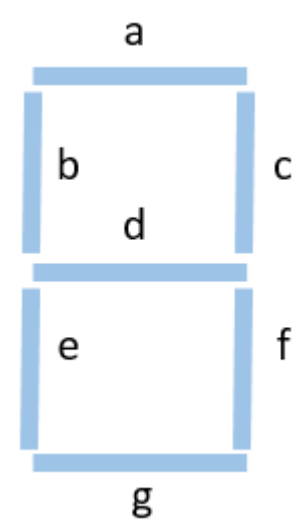
Se observa que la red ha quedado entrenada. La salida esperada coincide con la salida calculada.

Reconocimiento de números de un reloj digital

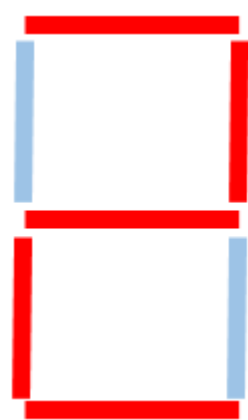
En la imagen, los números del 0 al 9 contruidos usando las barras verticales y horizontales. Típicos de un reloj digital.



Se quiere construir una red neuronal tipo perceptrón multicapa que dado ese número al estilo reloj digital se pueda deducir el número como tal. Para iniciar se pone un identificador a cada barra






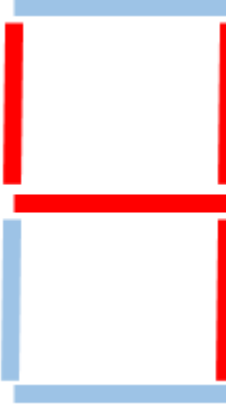
Luego se le da un valor de “1” a la barra que queda en rojo al construir el número y “0” a la barra que queda en azul claro. Por ejemplo:



Los valores de a,b,c,d,e,f,g serían: 1,0,1,1,1,0,1

Como se debe deducir que es un 2, este valor se convierte a binario 10_2 o 0,0,1,0 (para convertirlo en salida del perceptrón, como el máximo valor del reloj digital es 9 y este es 1,0,0,1 entonces el número de salidas es 4)

La tabla de entradas y salidas esperadas es:

Imagen	Valor de entrada	Valor de salida esperado
	1,1,1,0,1,1,1	0,0,0,0
	0,0,1,0,0,1,0	0,0,0,1
	1,0,1,1,1,0,1	0,0,1,0
	1,0,1,1,0,1,1	0,0,1,1
	0,1,1,1,0,1,0	0,1,0,0
	1,1,0,1,0,1,1	0,1,0,1

	1,1,0,1,1,1,1	0,1,1,0
	1,0,1,0,0,1,0	0,1,1,1
	1,1,1,1,1,1,1	1,0,0,0
	1,1,1,1,0,1,1	1,0,0,1

El código (se encuentra en la carpeta **RedNeuronal2** en GitHub) es el siguiente:

```
using System;
using System.Collections.Generic;

namespace AplicacionConsola {
    class Program {
        static void Main(string[] args) {
            //El número "dibujado" en el reloj digital
            int[,] relojEntra = {
                { 1, 1, 1, 0, 1, 1, 1 },
                { 0, 0, 1, 0, 0, 1, 0 },
                { 1, 0, 1, 1, 1, 0, 1 },
                { 1, 0, 1, 1, 0, 1, 1 },
                { 0, 1, 1, 1, 0, 1, 0 },
                { 1, 1, 0, 1, 0, 1, 1 },
                { 1, 1, 0, 1, 1, 1, 1 },
                { 1, 0, 1, 0, 0, 1, 0 },
                { 1, 1, 1, 1, 1, 1, 1 },
                { 1, 1, 1, 1, 0, 1, 1 }
            };

            //El número en binario
            int[,] numeroEsperado = {
                { 0, 0, 0, 0 },
                { 0, 0, 0, 1 },
                { 0, 0, 1, 0 },
                { 0, 0, 1, 1 },
                { 0, 1, 0, 0 },
                { 0, 1, 0, 1 },
                { 0, 1, 1, 0 },
                { 0, 1, 1, 1 },
                { 1, 0, 0, 0 },
                { 1, 0, 0, 1 }
            };

            int numEntradas = 7; //Número de entradas
            int capa0 = 5; //Total neuronas en la capa 0
            int capa1 = 5; //Total neuronas en la capa 1
            int capa2 = 4; //Total neuronas en la capa 2
            Perceptron perceptron = new Perceptron(numEntradas, capa0, capa1, capa2);

            //Estas serán las entradas externas al perceptrón
            List<double> entradas = new List<double>();
            entradas.Add(0);
            entradas.Add(0);
            entradas.Add(0);
            entradas.Add(0);
            entradas.Add(0);
            entradas.Add(0);
            entradas.Add(0);

            //Estas serán las salidas esperadas externas al perceptrón
            List<double> salidaEsperada = new List<double>();
            salidaEsperada.Add(0);
            salidaEsperada.Add(0);
            salidaEsperada.Add(0);
            salidaEsperada.Add(0);

            //Ciclo que entrena la red neuronal
            int totalCiclos = 8000; //Ciclos de entrenamiento
            for (int ciclo = 1; ciclo <= totalCiclos; ciclo++) {
                //Por cada ciclo, se entrena el perceptrón con todos los números

                //Cada 200 ciclos muestra como progresa el entrenamiento
                if (ciclo % 200 == 0) Console.WriteLine("Ciclo: " + ciclo.ToString());

                for (int conjunto = 0; conjunto < relojEntra.GetLength(0); conjunto++) {
                    //Entradas y salidas esperadas
                    entradas[0] = relojEntra[conjunto, 0];
                    entradas[1] = relojEntra[conjunto, 1];
                    entradas[2] = relojEntra[conjunto, 2];
                    entradas[3] = relojEntra[conjunto, 3];
                    entradas[4] = relojEntra[conjunto, 4];
                    entradas[5] = relojEntra[conjunto, 5];
                    entradas[6] = relojEntra[conjunto, 6];

                    salidaEsperada[0] = numeroEsperado[conjunto, 0];
                    salidaEsperada[1] = numeroEsperado[conjunto, 1];
                    salidaEsperada[2] = numeroEsperado[conjunto, 2];
                    salidaEsperada[3] = numeroEsperado[conjunto, 3];

                    //Primero calcula la salida del perceptrón con esas entradas
                    perceptron.calculaSalida(entradas);

                    //Luego entrena el perceptrón para ajustar los pesos y umbrales
                    perceptron.Entrena(entradas, salidaEsperada);

                    //Cada 200 ciclos muestra como progresa el entrenamiento
                    if (ciclo % 200 == 0) perceptron.SalidaPerceptron(entradas, salidaEsperada);
                }
            }

            Console.WriteLine("Finaliza");
            Console.ReadKey();
        }
    }

    class Perceptron {
```



```

public List<Capa> capas;

//Imprime los datos de las diferentes capas
public void SalidaPerceptron(List<double> entradas, List<double> salidaesperada) {
    for (int cont = 0; cont < entradas.Count; cont++) {
        Console.Write(entradas[cont].ToString() + " ");
    }
    Console.Write(" Esperada: ");
    for (int cont = 0; cont < salidaesperada.Count; cont++) {
        Console.Write(salidaesperada[cont].ToString() + " ");
    }
    Console.Write(" Calculada: ");
    for (int cont = 0; cont < capas[2].salidas.Count; cont++) {
        if (capas[2].salidas[cont] >= 0.5)
            Console.Write(" 1 ");
        else
            Console.Write(" 0 ");
        //Console.Write(capas[2].salidas[cont].ToString() + " | ");
    }
    Console.WriteLine(" ");
}

//Crea las diversas capas
public Perceptron(int numEntrada, int capa0, int capa1, int capa2) {
    Random azar = new Random();
    capas = new List<Capa>();
    capas.Add(new Capa(azar, capa0, numEntrada)); //Crea la capa 0
    capas.Add(new Capa(azar, capa1, capa0)); //Crea la capa 1
    capas.Add(new Capa(azar, capa2, capa1)); //Crea la capa 2
}

//Dada las entradas al perceptrón, se calcula la salida de cada capa.
//Con eso se sabrá que salidas se obtienen con los pesos y umbrales actuales.
//Esas salidas son requeridas para el algoritmo de entrenamiento.
public void calculaSalida(List<double> entradas) {
    capas[0].CalculaCapa(entradas);
    capas[1].CalculaCapa(capas[0].salidas);
    capas[2].CalculaCapa(capas[1].salidas);
}

//Con las salidas previamente calculadas con unas determinadas entradas
//se ejecuta el algoritmo de entrenamiento "Backpropagation"
public void Entrena(List<double> entradas, List<double> salidaEsperada) {
    int capa0 = capas[0].neuronas.Count;
    int capa1 = capas[1].neuronas.Count;
    int capa2 = capas[2].neuronas.Count;

    //Factor de aprendizaje
    double alpha = 0.4;

    //Procesa pesos capa 2
    for (int j = 0; j < capa1; j++) //Va de neurona en neurona de la capa 1
        for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
            double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
            double Si = salidaEsperada[i]; //Salida esperada
            double alj = capas[1].salidas[j]; //Salida de la capa 1
            double dE2 = alj * (Yi - Si) * Yi * (1 - Yi); //La fórmula del error
            capas[2].neuronas[i].nuevospesos[j] = capas[2].neuronas[i].pesos[j] - alpha * dE2; //Ajusta el nuevo peso
        }

    //Procesa pesos capa 1
    for (int j = 0; j < capa0; j++) //Va de neurona en neurona de la capa 0
        for (int k = 0; k < capa1; k++) { //Va de neurona en neurona de la capa 1
            double acum = 0;
            for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                double Yi = capas[2].salidas[i]; //Salida de la capa 2
                double Si = salidaEsperada[i]; //Salida esperada
                double W2ki = capas[2].neuronas[i].pesos[k];
                acum += W2ki * (Yi - Si) * Yi * (1 - Yi); //Sumatoria
            }
            double a0j = capas[0].salidas[j];
            double alk = capas[1].salidas[k];
            double dE1 = a0j * alk * (1 - alk) * acum;
            capas[1].neuronas[k].nuevospesos[j] = capas[1].neuronas[k].pesos[j] - alpha * dE1;
        }

    //Procesa pesos capa 0
    for (int j = 0; j < entradas.Count; j++) //Va de entrada en entrada
        for (int k = 0; k < capa0; k++) { //Va de neurona en neurona de la capa 0
            double acumular = 0;
            for (int p = 0; p < capa1; p++) { //Va de neurona en neurona de la capa 1
                double acum = 0;
                for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                    double Yi = capas[2].salidas[i];
                    double Si = salidaEsperada[i]; //Salida esperada
                    double W2pi = capas[2].neuronas[i].pesos[p];
                    acum += W2pi * (Yi - Si) * Yi * (1 - Yi); //Sumatoria interna
                }
                double W1kp = capas[1].neuronas[p].pesos[k];
                double alp = capas[1].salidas[p];
                acumular += W1kp * alp * (1 - alp) * acum; //Sumatoria externa
            }
            double xj = entradas[j];
            double a0k = capas[0].salidas[k];
            double dE0 = xj * a0k * (1 - a0k) * acumular;
            double W0jk = capas[0].neuronas[k].pesos[j];

```

```

        capas[0].neuronas[k].nuevospesos[j] = W0jk - alpha * dE0;
    }

    //Procesa umbrales capa 2
    for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
        double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
        double Si = salidaEsperada[i]; //Salida esperada
        double dE2 = (Yi - Si) * Yi * (1 - Yi);
        capas[2].neuronas[i].nuevoumbrales = capas[2].neuronas[i].umbrales - alpha * dE2;
    }

    //Procesa umbrales capa 1
    for (int k = 0; k < capa1; k++) { //Va de neurona en neurona de la capa 1
        double acum = 0;
        for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
            double Yi = capas[2].salidas[i]; //Salida de la capa 2
            double Si = salidaEsperada[i];
            double W2ki = capas[2].neuronas[i].pesos[k];
            acum += W2ki * (Yi - Si) * Yi * (1 - Yi);
        }
        double a1k = capas[1].salidas[k];
        double dE1 = a1k * (1 - a1k) * acum;
        capas[1].neuronas[k].nuevoumbrales = capas[1].neuronas[k].umbrales - alpha * dE1;
    }

    //Procesa umbrales capa 0
    for (int k = 0; k < capa0; k++) { //Va de neurona en neurona de la capa 0
        double acumular = 0;
        for (int p = 0; p < capa1; p++) { //Va de neurona en neurona de la capa 1
            double acum = 0;
            for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                double Yi = capas[2].salidas[i];
                double Si = salidaEsperada[i];
                double W2pi = capas[2].neuronas[i].pesos[p];
                acum += W2pi * (Yi - Si) * Yi * (1 - Yi);
            }
            double W1kp = capas[1].neuronas[p].pesos[k];
            double alp = capas[1].salidas[p];
            acumular += W1kp * alp * (1 - alp) * acum;
        }
        double a0k = capas[0].salidas[k];
        double dE0 = a0k * (1 - a0k) * acumular;
        capas[0].neuronas[k].nuevoumbrales = capas[0].neuronas[k].umbrales - alpha * dE0;
    }

    //Actualiza los pesos
    capas[0].actualiza();
    capas[1].actualiza();
    capas[2].actualiza();
}

}

class Capa {
    public List<Neurona> neuronas; //Las neuronas que tendrá la capa
    public List<double> salidas; //Almacena las salidas de cada neurona

    public Capa(Random azar, int totalNeuronas, int totalEntradas) {
        neuronas = new List<Neurona>();
        salidas = new List<double>();
        //Genera las neuronas
        for (int cont = 0; cont < totalNeuronas; cont++) {
            neuronas.Add(new Neurona(azar, totalEntradas));
            salidas.Add(0);
        }
    }

    //Calcula las salidas de cada neurona de la capa
    public void CalculaCapa(List<double> entradas) {
        for (int cont = 0; cont < neuronas.Count; cont++) {
            salidas[cont] = neuronas[cont].calculaSalida(entradas);
        }
    }

    //Actualiza los pesos y umbrales de las neuronas
    public void actualiza(){
        for (int cont = 0; cont < neuronas.Count; cont++) {
            neuronas[cont].actualiza();
        }
    }
}

class Neurona {
    public List<double> pesos; //Los pesos para cada entrada
    public List<double> nuevospesos; //Nuevos pesos dados por el algoritmo de "backpropagation"
    public double umbral; //El peso del umbral
    public double nuevoumbrales; //Nuevo umbral dado por el algoritmo de "backpropagation"

    //Inicializa los pesos y umbral con un valor al azar
    public Neurona(Random azar, int totalEntradas) {
        pesos = new List<double>();
        nuevospesos = new List<double>();
        for (int cont = 0; cont < totalEntradas; cont++) {
            pesos.Add(azar.NextDouble());
            nuevospesos.Add(0);
        }
    }
}

```

```
        umbral = azar.NextDouble();
        nuevoumbral = 0;
    }

    //Calcula la salida de la neurona dependiendo de las entradas
    public double calculaSalida(List<double> entradas) {
        double valor = 0;
        for (int cont = 0; cont < pesos.Count; cont++) {
            valor += entradas[cont] * pesos[cont];
        }
        valor += umbral;
        return 1 / (1 + Math.Exp(-valor));
    }

    //Reemplaza viejos pesos por nuevos
    public void actualiza(){
        for (int cont = 0; cont < pesos.Count; cont++){
            pesos[cont] = nuevospesos[cont];
        }
        umbral = nuevoumbral;
    }

}

}
```

Esta sería su ejecución

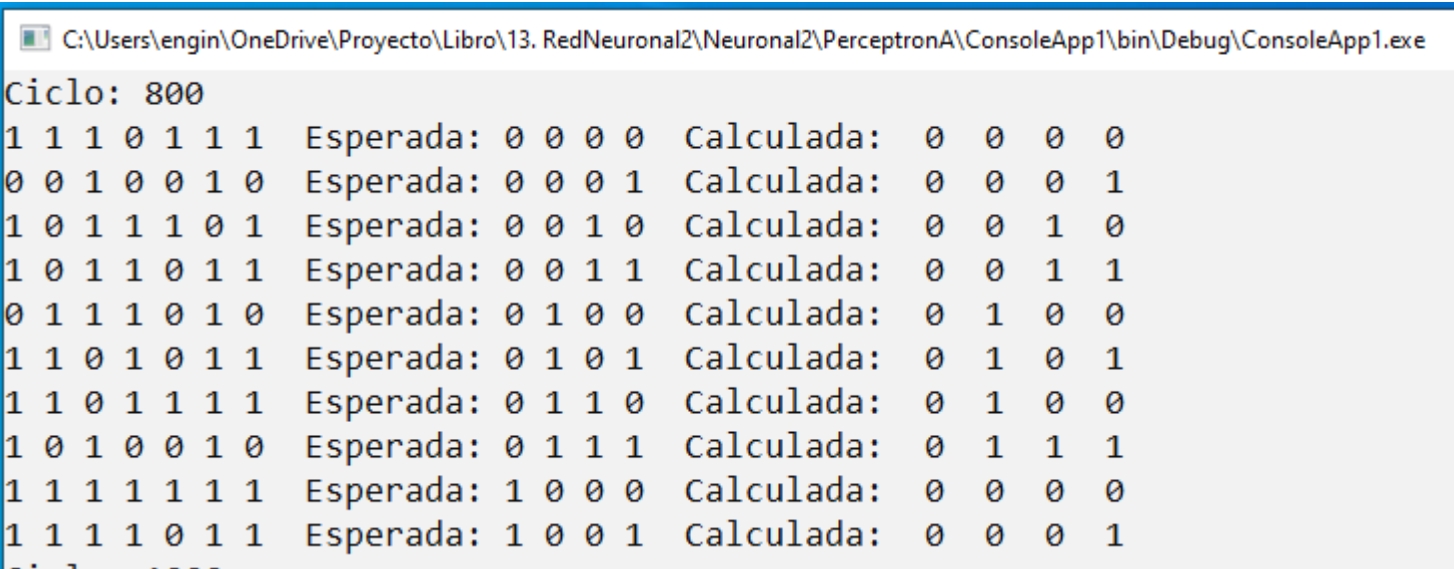


Ilustración 54: Inicio de aprendizaje de lectura de números de un reloj digital

En la iteración 800, comienza a notarse que la red neuronal está aprendiendo el patrón. Requiere más iteraciones

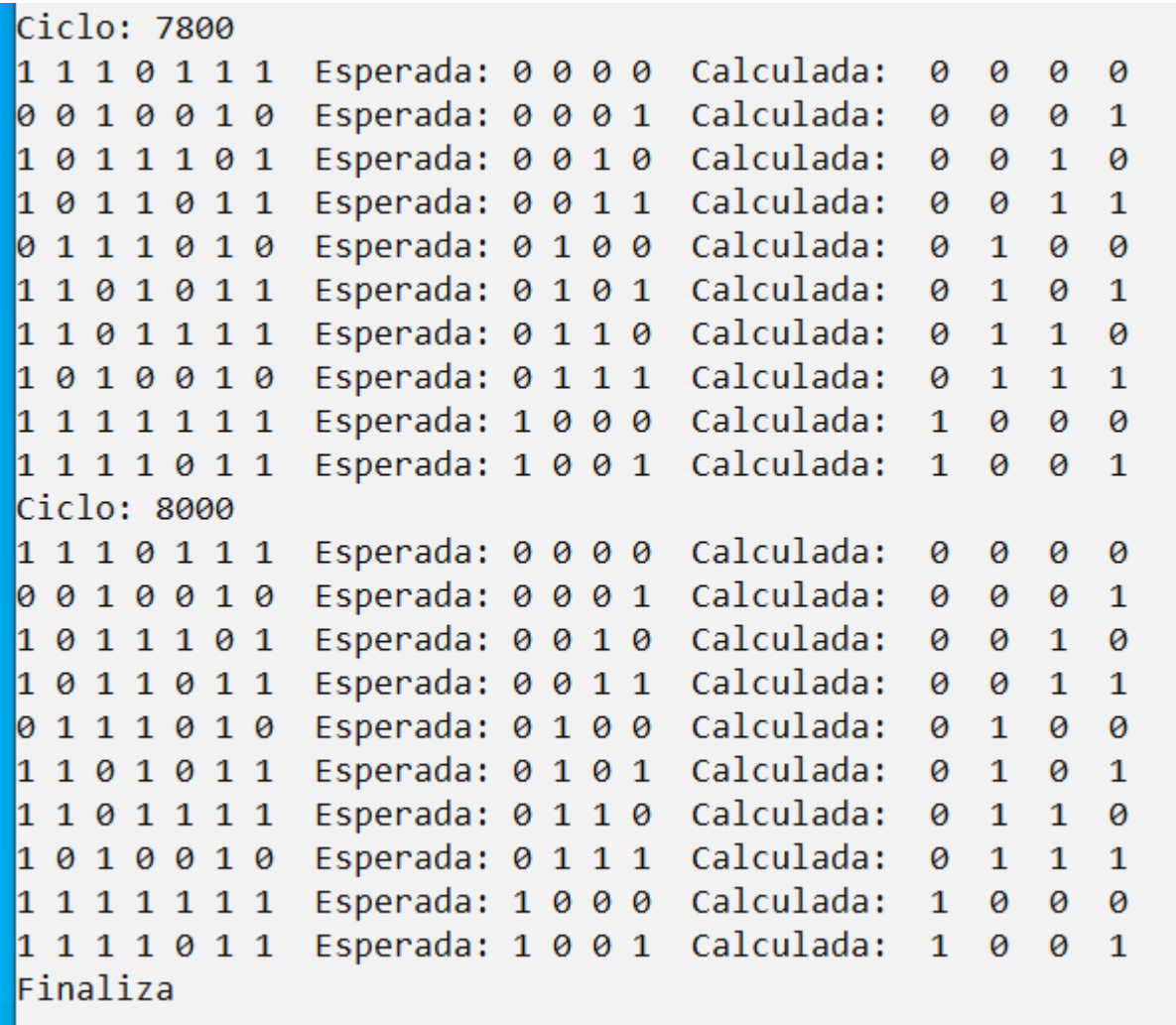


Ilustración 55: Final de aprendizaje de lectura de números de un reloj digital

En la iteración 7800 ya se observa que la red está entrenada completamente para reconocer los números. Solo faltaría implementar que se detenga el entrenamiento cuando la red neuronal ofrezca las salidas esperadas y así evitar iteraciones de más.

En los dos ejemplos anteriores, los valores de las entradas externas del perceptrón fueron 0 o 1. Pero no está limitado a eso, las entradas externas pueden tener valores entre 0 y 1 (incluyendo el 0 y el 1) por ejemplo: 0.7321, 0.21896, 0.9173418

El problema que se plantea es dado el comportamiento de un evento en el tiempo, ¿podrá la red neuronal deducir el patrón?

Ejemplo: Se tiene esta tabla

X	Y
0	0
5	0.43577871
10	1.73648178
15	3.88228568
20	6.84040287
25	10.5654565
30	15
35	20.0751753
40	25.7115044
45	31.8198052
50	38.3022222
55	45.0533624
60	51.9615242
65	58.9100062
70	65.7784835
75	72.444437
80	78.7846202
85	84.6765493
90	90
95	94.6384963
100	98.4807753

X es la variable independiente y Y es la variable dependiente, en otras palabras $y=f(x)$. El problema es que no se sabe $f(\)$, sólo están los datos (por motivos prácticos se muestra la tabla con X llegando hasta 100, realmente llega hasta 1800). Esta sería la gráfica.

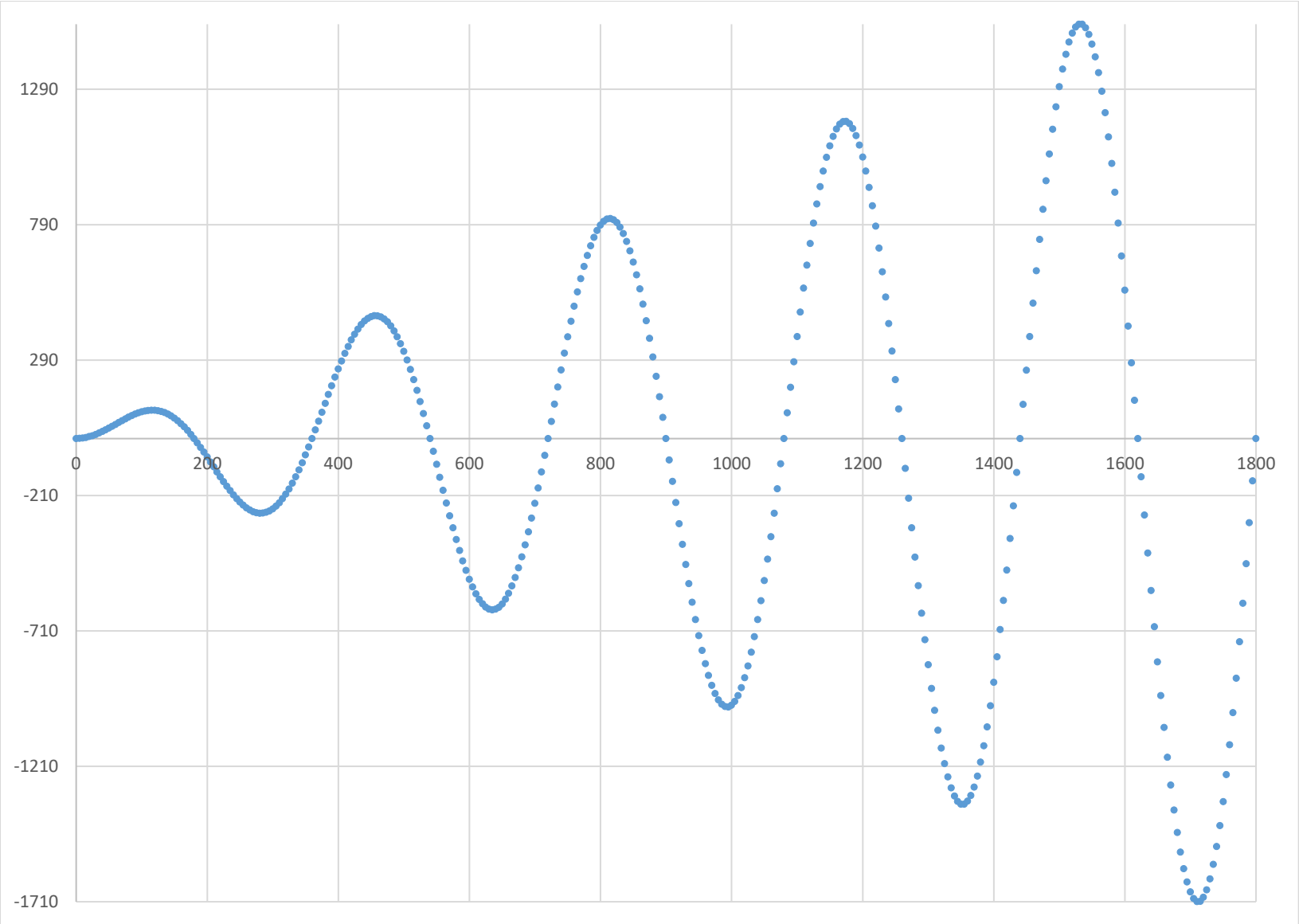


Ilustración 56: Gráfico generado por los puntos

Uniendo los puntos, se obtendría

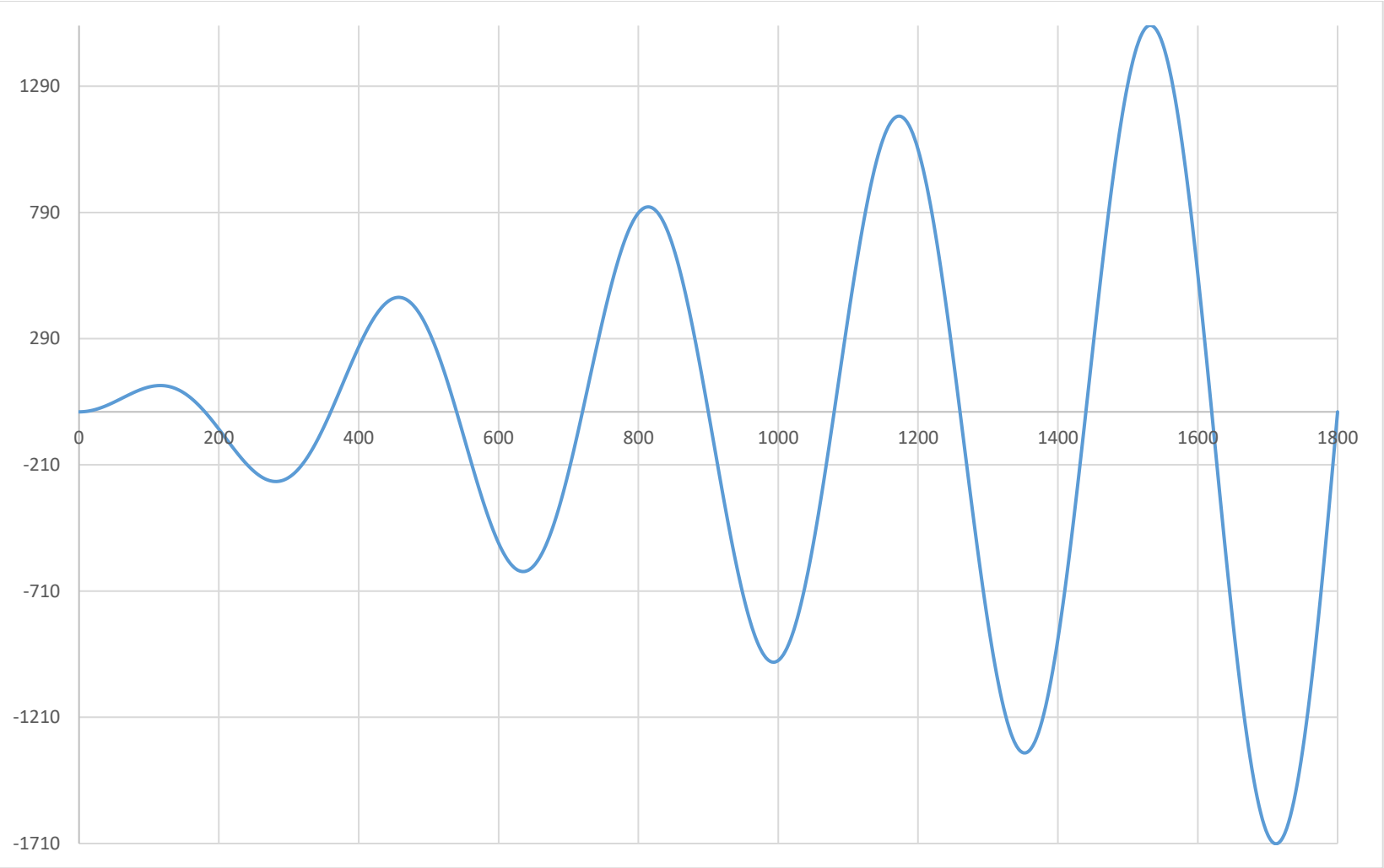


Ilustración 57: Gráfico uniendo los puntos

Los datos están en un archivo plano

```
C:\Users\engin\OneDrive\Proyecto\Libro\13. RedNeuronal2\Neuronal3\PerceptronA\ConsoleApp...
Archivo  Editar  Buscar  Vista  Codificación  Lenguaje  Configuración  Herramientas  Macro  Ejecutar  Plugins
Ventana  ?
[Icons]
datos.tendencia x Recorte de ventana
1  0→;→0
2  5→;→0,435778714
3  10→;→1,736481777
4  15→;→3,882285677
5  20→;→6,840402867
6  25→;→10,56545654
7  30→;→15
8  35→;→20,07517527
9  40→;→25,71150439
10 45→;→31,81980515
11 50→;→38,30222216
12 55→;→45,05336244
13 60→;→51,96152423
14 65→;→58,91000616
15 70→;→65,77848346
16 75→;→72,44443697
length : 6.783  lines : 361  Ln : 1  Col : 1  Sel : 0|0  Windows (CR LF)  UTF-8  INS
```

Ilustración 58: Archivo plano que almacena los valores de los puntos X, Y

Se hace entonces una normalización usando la siguiente fórmula




$$X_{normalizado} = \frac{X_{original} - MinimoX}{MaximoX - MinimoX}$$

$$Y_{normalizado} = \frac{Y_{original} - MinimoY}{MaximoY - MinimoY}$$

357	1775	-750,147415
358	1780	-608,795855
359	1785	-461,991996
360	1790	-310,830238
361	1795	-156,444558
362	1800	-2,2053E-12
363		
364	Minimo X	Minimo Y
365	0	-1710
366		
367	Maximo X	Maximo Y
368	1800	1530
369		
370		

355	1765	-1012,36241
356	1770	-885
357	1775	-750,147415
358	1780	-608,795855
359	1785	-461,991996
360	1790	-310,830238
361	1795	-156,444558
362	1800	-2,2053E-12
363		
364	Minimo X	Minimo Y
365	=MIN(A2:A362)	
366		
367	Maximo X	Maximo Y
368	1800	1530
369		

354	1760	-1131,30619
355	1765	-1012,36241
356	1770	-885
357	1775	-750,147415
358	1780	-608,795855
359	1785	-461,991996
360	1790	-310,830238
361	1795	-156,444558
362	1800	-2,2053E-12
363		
364	Minimo X	Minimo Y
365	0	-1710
366		
367	Maximo X	Maximo Y
368	1800	=MAX(B2:B362)
369		

C2	:	  	=(A2-\$A\$365)/(\$A\$368-\$A\$365)	
	A	B	C	D
1	Xreal	Yreal	Xnormalizado	Ynormalizado
2	0	0	0	0,527777778
3	5	0,4357787	0,002777778	0,527912277
4	10	1,7364818	0,005555556	0,528313729
5	15	3,8822857	0,008333333	0,528976014
6	20	6,8404029	0,011111111	0,529889013
7	25	10,565457	0,013888889	0,531038721

Se realiza la gráfica con los valores normalizados (datos entre 0 y 1 tanto en X como en Y)

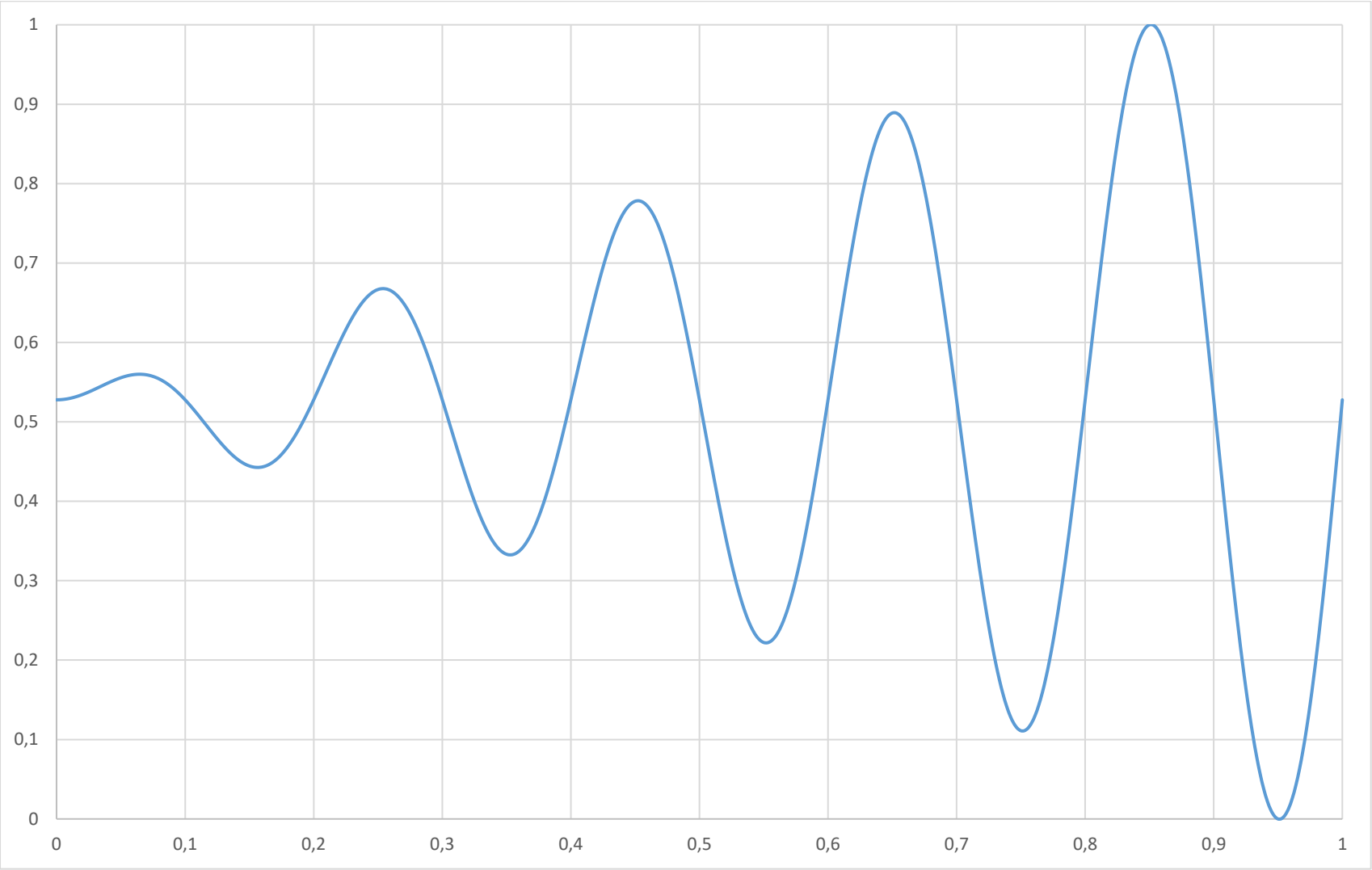


Ilustración 59: Gráfico al normalizar los datos

Con esos datos normalizados, se alimenta la red neuronal para entrenarla. Una vez termine el entrenamiento se procede a “desnormalizar” los resultados.

A continuación, el código completo (en la carpeta **RedNeuronal3** en GitHub):

```
using System;
using System.Collections.Generic;

namespace AplicacionConsola {
    class Program {
        static void Main(string[] args) {
            //Lee los datos de un archivo plano
            int MaximosRegistros = 2000;
            double[] entrada = new double[MaximosRegistros + 1];
            double[] salidas = new double[MaximosRegistros + 1];
            const string urlArchivo = "datos.tendencia";
            int ConjuntoEntradas = LeeDatosArchivo(urlArchivo, entrada, salidas);

            //Normaliza los valores entre 0 y 1 que es lo que requiere el perceptrón
            double minimoX = entrada[0], maximoX = entrada[0];
            double minimoY = salidas[0], maximoY = salidas[0];
            for (int cont = 0; cont < ConjuntoEntradas; cont++) {
                if (entrada[cont] > maximoX) maximoX = entrada[cont];
                if (salidas[cont] > maximoY) maximoY = salidas[cont];
                if (entrada[cont] < minimoX) minimoX = entrada[cont];
                if (salidas[cont] < minimoY) minimoY = salidas[cont];
            }

            for (int cont = 0; cont < ConjuntoEntradas; cont++) {
                entrada[cont] = (entrada[cont] - minimoX) / (maximoX - minimoX);
                salidas[cont] = (salidas[cont] - minimoY) / (maximoY - minimoY);
            }

            int numEntradas = 1; //Número de entradas
            int capa0 = 5; //Total neuronas en la capa 0
            int capa1 = 5; //Total neuronas en la capa 1
            int capa2 = 1; //Total neuronas en la capa 2
            Perceptron perceptron = new Perceptron(numEntradas, capa0, capa1, capa2);

            //Estas serán las entradas externas al perceptrón
            List<double> entradas = new List<double>();
            entradas.Add(0);

            //Estas serán las salidas esperadas externas al perceptrón
            List<double> salidaEsperada = new List<double>();
            salidaEsperada.Add(0);

            //Ciclo que entrena la red neuronal
            int totalCiclos = 8000; //Ciclos de entrenamiento
            for (int ciclo = 1; ciclo <= totalCiclos; ciclo++) {
                //Por cada ciclo, se entrena el perceptrón con todos los valores

                for (int conjunto = 0; conjunto < ConjuntoEntradas; conjunto++) {
                    //Entradas y salidas esperadas
                    entradas[0] = entrada[conjunto];
                    salidaEsperada[0] = salidas[conjunto];

                    //Primero calcula la salida del perceptrón con esas entradas
                    perceptron.calculaSalida(entradas);

                    //Luego entrena el perceptrón para ajustar los pesos y umbrales
                    perceptron.Entrena(entradas, salidaEsperada);
                }
            }

            //Muestra el resultado
            for (int conjunto = 0; conjunto < ConjuntoEntradas; conjunto++) {
                //Entradas y salidas esperadas
                entradas[0] = entrada[conjunto];
                salidaEsperada[0] = salidas[conjunto];

                //Calcula la salida del perceptrón con esas entradas
                perceptron.calculaSalida(entradas);

                //Muestra la salida
                perceptron.SalidaPerceptron(entradas, salidaEsperada);
            }

            Console.WriteLine("Finaliza");
            Console.ReadKey();
        }

        private static int LeeDatosArchivo(string urlArchivo, double[] entrada, double[] salida) {
            var archivo = new System.IO.StreamReader(urlArchivo);
            archivo.ReadLine(); //La línea de simple serie
            archivo.ReadLine(); //La línea de título de cada columna de datos
            string leelinea;

            int limValores = 0;
            while ((leelinea = archivo.ReadLine()) != null) {
                limValores++;
                double valX = TraerNumeroCadena(leelinea, ';', 1);
                double valY = TraerNumeroCadena(leelinea, ';', 2);
                entrada[limValores] = valX;
                salida[limValores] = valY;
            }
            archivo.Close();
            return limValores;
        }
    }
}
```



```

//Dada una cadena con separaciones por delimitador, trae determinado ítem
private static double TraerNumeroCadena(string linea, char delimitador, int numeroToken) {
    string numero = "";
    int numTrae = 0;
    foreach (char t in linea) {
        if (t != delimitador)
            numero += t;
        else {
            numTrae += 1;
            if (numTrae == numeroToken) {
                numero = numero.Trim();
                if (numero == "") return 0;
                return Convert.ToDouble(numero);
            }
            numero = "";
        }
    }
    numero = numero.Trim();
    if (numero == "") return 0;
    return Convert.ToDouble(numero);
}

}

class Perceptron {
    public List<Capa> capas;

    //Imprime los datos de las diferentes capas
    public void SalidaPerceptron(List<double> entradas, List<double> salidaesperada) {
        for (int cont = 0; cont < entradas.Count; cont++) {
            Console.Write(entradas[cont].ToString() + " ");
        }
        Console.Write(" | ");
        for (int cont = 0; cont < salidaesperada.Count; cont++) {
            Console.Write(salidaesperada[cont].ToString() + " ");
        }
        Console.Write(" | ");
        for (int cont = 0; cont < capas[2].salidas.Count; cont++) {
            Console.Write(capas[2].salidas[cont].ToString() + " | ");
        }
        Console.WriteLine(" ");
    }

    //Crea las diversas capas
    public Perceptron(int numEntrada, int capa0, int capa1, int capa2) {
        Random azar = new Random();
        capas = new List<Capa>();
        capas.Add(new Capa(azar, capa0, numEntrada)); //Crea la capa 0
        capas.Add(new Capa(azar, capa1, capa0)); //Crea la capa 1
        capas.Add(new Capa(azar, capa2, capa1)); //Crea la capa 2
    }

    //Dada las entradas al perceptrón, se calcula la salida de cada capa.
    //Con eso se sabrá que salidas se obtienen con los pesos y umbrales actuales.
    //Esas salidas son requeridas para el algoritmo de entrenamiento.
    public void calculaSalida(List<double> entradas) {
        capas[0].CalculaCapa(entradas);
        capas[1].CalculaCapa(capas[0].salidas);
        capas[2].CalculaCapa(capas[1].salidas);
    }

    //Con las salidas previamente calculadas con unas determinadas entradas
    //se ejecuta el algoritmo de entrenamiento "Backpropagation"
    public void Entrena(List<double> entradas, List<double> salidaEsperada) {
        int capa0 = capas[0].neuronas.Count;
        int capa1 = capas[1].neuronas.Count;
        int capa2 = capas[2].neuronas.Count;

        //Factor de aprendizaje
        double alpha = 0.4;

        //Procesa pesos capa 2
        for (int j = 0; j < capa1; j++) //Va de neurona en neurona de la capa 1
            for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
                double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
                double Si = salidaEsperada[i]; //Salida esperada
                double alj = capas[1].salidas[j]; //Salida de la capa 1
                double dE2 = alj * (Yi - Si) * Yi * (1 - Yi); //La fórmula del error
                capas[2].neuronas[i].nuevospesos[j] = capas[2].neuronas[i].pesos[j] - alpha * dE2; //Ajusta el nuevo peso
            }

        //Procesa pesos capa 1
        for (int j = 0; j < capa0; j++) //Va de neurona en neurona de la capa 0
            for (int k = 0; k < capa1; k++) { //Va de neurona en neurona de la capa 1
                double acum = 0;
                for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                    double Yi = capas[2].salidas[i]; //Salida de la capa 2
                    double Si = salidaEsperada[i]; //Salida esperada
                    double W2ki = capas[2].neuronas[i].pesos[k];
                    acum += W2ki * (Yi - Si) * Yi * (1 - Yi); //Sumatoria
                }
                double a0j = capas[0].salidas[j];
                double alj = capas[1].salidas[k];
                double dE1 = a0j * alj * (1 - alj) * acum;
                capas[1].neuronas[k].nuevospesos[j] = capas[1].neuronas[k].pesos[j] - alpha * dE1;
            }
    }
}

```

```

//Procesa pesos capa 0
for (int j = 0; j < entradas.Count; j++) //Va de entrada en entrada
    for (int k = 0; k < capa0; k++) { //Va de neurona en neurona de la capa 0
        double acumular = 0;
        for (int p = 0; p < capa1; p++) { //Va de neurona en neurona de la capa 1
            double acum = 0;
            for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                double Yi = capas[2].salidas[i];
                double Si = salidaEsperada[i]; //Salida esperada
                double W2pi = capas[2].neuronas[i].pesos[p];
                acum += W2pi * (Yi - Si) * Yi * (1 - Yi); //Sumatoria interna
            }
            double W1kp = capas[1].neuronas[p].pesos[k];
            double alp = capas[1].salidas[p];
            acumular += W1kp * alp * (1 - alp) * acum; //Sumatoria externa
        }
        double xj = entradas[j];
        double a0k = capas[0].salidas[k];
        double dE0 = xj * a0k * (1 - a0k) * acumular;
        double W0jk = capas[0].neuronas[k].pesos[j];
        capas[0].neuronas[k].nuevospesos[j] = W0jk - alpha * dE0;
    }

//Procesa umbrales capa 2
for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
    double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
    double Si = salidaEsperada[i]; //Salida esperada
    double dE2 = (Yi - Si) * Yi * (1 - Yi);
    capas[2].neuronas[i].nuevoumbrales = capas[2].neuronas[i].umbral - alpha * dE2;
}

//Procesa umbrales capa 1
for (int k = 0; k < capa1; k++) { //Va de neurona en neurona de la capa 1
    double acum = 0;
    for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
        double Yi = capas[2].salidas[i]; //Salida de la capa 2
        double Si = salidaEsperada[i];
        double W2ki = capas[2].neuronas[i].pesos[k];
        acum += W2ki * (Yi - Si) * Yi * (1 - Yi);
    }
    double alk = capas[1].salidas[k];
    double dE1 = alk * (1 - alk) * acum;
    capas[1].neuronas[k].nuevoumbrales = capas[1].neuronas[k].umbral - alpha * dE1;
}

//Procesa umbrales capa 0
for (int k = 0; k < capa0; k++) { //Va de neurona en neurona de la capa 0
    double acumular = 0;
    for (int p = 0; p < capa1; p++) { //Va de neurona en neurona de la capa 1
        double acum = 0;
        for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
            double Yi = capas[2].salidas[i];
            double Si = salidaEsperada[i];
            double W2pi = capas[2].neuronas[i].pesos[p];
            acum += W2pi * (Yi - Si) * Yi * (1 - Yi);
        }
        double W1kp = capas[1].neuronas[p].pesos[k];
        double alp = capas[1].salidas[p];
        acumular += W1kp * alp * (1 - alp) * acum;
    }
    double a0k = capas[0].salidas[k];
    double dE0 = a0k * (1 - a0k) * acumular;
    capas[0].neuronas[k].nuevoumbrales = capas[0].neuronas[k].umbral - alpha * dE0;
}

//Actualiza los pesos
capas[0].actualiza();
capas[1].actualiza();
capas[2].actualiza();
}

}

class Capa {
public List<Neurona> neuronas; //Las neuronas que tendrá la capa
public List<double> salidas; //Almacena las salidas de cada neurona

public Capa(Random azar, int totalNeuronas, int totalEntradas) {
    neuronas = new List<Neurona>();
    salidas = new List<double>();
    //Genera las neuronas
    for (int cont = 0; cont < totalNeuronas; cont++) {
        neuronas.Add(new Neurona(azar, totalEntradas));
        salidas.Add(0);
    }
}

//Calcula las salidas de cada neurona de la capa
public void CalculaCapa(List<double> entradas) {
    for (int cont = 0; cont < neuronas.Count; cont++) {
        salidas[cont] = neuronas[cont].calculaSalida(entradas);
    }
}

//Actualiza los pesos y umbrales de las neuronas

```

```

        public void actualiza(){
            for (int cont = 0; cont < neuronas.Count; cont++) {
                neuronas[cont].actualiza();
            }
        }
    }

class Neurona {
    public List<double> pesos; //Los pesos para cada entrada
    public List<double> nuevospesos; //Nuevos pesos dados por el algoritmo de "backpropagation"
    public double umbral; //El peso del umbral
    public double nuevoumbral; //Nuevo umbral dado por el algoritmo de "backpropagation"

    //Inicializa los pesos y umbral con un valor al azar
    public Neurona(Random azar, int totalEntradas) {
        pesos = new List<double>();
        nuevospesos = new List<double>();
        for (int cont = 0; cont < totalEntradas; cont++) {
            pesos.Add(azar.NextDouble());
            nuevospesos.Add(0);
        }
        umbral = azar.NextDouble();
        nuevoumbral = 0;
    }

    //Calcula la salida de la neurona dependiendo de las entradas
    public double calculaSalida(List<double> entradas) {
        double valor = 0;
        for (int cont = 0; cont < pesos.Count; cont++) {
            valor += entradas[cont] * pesos[cont];
        }
        valor += umbral;
        return 1 / (1 + Math.Exp(-valor));
    }

    //Reemplaza viejos pesos por nuevos
    public void actualiza(){
        for (int cont = 0; cont < pesos.Count; cont++){
            pesos[cont] = nuevospesos[cont];
        }
        umbral = nuevoumbral;
    }
}
}

```

Ejemplo de ejecución

```
C:\Users\engin\OneDrive\Proyecto\Libro\13. RedNeuronal2\Neuronal3\PerceptronA\ConsoleApp1\bin\Debug\ConsoleApp1.exe
0,924791086350975 | 0,19844832345679 | 0,189374918235469 |
0,927576601671309 | 0,164403459567901 | 0,178917943783181 |
0,930362116991644 | 0,132933882716049 | 0,170849556819221 |
0,933147632311978 | 0,10429639691358 | 0,164598921037726 |
0,935933147632312 | 0,0787275685185185 | 0,15973706443211 |
0,938718662952646 | 0,0564417836419753 | 0,155940984712764 |
0,941504178272981 | 0,0376294663580247 | 0,15296651952702 |
0,944289693593315 | 0,0224554703703704 | 0,15062822230495 |
0,947075208913649 | 0,0110576604938272 | 0,148784537252624 |
0,949860724233983 | 0,00354569135802468 | 0,147326881577883 |
0,952646239554318 | 0 | 0,146171578754946 |
0,955431754874652 | 0,000471016358024675 | 0,145253867785258 |
0,958217270194986 | 0,00497860030864195 | 0,144523427982232 |
0,96100278551532 | 0,0135117129629629 | 0,14394101573241 |
0,963788300835655 | 0,0260283228395062 | 0,143475922239083 |
0,966573816155989 | 0,0424555521604938 | 0,143104041413668 |
0,969359331476323 | 0,0626900608024691 | 0,142806394170429 |
0,972144846796657 | 0,0865986675925926 | 0,142567996210143 |
0,974930362116992 | 0,114019205246914 | 0,142376985767829 |
0,977715877437326 | 0,14476160462963 | 0,142223949114293 |
0,98050139275766 | 0,178609199691358 | 0,142101397180982 |
0,983286908077994 | 0,215320243827161 | 0,142003358153119 |
0,986072423398329 | 0,25462962962963 | 0,141925059389348 |
0,988857938718663 | 0,296250797962963 | 0,141862678383485 |
0,991643454038997 | 0,3398778225 | 0,14181314725965 |
0,994428969359331 | 0,385187655709877 | 0,141773998898508 |
0,997214484679666 | 0,431842519135802 | 0,141743245528822 |
1 | 0,479492420308642 | 0,141719282703529 |
```

Ilustración 60: Resultado de la ejecución del programa de detección de patrones en series X,Y

Llevando el resultado numérico al gráfico, en azul los datos esperados, en naranja lo aprendido por la red neuronal.

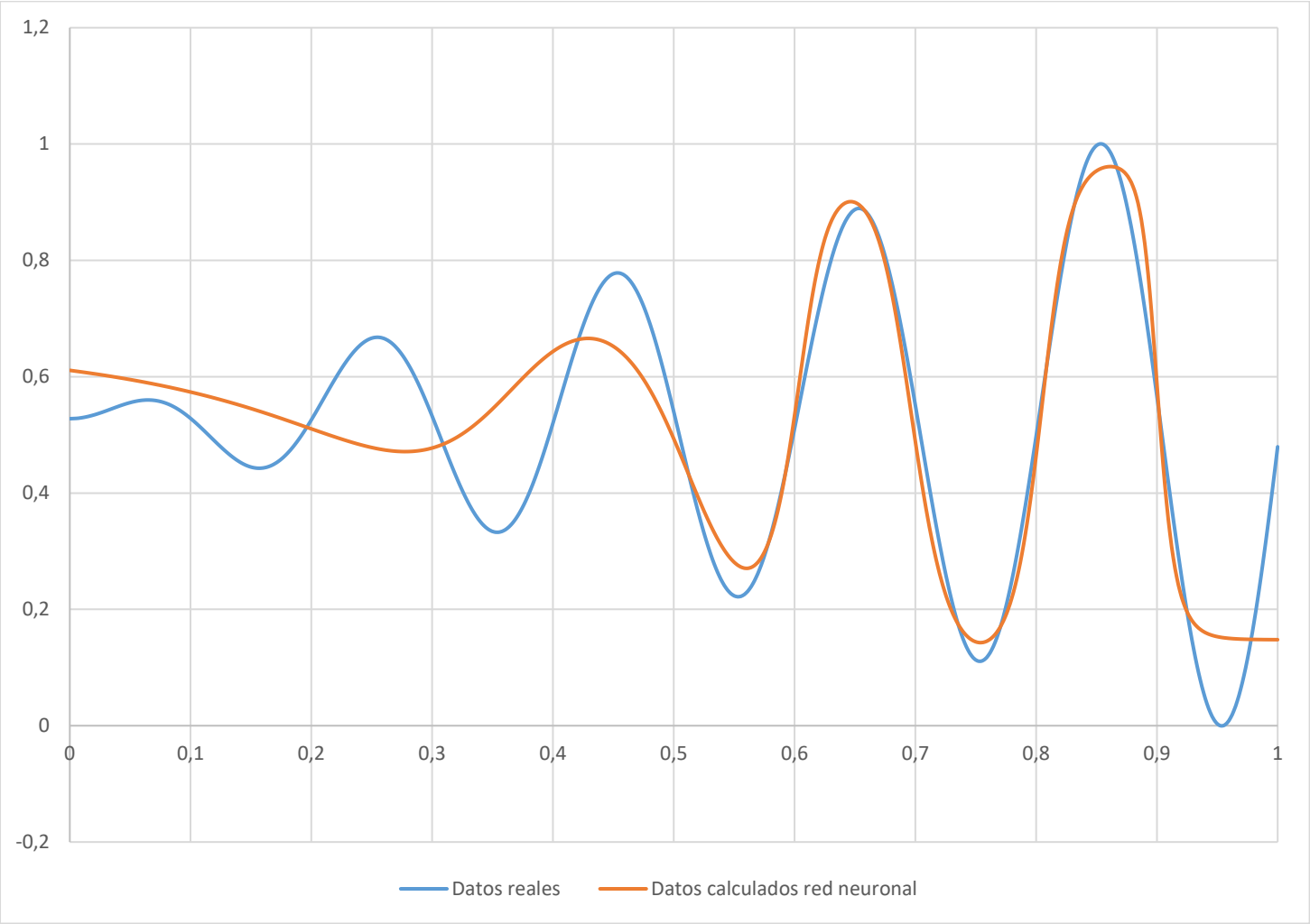


Ilustración 61: Comparativa entre los datos reales y los datos calculados por la red neuronal

Este proceso es lento. Puede acelerarse disminuyendo el número de neuronas en las capas ocultas a 4 (en ambas), mejorará la velocidad, pero bajará la precisión del ajuste en curva.

El sobreajuste

Un problema que sucede con las redes neuronales es el sobreajuste, es decir, que la red neuronal se ajusta tanto a tal punto que logra prácticamente acertar a todas o casi todas las salidas esperadas, pero falla estrepitosamente en interpolar. Obsérvese el gráfico:

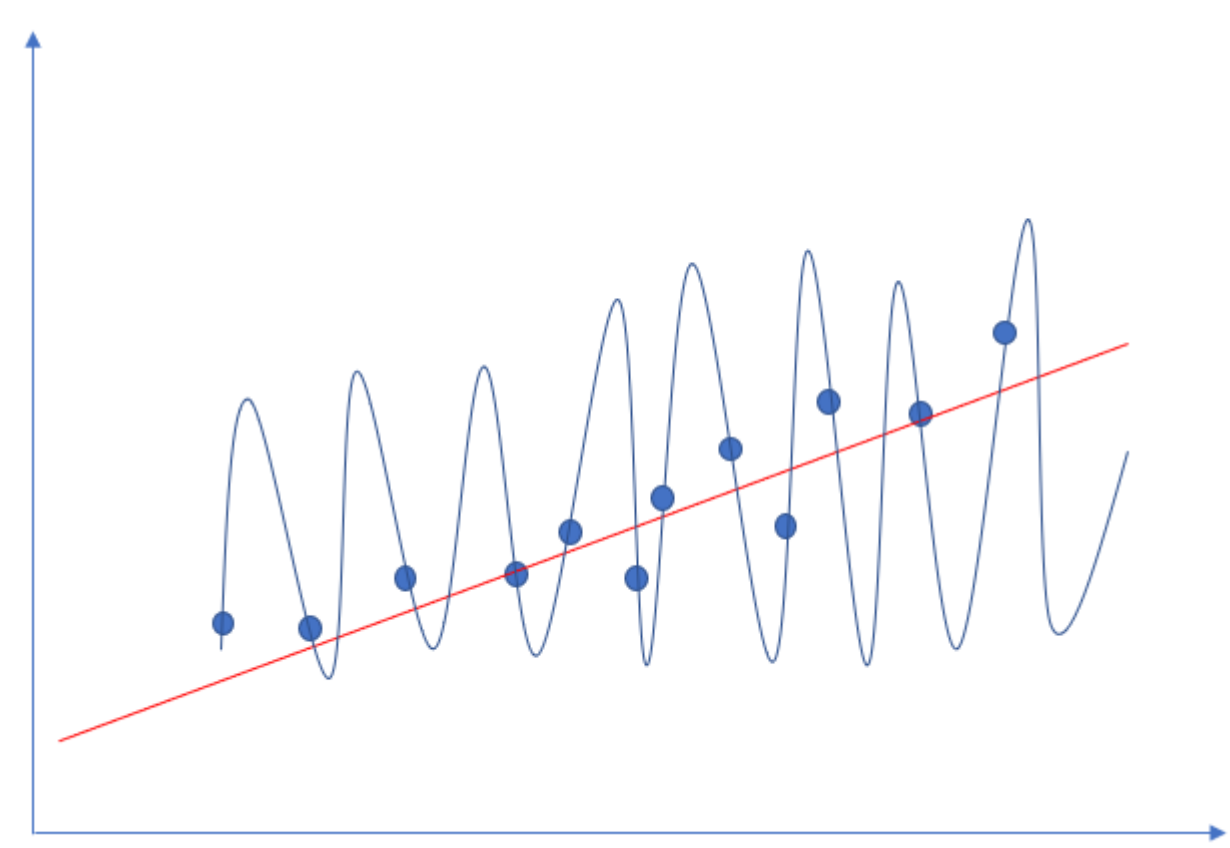


Ilustración 62: Línea roja vs Línea con varias curvas

Los puntos azules son los datos a los cuáles vamos a buscar un patrón. La línea roja es la línea de tendencia recta que mejor se ajusta a esos puntos, a simple vista, no acierta a todos. En cambio, la línea llena de curvas en azul acierta a todos los puntos. En una primera lectura, se puede juzgar que la línea llena de curvas lo hace mucho mejor que la línea roja, pero hay un gran problema y es en la **interpolación**, porque mientras la línea roja es más conservadora al deducir el valor que puede haber entre una pareja seguida de dos puntos azules, por el otro lado, al usar la línea llena de curvas, hay unas crestas y valles muy pronunciados por lo que el valor dada con esa interpolación no sería realista.

¿Cómo evitar el sobreajuste? No es fácil. Una técnica es tomar el total de los datos de entrada y salidas esperados, usar el 80% de estos datos para entrenar la red neuronal y el 20% restante para verificar si hay sobreajuste o no. Los porcentajes pueden variar. Ese 80% se escoge al azar de los datos de entrada y salidas esperados, el 20% restante es para validar sobreajuste.

La idea es simple: se entrena con el 80% y se valida el ajuste con el 20%, ese ajuste (menor es mejor) con los datos de validación debe mejorar en cada ciclo, si comienza a alejarse, entonces el entrenamiento se detiene. Observar la gráfica:

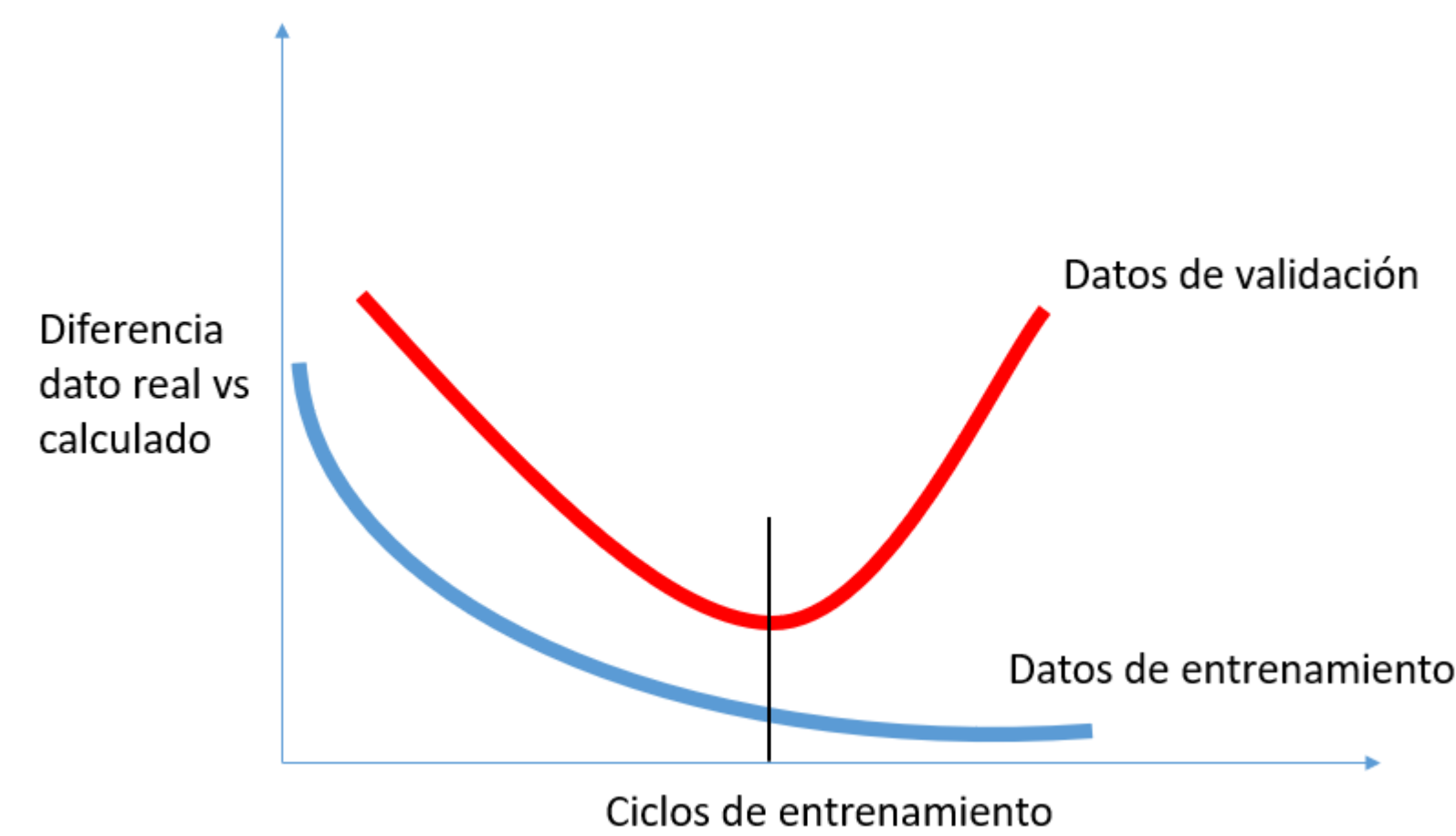


Ilustración 63: Ciclos de entrenamiento y su resultado con los datos de entrenamiento vs datos de validación

En el punto mostrado con la línea vertical en negro es donde se muestra que el ajuste logra su mejor valor y debe detenerse el entrenamiento.

El siguiente programa muestra las clases principal y perceptrón:

```
class Program {
    static void Main(string[] args) {
        //Lee los datos de un archivo plano
        int MaximosRegistros = 2000;
        double[] entrada = new double[MaximosRegistros + 1];
        double[] salidas = new double[MaximosRegistros + 1];

        //Qué valores de entrada se usarán para entrenamiento y cuáles para validación de "overfitting"
        bool[] entraValida = new bool[MaximosRegistros + 1];

        const string urlArchivo = "datos.tendencia";
        int ConjuntoEntradas = LeeDatosArchivo(urlArchivo, entrada, salidas);

        //Normaliza los valores entre 0 y 1 que es lo que requiere el perceptrón
        double minimoX = entrada[0], maximoX = entrada[0];
        double minimoY = salidas[0], maximoY = salidas[0];
        for (int cont = 0; cont < ConjuntoEntradas; cont++) {
            if (entrada[cont] > maximoX) maximoX = entrada[cont];
            if (salidas[cont] > maximoY) maximoY = salidas[cont];
            if (entrada[cont] < minimoX) minimoX = entrada[cont];
            if (salidas[cont] < minimoY) minimoY = salidas[cont];
        }

        for (int cont = 0; cont < ConjuntoEntradas; cont++) {
            entrada[cont] = (entrada[cont] - minimoX) / (maximoX - minimoX);
            salidas[cont] = (salidas[cont] - minimoY) / (maximoY - minimoY);
            entraValida[cont] = false;
        }

        //Marca al azar que valores usará para entrenar y cuáles para validar "overfitting"
        int porcentaje = 20;
        int totalValida = ConjuntoEntradas * porcentaje / 100;
        int cuentaValida = 0;
        Random aleatorio = new Random();
        do {
            int pos = aleatorio.Next(ConjuntoEntradas);
            if (entraValida[pos] == false) {
                entraValida[pos] = true;
                cuentaValida++;
            }
        } while (cuentaValida < totalValida);

        int numEntradas = 1; //Número de entradas
        int capa0 = 5; //Total neuronas en la capa 0
        int capa1 = 5; //Total neuronas en la capa 1
        int capa2 = 1; //Total neuronas en la capa 2

        Perceptron perceptron = new Perceptron(numEntradas, capa0, capa1, capa2);

        //Estas serán las entradas externas al perceptrón
        List<double> entradas = new List<double>();
        entradas.Add(0);

        //Estas serán las salidas esperadas externas al perceptrón
        List<double> salidaEsperada = new List<double>();
        salidaEsperada.Add(0);

        //Ciclo que entrena la red neuronal
        double anteriorAjuste = Double.MaxValue; //Para validar el "overfitting"
        int ciclos = 0;
        while (true) {
            ciclos++;

            //Por cada ciclo, se entrena el perceptrón con todos los valores
            for (int conjunto = 0; conjunto < ConjuntoEntradas; conjunto++) {
                //Si el valor es para validar no lo usa para entrenar
                if (entraValida[conjunto] == true) continue;

                //Entradas y salidas esperadas
                entradas[0] = entrada[conjunto];
                salidaEsperada[0] = salidas[conjunto];

                //Primero calcula la salida del perceptrón con esas entradas
                perceptron.calculaSalida(entradas);

                //Luego entrena el perceptrón para ajustar los pesos y umbrales
                perceptron.Entrena(entradas, salidaEsperada);
            }

            //Valida el "overfitting"
            double ajusteValidando = 0;
            for (int conjunto = 0; conjunto < ConjuntoEntradas; conjunto++) {
                //Si el valor es para entrenar no lo usa para validar
                if (entraValida[conjunto] == false) continue;

                //Entradas y salidas esperadas
                entradas[0] = entrada[conjunto];
                salidaEsperada[0] = salidas[conjunto];
                ajusteValidando += perceptron.CalculaAjuste(entradas, salidaEsperada);
            }
        }
    }
}
```

```

        //Si el nuevo entrenamiento aleja la precisión entonces termina el entrenamiento
        if (ajusteValidando > anteriorAjuste) {
            Console.WriteLine("Posible Overfitting en: " + ciclos.ToString());
            break;
        }
        else
            anteriorAjuste = ajusteValidando;
    }

    //Muestra el resultado del entrenamiento
    for (int conjunto = 0; conjunto < ConjuntoEntradas; conjunto++) {
        if (entraValida[conjunto] == false) continue;

        //Entradas y salidas esperadas
        entradas[0] = entrada[conjunto];
        salidaEsperada[0] = salidas[conjunto];

        //Calcula la salida del perceptrón con esas entradas
        perceptron.calculaSalida(entradas);

        //Muestra la salida
        perceptron.SalidaPerceptron(entradas, salidaEsperada);
    }

    Console.WriteLine("Finaliza");
    Console.ReadKey();
}

private static int LeeDatosArchivo(string urlArchivo, double[] entrada, double[] salida) {
    var archivo = new System.IO.StreamReader(urlArchivo);
    archivo.ReadLine(); //La línea de simple serie
    archivo.ReadLine(); //La línea de título de cada columna de datos
    string leelinea;

    int limValores = 0;
    while ((leelinea = archivo.ReadLine()) != null) {
        limValores++;
        double valX = TraerNumeroCadena(leelinea, ';', 1);
        double valY = TraerNumeroCadena(leelinea, ';', 2);
        entrada[limValores] = valX;
        salida[limValores] = valY;
    }
    archivo.Close();
    return limValores;
}

//Dada una cadena con separaciones por delimitador, trae determinado ítem
private static double TraerNumeroCadena(string linea, char delimitador, int numeroToken) {
    string numero = "";
    int numTrae = 0;
    foreach (char t in linea) {
        if (t != delimitador)
            numero += t;
        else {
            numTrae += 1;
            if (numTrae == numeroToken) {
                numero = numero.Trim();
                if (numero == "") return 0;
                return Convert.ToDouble(numero);
            }
            numero = "";
        }
    }
    numero = numero.Trim();
    if (numero == "") return 0;
    return Convert.ToDouble(numero);
}

}

class Perceptron {
    public List<Capa> capas;

    //Calcula el ajuste para validar el "overfitting"
    public double CalculaAjuste(List<double> entradas, List<double> salidaesperada) {
        double acum = 0;
        calculaSalida(entradas);
        for (int cont = 0; cont < salidaesperada.Count; cont++) {
            double calculado = capas[2].salidas[cont];
            double esperado = salidaesperada[cont];
            acum += (calculado - esperado) * (calculado - esperado);
        }
        return acum;
    }

    //Imprime los datos de las diferentes capas
    public void SalidaPerceptron(List<double> entradas, List<double> salidaesperada) {
        for (int cont = 0; cont < entradas.Count; cont++) {
            Console.Write(entradas[cont].ToString() + " ");
        }
        Console.Write(" | ");
        for (int cont = 0; cont < salidaesperada.Count; cont++) {
            Console.Write(salidaesperada[cont].ToString() + " ");
        }
        Console.Write(" | ");
    }
}

```



```

double acum = 0;
for (int cont = 0; cont < capas[2].salidas.Count; cont++) {
    double calculado = capas[2].salidas[cont];
    double esperado = salidaesperada[cont];
    acum += (calculado - esperado) * (calculado - esperado);
    Console.WriteLine(capas[2].salidas[cont].ToString() + " | ");
}
Console.WriteLine(acum.ToString());
}

//Crea las diversas capas
public Perceptron(int numEntrada, int capa0, int capa1, int capa2) {
    Random azar = new Random();
    capas = new List<Capa>();
    capas.Add(new Capa(azar, capa0, numEntrada)); //Crea la capa 0
    capas.Add(new Capa(azar, capa1, capa0)); //Crea la capa 1
    capas.Add(new Capa(azar, capa2, capa1)); //Crea la capa 2
}

//Dada las entradas al perceptrón, se calcula la salida de cada capa.
//Con eso se sabrá que salidas se obtienen con los pesos y umbrales actuales.
//Esas salidas son requeridas para el algoritmo de entrenamiento.
public void calculaSalida(List<double> entradas) {
    capas[0].CalculaCapa(entradas);
    capas[1].CalculaCapa(capas[0].salidas);
    capas[2].CalculaCapa(capas[1].salidas);
}

//Con las salidas previamente calculadas con unas determinadas entradas
//se ejecuta el algoritmo de entrenamiento "Backpropagation"
public void Entrena(List<double> entradas, List<double> salidaEsperada) {
    int capa0 = capas[0].neuronas.Count;
    int capa1 = capas[1].neuronas.Count;
    int capa2 = capas[2].neuronas.Count;

    //Factor de aprendizaje
    double alpha = 0.4;

    //Procesa pesos capa 2
    for (int j = 0; j < capa1; j++) //Va de neurona en neurona de la capa 1
        for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
            double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
            double Si = salidaEsperada[i]; //Salida esperada
            double alj = capas[1].salidas[j]; //Salida de la capa 1
            double dE2 = alj * (Yi - Si) * Yi * (1 - Yi); //La fórmula del error
            capas[2].neuronas[i].nuevospesos[j] = capas[2].neuronas[i].pesos[j] - alpha * dE2; //Ajusta el nuevo peso
        }

    //Procesa pesos capa 1
    for (int j = 0; j < capa0; j++) //Va de neurona en neurona de la capa 0
        for (int k = 0; k < capa1; k++) { //Va de neurona en neurona de la capa 1
            double acum = 0;
            for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                double Yi = capas[2].salidas[i]; //Salida de la capa 2
                double Si = salidaEsperada[i]; //Salida esperada
                double W2ki = capas[2].neuronas[i].pesos[k];
                acum += W2ki * (Yi - Si) * Yi * (1 - Yi); //Sumatoria
            }
            double a0j = capas[0].salidas[j];
            double alk = capas[1].salidas[k];
            double dE1 = a0j * alk * (1 - alk) * acum;
            capas[1].neuronas[k].nuevospesos[j] = capas[1].neuronas[k].pesos[j] - alpha * dE1;
        }

    //Procesa pesos capa 0
    for (int j = 0; j < entradas.Count; j++) //Va de entrada en entrada
        for (int k = 0; k < capa0; k++) { //Va de neurona en neurona de la capa 0
            double acumular = 0;
            for (int p = 0; p < capa1; p++) { //Va de neurona en neurona de la capa 1
                double acum = 0;
                for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                    double Yi = capas[2].salidas[i];
                    double Si = salidaEsperada[i]; //Salida esperada
                    double W2pi = capas[2].neuronas[i].pesos[p];
                    acum += W2pi * (Yi - Si) * Yi * (1 - Yi); //Sumatoria interna
                }
                double W1kp = capas[1].neuronas[p].pesos[k];
                double alp = capas[1].salidas[p];
                acumular += W1kp * alp * (1 - alp) * acum; //Sumatoria externa
            }
            double xj = entradas[j];
            double a0k = capas[0].salidas[k];
            double dE0 = xj * a0k * (1 - a0k) * acumular;
            double W0jk = capas[0].neuronas[k].pesos[j];
            capas[0].neuronas[k].nuevospesos[j] = W0jk - alpha * dE0;
        }

    //Procesa umbrales capa 2
    for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa de salida (capa 2)
        double Yi = capas[2].salidas[i]; //Salida de la neurona de la capa de salida
        double Si = salidaEsperada[i]; //Salida esperada
        double dE2 = (Yi - Si) * Yi * (1 - Yi);
        capas[2].neuronas[i].nuevolumbral = capas[2].neuronas[i].umbral - alpha * dE2;
    }

    //Procesa umbrales capa 1

```

```

    for (int k = 0; k < capa1; k++) { //Va de neurona en neurona de la capa 1
        double acum = 0;
        for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
            double Yi = capas[2].salidas[i]; //Salida de la capa 2
            double Si = salidaEsperada[i];
            double W2ki = capas[2].neuronas[i].pesos[k];
            acum += W2ki * (Yi - Si) * Yi * (1 - Yi);
        }
        double a1k = capas[1].salidas[k];
        double dE1 = a1k * (1 - a1k) * acum;
        capas[1].neuronas[k].nuevoumbra1 = capas[1].neuronas[k].umbra1 - alpha * dE1;
    }

    //Procesa umbrales capa 0
    for (int k = 0; k < capa0; k++) { //Va de neurona en neurona de la capa 0
        double acumular = 0;
        for (int p = 0; p < capa1; p++) { //Va de neurona en neurona de la capa 1
            double acum = 0;
            for (int i = 0; i < capa2; i++) { //Va de neurona en neurona de la capa 2
                double Yi = capas[2].salidas[i];
                double Si = salidaEsperada[i];
                double W2pi = capas[2].neuronas[i].pesos[p];
                acum += W2pi * (Yi - Si) * Yi * (1 - Yi);
            }
            double W1kp = capas[1].neuronas[p].pesos[k];
            double alp = capas[1].salidas[p];
            acumular += W1kp * alp * (1 - alp) * acum;
        }
        double a0k = capas[0].salidas[k];
        double dE0 = a0k * (1 - a0k) * acumular;
        capas[0].neuronas[k].nuevoumbra1 = capas[0].neuronas[k].umbra1 - alpha * dE0;
    }

    //Actualiza los pesos
    capas[0].actualiza();
    capas[1].actualiza();
    capas[2].actualiza();
}
}

```

```

class Capa {
    public List<Neurona> neuronas; //Las neuronas que tendrá la capa
    public List<double> salidas; //Almacena las salidas de cada neurona

    public Capa(Random azar, int totalNeuronas, int totalEntradas) {
        neuronas = new List<Neurona>();
        salidas = new List<double>();
        //Genera las neuronas
        for (int cont = 0; cont < totalNeuronas; cont++) {
            neuronas.Add(new Neurona(azar, totalEntradas));
            salidas.Add(0);
        }
    }

    //Calcula las salidas de cada neurona de la capa
    public void CalculaCapa(List<double> entradas) {
        for (int cont = 0; cont < neuronas.Count; cont++) {
            salidas[cont] = neuronas[cont].calculaSalida(entradas);
        }
    }

    //Actualiza los pesos y umbrales de las neuronas
    public void actualiza() {
        for (int cont = 0; cont < neuronas.Count; cont++) {
            neuronas[cont].actualiza();
        }
    }
}

class Neurona {
    public List<double> pesos; //Los pesos para cada entrada
    public List<double> nuevospesos; //Nuevos pesos dados por el algoritmo de "backpropagation"
    public double umbra1; //El peso del umbra1
    public double nuevoumbra1; //Nuevo umbra1 dado por el algoritmo de "backpropagation"

    //Inicializa los pesos y umbra1 con un valor al azar
    public Neurona(Random azar, int totalEntradas) {
        pesos = new List<double>();
        nuevospesos = new List<double>();
        for (int cont = 0; cont < totalEntradas; cont++) {
            pesos.Add(azar.NextDouble());
            nuevospesos.Add(0);
        }
        umbra1 = azar.NextDouble();
        nuevoumbra1 = 0;
    }

    //Calcula la salida de la neurona dependiendo de las entradas
    public double calculaSalida(List<double> entradas) {
        double valor = 0;
        for (int cont = 0; cont < pesos.Count; cont++) {
            valor += entradas[cont] * pesos[cont];
        }
    }
}

```

```

        valor += umbral;
        return 1 / (1 + Math.Exp(-valor));
    }

    //Reemplaza viejos pesos por nuevos
    public void actualiza() {
        for (int cont = 0; cont < pesos.Count; cont++) {
            pesos[cont] = nuevospesos[cont];
        }
        umbral = nuevoumbral;
    }
}

```

¿Funciona? Realmente no, la sospecha es que durante las pruebas el programa detenía el entrenamiento en pocos ciclos (no pasaba de 1000) porque detectaba que el ajuste comenzaba a subir (cuando debe bajar). Pero el aprendizaje a simple vista no era muy bueno.

Sin validar el sobreajuste y que haya 10 mil ciclos de entrenamiento, se obtiene este gráfico, donde en azul son los valores esperados y en naranja lo que aprendió la red neuronal:

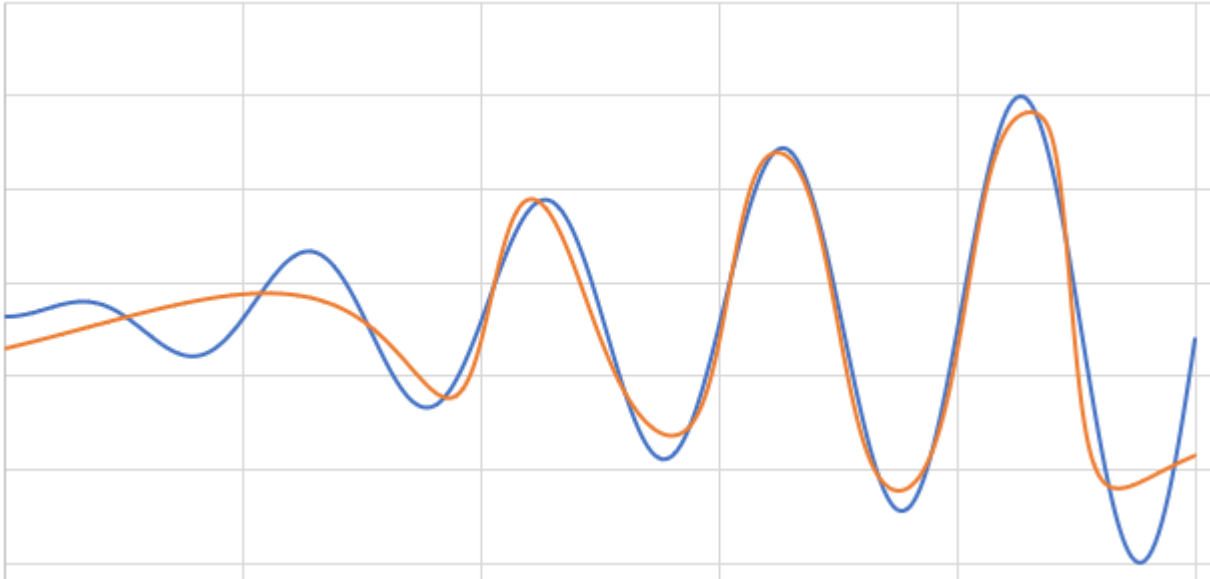


Ilustración 64: Red neuronal muestra que ha aprendido el comportamiento de los datos

Validando el sobreajuste se obtiene este gráfico en cambio:

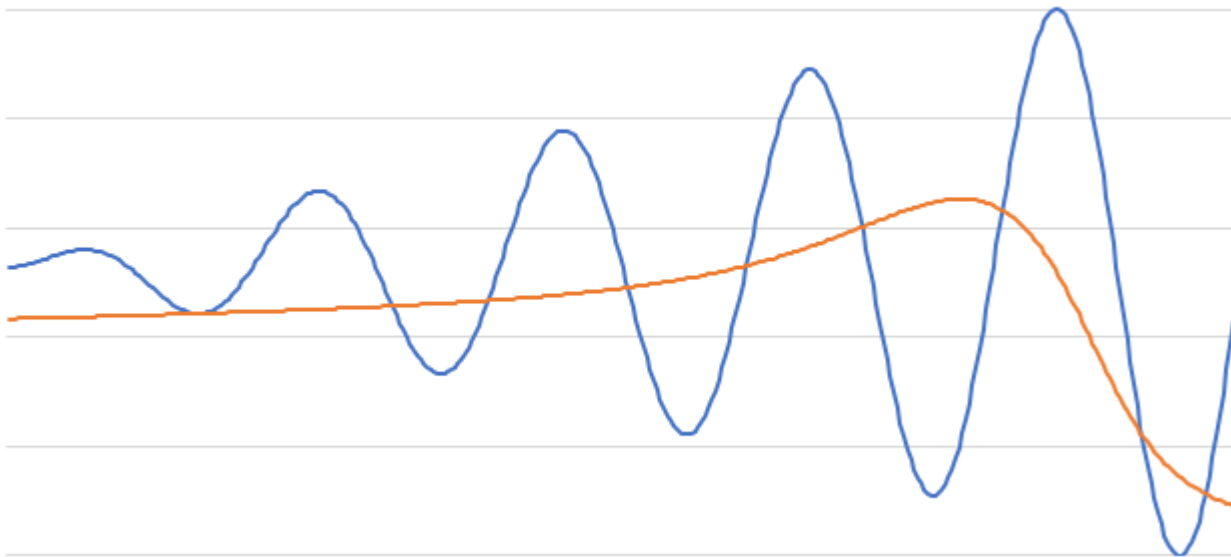


Ilustración 65: No es muy claro que la red neuronal haya aprendido el comportamiento de los datos

Es notable que dejando operar con más ciclos, hay un mejor ajuste. Entonces ¿Por qué el programa en C# anterior falla? Un cambio al programa para que muestre como se comporta el ajuste con los datos de validación a medida que se ejecuta el entrenamiento da la respuesta:

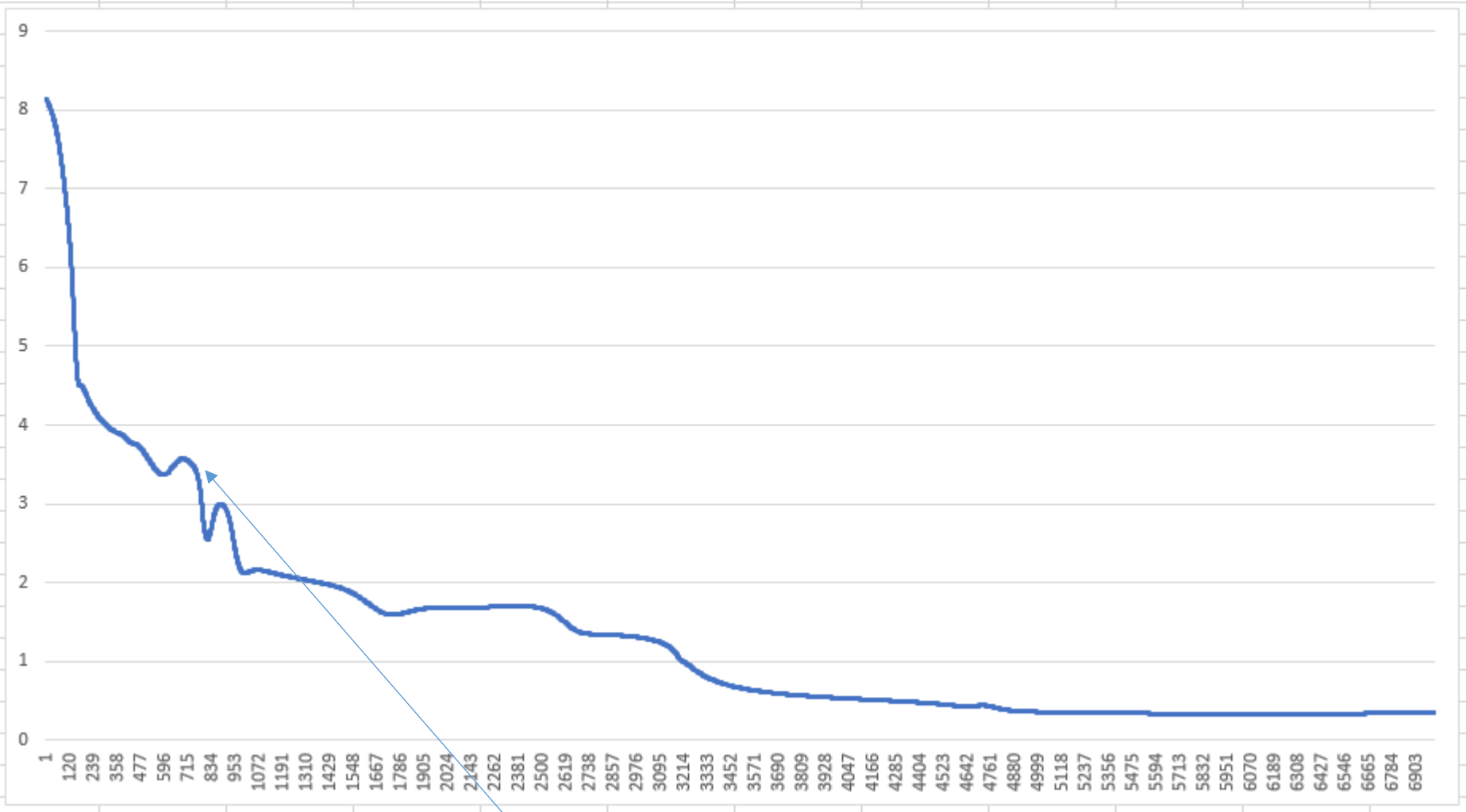


Ilustración 66: A más ciclos de entrenamiento mejor ajuste con los datos de validación, pero hay altibajos

Como se puede observar, el cálculo del ajuste con los datos de validación va bajando (que significa que la red neuronal está entrenando bien), pero en ocasiones hay un retroceso en el ajuste y en ese momento el programa en C# anterior se detiene, pero si se dejara continuar, vuelve a bajar y mantiene esa tónica en general. Luego habría que modificar el programa en C# para que no tenga en cuenta esos altibajos y vea el comportamiento en conjunto. ¿Cómo hacerlo? En el momento que escribo este libro, lo estoy investigando.

Bibliografía y webgrafía

Redes Neuronales: Fácil y desde cero.

Autor: Javier García

https://www.youtube.com/watch?v=jaEIv_E29sk&list=PLAnA8FVrBl8AWkZmbswwWiF8a_52dQ3JQ

Dot CSV

<https://www.youtube.com/channel/UCy5znSnfMsDwaLIROnZ7Qbg>

El traductor de Ingeniería

<https://www.youtube.com/watch?v=ojiMGOqwwCE&list=PLTIdiuUvwYCEVuBjYVwz8Pd77wSoj-EsO>