C# Y .NET 8
Parte 11. Redes Neuronales

2024-07

Rafael Alberto Moreno Parra
ramsoftware@gmail.com

Contenido

Tabla de ilustraciones

Acerca del autor

Rafael Alberto Moreno Parra

ramsoftware@gmail.com o enginelife@hotmail.com

Sitio Web: http://darwin.50webs.com (dedicado a la investigación de algoritmos evolutivos y vida artificial).

Github: https://github.com/ramsoftware

Youtube: https://www.youtube.com/@RafaelMorenoP

Licencia de este libro

Licencia del software

Marcas registradas

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 2022 ® Enlace: https://visualstudio.microsoft.com/es/vs/

Dedicatoria

A mis padres, a mi hermana….

Y a mi tropa gatuna: Sally, Suini, Grisú, Capuchina, Milú, Arián, Frac y mis recordados Tinita, Tammy, Vikingo y Michu.

Iniciando

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

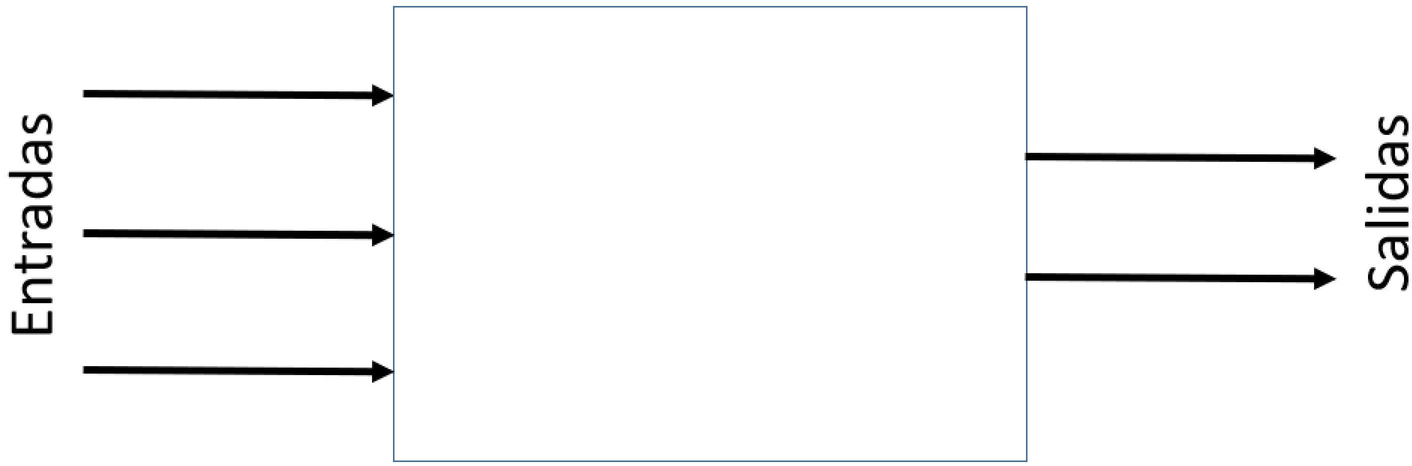


Ilustración 1: Caja negra, entradas y salidas

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

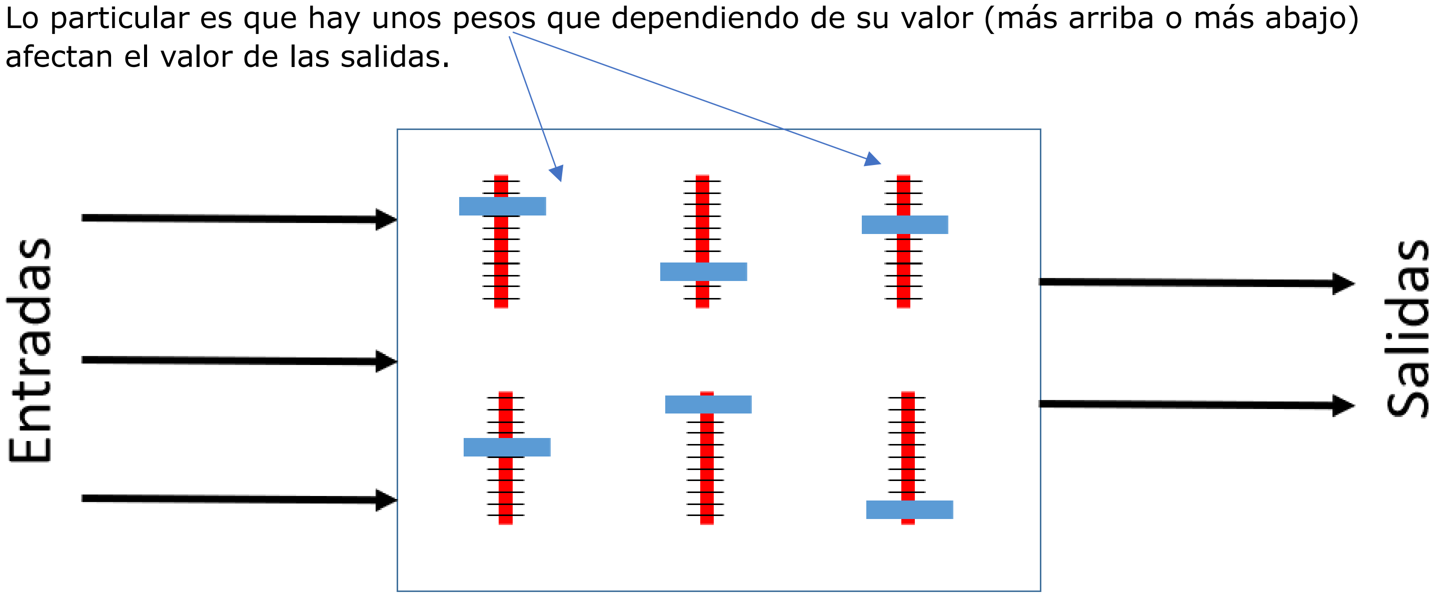


Ilustración 2: Pesos al interior de la caja

En el ejemplo, están 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).

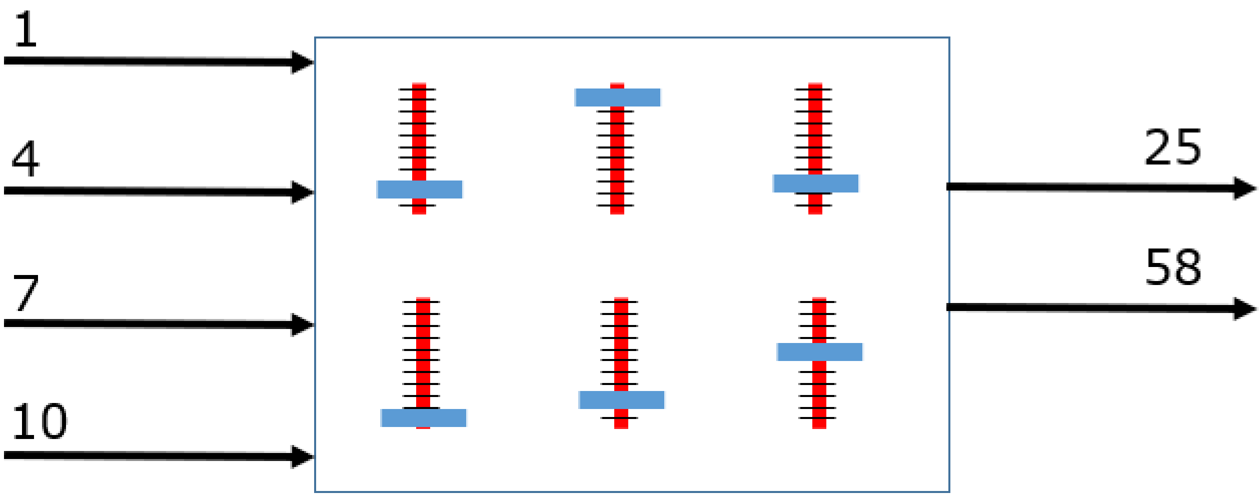


Ilustración 3: Esos pesos ya operan con el ejemplo A

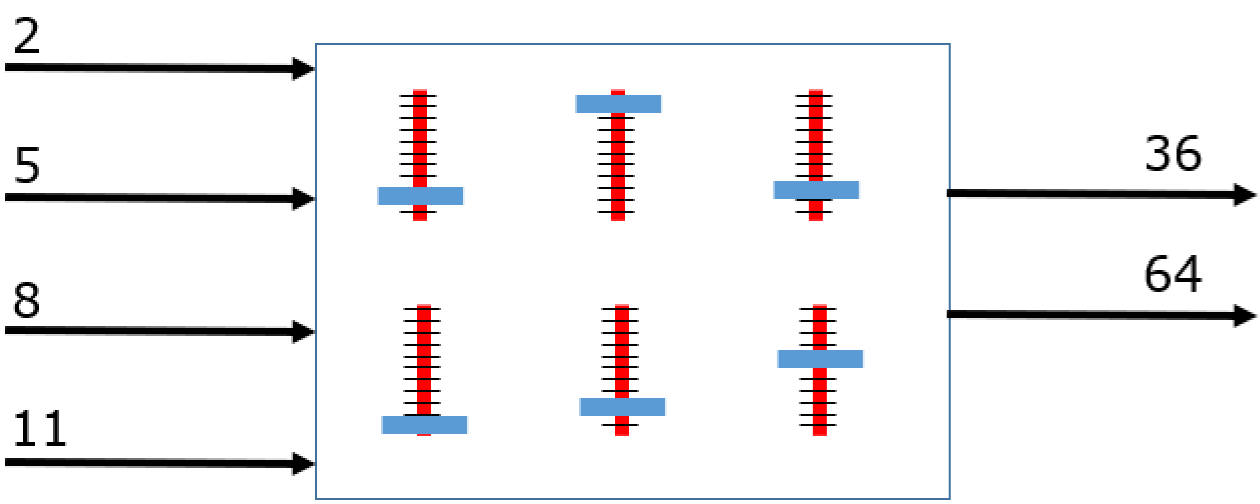


Ilustración 4: Los mismos pesos funcionan para el ejemplo B

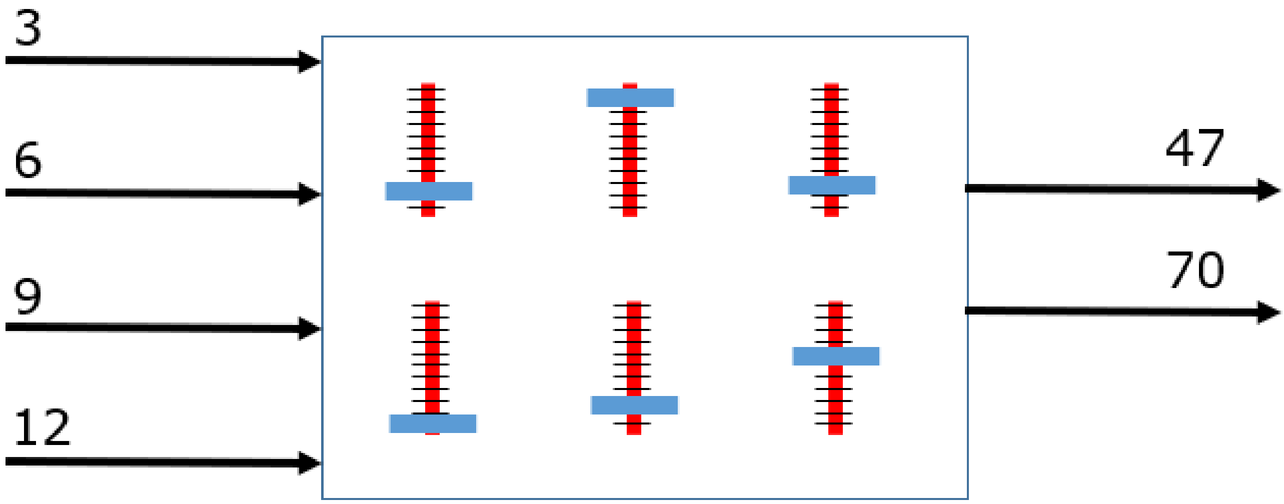


Ilustración 5: Y esos mismos pesos funcionan para el ejemplo C

¿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.

Perceptrón simple

Se inicia con una neurona. Esta es su representación:

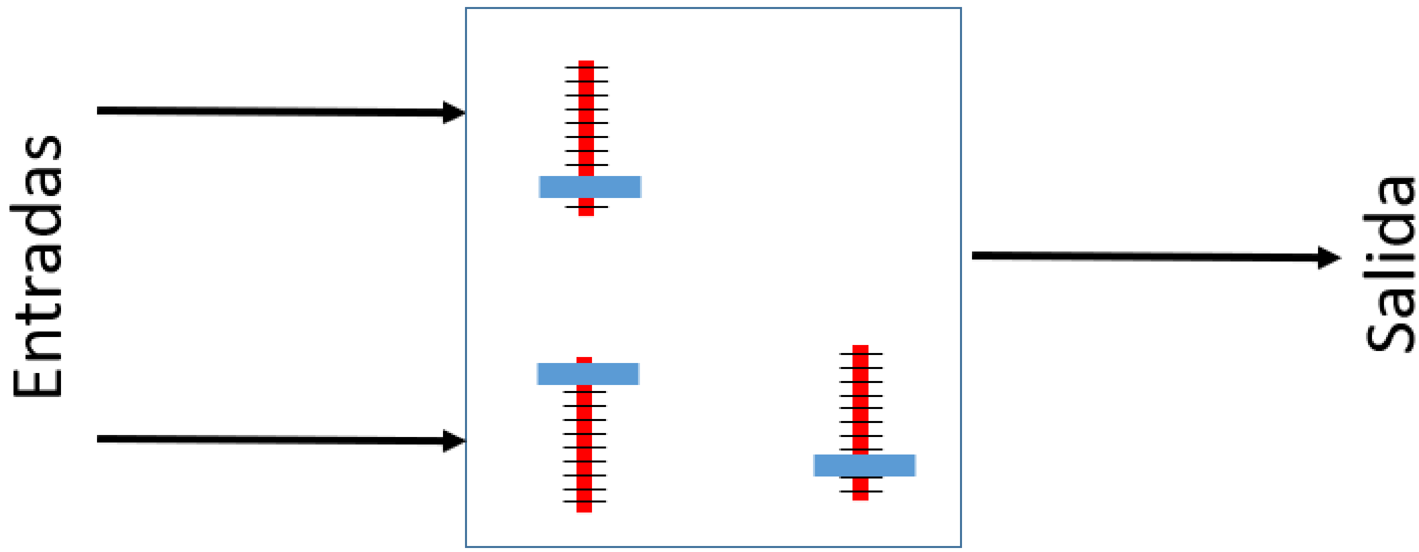


Ilustración 6: Perceptrón simple

Dos entradas, una salida y tres pesos. Se demostrará 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 Perceptrón 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 razón de este cambio es que se requieren valores cuantitativos para ser usados dentro de fórmulas matemáticas.

Paso 2: Diseñando la neurona:

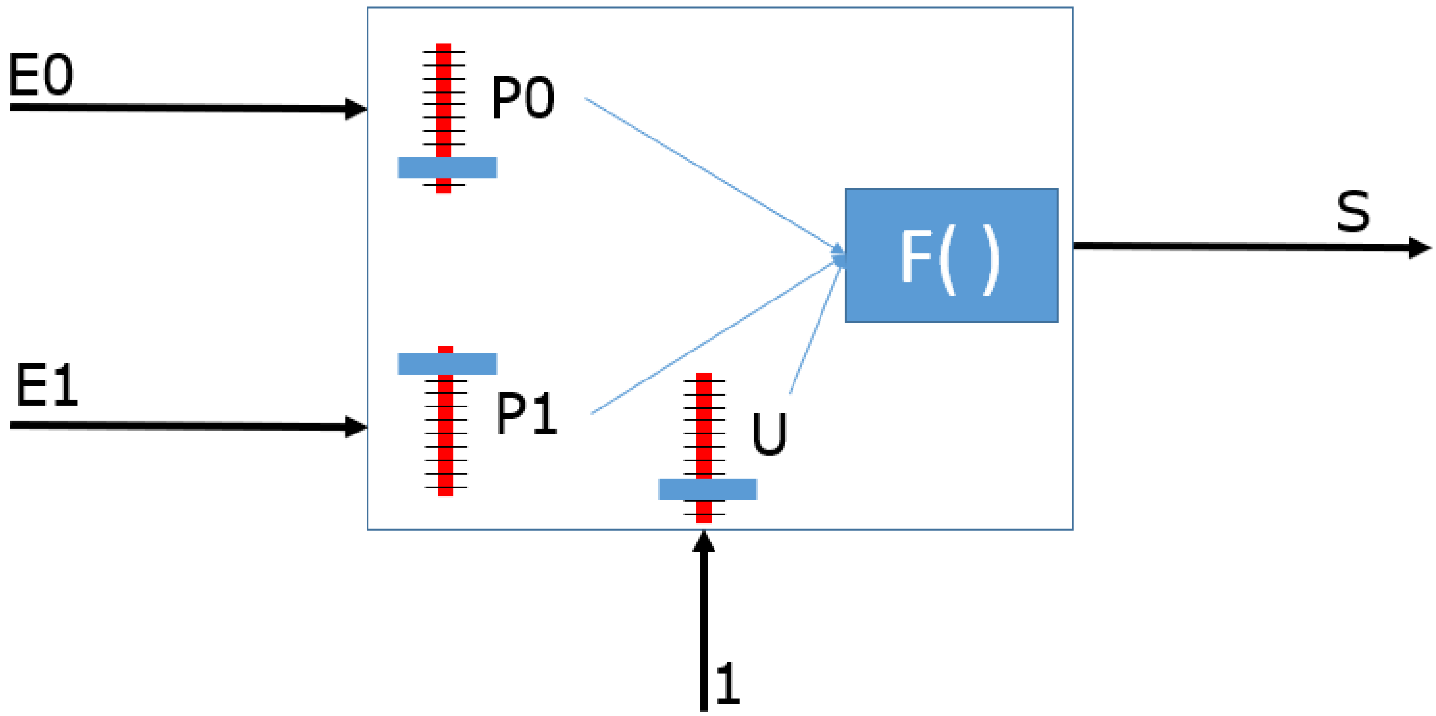


Ilustración 7: Funcionamiento del perceptrón simple

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 > 0.5 entonces
             retorne 1
     de lo contrario
             retorne 0
     fin si
Fin
```

Continuando con el ejemplo:

S = F (1.4462)
S = 1

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

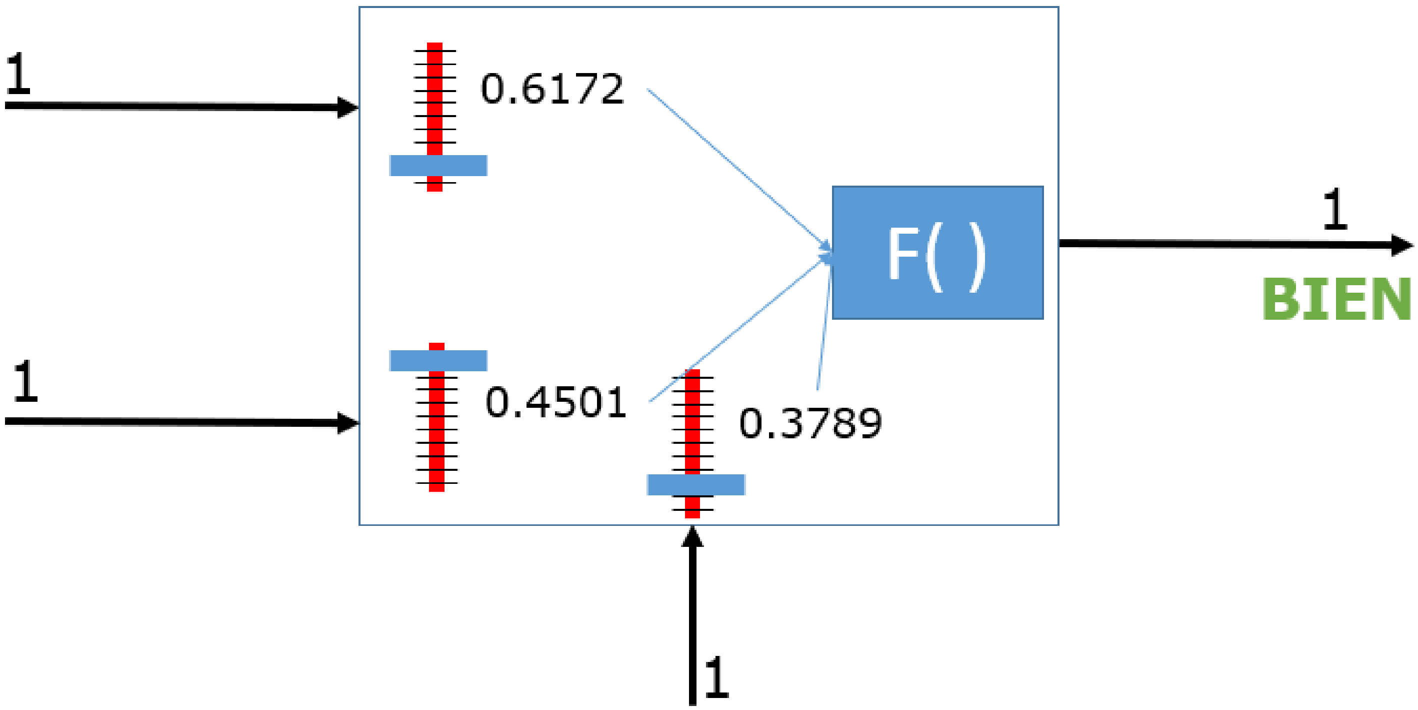


Ilustración 8: Los pesos funcionan para esa regla

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

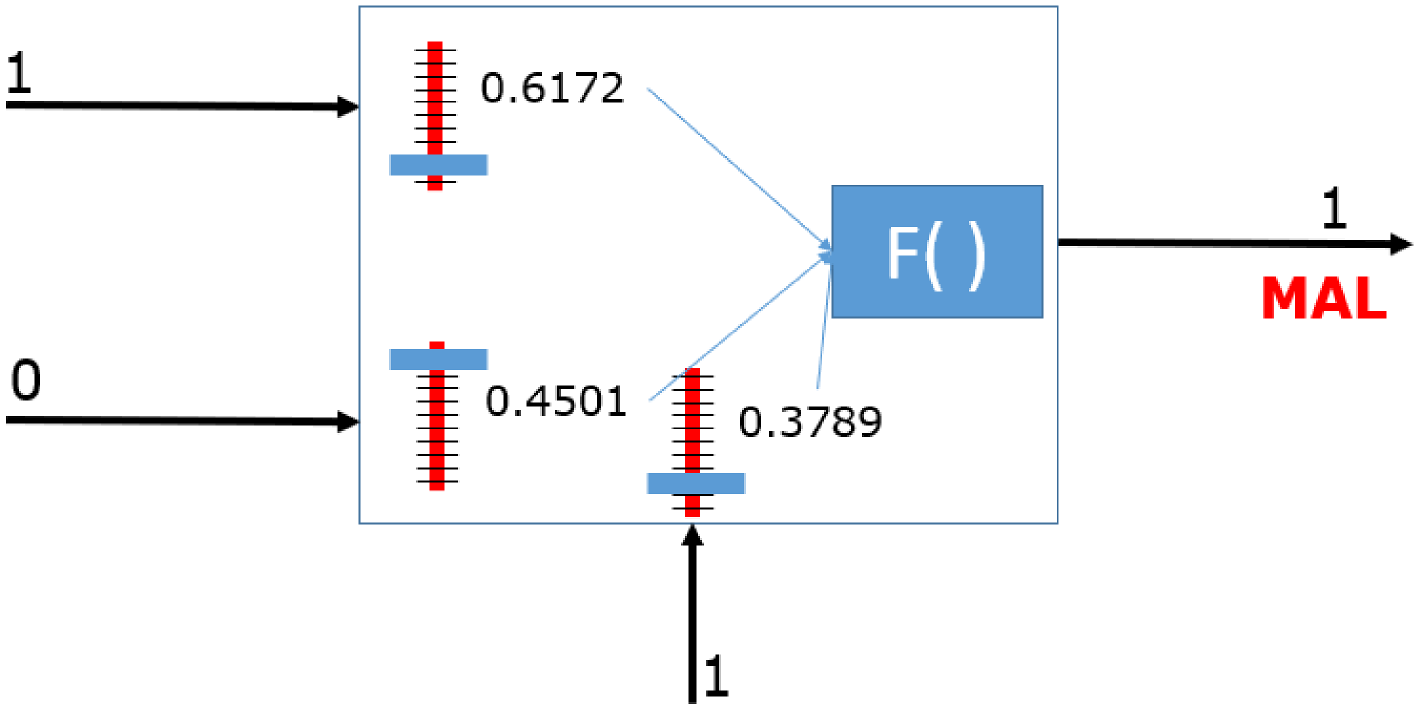


Ilustración 9: Esos pesos fallan con la segunda regla

¿Qué sigue? 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.

K/001.cs

```
namespace Ejemplo {
   internal class Program {
      static void Main() {
         //Único generador de números aleatorios
         Random Azar = new();

         //Tabla AND
         int[][] Entra = [
            [1, 1],
            [1, 0],
            [0, 1],
            [0, 0]
         ];
         int[] Sale = [1, 0, 0, 0];

         //Los pesos
         double P0, P1, U;

         //Mantiene el proceso activo
```

```
bool Proceso;

//Número de iteraciones
int Iteracion = 0;

//Hasta que aprenda la tabla AND
do {
   Iteracion++;

   //Pesos al azar
   P0 = Azar.NextDouble();
   P1 = Azar.NextDouble();
   U = Azar.NextDouble();

   //Prueba la tabla AND
   Proceso = false;
   for (int Con = 0; Con < Entra.GetLength(0); Con++) {

      //Calcula el valor de entrada a la función
      double Oper = Entra[Con][0] * P0 + Entra[Con][1] * P1 + U;

      //Función de activación
      int Salida = Oper > 0.5 ? 1 : 0;

      //Si la salida no coincide con lo esperado,
      //cambia los pesos
      if (Salida != Sale[Con]) {
         Proceso = true;
         break;
      }
   }
} while (Proceso);

//Muestra aprendizaje perceptrón simple
for (int Cont = 0; Cont < Entra.GetLength(0); Cont++) {
   double Oper = Entra[Cont][0] * P0 + Entra[Cont][1] * P1 + U;

   //Función de activación
   int Salida = Oper > 0.5 ? 1 : 0;

   Console.Write("Entradas: " + Entra[Cont][0]);
   Console.Write(" y " + Entra[Cont][1] + " = ");
   Console.WriteLine(Sale[Cont] + " red: " + Salida);

}

Console.Write("Pesos encontrados P0= " + P0);
Console.WriteLine(" P1= " + P1 + " U= " + U);
```

```
            Console.WriteLine("Total Iteraciones: " + Iteracion);
        }
    }
}
```

```
Consola de depuración de Mi
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= 0,1864582462399923 P1= 0,32470435638036177 U= 0,1552258204430117
Total Iteraciones: 13
```

Ilustración 10: Dar con los pesos con sólo azar

```
Consola de depuración de Mi
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= 0,3709107516082192 P1= 0,22883767769211982 U= 0,04995841495690845
Total Iteraciones: 106
```

Ilustración 11: Dar con los pesos con sólo azar

```
Consola de depuración de Mi
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= 0,2729037576898937 P1= 0,3652514916457045 U= 0,12023829796275443
Total Iteraciones: 65
```

Ilustración 12: Dar con los pesos con sólo azar

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 se observa que en una ejecución requirió 106 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 se 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). A continuación, el código en C#

K/002.cs

```
namespace Ejemplo {
   internal class Program {
      static void Main() {
         //Único generador de números aleatorios
         Random Azar = new();

         //Tabla AND
         int[][] Entra = [
            [1, 1],
            [1, 0],
            [0, 1],
            [0, 0]
         ];
         int[] Sale = [1, 0, 0, 0];

         //Los pesos
         double P0, P1, U;

         //Mantiene el proceso activo
         bool Proceso;

         //Número de iteraciones
         int Iteracion = 0;

         //Tasa de aprendizaje
         double TasaAprende = 0.3;

         //Pesos inician al azar
         P0 = Azar.NextDouble();
         P1 = Azar.NextDouble();
```

```
        U = Azar.NextDouble();

        //Hasta que aprenda la tabla AND
        do {
           Iteracion++;

           //Prueba la tabla AND
           Proceso = false;
           for (int Cont = 0; Cont < Entra.GetLength(0); Cont++) {

              //Calcula el valor de entrada a la función
              double Oper = Entra[Cont][0] * P0 + Entra[Cont][1] * P1 + U;

              //Función de activación
              int Salida = Oper > 0.5 ? 1 : 0;

              //El error
              int Error = Sale[Cont] - Salida;

              //Si hay error, cambia los pesos con
              //la Tasa de Aprendizaje
              if (Error != 0) {
                 P0 += TasaAprende * Error * Entra[Cont][0];
                 P1 += TasaAprende * Error * Entra[Cont][1];
                 U += TasaAprende * Error * 1;
                 Proceso = true;
              }
           }
        } while (Proceso);

        //Muestra aprendizaje perceptrón simple
        for (int Cont = 0; Cont < Entra.GetLength(0); Cont++) {
           double Oper = Entra[Cont][0] * P0 + Entra[Cont][1] * P1 + U;

           //Función de activación
           int Salida = Oper > 0.5 ? 1 : 0;

           Console.Write("Entradas: " + Entra[Cont][0]);
           Console.Write(" y " + Entra[Cont][1] + " = ");
           Console.WriteLine(Sale[Cont] + " red: " + Salida);
        }

        Console.Write("Pesos encontrados P0= " + P0);
        Console.WriteLine(" P1= " + P1 + " U= " + U);
        Console.WriteLine("Total Iteraciones: " + Iteracion);
     }
  }
}
```

```
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= 0,5218730868070858 P1= 0,6514357002751983 U= -0,4271222277132198
Total Iteraciones: 7
```

Ilustración 13: Encontrando los pesos más rápido

```
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= 0,42514198541349996 P1= 0,6067804235862224 U= -0,3399274573373506
Total Iteraciones: 7
```

Ilustración 14: Encontrando los pesos más rápido

```
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= 0,5705308605404625 P1= 0,5039279097470348 U= -0,5078508611506496
Total Iteraciones: 3
```

Ilustración 15: Encontrando los pesos más rápido

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

Perceptrón simple: Aprendiendo la tabla del OR

El ejemplo anterior el perceptrón simple aprendía la tabla AND, ¿y 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 sólo cambiar estas líneas del programa:

```
//Tabla OR
int[][] Entradas = new int[][] {
    new int[] {1, 1},
    new int[] {1, 0},
    new int[] {0, 1},
    new int[] {0, 0}
};
int[] Salidas = new int[] { 1, 1, 1, 0 };
```

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:

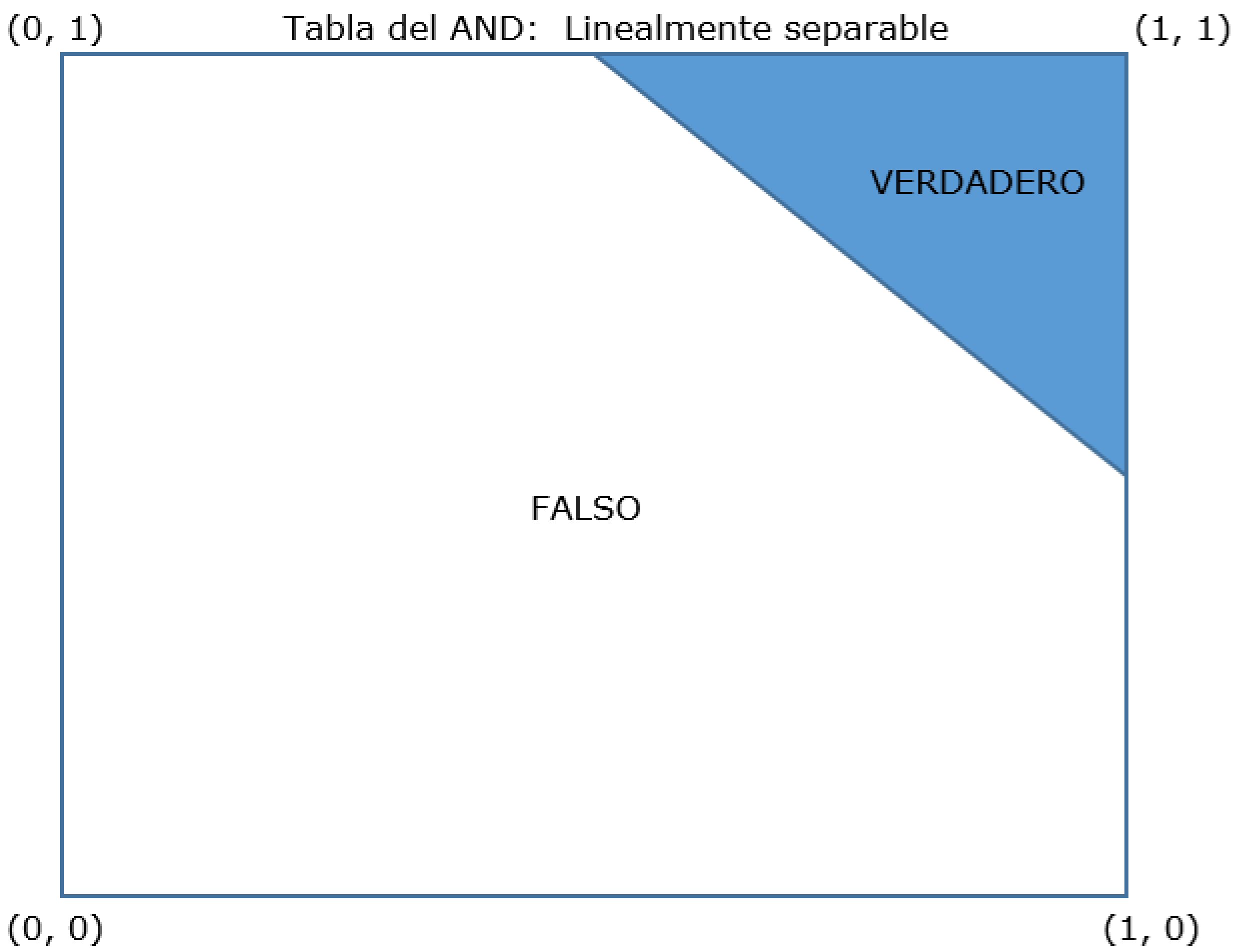


Ilustración 16: Tabla del AND

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

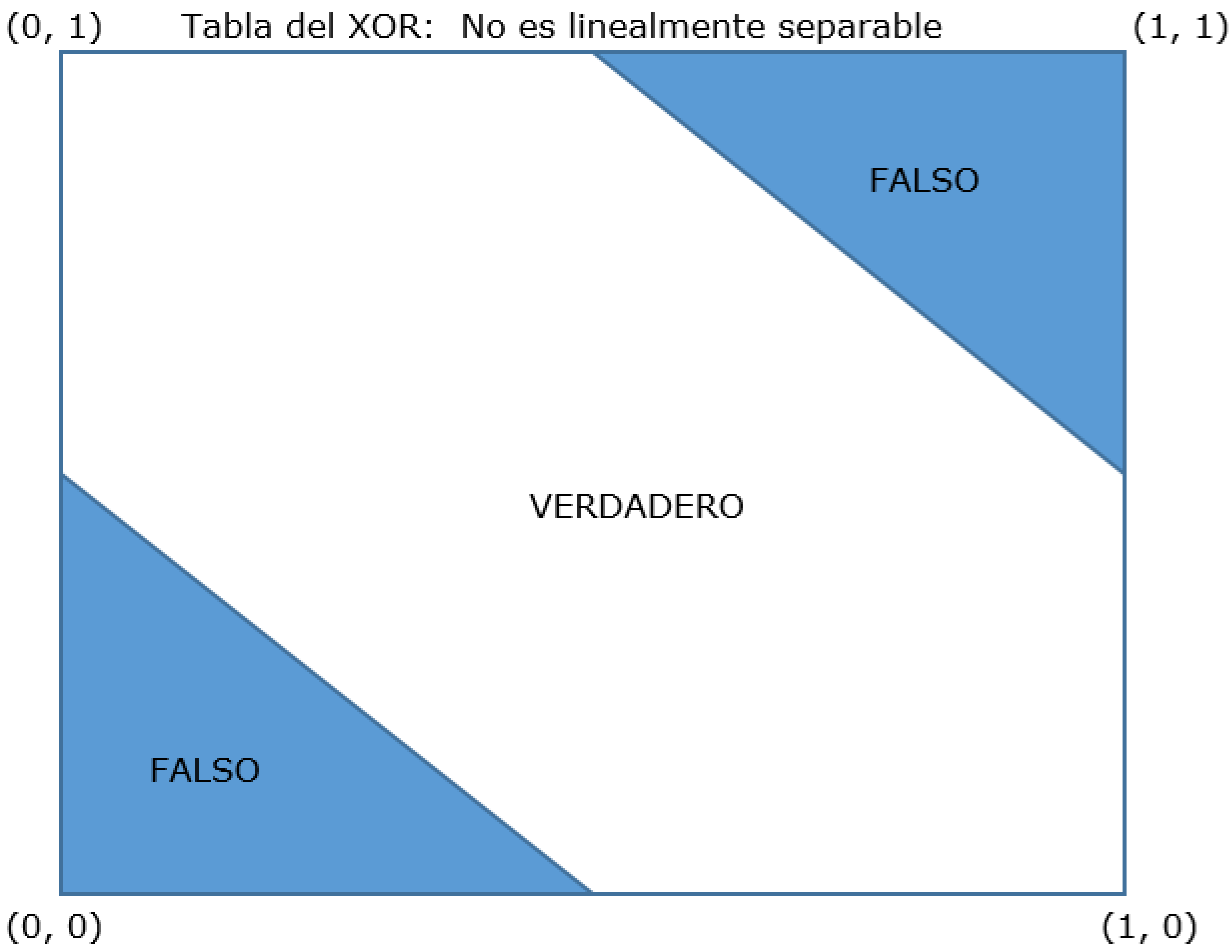


Ilustración 17: Tabla del XOR

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

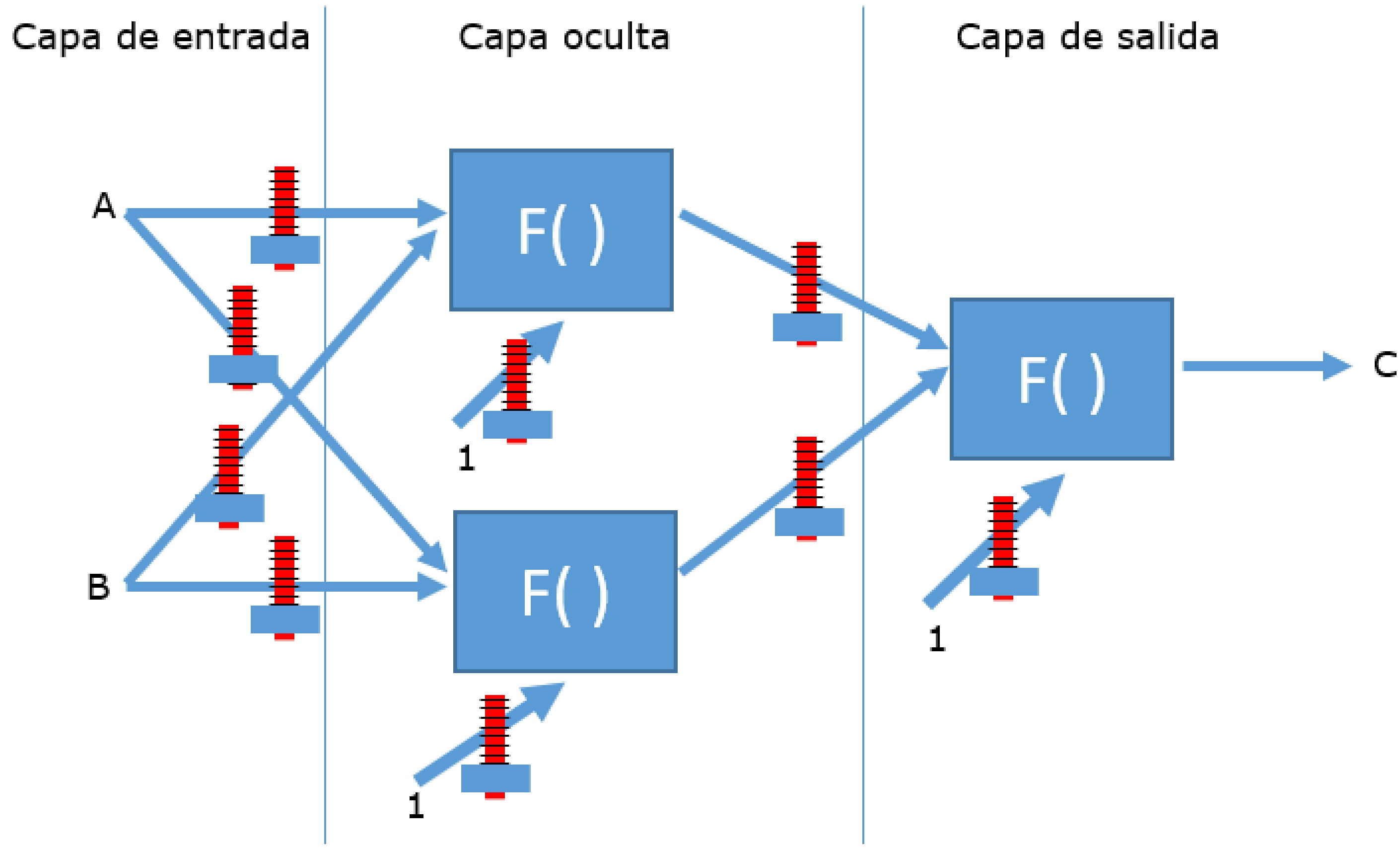


Ilustración 18: Red neuronal con 3 neuronas

O así

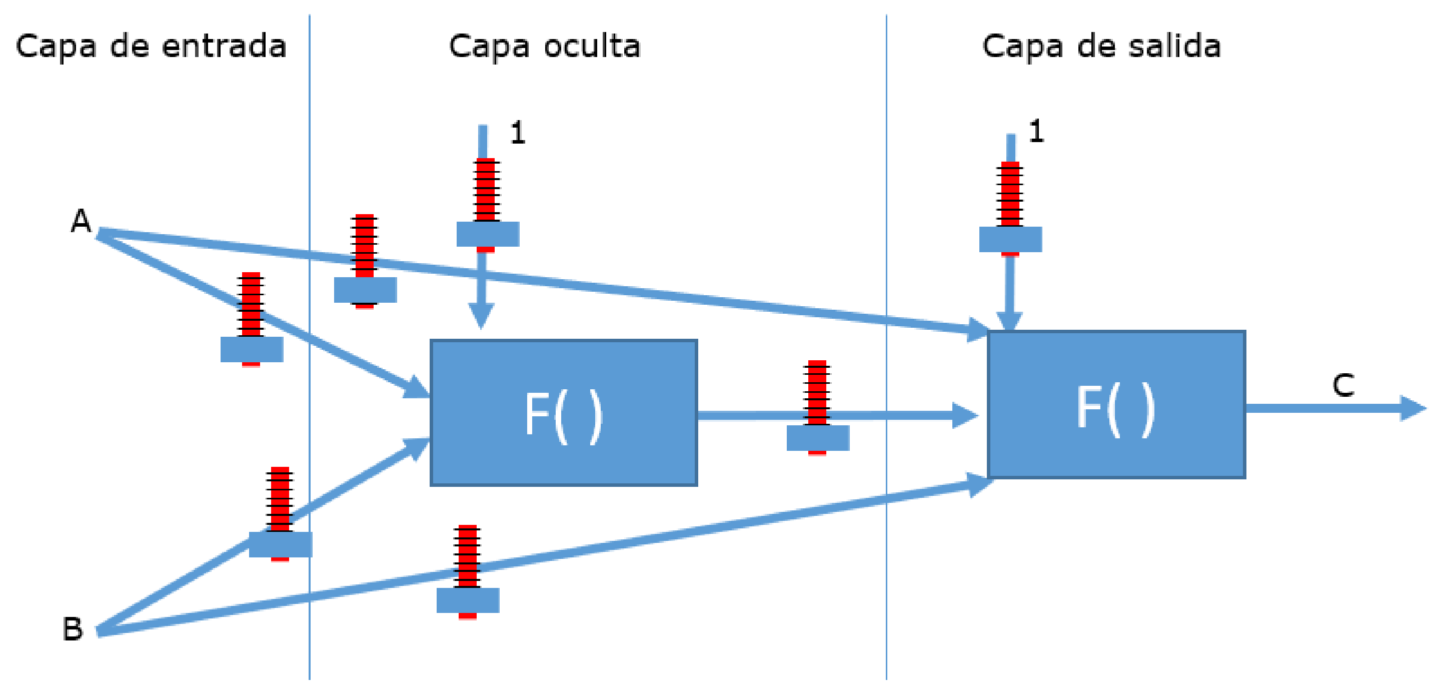


Ilustración 19: Red neuronal con otro tipo de conexiones

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.

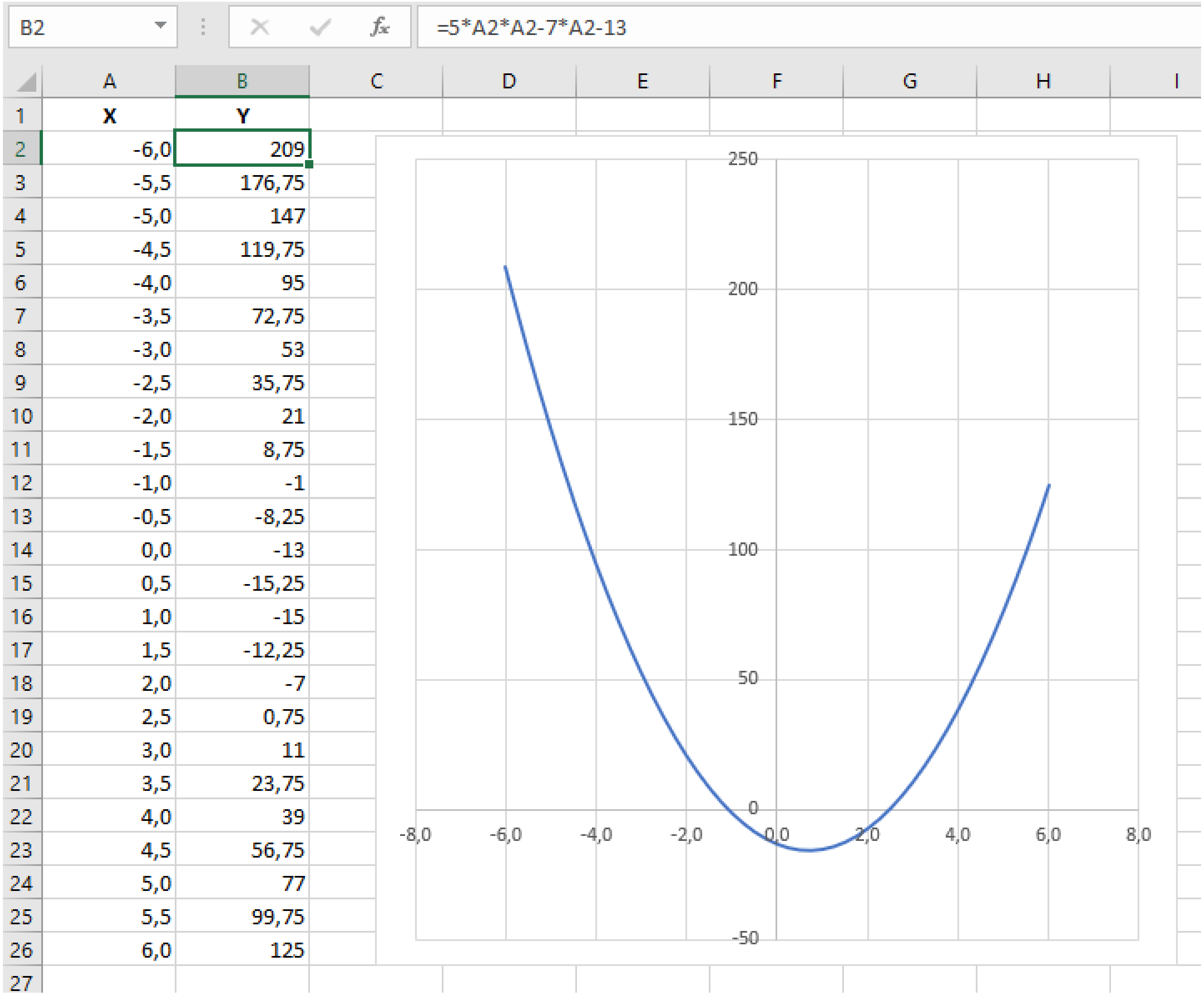
Encontrando el mínimo en una ecuación

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



X	Y
-6,0	209
-5,5	176,75
-5,0	147
-4,5	119,75
-4,0	95
-3,5	72,75
-3,0	53
-2,5	35,75
-2,0	21
-1,5	8,75
-1,0	-1
-0,5	-8,25
0,0	-13
0,5	-15,25
1,0	-15
1,5	-12,25
2,0	-7
2,5	0,75
3,0	11
3,5	23,75
4,0	39
4,5	56,75
5,0	77
5,5	99,75
6,0	125

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

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

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

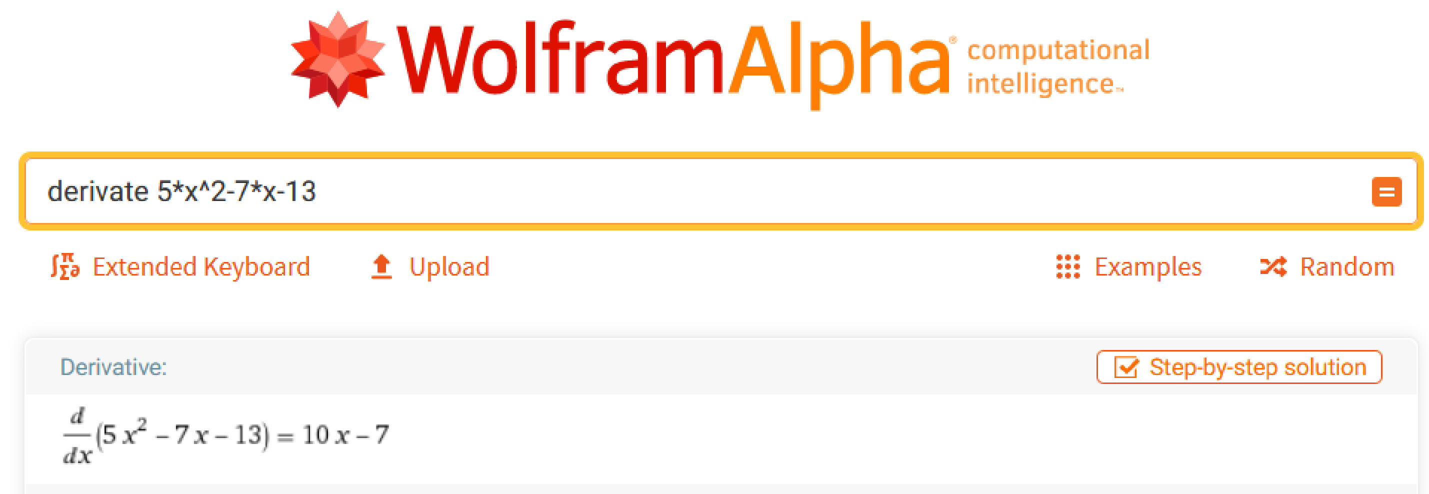


Ilustración 21: Derivada usando en WolframAlpha

Luego esa derivada se iguala a cero

$$0 = 10 * x - 7$$

Se resuelve el valor de x

$$x = 7/10$$

$$x = 0.7$$

Se deduce 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. Esta es su implementación en C#:

K/003.cs

```
namespace Ejemplo {
   internal class Program {
      static void Main() {
         double X = 1; //valor inicial
         double Yini = Ecuacion(X);
         double Variacion = 1;

         while (Math.Abs(Variacion) > 0.00001) {
            double Ysigue = Ecuacion(X + Variacion);
```

```
            //Si en vez de disminuir Y,
            //lo que hace es aumentar,
            //cambia de dirección a un paso menor
            if (Ysigue > Yini) {
               Variacion *= -1;
               Variacion /= 10;
            }
            else { //Está disminuyendo Y
               Yini = Ysigue;
               X += Variacion;
               Console.WriteLine("X: " + X + " Y:" + Yini);
            }
         }
         Console.WriteLine("Respuesta: " + X);
      }

      //Ecuación a analizar
      static double Ecuacion(double X) {
         double Y = 2 * Math.Sin(3 * X - 4);
         Y += 5 * Math.Sin(-4 * X + 5);
         Y -= 4 * Math.Sin(5 * X - 7);
         return Y;
      }
   }
}
```

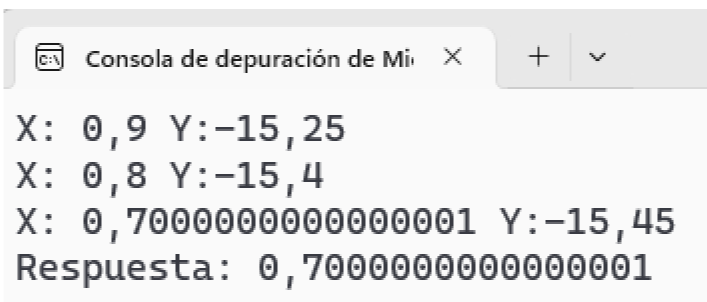


Ilustración 22: Buscando el mínimo

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.

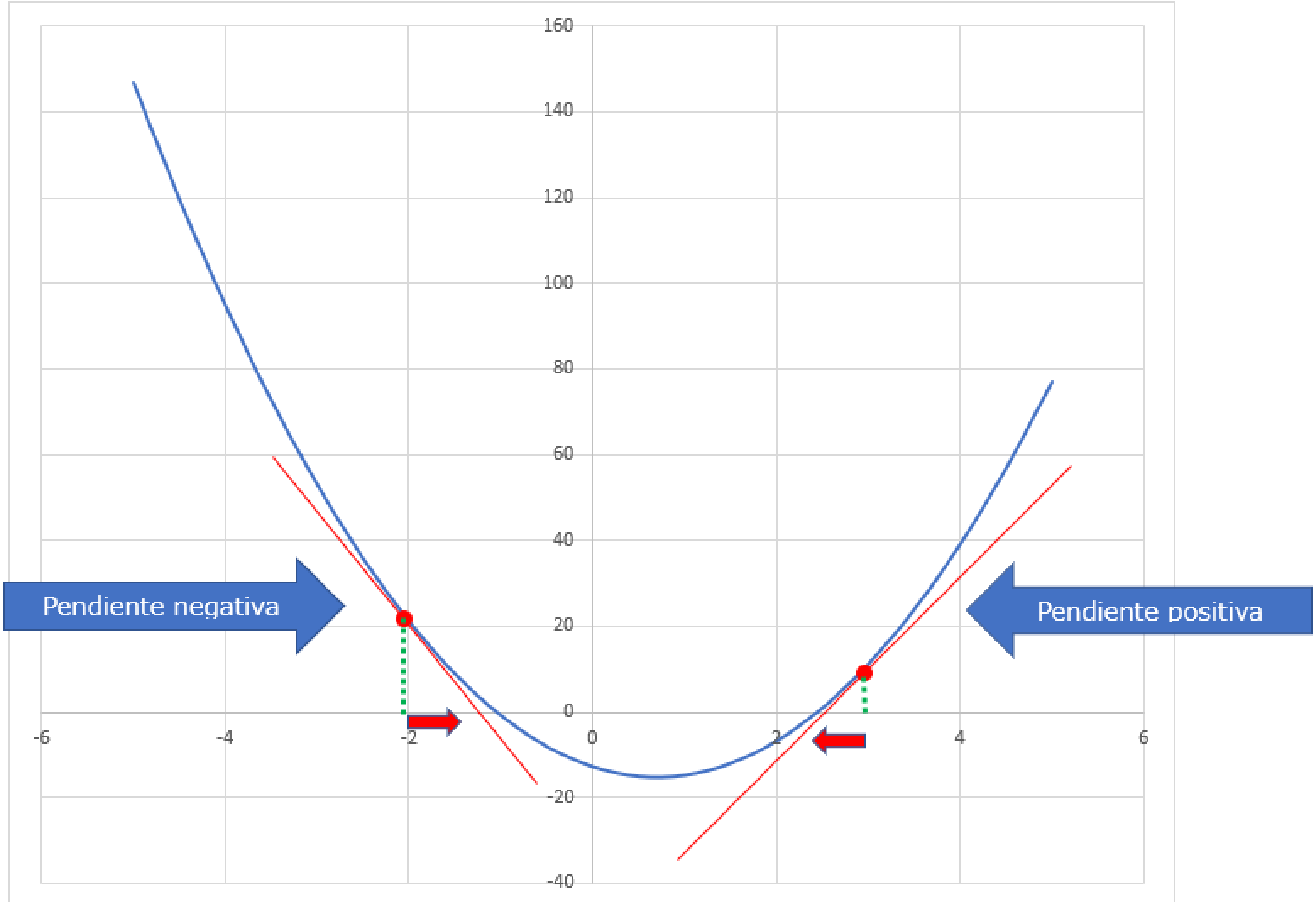


Ilustración 23: Pendientes

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:

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

Se agrega entonces un $\propto$ que es una constante muy pequeña, por ejemplo α=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:

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

En formato matemático

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

Mínimos locales y globales

La siguiente curva es generada por el siguiente polinomio

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

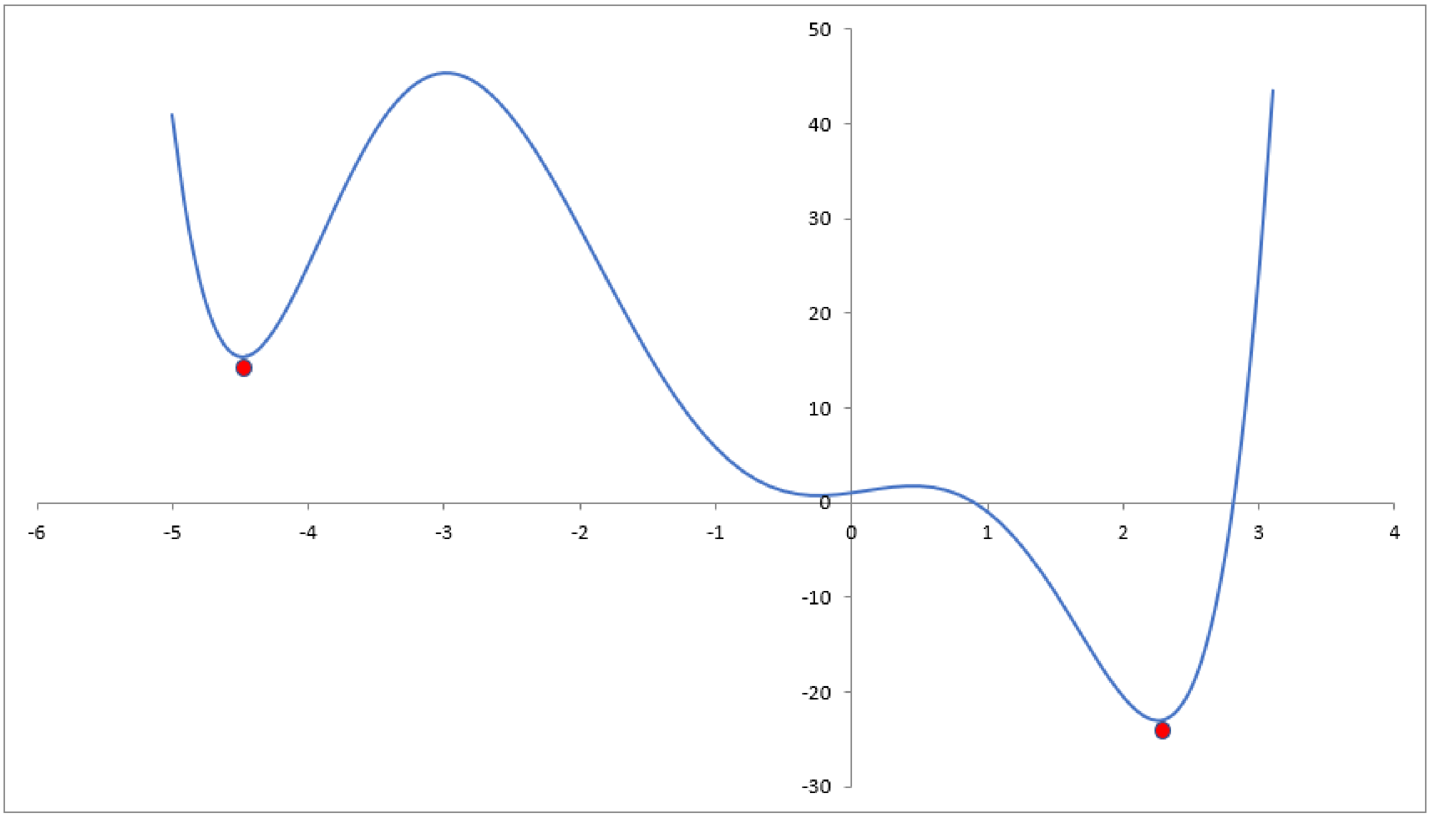


Ilustración 24: Gráfico de un polinomio de quinto grado

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:

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

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

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 evolutivos. Así que se deben probar otros valores de X para iniciar, si fuese X=2 se observa 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)

Búsqueda de mínimos y redes neuronales

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

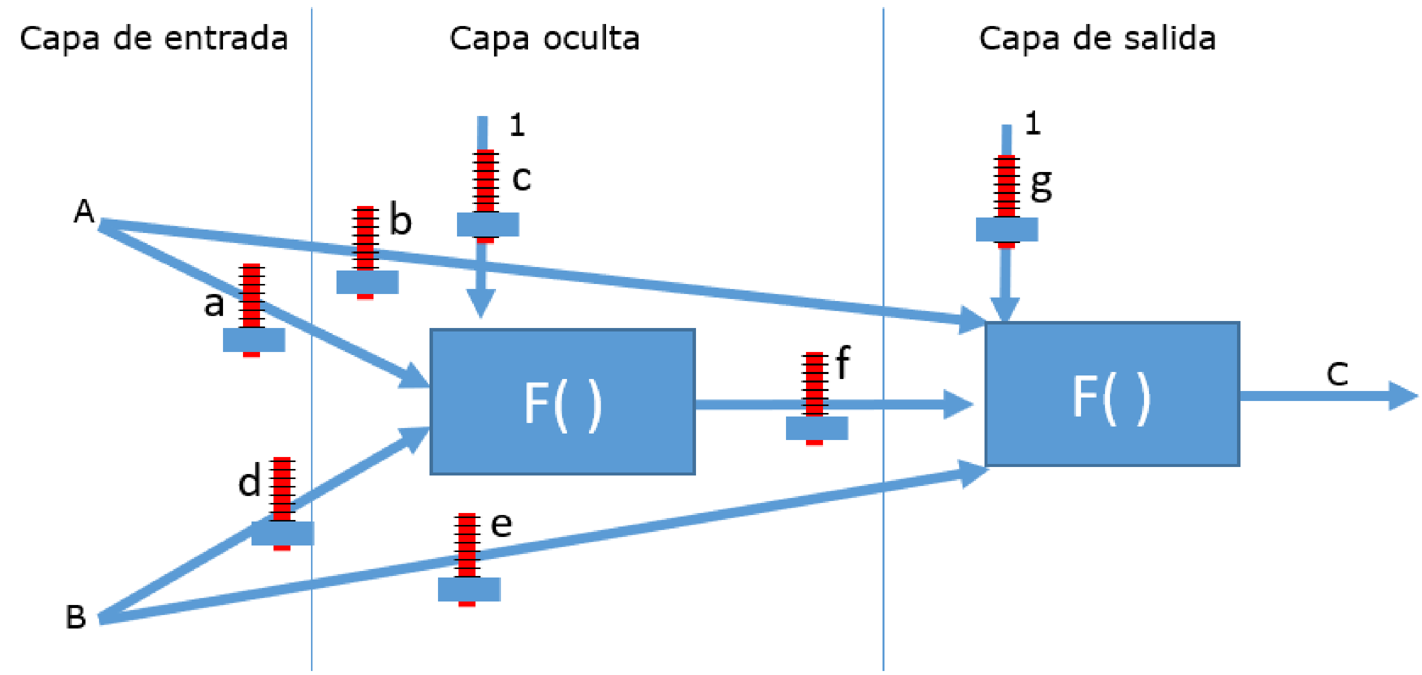


Ilustración 25: Red neuronal con varias conexiones distintas

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.

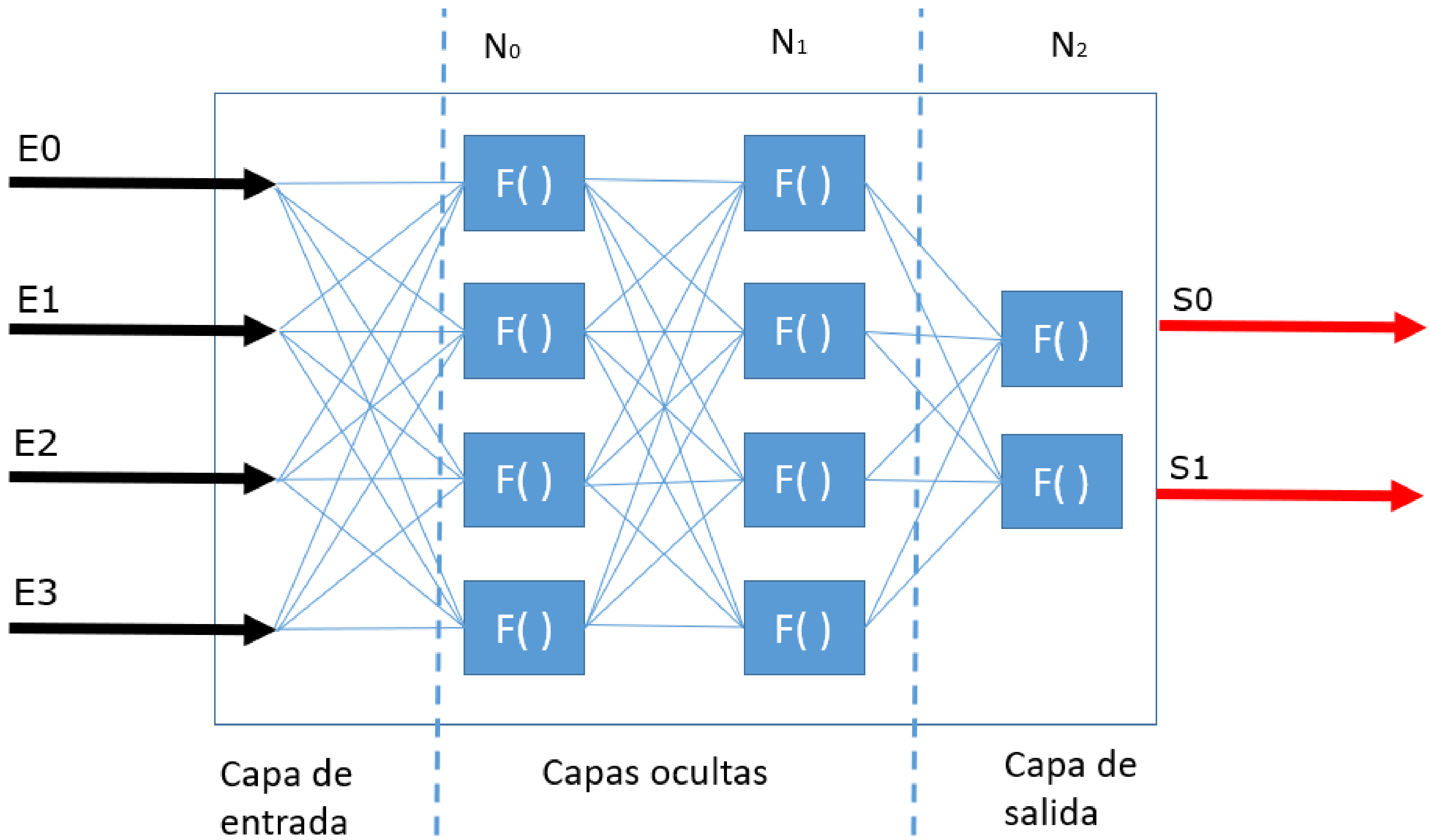


Ilustración 26: Perceptrón multicapa

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.

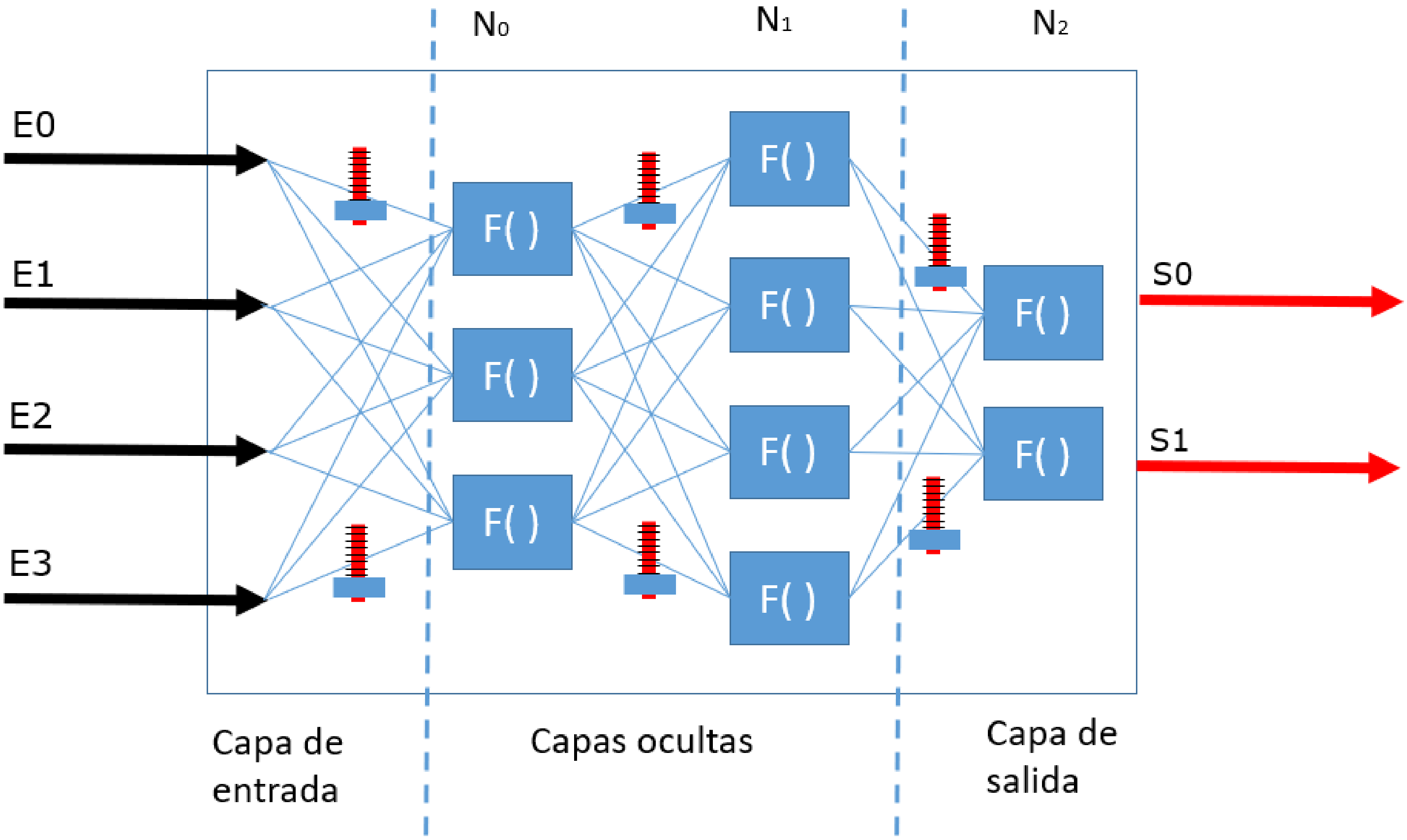


Ilustración 27: Esquema de una neurona

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

K/004.cs

```
namespace Ejemplo {
   class Neurona {
      public double CalculaSalida(double E0, double E1) {
         double S = 0;

         //Se hace una operación aquí

         return S;
      }
   }

   class Program {
      static void Main() {
         Neurona objA = new Neurona();
```

```
        Neurona objB = new Neurona();
      }
   }
}
```

En cada entrada hay un peso P0 y P1. Para la entrada interna, que siempre es 1, el peso se llama U

K/005.cs

```
namespace Ejemplo {
   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 = 0;

         //Se hace una operación aquí

         return S;
      }
   }

   class Program {
      static void Main() {
         Neurona objA = new Neurona();
         Neurona objB = 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.

K/006.cs

```
namespace Ejemplo {
   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 = 0;

         //Se hace una operación aquí

         return S;
      }
   }

   class Program {
      static void Main() {
         Random Azar = new();
         Neurona objA = new Neurona(Azar);
         Neurona objB = 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.

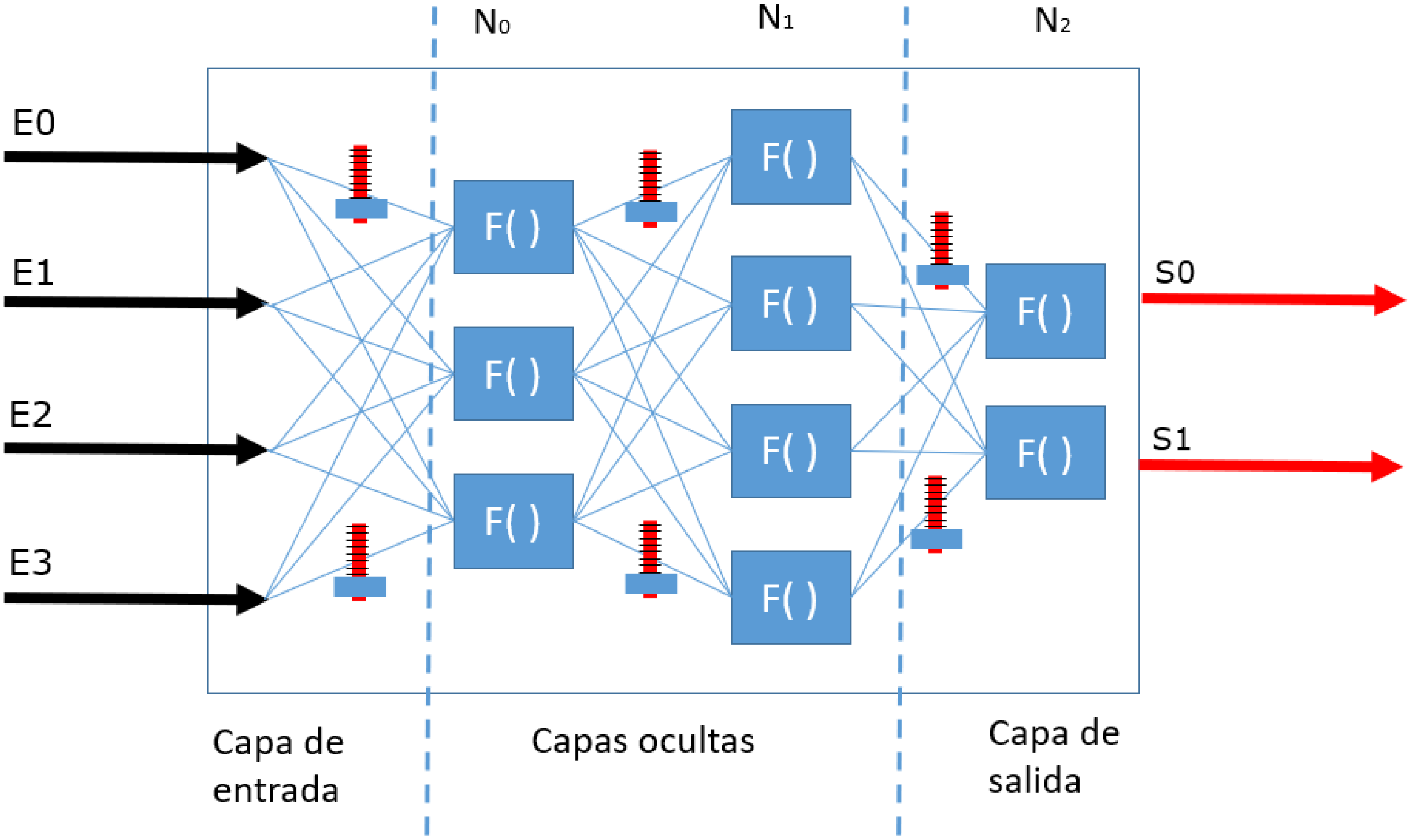


Ilustración 28: Esquema de un perceptrón multicapa

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*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:

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

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.

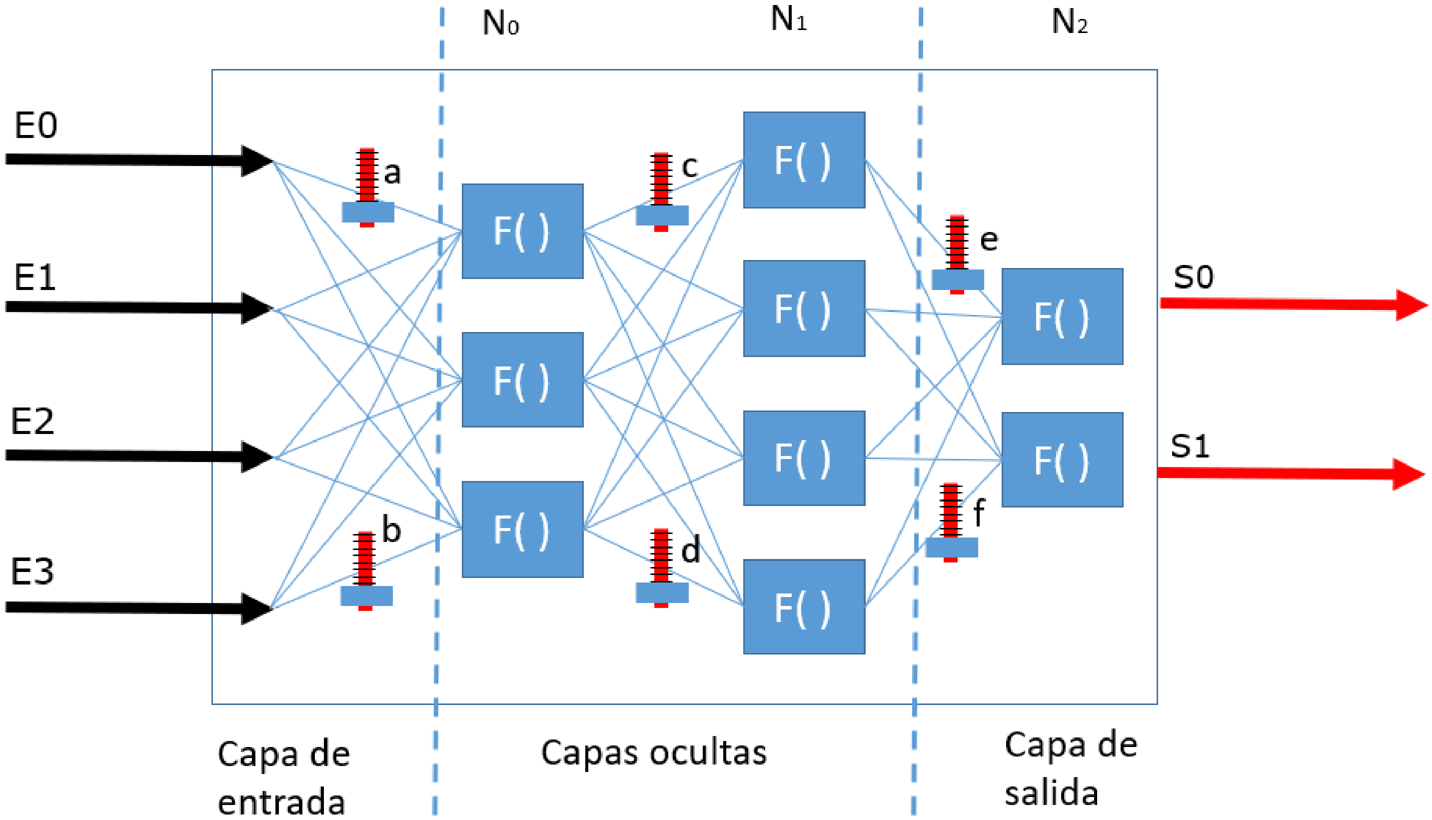


Ilustración 29: Nombrando los pesos con una letra

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

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

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

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	$w_{3,1}^{(2)}$

La función de activación de la neurona

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

```
Función F(valor)
Inicio
   Si valor > 0.5 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

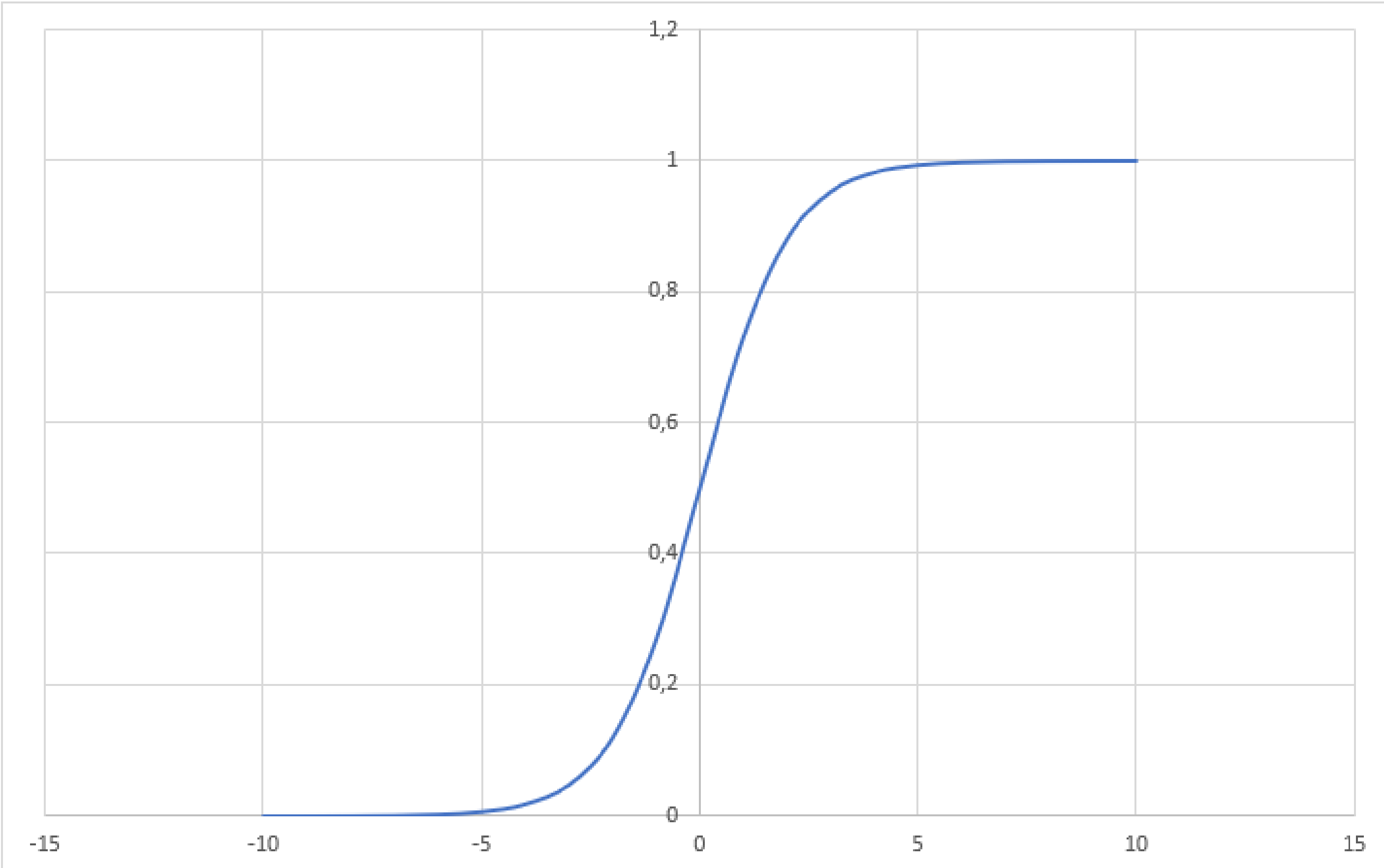


Ilustración 30: Gráfico de una función sigmoide

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:

$$\partial y = \frac{1}{1 + e^{-x}} * \left[1 - \frac{1}{1 + e^{-x}}\right]$$

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

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

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

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

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

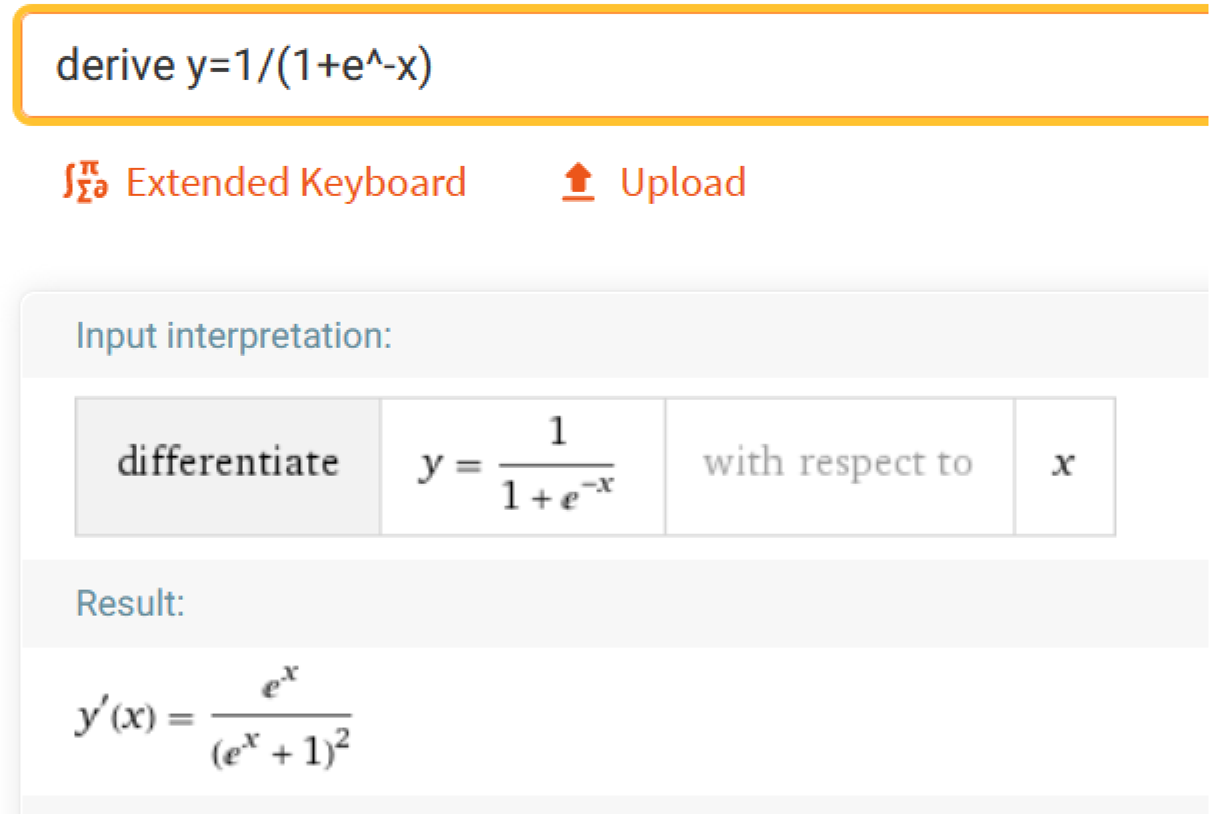


Ilustración 31: Derivada de la función sigmoide

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

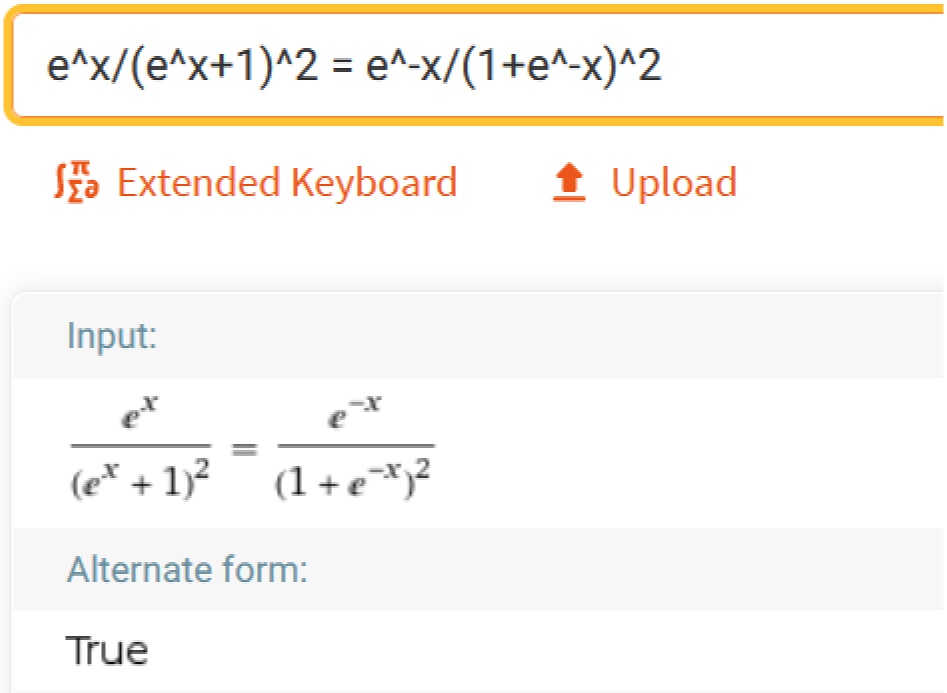


Ilustración 32: Comparativa de derivadas

El código del perceptrón multicapa progresa así:

K/007.cs

```
namespace Ejemplo {
   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();
      }

      //Calcula la salida de la neurona con las dos entradas E0 y E1
      public double CalculaSalida(double E0, double E1) {
         double Valor = E0 * P0 + E1 * P1 + 1 * U;
         return 1 / (1 + Math.Exp(-Valor));
      }
   }

   class Program {
      static void Main() {
         Random Azar = new();
         Neurona objA = new Neurona(Azar);
         Neurona objB = 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 se observa una conexión entre tres neuronas

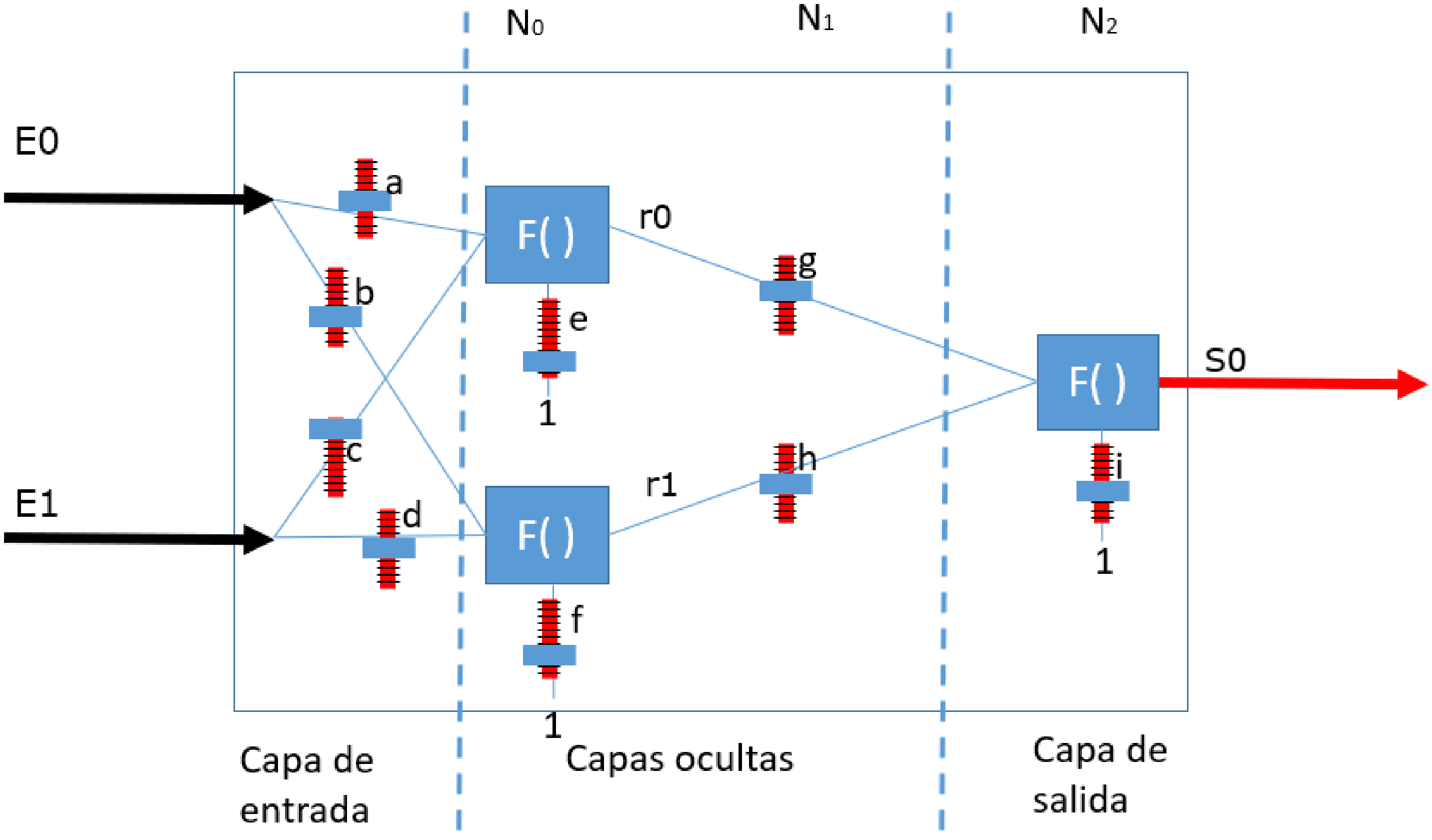


Ilustración 33: Esquema de un perceptrón multicapa

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 se quiere ajustar S0 al resultado esperado, entonces se retrocede a las entradas de esa neurona de la derecha ajustando sus pesos respectivos y por supuesto, ese ajuste hace retroceder más aún hasta mirar los pesos de las neuronas de arriba y abajo. Eso se conocerá como el algoritmo "Backpropagation".

Luego:

$$r0 = F(E0 * a + E1 * c + 1 * e)$$

$$r1 = F(E0 * b + E1 * d + 1 * f)$$

$$S0 = F(r0 * g + r1 * h + 1 * i)$$

Se concluye entonces que:

$$S0 = F(F(E0 * a + E1 * c + 1 * e) * g + F(E0 * b + E1 * d + 1 * f) * h + 1 * i)$$

Nombrando las entradas, pesos, umbrales, salidas y capas

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

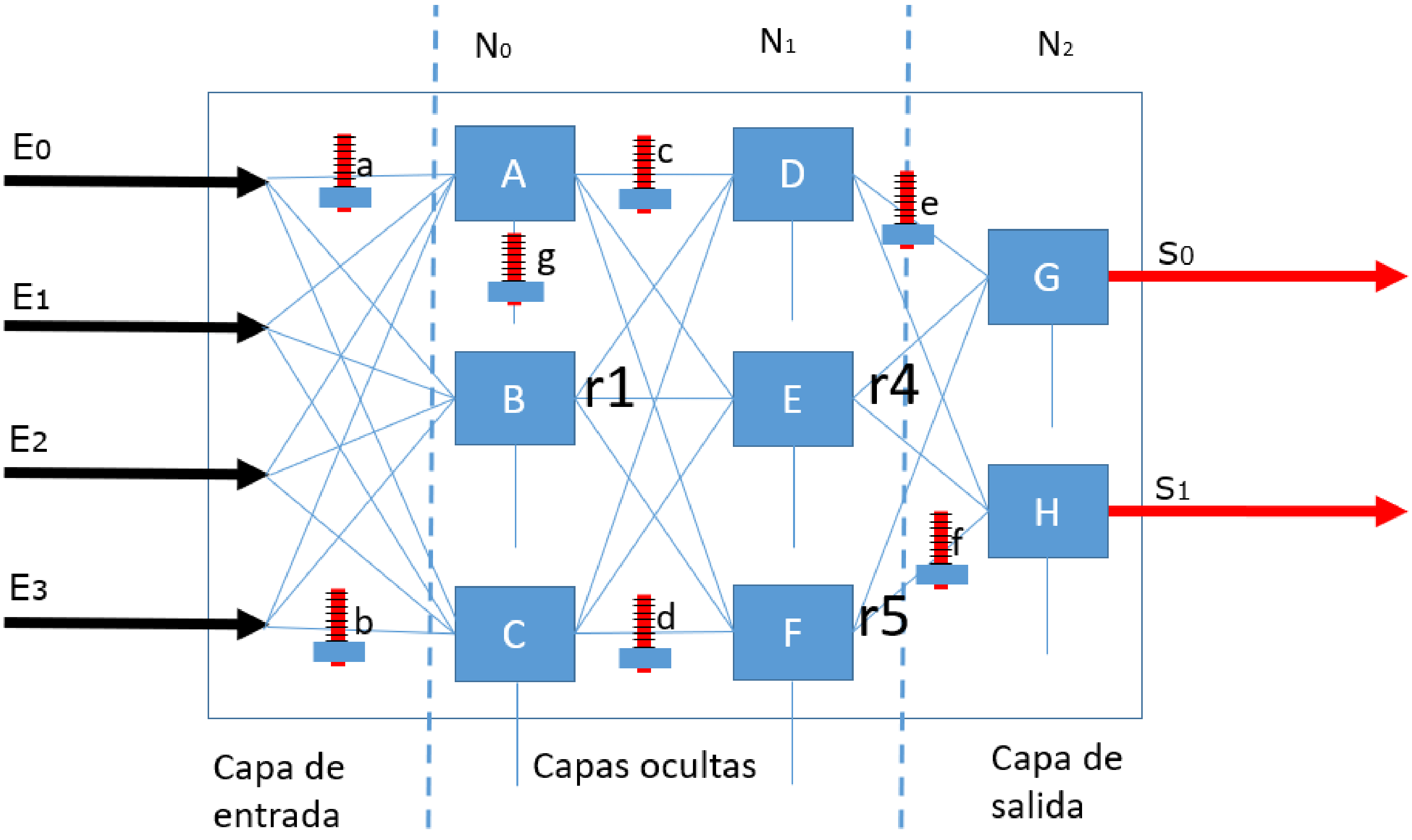


Ilustración 34: Partes de un perceptrón multicapa

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:

$$n_{número\ de\ la\ capa}$$

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 E_0, E_1, E_2 y E_3? 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:

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

$$a_1^{(0)} = F\left(a_0^{(E)} * w_{0,1}^{(0)} + a_1^{(E)} * w_{1,1}^{(0)} + a_2^{(E)} * w_{2,1}^{(0)} + a_2^{(E)} * w_{3,1}^{(0)} + u_1^{(0)}\right)$$

$$a_2^{(0)} = F\left(a_0^{(E)} * w_{0,2}^{(0)} + a_1^{(E)} * w_{1,2}^{(0)} + a_2^{(E)} * w_{2,2}^{(0)} + a_2^{(E)} * w_{3,2}^{(0)} + u_2^{(0)}\right)$$

¡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

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

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

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

La regla de la cadena funciona.

Derivadas parciales

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.

Ejemplo de una ecuación con tres variables independientes:

$$q = a^2 + b^3 + c^4$$

Su derivada parcial con respecto a la variable **a** sería

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

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

Su derivada parcial con respecto a la variable **b** sería

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

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

Su derivada parcial con respecto a la variable **c** sería

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

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

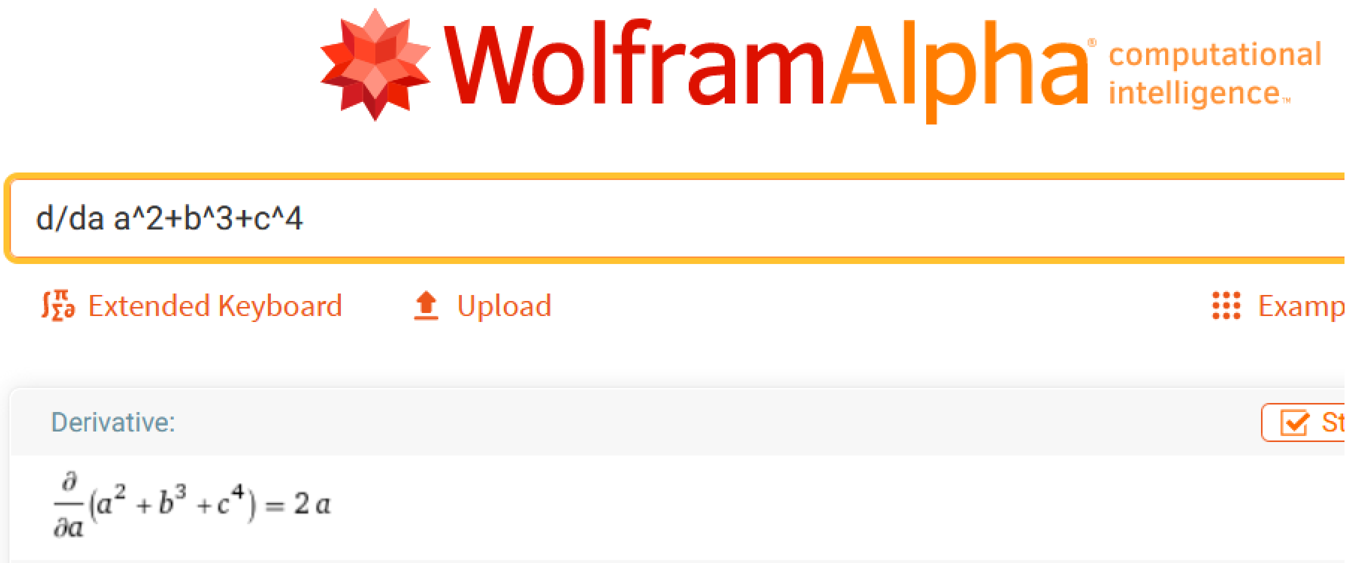


Ilustración 35: Derivada parcial

Las derivadas en el algoritmo de propagación hacia atrás

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

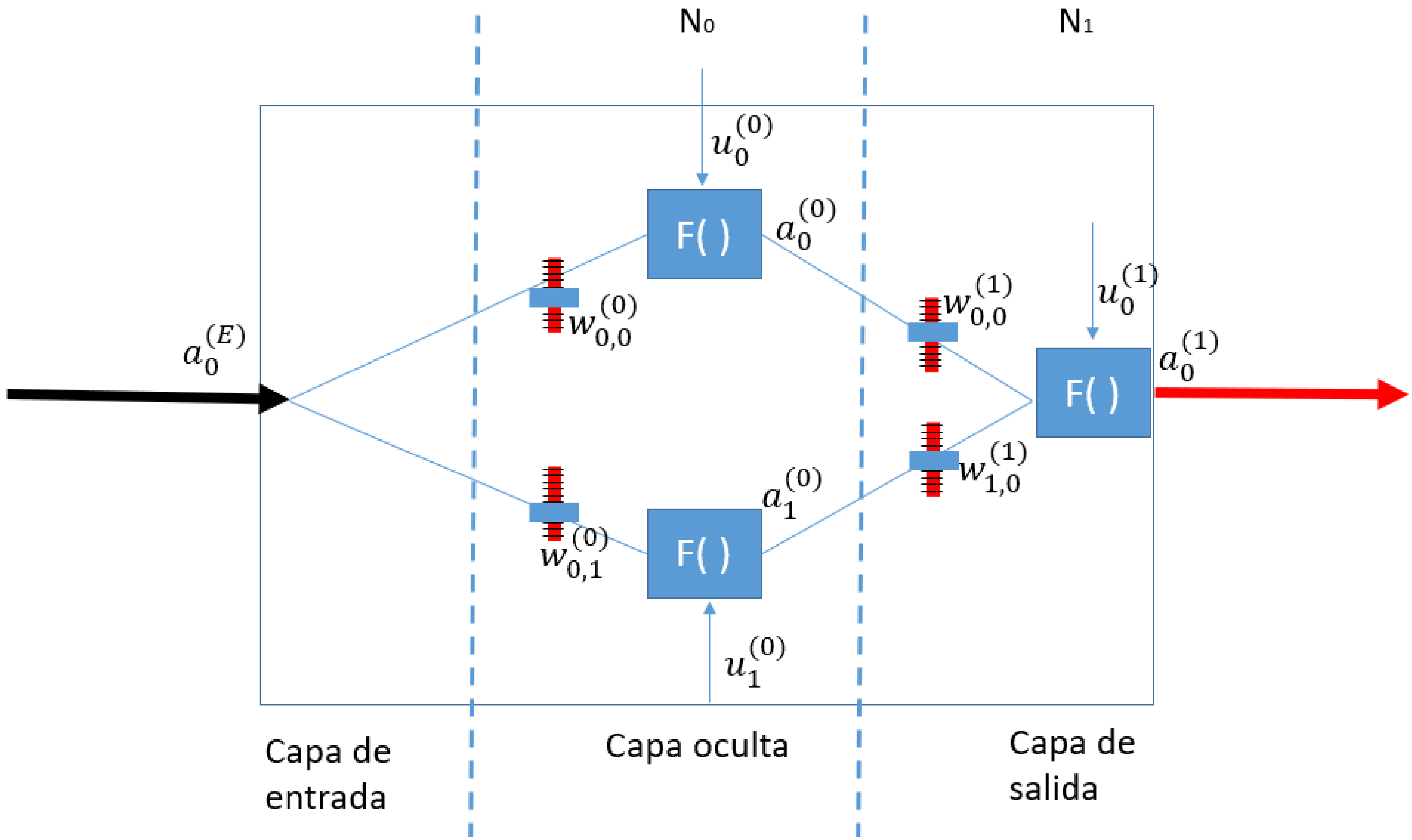


Ilustración 36: Nombramiento de pesos, umbrales y salidas

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

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

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

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

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

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

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

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:

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

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

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

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

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

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

Entonces:

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

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:

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

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

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

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

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

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

Entonces:

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

Generalizando

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

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:

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

Se comporta como una constante

Entonces

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

Y como

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

Entonces

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

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:

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

Entonces

Se comporta como una constante

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

Y como

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

Entonces

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

Generalizando

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

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:

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

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

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

$$\partial k = 1$$

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

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

Entonces:

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

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:

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

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

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

$$\partial k = 1$$

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

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

Entonces:

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

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

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

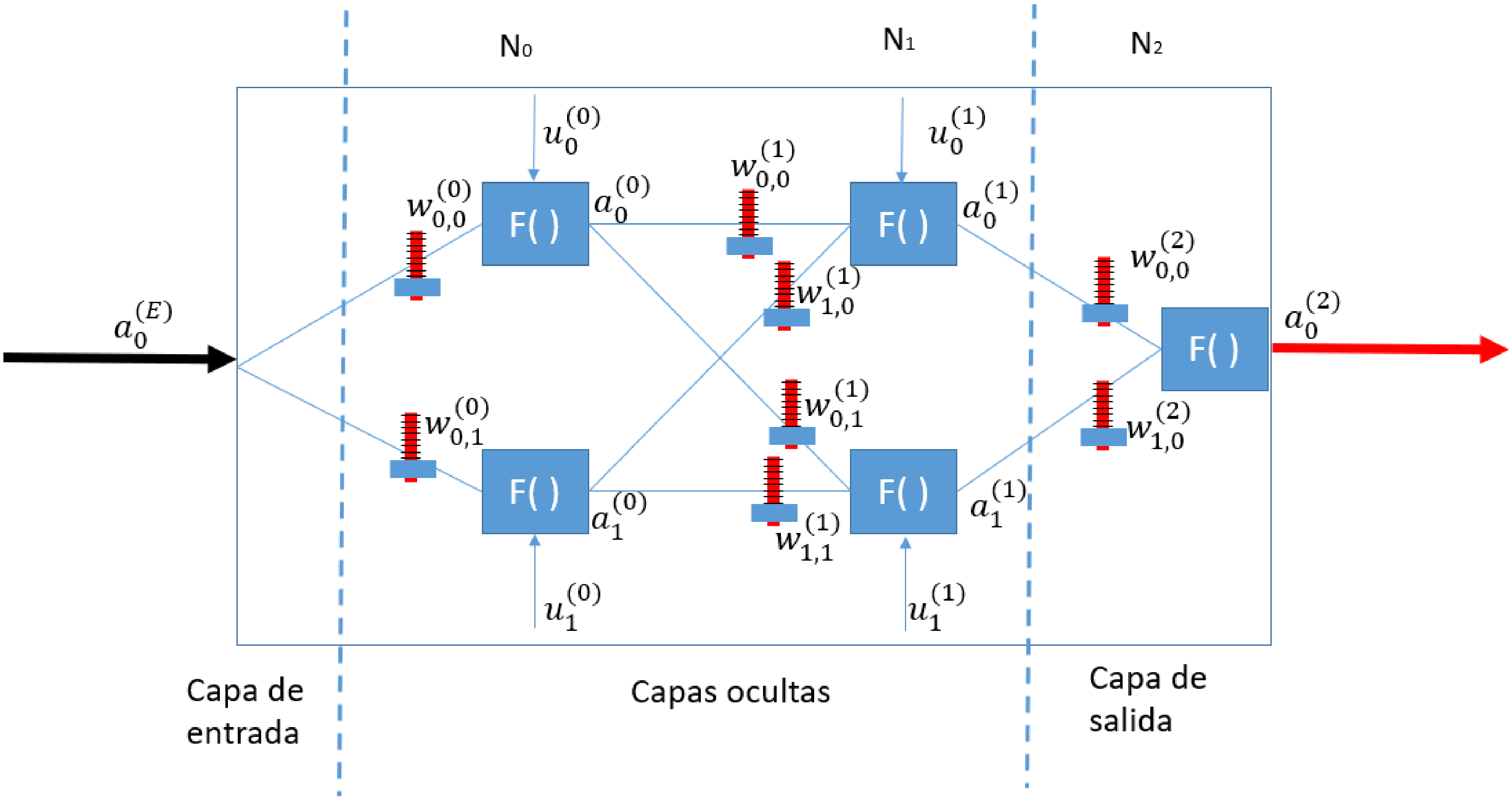


Ilustración 37: Esquema de un perceptrón multicapa

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

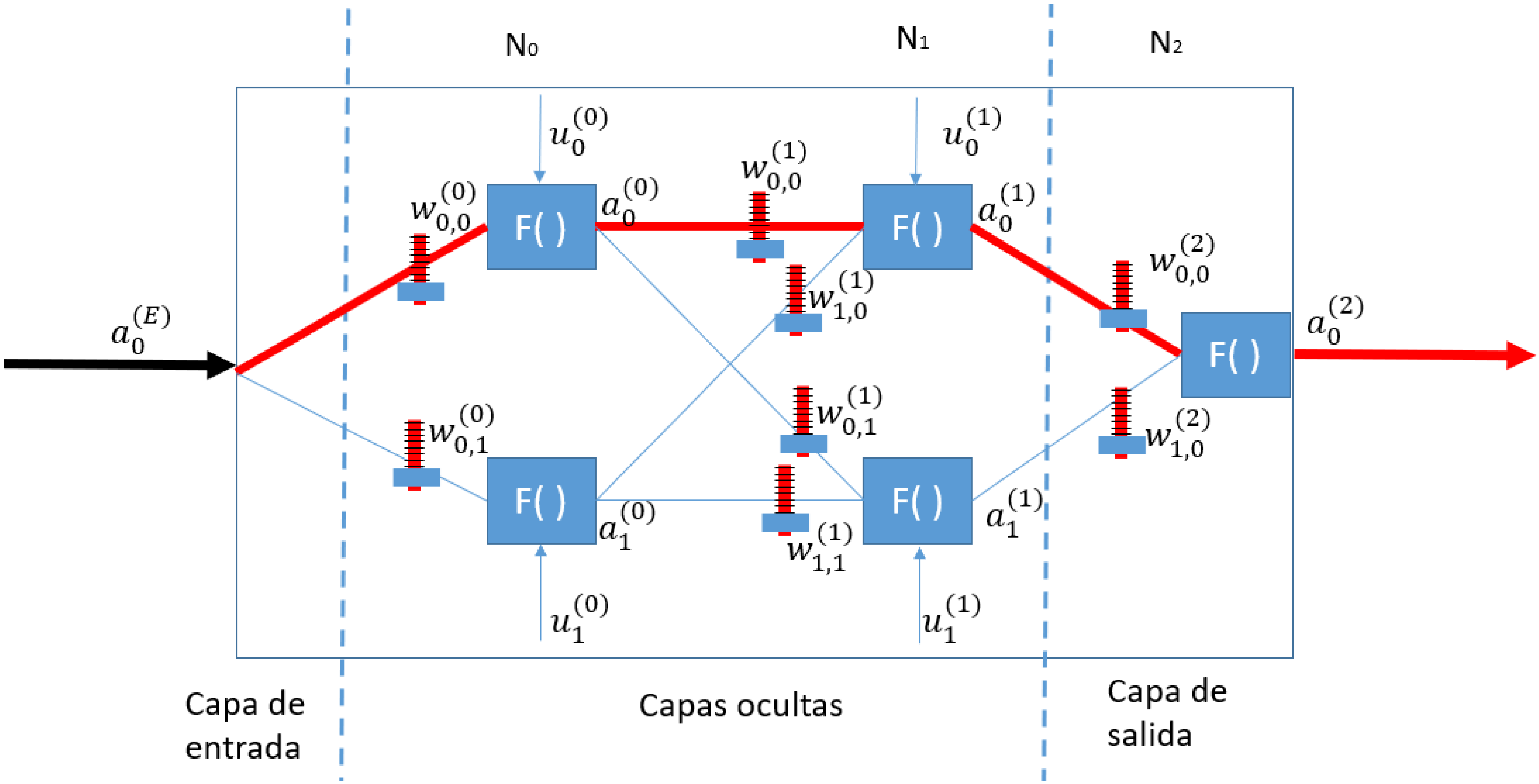


Ilustración 38: Primer camino para ese peso

Y

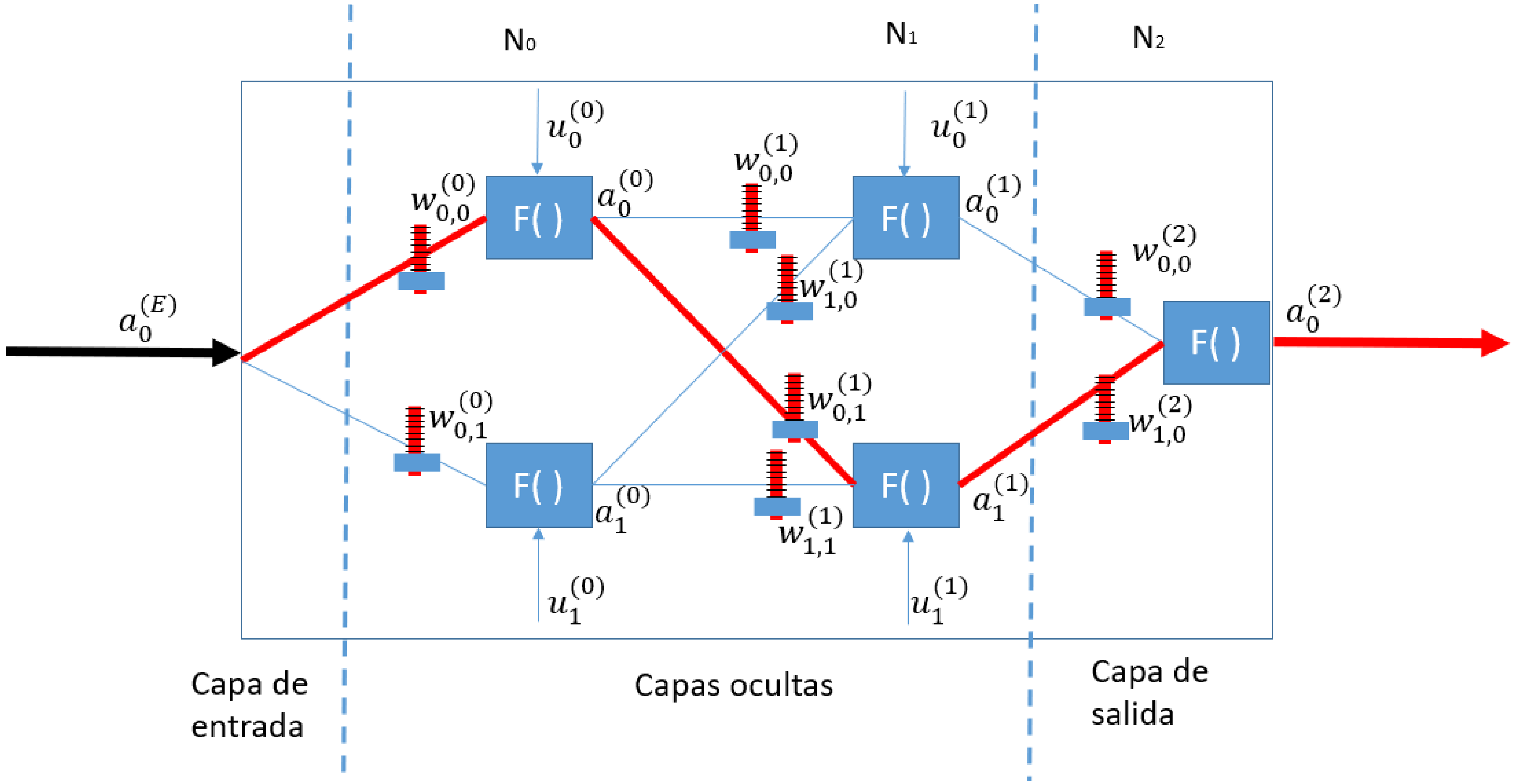


Ilustración 39: Segundo camino para ese peso

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

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

$$+$$

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

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

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

$$+$$

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

Y así poder generalizar

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

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

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

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:

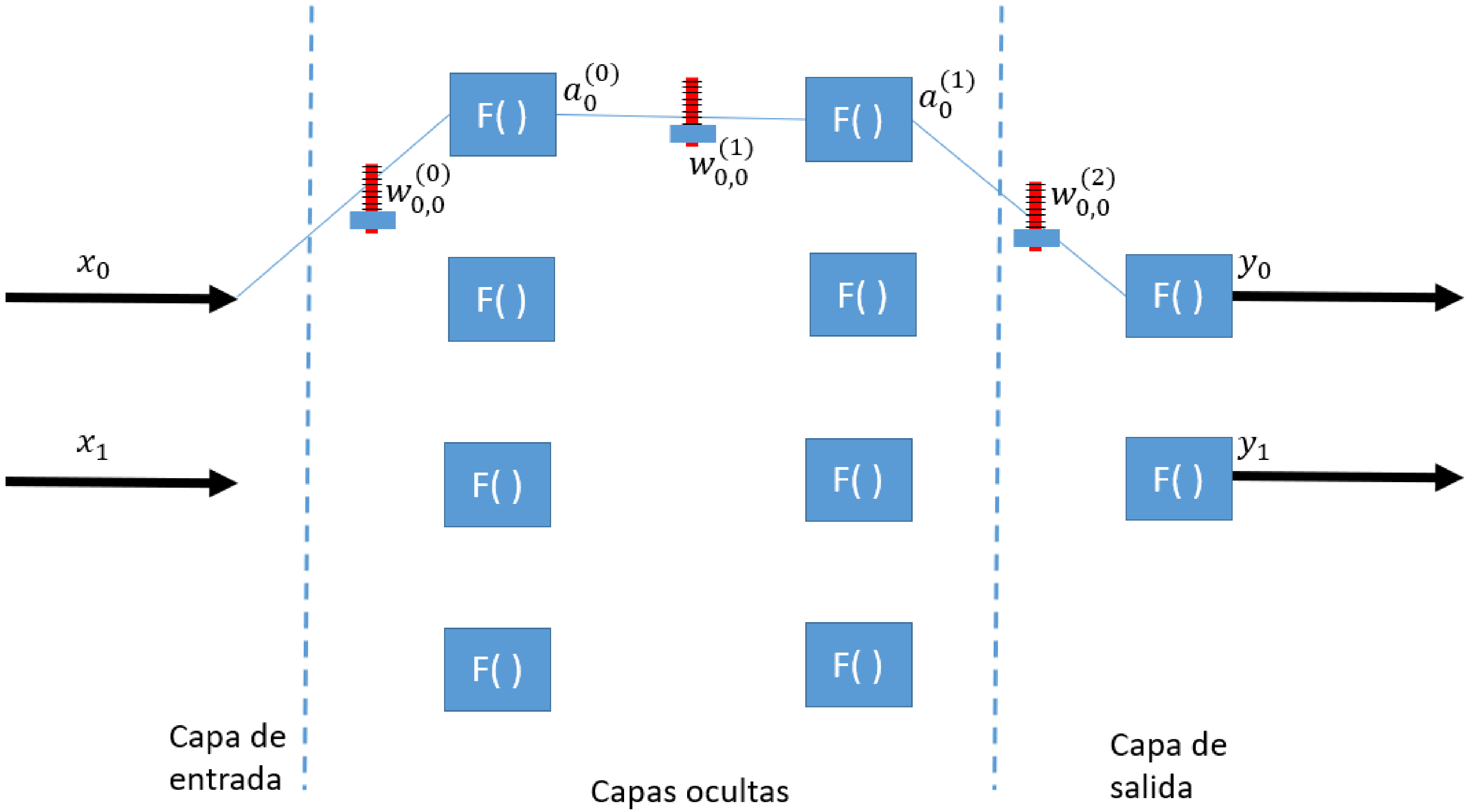


Ilustración 40: Primer camino

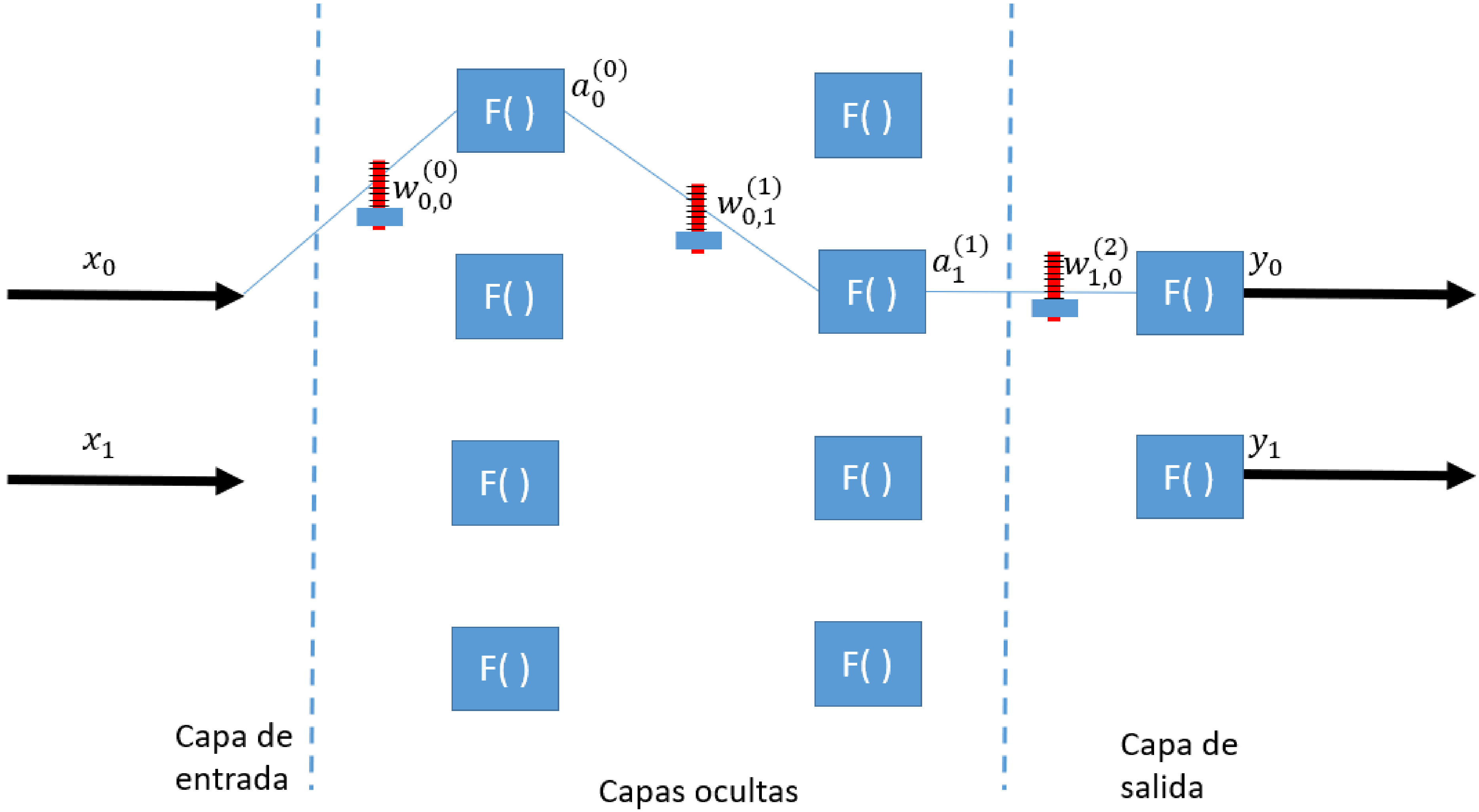


Ilustración 41: Segundo camino

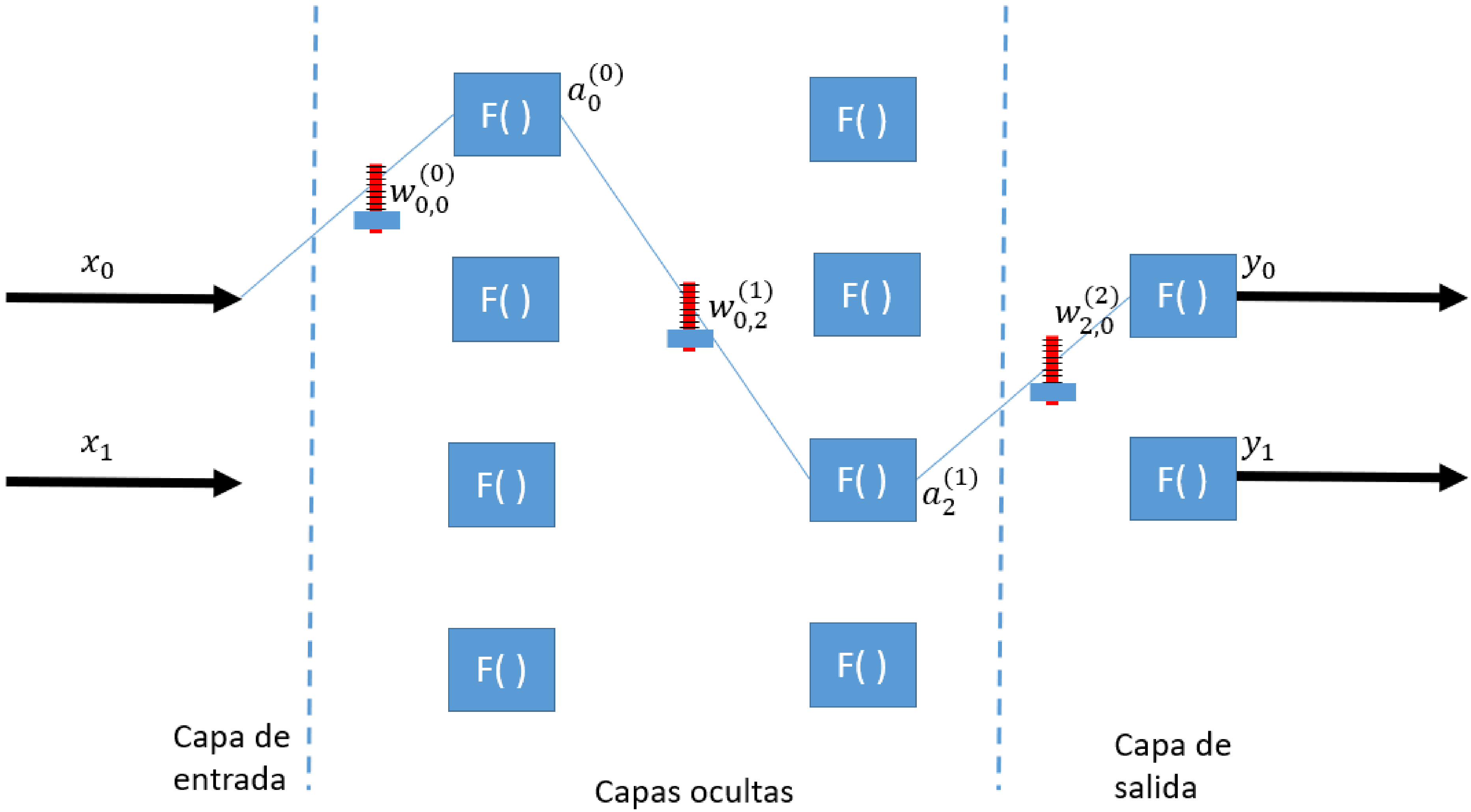


Ilustración 42: Tercer camino

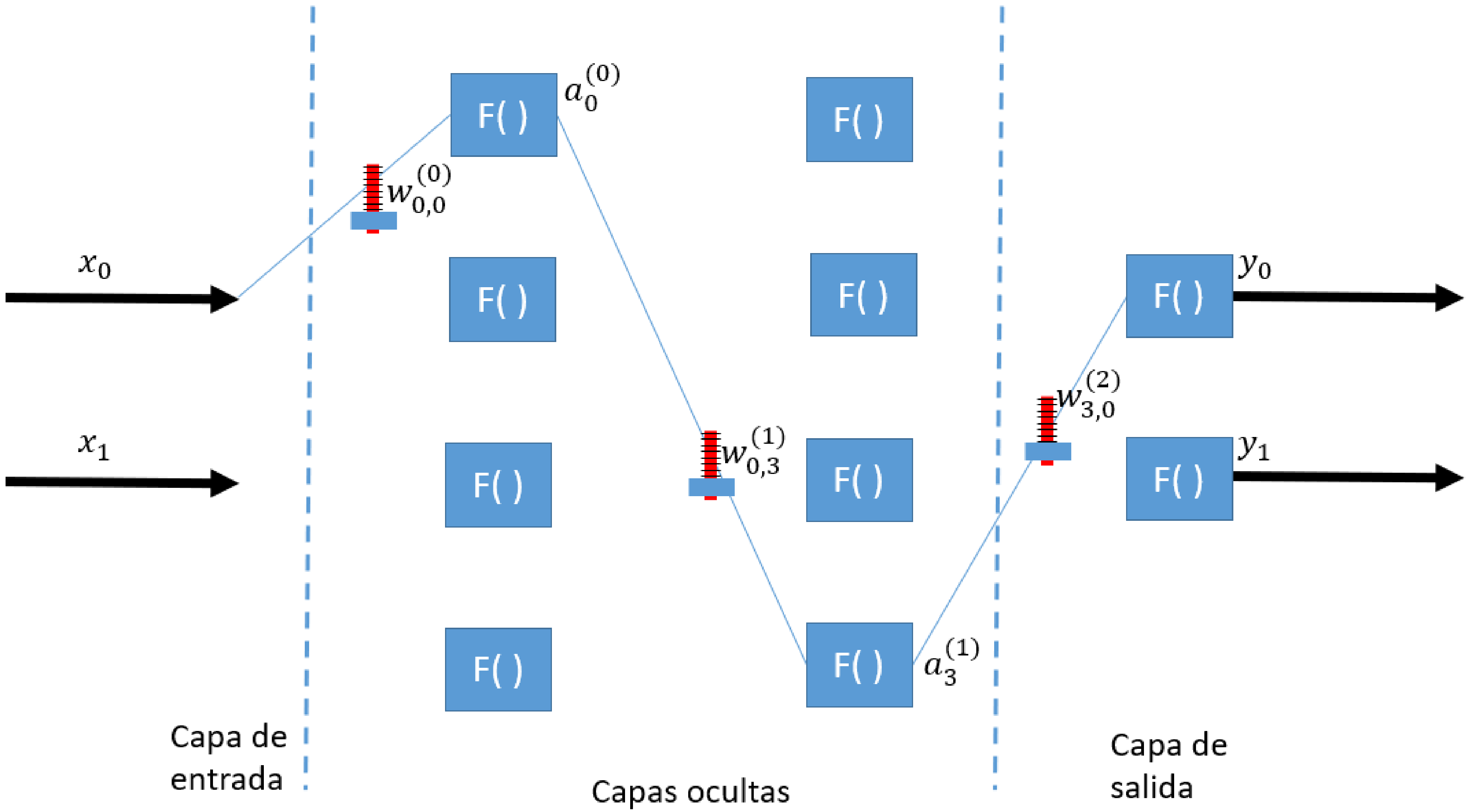


Ilustración 43: Cuarto camino

Luego

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

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 $u^{(2)}$?

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

Se tiene 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 se está 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.

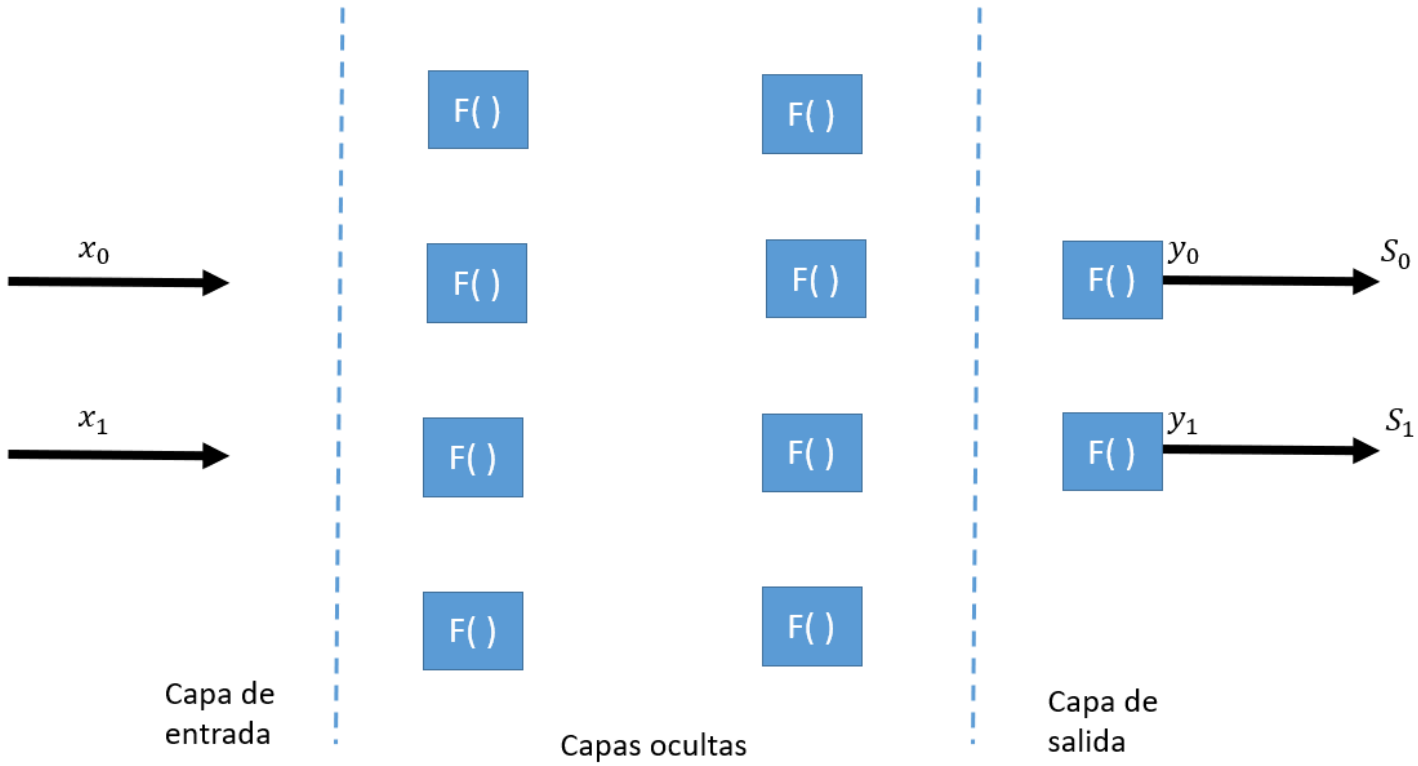


Ilustración 44: Dos entradas y dos salidas

Si se tomaran las salidas y_0 y y_1 como coordenadas e igualmente S_0 y S_1, se obtiene lo siguiente:

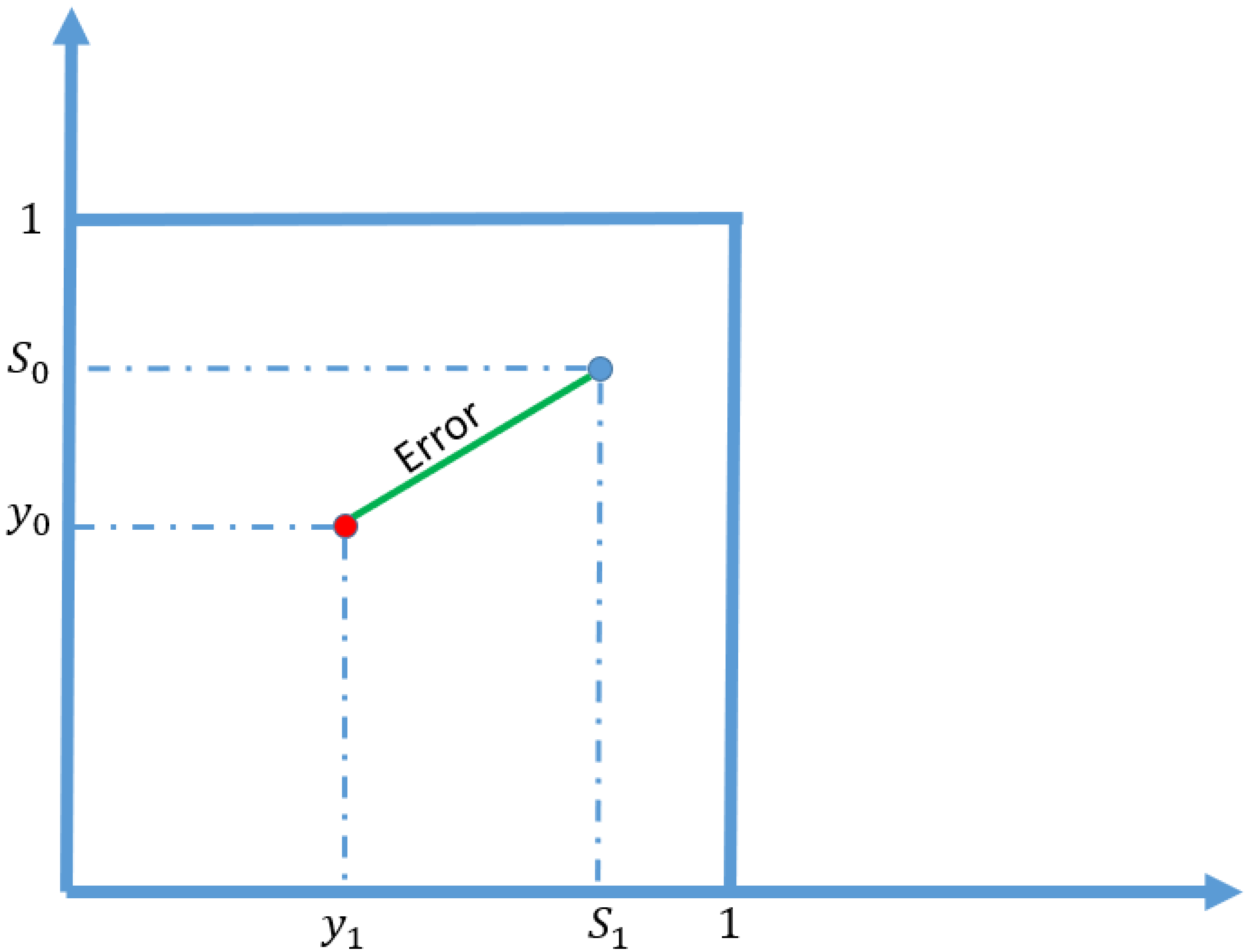


Ilustración 45: Representación del error

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:

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

Para minimizar ese error hay que considerar que dado:

$$F(x) = \sqrt[2]{g(x)}$$

Al derivar:

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

Y como hay que minimizar se iguala esa derivada a cero

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

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 se tiene 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

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

Entonces

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

Simplificando

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

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

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

Luego reemplazando en la expresión se obtiene:

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

Simplificando

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

Luego

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

En limpio las fórmulas para los pesos son:

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

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

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

Y para los umbrales sería:

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

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

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

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

La fórmula de variación de los pesos y umbrales es:

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

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

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

$$u_{i}^{(2)} \leftarrow u_{i}^{(2)} - \propto * \frac{\partial Error}{\partial u_{i}^{(2)}}$$

$$u_{k}^{(1)} \leftarrow u_{k}^{(1)} - \propto * \frac{\partial Error}{\partial u_{k}^{(1)}}$$

$$u_{k}^{(0)} \leftarrow u_{k}^{(0)} - \propto * \frac{\partial Error}{\partial u_{k}^{(0)}}$$

Donde $\propto$ es el factor de aprendizaje con un valor pequeño entre 0.1 y 0.9

Implementación del perceptrón multicapa

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

Ilustración 46: 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.

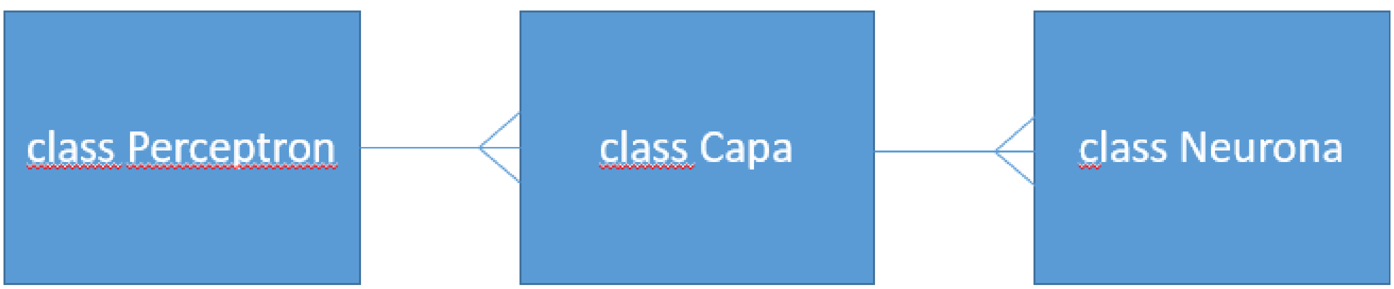


Ilustración 47: Modelo del perceptrón

Esta sería la plantilla del programa:

K/008.cs

```
namespace Ejemplo {
   class Program {
      static void Main() {
      }
   }

   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 y el umbral al azar. Así quedaría el código:

K/009.cs

```
namespace Ejemplo {
   class Program {
      static void Main() {
      }
   }

   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 = [];
         for (int Contador = 0; Contador < TotalEntradas; Contador++)
            Pesos.Add(Azar.NextDouble());
         Umbral = Azar.NextDouble();
      }
   }
}
```

Se añade a la clase neurona, el método CalculaSalida() que tiene como parámetro un arreglo unidimensional, el cual tiene los datos de entrada.

K/010.cs

```
namespace Ejemplo {
   class Program {
      static void Main() {
      }
   }

   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 = [];
         for (int Contador = 0; Contador < TotalEntradas; Contador++)
            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 Contador = 0; Contador < Pesos.Count; Contador++)
            Valor += Entradas[Contador] * Pesos[Contador];
         Valor += Umbral;
         return 1 / (1 + Math.Exp(-Valor));
      }
   }
}
```

La clase **Capa** almacena sus propias neuronas y la salida de cada una de esas neuronas en una 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".

K/011.cs

```
namespace Ejemplo {
   class Program {
      static void Main() {
      }
   }

   class Perceptron {
      List<Capa> Capas;
   }

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

      public Capa(Random Azar, int TotalNeuronas, int TotalEntradas) {
         Neuronas = [];
         Salidas = [];

         //Genera las neuronas e inicializa sus salidas
         for (int Contador = 0; Contador < TotalNeuronas; Contador++) {
            Neuronas.Add(new Neurona(Azar, TotalEntradas));
            Salidas.Add(0);
         }
      }

      //Calcula la salida 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 = [];
      for (int Contador = 0; Contador < TotalEntradas; Contador++)
         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 Contador = 0; Contador < Pesos.Count; Contador++)
			Valor += Entradas[Contador] * Pesos[Contador];
		Valor += Umbral;
		return 1 / (1 + Math.Exp(-Valor));
	}
}
```

K/012.cs

```
namespace Ejemplo {
   class Program {
      static void Main() {
      }
   }

   class Perceptron {
      List<Capa>? Capas;

      //Crea las diversas capas
      public void CreaCapas(Random Azar, int Entradas,
                        int NeuronasCapa0, int NeuronasCapa1,
                        int NeuronasCapa2) {
         Capas =
         [
            //Crea la capa 0
            new Capa(Azar, NeuronasCapa0, Entradas),

            //Crea la capa 1 (el número de entradas es
            //el número de neuronas de la capa anterior)
            new Capa(Azar, NeuronasCapa1, NeuronasCapa0),

            //Crea la capa 2 (el número de entradas es
            //el número de neuronas de la capa anterior)
            new Capa(Azar, NeuronasCapa2, NeuronasCapa1),
         ];
      }
   }

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

      public Capa(Random Azar, int TotalNeuronas, int TotalEntradas) {
         Neuronas = [];
         Salidas = [];

         //Genera las neuronas e inicializa sus salidas
         for (int Contador = 0; Contador < TotalNeuronas; Contador++) {
            Neuronas.Add(new Neurona(Azar, TotalEntradas));
            Salidas.Add(0);
         }
      }

      //Calcula la salida 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 = [];
      for (int Contador = 0; Contador < TotalEntradas; Contador++)
         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 Contador = 0; Contador < Pesos.Count; Contador++)
         Valor += Entradas[Contador] * Pesos[Contador];
      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.

K/013.cs

```
namespace Ejemplo {
   class Program {
      static void Main() {
      }
   }

   class Perceptron {
      List<Capa>? Capas;

      //Crea las diversas capas
      public void CreaCapas(Random Azar, int Entradas,
         int NeuronasCapa0, int NeuronasCapa1, int NeuronasCapa2) {
         Capas =
         [
            //Crea la capa 0
            new Capa(Azar, NeuronasCapa0, Entradas),

            //Crea la capa 1 (el número de entradas
            //es el número de neuronas de la capa anterior)
            new Capa(Azar, NeuronasCapa1, NeuronasCapa0),

            //Crea la capa 2 (el número de entradas
            //es el número de neuronas de la capa anterior)
            new Capa(Azar, NeuronasCapa2, NeuronasCapa1),
         ];
      }

      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 {
      //Las neuronas que tendrá la capa
      List<Neurona> Neuronas;

      //Almacena la salida de cada neurona
      public List<double> Salidas;
```

```
      public Capa(Random Azar, int TotalNeuronas, int TotalEntradas) {
         Neuronas = [];
         Salidas = [];

         //Genera las neuronas e inicializa sus salidas
         for (int Contador = 0; Contador < TotalNeuronas; Contador++) {
            Neuronas.Add(new Neurona(Azar, TotalEntradas));
            Salidas.Add(0);
         }
      }

      //Calcula la salida 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 = [];
      for (int Contador = 0; Contador < TotalEntradas; Contador++)
         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 Contador = 0; Contador < Pesos.Count; Contador++)
         Valor += Entradas[Contador] * Pesos[Contador];
      Valor += Umbral;
      return 1 / (1 + Math.Exp(-Valor));
   }
}
```

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

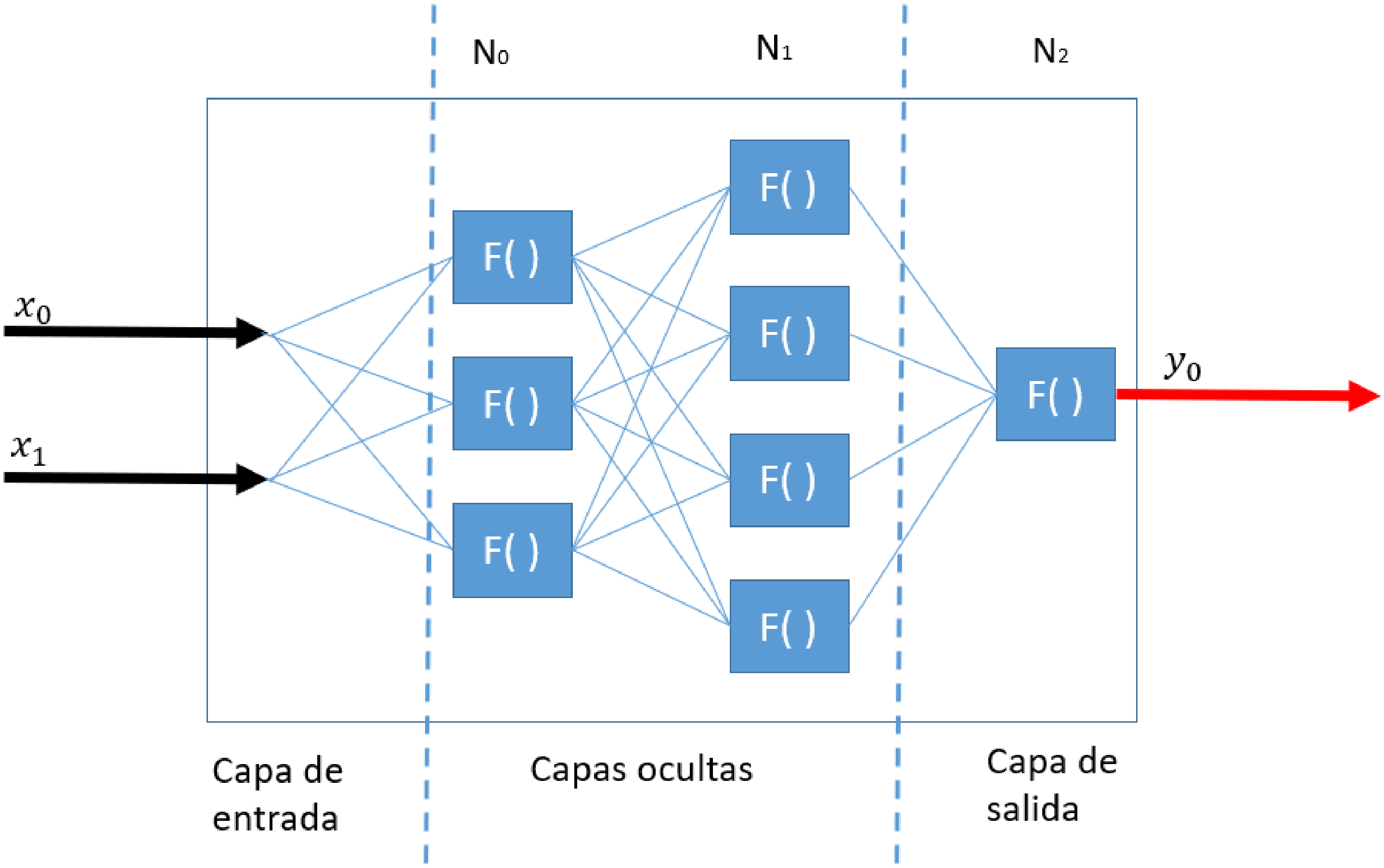


Ilustración 48: Un perceptrón multicapa

K/014.cs

```
class Program {
    static void Main() {
        //Un solo generador de números pseudo-aleatorios
        Random Azar = new();
        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);
    }
}
```

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

Algoritmo de retro propagación

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

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

Se implementa así:

```
for (int j = 0; j < NeuronasCapa1; j++)
 //Va de neurona en neurona de la capa 1
   for (int i = 0; i < NeuronasCapa2; 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
   }
```

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

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

Se implementa así:

```
for (int j = 0; j < NeuronasCapa0; j++)
 //Va de neurona en neurona de la capa 0
   for (int k = 0; k < NeuronasCapa1; k++) {
//Va de neurona en neurona de la capa 1
      double Acumula = 0;
      for (int i = 0; i < NeuronasCapa2; 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];
         Acumula += 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) * Acumula;
      Capas[1].Neuronas[k].NuevosPesos[j] = Capas[1].Neuronas[k].Pesos[j] -
Alpha * dE1;
   }
```

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

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

Se implementa así:

```
for (int j = 0; j < Entradas.Count; j++)
//Va de entrada en entrada
   for (int k = 0; k < NeuronasCapa0; k++) { //Va de neurona en neurona de
la capa 0
      double Acumula = 0;
      for (int p = 0; p < NeuronasCapa1; p++) { //Va de neurona en neurona
de la capa 1
         double InternoAcumula = 0;
         for (int i = 0; i < NeuronasCapa2; 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];
            InternoAcumula += W2pi * (Yi - Si) * Yi * (1 - Yi); //Sumatoria
interna
         }
         double W1kp = Capas[1].Neuronas[p].Pesos[k];
         double a1p = Capas[1].Salidas[p];
         Acumula += W1kp * a1p * (1 - a1p) * InternoAcumula; //Sumatoria
externa
      }
      double xj = Entradas[j];
      double a0k = Capas[0].Salidas[k];
      double dE0 = xj * a0k * (1 - a0k) * Acumula;
```

```
        double W0jk = Capas[0].Neuronas[k].Pesos[j];
        Capas[0].Neuronas[k].NuevosPesos[j] = W0jk - Alpha * dE0;
    }
```

Para los umbrales de la capa 2

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

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

Se implementa así:

```
for (int i = 0; i < NeuronasCapa2; 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].NuevoUmbral = Capas[2].Neuronas[i].Umbral - Alpha * dE2;
}
```

Para los umbrales de la capa 1

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

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

Se implementa así:

```
for (int k = 0; k < NeuronasCapa1; k++) {
//Va de neurona en neurona de la capa 1
   double Acumula = 0;
   for (int i = 0; i < NeuronasCapa2; 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];
      Acumula += W2ki * (Yi - Si) * Yi * (1 - Yi);
   }
   double a1k = Capas[1].Salidas[k];
   double dE1 = a1k * (1 - a1k) * Acumula;
   Capas[1].Neuronas[k].NuevoUmbral = Capas[1].Neuronas[k].Umbral - Alpha *
dE1;
}
```

Para los umbrales de la capa 0

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

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

Se implementa así:

```
for (int k = 0; k < NeuronasCapa0; k++) {
//Va de neurona en neurona de la capa 0
   double Acumula = 0;
   for (int p = 0; p < NeuronasCapa1; p++) { //Va de neurona en neurona de
la capa 1
      double InternoAcumula = 0;
      for (int i = 0; i < NeuronasCapa2; 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];
         InternoAcumula += W2pi * (Yi - Si) * Yi * (1 - Yi);
      }
      double W1kp = Capas[1].Neuronas[p].Pesos[k];
      double a1p = Capas[1].Salidas[p];
      Acumula += W1kp * a1p * (1 - a1p) * InternoAcumula;
   }
   double a0k = Capas[0].Salidas[k];
   double dE0 = a0k * (1 - a0k) * Acumula;
   Capas[0].Neuronas[k].NuevoUmbral = Capas[0].Neuronas[k].Umbral - Alpha *
dE0;
}
```

Como se puede observar 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"

```
public void Actualiza() {
   for (int Contador = 0; Contador < Pesos.Count; Contador++)
      Pesos[Contador] = NuevosPesos[Contador];
   Umbral = NuevoUmbral;
}
```

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

```
public void Actualiza() {
   for (int Contador = 0; Contador < Neuronas.Count; Contador++)
      Neuronas[Contador].Actualiza();
}
```

Código completo del perceptrón

A continuación, se muestra el código completo 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.

K/015.cs

```
namespace Ejemplo {
   class Program {
      static void Main() {
         //Tabla XOR
         int[][] XORentra = [
            [1, 1],
            [1, 0],
            [0, 1],
            [0, 0]
         ];
         int[] XORsale = [0, 1, 1, 0];

         int TotalEntradas = 2; //Número de entradas
         int NeuronasCapa0 = 3; //Total neuronas en la capa 0
         int NeuronasCapa1 = 2; //Total neuronas en la capa 1
         int NeuronasCapa2 = 1; //Total neuronas en la capa 2
         Perceptron RedNeuronal = new(TotalEntradas, NeuronasCapa0,
                         NeuronasCapa1, NeuronasCapa2);

         //Estas serán las dos entradas externas al perceptrón
         List<double> Entradas = [0, 0];

         //Esta será la salida esperada externa al perceptrón
         List<double> SalidaEsperada = [0];

         //Ciclo que entrena la red neuronal
         int TotalCiclos = 90000; //Ciclos de entrenamiento
         for (int Ciclo = 1; Ciclo <= TotalCiclos; Ciclo++) {

            if (Ciclo % 200 == 0) Console.WriteLine("\r\nCiclo: " + 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
            RedNeuronal.CalculaSalida(Entradas);

            //Luego entrena el perceptrón para
            //ajustar los pesos y umbrales
            RedNeuronal.Entrena(Entradas, SalidaEsperada);

            //Cada 200 ciclos muestra como
            //progresa el entrenamiento
            if (Ciclo % 200 == 0)
               RedNeuronal.SalidaPerceptron(Entradas, SalidaEsperada[0]);
         }
      }

      Console.WriteLine("Finaliza el entrenamiento");
   }
}

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] + " | ");

      Console.Write(" Esperada: " + SalidaEsperada + "  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] + " | ");
      }
      Console.WriteLine(" ");
   }

   //Crea las diversas capas
   public Perceptron(  int TotalEntradas, int NeuronasCapa0,
                 int NeuronasCapa1, int NeuronasCapa2) {
      Random Azar = new();
      Capas =
      [
         new Capa(Azar, NeuronasCapa0, TotalEntradas), //Crea la capa 0
```

```
         new Capa(Azar, NeuronasCapa1, NeuronasCapa0), //Crea la capa 1
         new Capa(Azar, NeuronasCapa2, NeuronasCapa1), //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 NeuronasCapa0 = Capas[0].Neuronas.Count;
      int NeuronasCapa1 = Capas[1].Neuronas.Count;
      int NeuronasCapa2 = Capas[2].Neuronas.Count;

      //Factor de aprendizaje
      double Alpha = 0.4;

      //=====================
      //Procesa pesos capa 2
      //=====================

      //Va de neurona en neurona de la capa 1
      for (int j = 0; j < NeuronasCapa1; j++)

         //Va de neurona en neurona de la capa de salida (capa 2)
         for (int i = 0; i < NeuronasCapa2; i++) {

            //Salida de la neurona de la capa de salida
            double Yi = Capas[2].Salidas[i];

            //Salida esperada
            double Si = SalidaEsperada[i];

            //Salida de la capa 1
            double a1j = Capas[1].Salidas[j];

            //La fórmula del error
```

```
          double dE2 = a1j * (Yi - Si) * Yi * (1 - Yi);

          //Ajusta el nuevo peso
          double Nuevo = Capas[2].Neuronas[i].Pesos[j] - Alpha * dE2;
          Capas[2].Neuronas[i].NuevosPesos[j] = Nuevo;
        }

      //=====================
      //Procesa pesos capa 1
      //=====================

      //Va de neurona en neurona de la capa 0
      for (int j = 0; j < NeuronasCapa0; j++)

        //Va de neurona en neurona de la capa 1
        for (int k = 0; k < NeuronasCapa1; k++) {
          double Acumula = 0;

          //Va de neurona en neurona de la capa 2
          for (int i = 0; i < NeuronasCapa2; i++) {

            //Salida de la capa 2
            double Yi = Capas[2].Salidas[i];

            //Salida esperada
            double Si = SalidaEsperada[i];
            double W2ki = Capas[2].Neuronas[i].Pesos[k];
            Acumula += 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) * Acumula;
          double Nuevo = Capas[1].Neuronas[k].Pesos[j] - Alpha * dE1;
          Capas[1].Neuronas[k].NuevosPesos[j] = Nuevo;
        }

      //=====================
      //Procesa pesos capa 0
      //=====================

      //Va de entrada en entrada
      for (int j = 0; j < Entradas.Count; j++)

        //Va de neurona en neurona de la capa 0
        for (int k = 0; k < NeuronasCapa0; k++) {
          double Acumula = 0;

          //Va de neurona en neurona de la capa 1
```

```
         for (int p = 0; p < NeuronasCapa1; p++) {
            double InternoAcumula = 0;

            //Va de neurona en neurona de la capa 2
            for (int i = 0; i < NeuronasCapa2; i++) {
               double Yi = Capas[2].Salidas[i];
               double Si = SalidaEsperada[i]; //Salida esperada
               double W2pi = Capas[2].Neuronas[i].Pesos[p];

               //Sumatoria interna
               InternoAcumula += W2pi * (Yi - Si) * Yi * (1 - Yi);
            }
            double W1kp = Capas[1].Neuronas[p].Pesos[k];
            double a1p = Capas[1].Salidas[p];

            //Sumatoria externa
            Acumula += W1kp * a1p * (1 - a1p) * InternoAcumula;
         }
         double xj = Entradas[j];
         double a0k = Capas[0].Salidas[k];
         double dE0 = xj * a0k * (1 - a0k) * Acumula;
         double W0jk = Capas[0].Neuronas[k].Pesos[j];
         Capas[0].Neuronas[k].NuevosPesos[j] = W0jk - Alpha * dE0;
      }

   //========================
   //Procesa umbrales capa 2
   //========================

   //Va de neurona en neurona de la capa de salida (capa 2)
   for (int i = 0; i < NeuronasCapa2; i++) {

      //Salida de la neurona de la capa de salida
      double Yi = Capas[2].Salidas[i];

      //Salida esperada
      double Si = SalidaEsperada[i];
      double dE2 = (Yi - Si) * Yi * (1 - Yi);
      double Nuevo = Capas[2].Neuronas[i].Umbral - Alpha * dE2;
      Capas[2].Neuronas[i].NuevoUmbral = Nuevo;
   }

   //========================
   //Procesa umbrales capa 1
   //========================

   //Va de neurona en neurona de la capa 1
   for (int k = 0; k < NeuronasCapa1; k++) {
```

```
            double Acumula = 0;

            //Va de neurona en neurona de la capa 2
            for (int i = 0; i < NeuronasCapa2; i++) {

               //Salida de la capa 2
               double Yi = Capas[2].Salidas[i];
               double Si = SalidaEsperada[i];
               double W2ki = Capas[2].Neuronas[i].Pesos[k];
               Acumula += W2ki * (Yi - Si) * Yi * (1 - Yi);
            }
            double a1k = Capas[1].Salidas[k];
            double dE1 = a1k * (1 - a1k) * Acumula;
            double Nuevo = Capas[1].Neuronas[k].Umbral - Alpha * dE1;
            Capas[1].Neuronas[k].NuevoUmbral = Nuevo;
         }

         //========================
         //Procesa umbrales capa 0
         //========================

         //Va de neurona en neurona de la capa 0
         for (int k = 0; k < NeuronasCapa0; k++) {
            double Acumula = 0;

            //Va de neurona en neurona de la capa 1
            for (int p = 0; p < NeuronasCapa1; p++) {
               double InternoAcumula = 0;

               //Va de neurona en neurona de la capa 2
               for (int i = 0; i < NeuronasCapa2; i++) {
                  double Yi = Capas[2].Salidas[i];
                  double Si = SalidaEsperada[i];
                  double W2pi = Capas[2].Neuronas[i].Pesos[p];
                  InternoAcumula += W2pi * (Yi - Si) * Yi * (1 - Yi);
               }
               double W1kp = Capas[1].Neuronas[p].Pesos[k];
               double a1p = Capas[1].Salidas[p];
               Acumula += W1kp * a1p * (1 - a1p) * InternoAcumula;
            }
            double a0k = Capas[0].Salidas[k];
            double dE0 = a0k * (1 - a0k) * Acumula;
            double Nuevo = Capas[0].Neuronas[k].Umbral - Alpha * dE0;
            Capas[0].Neuronas[k].NuevoUmbral = Nuevo;
         }

         //Actualiza los pesos
         Capas[0].Actualiza();
```

```
         Capas[1].Actualiza();
         Capas[2].Actualiza();
      }

   }

   class Capa {
      //Las neuronas que tendrá la capa
      public List<Neurona> Neuronas;

      //Almacena las salidas de cada neurona
      public List<double> Salidas;

      public Capa(Random Azar, int TotalNeuronas, int TotalEntradas) {
         Neuronas = [];
         Salidas = [];

         //Genera las neuronas
         for (int Contador = 0; Contador < TotalNeuronas; Contador++) {
            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 Contador = 0; Contador < Neuronas.Count; Contador++)
            Salidas[Contador] = Neuronas[Contador].CalculaSalida(Entradas);
      }

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

   class Neurona {
      //Los pesos para cada entrada
      public List<double> Pesos;

      //Nuevos pesos dados por el algoritmo de "backpropagation"
      public List<double> NuevosPesos;

      //El peso del umbral
      public double Umbral;

      //Nuevo umbral dado por el algoritmo de "backpropagation"
```

```
      public double NuevoUmbral;

      //Inicializa los pesos y umbral con un valor al azar
      public Neurona(Random Azar, int TotalEntradas) {
         Pesos = [];
         NuevosPesos = [];
         for (int Contador = 0; Contador < TotalEntradas; Contador++) {
            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 Contador = 0; Contador < Pesos.Count; Contador++)
            Valor += Entradas[Contador] * Pesos[Contador];
         Valor += Umbral;
         return 1 / (1 + Math.Exp(-Valor));
      }

      //Reemplaza viejos pesos por nuevos
      public void Actualiza() {
         for (int Contador = 0; Contador < Pesos.Count; Contador++)
            Pesos[Contador] = NuevosPesos[Contador];
         Umbral = NuevoUmbral;
      }
   }
}
```

```
Ciclo: 89800
1 | 1 |  Esperada: 0  Calculada:  0 | 0,003702174383143816 |
1 | 0 |  Esperada: 1  Calculada:  1 | 0,9953775432424248 |
0 | 1 |  Esperada: 1  Calculada:  1 | 0,9953696734810018 |
0 | 0 |  Esperada: 0  Calculada:  0 | 0,0031923050449013304 |

Ciclo: 90000
1 | 1 |  Esperada: 0  Calculada:  0 | 0,003695850644007681 |
1 | 0 |  Esperada: 1  Calculada:  1 | 0,9953851916858888 |
0 | 1 |  Esperada: 1  Calculada:  1 | 0,9953773370654849 |
0 | 0 |  Esperada: 0  Calculada:  0 | 0,0031871208730737985 |
Finaliza el entrenamiento
```

Ilustración 49: Perceptrón aprendiendo la tabla del XOR

Reconocimiento de números de un reloj digital

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

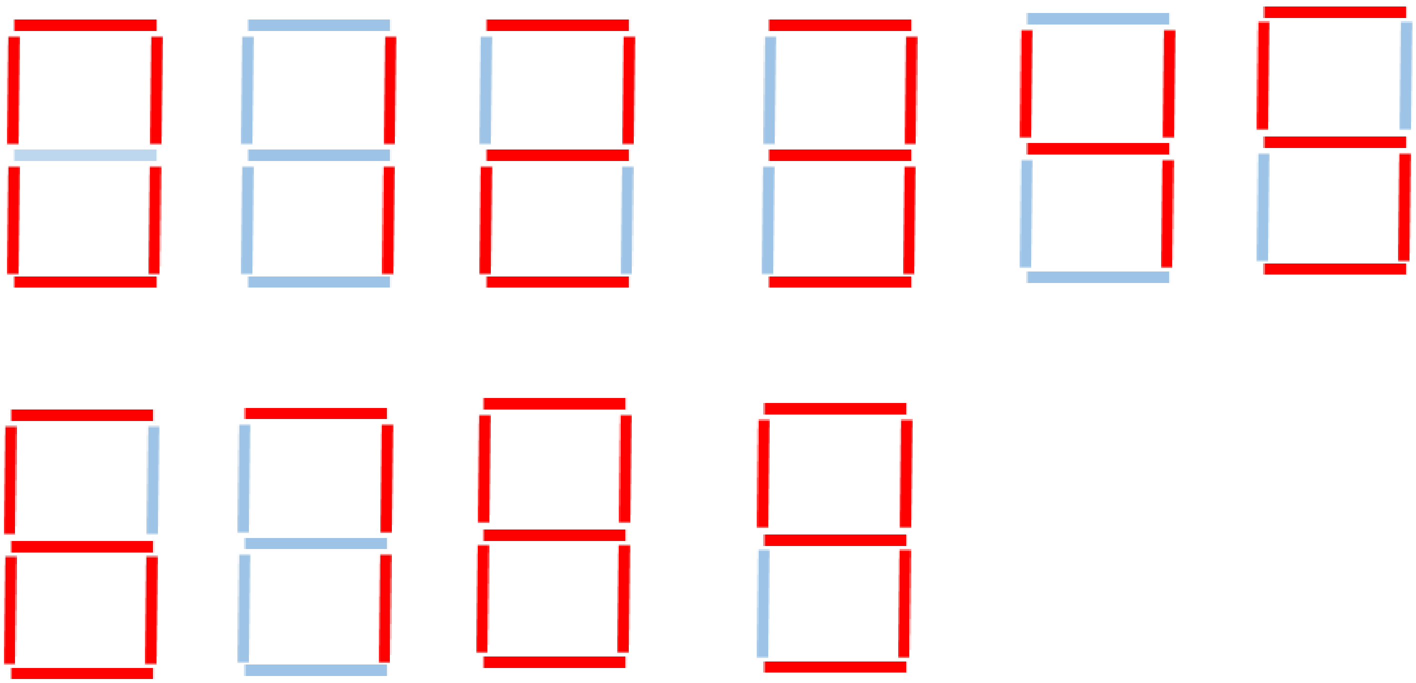
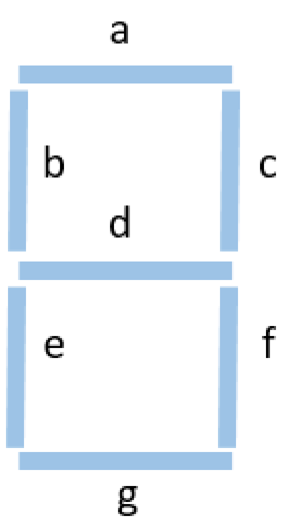
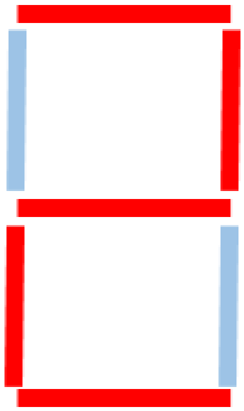


Ilustración 50: Números en un reloj digital

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



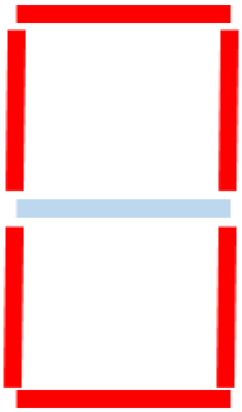
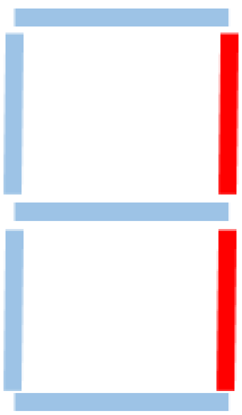
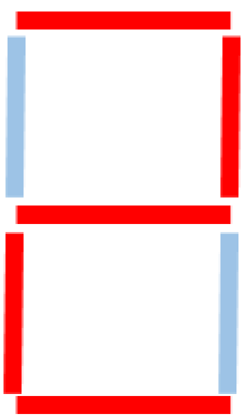
Luego se le da un valor de "1" a la barra que queda en rojo al construir el número y "0" a la barra que queda en azul claro. Por ejemplo:

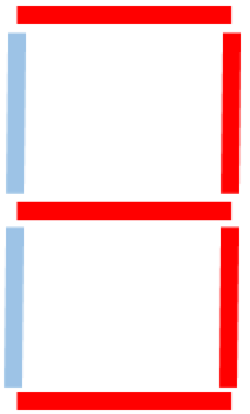
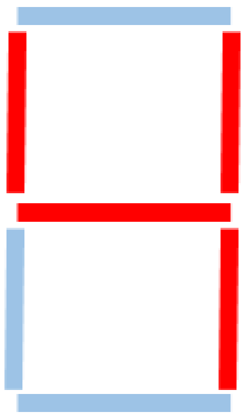
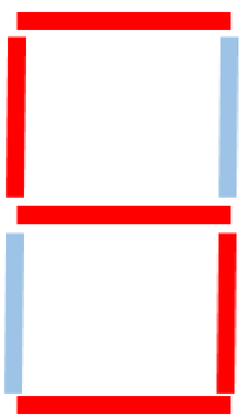
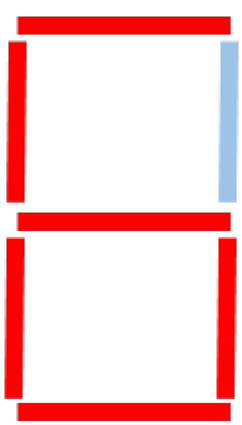


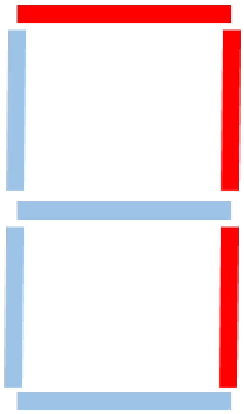
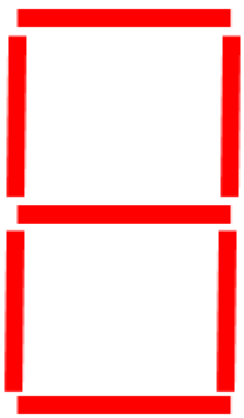
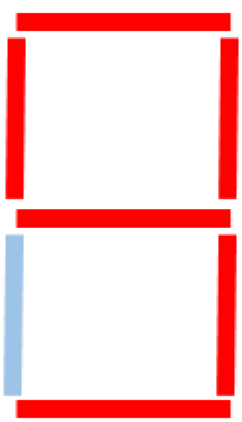
Los valores de a,b,c,d,e,f,g serían: 1,0,1,1,1,0,1

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 es el siguiente:

K/016.cs

```
namespace Ejemplo {
   class Program {
      static void Main() {
         //El número "dibujado" en el reloj digital
         int[][] Reloj = [
            [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]
         ];

         //Número esperado en binario
         int[][] Esperado = [
            [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 TotalEntradas = 7; //Número de entradas
         int NeuronasCapa0 = 5; //Total neuronas en la capa 0
         int NeuronasCapa1 = 5; //Total neuronas en la capa 1
         int NeuronasCapa2 = 4; //Total neuronas en la capa 2
         Perceptron RedNeuronal = new(TotalEntradas, NeuronasCapa0,
                                   NeuronasCapa1, NeuronasCapa2);

         //Estas serán las dos entradas externas al perceptrón
         List<double> Entradas = [0, 0, 0, 0, 0, 0, 0];

         //Esta será la salida esperada externa al perceptrón
         List<double> SalidaEsperada = [0, 0, 0, 0];

         //Ciclo que entrena la red neuronal
```

```
      int TotalCiclos = 10000; //Ciclos de entrenamiento
      for (int Ciclo = 1; Ciclo <= TotalCiclos; Ciclo++) {

         if (Ciclo % 1000 == 0)
            Console.WriteLine("\r\nCiclo: " + Ciclo);

         //Por cada ciclo, se entrena el perceptrón
         //con toda la tabla
         for (int Conj = 0; Conj < Esperado.GetLength(0); Conj++) {

            //Entradas y salidas esperadas
            for (int Entra = 0; Entra < Reloj[0].GetLength(0); Entra++)
               Entradas[Entra] = Reloj[Conj][Entra];

            for (int Sale = 0; Sale < Esperado[0].GetLength(0); Sale++)
               SalidaEsperada[Sale] = Esperado[Conj][Sale];

            //Primero calcula la salida del
            //perceptrón con esas entradas
            RedNeuronal.CalculaSalida(Entradas);

            //Luego entrena el perceptrón para
            //ajustar los pesos y umbrales
            RedNeuronal.Entrena(Entradas, SalidaEsperada);

            //Cada 1000 ciclos muestra como
            //progresa el entrenamiento
            if (Ciclo % 1000 == 0)
               RedNeuronal.SalidaPerceptron(Entradas, SalidaEsperada);
         }
      }

      Console.WriteLine("Finaliza el entrenamiento");
   }
}

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] + " ");
      }

      Console.Write(" Esperada: ");
```

```
        for (int cont = 0; cont < SalidaEsperada.Count; cont++) {
           Console.Write(SalidaEsperada[cont] + " ");
        }

        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.WriteLine(" ");
     }

     //Crea las diversas capas
     public Perceptron(int TotalEntradas, int NeuronasCapa0,
                       int NeuronasCapa1, int NeuronasCapa2) {
        Random Azar = new();
        Capas =
        [
           new Capa(Azar, NeuronasCapa0, TotalEntradas), //Crea la capa 0
           new Capa(Azar, NeuronasCapa1, NeuronasCapa0), //Crea la capa 1
           new Capa(Azar, NeuronasCapa2, NeuronasCapa1), //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 NeuronasCapa0 = Capas[0].Neuronas.Count;
        int NeuronasCapa1 = Capas[1].Neuronas.Count;
        int NeuronasCapa2 = Capas[2].Neuronas.Count;
```

```
        //Factor de aprendizaje
        double Alpha = 0.4;

        //====================
        //Procesa pesos capa 2
        //====================

        //Va de neurona en neurona de la capa 1
        for (int j = 0; j < NeuronasCapa1; j++)

           //Va de neurona en neurona de la capa de salida (capa 2)
           for (int i = 0; i < NeuronasCapa2; i++) {

              //Salida de la neurona de la capa de salida
              double Yi = Capas[2].Salidas[i];

              //Salida esperada
              double Si = SalidaEsperada[i];

              //Salida de la capa 1
              double a1j = Capas[1].Salidas[j];

              //La fórmula del error
              double dE2 = a1j * (Yi - Si) * Yi * (1 - Yi);

              //Ajusta el nuevo peso
              double Nuevo = Capas[2].Neuronas[i].Pesos[j] - Alpha * dE2;
              Capas[2].Neuronas[i].NuevosPesos[j] = Nuevo;
           }

        //====================
        //Procesa pesos capa 1
        //====================

        //Va de neurona en neurona de la capa 0
        for (int j = 0; j < NeuronasCapa0; j++)

           //Va de neurona en neurona de la capa 1
           for (int k = 0; k < NeuronasCapa1; k++) {
              double Acumula = 0;

              //Va de neurona en neurona de la capa 2
              for (int i = 0; i < NeuronasCapa2; i++) {

                 //Salida de la capa 2
                 double Yi = Capas[2].Salidas[i];

                 //Salida esperada
```

```
            double Si = SalidaEsperada[i];
            double W2ki = Capas[2].Neuronas[i].Pesos[k];
            Acumula += 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) * Acumula;
         double Nuevo = Capas[1].Neuronas[k].Pesos[j] - Alpha * dE1;
         Capas[1].Neuronas[k].NuevosPesos[j] = Nuevo;
      }

   //=====================
   //Procesa pesos capa 0
   //=====================

   //Va de entrada en entrada
   for (int j = 0; j < Entradas.Count; j++)

      //Va de neurona en neurona de la capa 0
      for (int k = 0; k < NeuronasCapa0; k++) {
         double Acumula = 0;

         //Va de neurona en neurona de la capa 1
         for (int p = 0; p < NeuronasCapa1; p++) {
            double InternoAcumula = 0;

            //Va de neurona en neurona de la capa 2
            for (int i = 0; i < NeuronasCapa2; i++) {
               double Yi = Capas[2].Salidas[i];
               double Si = SalidaEsperada[i]; //Salida esperada
               double W2pi = Capas[2].Neuronas[i].Pesos[p];

               //Sumatoria interna
               InternoAcumula += W2pi * (Yi - Si) * Yi * (1 - Yi);
            }
            double W1kp = Capas[1].Neuronas[p].Pesos[k];
            double a1p = Capas[1].Salidas[p];

            //Sumatoria externa
            Acumula += W1kp * a1p * (1 - a1p) * InternoAcumula;
         }
         double xj = Entradas[j];
         double a0k = Capas[0].Salidas[k];
         double dE0 = xj * a0k * (1 - a0k) * Acumula;
         double W0jk = Capas[0].Neuronas[k].Pesos[j];
         Capas[0].Neuronas[k].NuevosPesos[j] = W0jk - Alpha * dE0;
      }
```

```
//=======================
//Procesa umbrales capa 2
//=======================

//Va de neurona en neurona de la capa de salida (capa 2)
for (int i = 0; i < NeuronasCapa2; i++) {

   //Salida de la neurona de la capa de salida
   double Yi = Capas[2].Salidas[i];

   //Salida esperada
   double Si = SalidaEsperada[i];
   double dE2 = (Yi - Si) * Yi * (1 - Yi);
   double Nuevo = Capas[2].Neuronas[i].Umbral - Alpha * dE2;
   Capas[2].Neuronas[i].NuevoUmbral = Nuevo;
}

//=======================
//Procesa umbrales capa 1
//=======================

//Va de neurona en neurona de la capa 1
for (int k = 0; k < NeuronasCapa1; k++) {
   double Acumula = 0;

   //Va de neurona en neurona de la capa 2
   for (int i = 0; i < NeuronasCapa2; i++) {

      //Salida de la capa 2
      double Yi = Capas[2].Salidas[i];
      double Si = SalidaEsperada[i];
      double W2ki = Capas[2].Neuronas[i].Pesos[k];
      Acumula += W2ki * (Yi - Si) * Yi * (1 - Yi);
   }
   double a1k = Capas[1].Salidas[k];
   double dE1 = a1k * (1 - a1k) * Acumula;
   double Nuevo = Capas[1].Neuronas[k].Umbral - Alpha * dE1;
   Capas[1].Neuronas[k].NuevoUmbral = Nuevo;
}

//=======================
//Procesa umbrales capa 0
//=======================

//Va de neurona en neurona de la capa 0
for (int k = 0; k < NeuronasCapa0; k++) {
   double Acumula = 0;
```

```
         //Va de neurona en neurona de la capa 1
         for (int p = 0; p < NeuronasCapa1; p++) {
            double InternoAcumula = 0;

            //Va de neurona en neurona de la capa 2
            for (int i = 0; i < NeuronasCapa2; i++) {
               double Yi = Capas[2].Salidas[i];
               double Si = SalidaEsperada[i];
               double W2pi = Capas[2].Neuronas[i].Pesos[p];
               InternoAcumula += W2pi * (Yi - Si) * Yi * (1 - Yi);
            }
            double W1kp = Capas[1].Neuronas[p].Pesos[k];
            double a1p = Capas[1].Salidas[p];
            Acumula += W1kp * a1p * (1 - a1p) * InternoAcumula;
         }
         double a0k = Capas[0].Salidas[k];
         double dE0 = a0k * (1 - a0k) * Acumula;
         double Nuevo = Capas[0].Neuronas[k].Umbral - Alpha * dE0;
         Capas[0].Neuronas[k].NuevoUmbral = Nuevo;
      }

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

}

class Capa {
   //Las neuronas que tendrá la capa
   public List<Neurona> Neuronas;

   //Almacena las salidas de cada neurona
   public List<double> Salidas;

   public Capa(Random Azar, int TotalNeuronas, int TotalEntradas) {
      Neuronas = [];
      Salidas = [];

      //Genera las neuronas
      for (int Contador = 0; Contador < TotalNeuronas; Contador++) {
         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 Contador = 0; Contador < Neuronas.Count; Contador++)
			Salidas[Contador] = Neuronas[Contador].CalculaSalida(Entradas);
	}

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

class Neurona {
	//Los pesos para cada entrada
	public List<double> Pesos;

	//Nuevos pesos dados por el algoritmo de "backpropagation"
	public List<double> NuevosPesos;

	//El peso del umbral
	public double Umbral;

	//Nuevo umbral dado por el algoritmo de "backpropagation"
	public double NuevoUmbral;

	//Inicializa los pesos y umbral con un valor al azar
	public Neurona(Random Azar, int TotalEntradas) {
		Pesos = [];
		NuevosPesos = [];
		for (int Contador = 0; Contador < TotalEntradas; Contador++) {
			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 Contador = 0; Contador < Pesos.Count; Contador++)
			Valor += Entradas[Contador] * Pesos[Contador];
		Valor += Umbral;
		return 1 / (1 + Math.Exp(-Valor));
	}

	//Reemplaza viejos pesos por nuevos
	public void Actualiza() {
```

```
            for (int Contador = 0; Contador < Pesos.Count; Contador++)
               Pesos[Contador] = NuevosPesos[Contador];
            Umbral = NuevoUmbral;
         }
      }
   }
```

```
Consola de depuración de Mi

Ciclo: 1000
1 1 1 0 1 1 1  Esperada: 0 0 0 0  Calculada: 0 0 0 0
0 0 1 0 0 1 0  Esperada: 0 0 0 1  Calculada: 0 0 0 1
1 0 1 1 1 0 1  Esperada: 0 0 1 0  Calculada: 0 0 1 0
1 0 1 1 0 1 1  Esperada: 0 0 1 1  Calculada: 0 0 1 1
0 1 1 1 0 1 0  Esperada: 0 1 0 0  Calculada: 1 0 0 0
1 1 0 1 0 1 1  Esperada: 0 1 0 1  Calculada: 0 1 0 1
1 1 0 1 1 1 1  Esperada: 0 1 1 0  Calculada: 0 0 1 0
1 0 1 0 0 1 0  Esperada: 0 1 1 1  Calculada: 0 1 1 1
1 1 1 1 1 1 1  Esperada: 1 0 0 0  Calculada: 1 0 0 0
1 1 1 1 0 1 1  Esperada: 1 0 0 1  Calculada: 0 1 0 1
```

Ilustración 51: Aprendiendo los patrones para identificar el número digital.

```
Ciclo: 10000
1 1 1 0 1 1 1  Esperada: 0 0 0 0  Calculada: 0 0 0 0
0 0 1 0 0 1 0  Esperada: 0 0 0 1  Calculada: 0 0 0 1
1 0 1 1 1 0 1  Esperada: 0 0 1 0  Calculada: 0 0 1 0
1 0 1 1 0 1 1  Esperada: 0 0 1 1  Calculada: 0 0 1 1
0 1 1 1 0 1 0  Esperada: 0 1 0 0  Calculada: 0 1 0 0
1 1 0 1 0 1 1  Esperada: 0 1 0 1  Calculada: 0 1 0 1
1 1 0 1 1 1 1  Esperada: 0 1 1 0  Calculada: 0 1 1 0
1 0 1 0 0 1 0  Esperada: 0 1 1 1  Calculada: 0 1 1 1
1 1 1 1 1 1 1  Esperada: 1 0 0 0  Calculada: 1 0 0 0
1 1 1 1 0 1 1  Esperada: 1 0 0 1  Calculada: 1 0 0 1
Finaliza el entrenamiento
```

Ilustración 52: Aprendió los patrones para identificar el número digital.

Detección de patrones en series de tiempo

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 mientras 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.

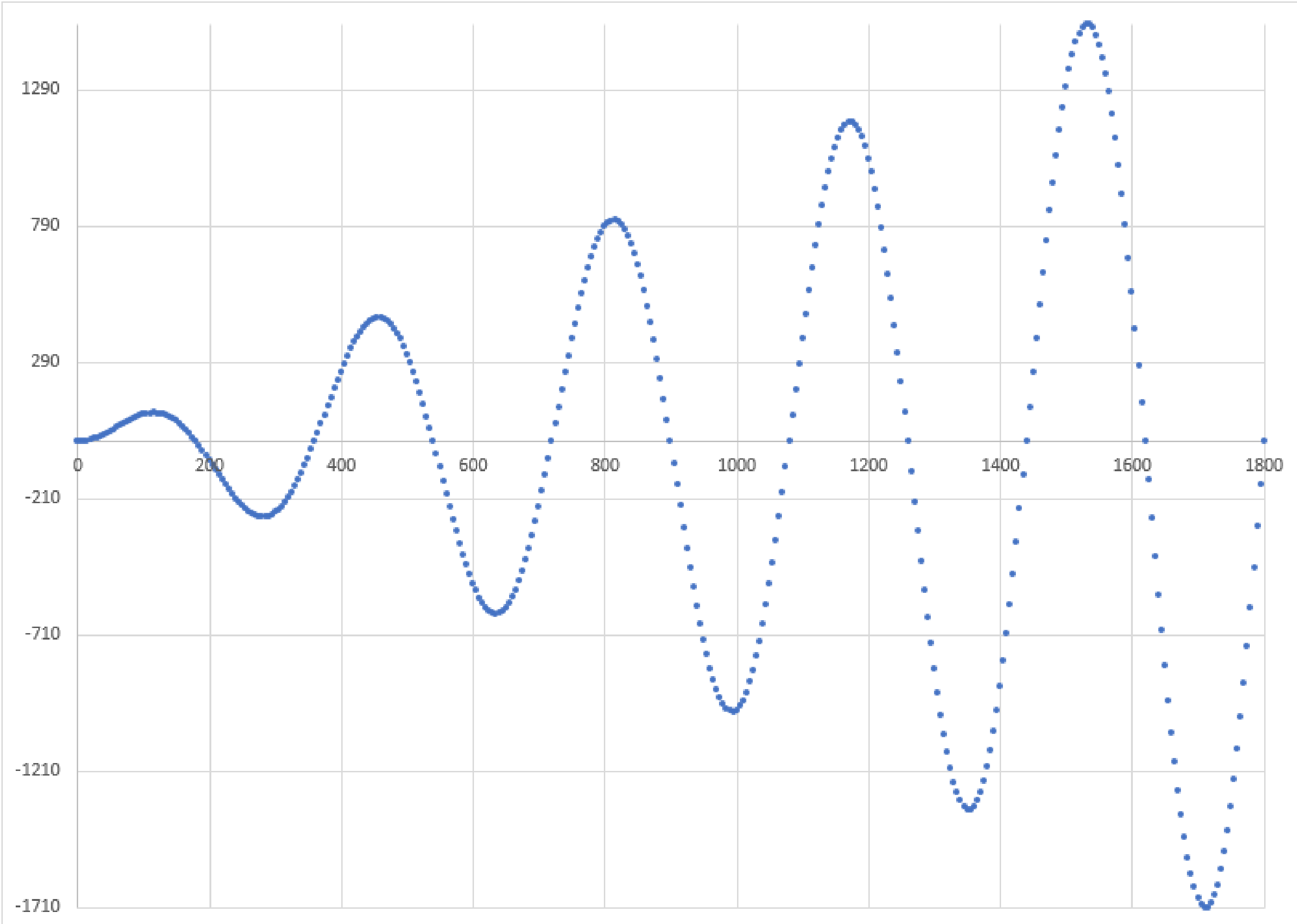


Ilustración 53: Gráfico generado por los puntos

Uniendo los puntos, se obtendría

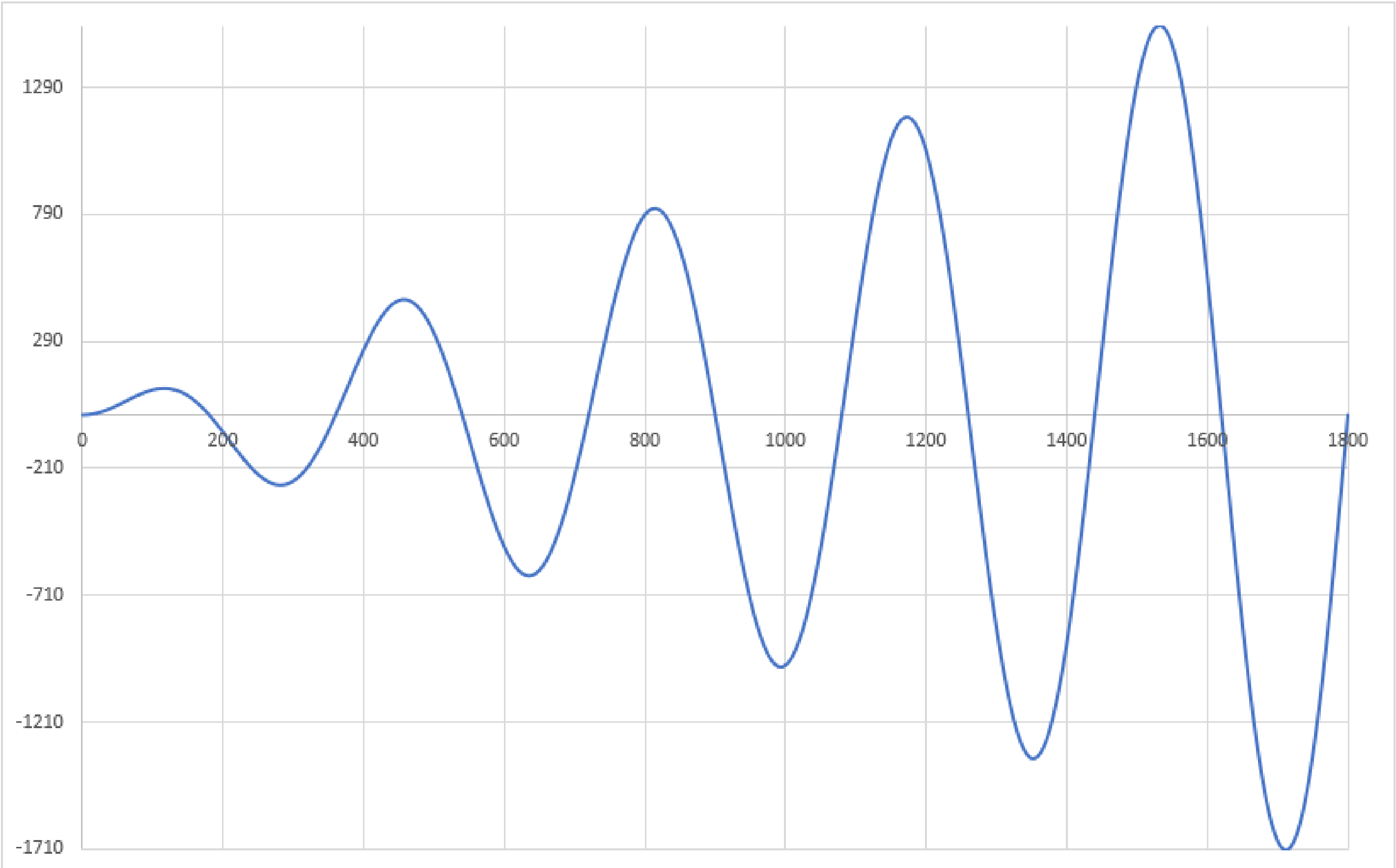


Ilustración 54: Gráfico uniendo los puntos

El primer paso es convertir los valores de X y de Y que dan origen al gráfico, en valores entre 0 y 1 porque la función de activación del perceptrón multicapa $y = \frac{1}{1+e^{-x}}$ solo genera números entre 0 y 1.

Se hace entonces una normalización usando la siguiente fórmula

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

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

Como indica la fórmula, habría que recorrer todos los datos de X y Y para deducir el mayor valor de X, menor valor de X, mayor valor de Y y menor valor de Y.

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

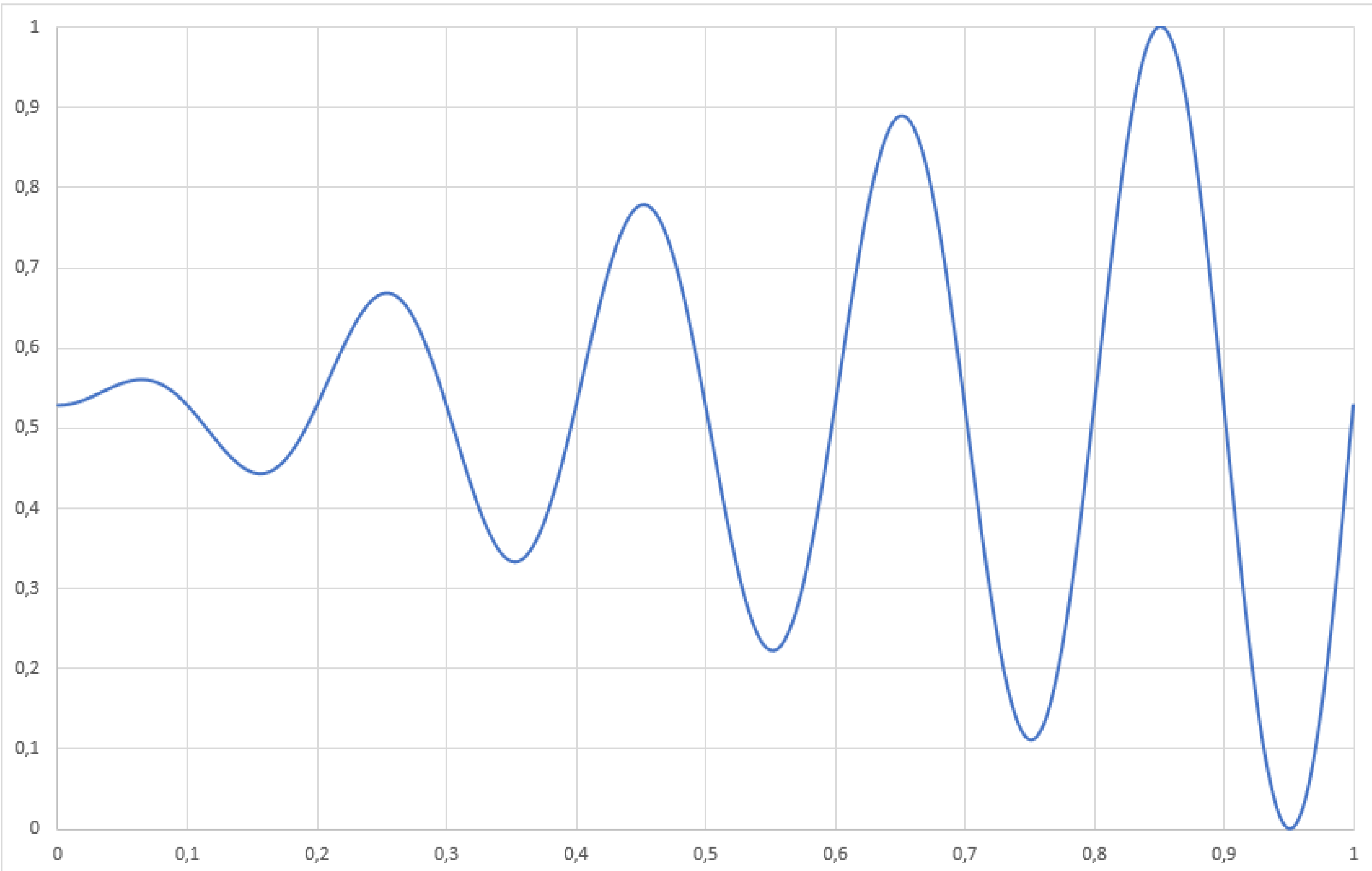


Ilustración 55: Gráfico al normalizar los datos

Con esos datos normalizados, se alimenta la red neuronal para entrenarla. Este sería el código en C#:

K/017.cs

```
namespace Ejemplo {
   class Program {
      static void Main()  {
         Random Azar = new();

         //Genera una ecuación al azar para generar el dataset
         double cfA, cfB, cfC, cfD;
```

```
double cfE, cfF, cfG, cfH, cfI;

double cfMin = -10;
double cfMax = 10;
cfA = Azar.NextDouble() * (cfMax - cfMin) + cfMin;
cfB = Azar.NextDouble() * (cfMax - cfMin) + cfMin;
cfC = Azar.NextDouble() * (cfMax - cfMin) + cfMin;
cfD = Azar.NextDouble() * (cfMax - cfMin) + cfMin;
cfE = Azar.NextDouble() * (cfMax - cfMin) + cfMin;
cfF = Azar.NextDouble() * (cfMax - cfMin) + cfMin;
cfG = Azar.NextDouble() * (cfMax - cfMin) + cfMin;
cfH = Azar.NextDouble() * (cfMax - cfMin) + cfMin;
cfI = Azar.NextDouble() * (cfMax - cfMin) + cfMin;

List<double> Xentra = new List<double>();
List<double> Ysale = new List<double>();

//Genera el dataset con esta ecuación
double Y;
for (double X = -360; X <= 360; X++) {
   Y = cfA * Math.Sin((cfB * X + cfC) * Math.PI / 180);
   Y += cfD * Math.Sin((cfE * X + cfF) * Math.PI / 180);
   Y += cfG * Math.Sin((cfH * X + cfI) * Math.PI / 180);

   Xentra.Add(X);
   Ysale.Add(Y);
}

//Con los datos de entrada y salida
//generados (el dataset), ahora procede
//a normalizarlos
double MinX = double.MaxValue;
double MaxX = double.MinValue;

double MinY = double.MaxValue;
double MaxY = double.MinValue;

for (int Datos = 0; Datos < Ysale.Count; Datos++) {
   if (Xentra[Datos] < MinX) MinX = Xentra[Datos];
   if (Xentra[Datos] > MaxX) MaxX = Xentra[Datos];
   if (Ysale[Datos] < MinY) MinY = Ysale[Datos];
   if (Ysale[Datos] > MaxY) MaxY = Ysale[Datos];
}

for (int Datos = 0; Datos < Ysale.Count; Datos++) {
   Xentra[Datos] = (Xentra[Datos] - MinX) / (MaxX - MinX);
   Ysale[Datos] = (Ysale[Datos] - MinY) / (MaxY - MinY);
}
```

```
//Con los datos normalizados, ahora se entrena
//a la red neuronal para que detecte el patrón
int TotalEntradas = 1; //Número de entradas
int NeuronasCapa0 = 5; //Total neuronas en la capa 0
int NeuronasCapa1 = 5; //Total neuronas en la capa 1
int NeuronasCapa2 = 1; //Total neuronas en la capa 2
Perceptron RedNeuronal = new(TotalEntradas, NeuronasCapa0,
                   NeuronasCapa1, NeuronasCapa2);

//Esta será la única entrada externa
//al perceptrón, es decir, X
List<double> Entrada = [0];

//Esta será la salida esperada
//externa al perceptrón, es decir, Y
List<double> SalidaEsperada = [0];

//Ciclo que entrena la red neuronal
int TotalCiclos = 25000; //Ciclos de entrenamiento
for (int Ciclo = 1; Ciclo <= TotalCiclos; Ciclo++) {

   if (Ciclo % 5000 == 0)
      Console.WriteLine("Ciclo: " + Ciclo);

   //Por cada ciclo, se entrena
   //el perceptrón con toda la tabla
   for (int Cnj = 0; Cnj < Xentra.Count; Cnj++) {

      //Entrada y salida esperadas
      Entrada[0] = Xentra[Cnj];
      SalidaEsperada[0] = Ysale[Cnj];

      //Primero calcula la salida
      //del perceptrón con esa entrada
      RedNeuronal.CalculaSalida(Entrada);

      //Luego entrena el perceptrón para
      //ajustar los pesos y umbrales
      RedNeuronal.Entrena(Entrada, SalidaEsperada);
   }
}

Console.Write("Entrada normalizada;");
Console.Write("Salida esperada normalizada;");
Console.WriteLine("Salida perceptrón normalizada");

for (int Cnj = 0; Cnj < Xentra.Count; Cnj++) {
```

```
         //Entradas y salidas esperadas
         Entrada[0] = Xentra[Cnj];
         SalidaEsperada[0] = Ysale[Cnj];

         //Calcula la salida del perceptrón con esas entradas
         RedNeuronal.CalculaSalida(Entrada);

         //Muestra la salida
         RedNeuronal.SalidaPerceptron(Entrada, SalidaEsperada);
      }
      Console.WriteLine("Finaliza el entrenamiento");
   }
}

class Perceptron {
   public List<Capa> Capas;

   //Imprime los datos de las diferentes capas
   public void SalidaPerceptron(List<double> Entradas,
                         List<double> SalidaEsperada) {
      //Personalizado para series de tiempo:
      //Una entrada y una salida
      Console.Write(Entradas[0] + ";");
      Console.Write(SalidaEsperada[0] + ";");
      Console.WriteLine(Capas[2].Salidas[0]);
   }

   //Crea las diversas capas
   public Perceptron(int TotalEntradas, int NeuronasCapa0,
               int NeuronasCapa1, int NeuronasCapa2) {
      Random Azar = new();
      Capas =
      [
         new Capa(Azar, NeuronasCapa0, TotalEntradas), //Crea la capa 0
         new Capa(Azar, NeuronasCapa1, NeuronasCapa0), //Crea la capa 1
         new Capa(Azar, NeuronasCapa2, NeuronasCapa1), //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 NeuronasCapa0 = Capas[0].Neuronas.Count;
   int NeuronasCapa1 = Capas[1].Neuronas.Count;
   int NeuronasCapa2 = Capas[2].Neuronas.Count;

   //Factor de aprendizaje
   double Alpha = 0.4;

   //=====================
   //Procesa pesos capa 2
   //=====================

   //Va de neurona en neurona de la capa 1
   for (int j = 0; j < NeuronasCapa1; j++)

      //Va de neurona en neurona de la capa de salida (capa 2)
      for (int i = 0; i < NeuronasCapa2; i++) {

         //Salida de la neurona de la capa de salida
         double Yi = Capas[2].Salidas[i];

         //Salida esperada
         double Si = SalidaEsperada[i];

         //Salida de la capa 1
         double a1j = Capas[1].Salidas[j];

         //La fórmula del error
         double dE2 = a1j * (Yi - Si) * Yi * (1 - Yi);

         //Ajusta el nuevo peso
         double Nuevo = Capas[2].Neuronas[i].Pesos[j] - Alpha * dE2;
         Capas[2].Neuronas[i].NuevosPesos[j] = Nuevo;
      }

   //=====================
   //Procesa pesos capa 1
   //=====================

   //Va de neurona en neurona de la capa 0
   for (int j = 0; j < NeuronasCapa0; j++)
```

```
        //Va de neurona en neurona de la capa 1
        for (int k = 0; k < NeuronasCapa1; k++) {
          double Acumula = 0;

          //Va de neurona en neurona de la capa 2
          for (int i = 0; i < NeuronasCapa2; i++) {

            //Salida de la capa 2
            double Yi = Capas[2].Salidas[i];

            //Salida esperada
            double Si = SalidaEsperada[i];
            double W2ki = Capas[2].Neuronas[i].Pesos[k];
            Acumula += 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) * Acumula;
          double Nuevo = Capas[1].Neuronas[k].Pesos[j] - Alpha * dE1;
          Capas[1].Neuronas[k].NuevosPesos[j] = Nuevo;
        }

      //=====================
      //Procesa pesos capa 0
      //=====================

      //Va de entrada en entrada
      for (int j = 0; j < Entradas.Count; j++)

        //Va de neurona en neurona de la capa 0
        for (int k = 0; k < NeuronasCapa0; k++) {
          double Acumula = 0;

          //Va de neurona en neurona de la capa 1
          for (int p = 0; p < NeuronasCapa1; p++) {
            double InternoAcumula = 0;

            //Va de neurona en neurona de la capa 2
            for (int i = 0; i < NeuronasCapa2; i++) {
              double Yi = Capas[2].Salidas[i];
              double Si = SalidaEsperada[i]; //Salida esperada
              double W2pi = Capas[2].Neuronas[i].Pesos[p];

              //Sumatoria interna
              InternoAcumula += W2pi * (Yi - Si) * Yi * (1 - Yi);
            }
            double W1kp = Capas[1].Neuronas[p].Pesos[k];
            double a1p = Capas[1].Salidas[p];
```

```
          //Sumatoria externa
          Acumula += W1kp * a1p * (1 - a1p) * InternoAcumula;
        }
        double xj = Entradas[j];
        double a0k = Capas[0].Salidas[k];
        double dE0 = xj * a0k * (1 - a0k) * Acumula;
        double W0jk = Capas[0].Neuronas[k].Pesos[j];
        Capas[0].Neuronas[k].NuevosPesos[j] = W0jk - Alpha * dE0;
      }

    //========================
    //Procesa umbrales capa 2
    //========================

    //Va de neurona en neurona de la capa de salida (capa 2)
    for (int i = 0; i < NeuronasCapa2; i++) {

      //Salida de la neurona de la capa de salida
      double Yi = Capas[2].Salidas[i];

      //Salida esperada
      double Si = SalidaEsperada[i];
      double dE2 = (Yi - Si) * Yi * (1 - Yi);
      double Nuevo = Capas[2].Neuronas[i].Umbral - Alpha * dE2;
      Capas[2].Neuronas[i].NuevoUmbral = Nuevo;
    }

    //========================
    //Procesa umbrales capa 1
    //========================

    //Va de neurona en neurona de la capa 1
    for (int k = 0; k < NeuronasCapa1; k++) {
      double Acumula = 0;

      //Va de neurona en neurona de la capa 2
      for (int i = 0; i < NeuronasCapa2; i++) {

        //Salida de la capa 2
        double Yi = Capas[2].Salidas[i];
        double Si = SalidaEsperada[i];
        double W2ki = Capas[2].Neuronas[i].Pesos[k];
        Acumula += W2ki * (Yi - Si) * Yi * (1 - Yi);
      }
      double a1k = Capas[1].Salidas[k];
      double dE1 = a1k * (1 - a1k) * Acumula;
      double Nuevo = Capas[1].Neuronas[k].Umbral - Alpha * dE1;
```

```
         Capas[1].Neuronas[k].NuevoUmbral = Nuevo;
      }

      //=======================
      //Procesa umbrales capa 0
      //=======================

      //Va de neurona en neurona de la capa 0
      for (int k = 0; k < NeuronasCapa0; k++) {
         double Acumula = 0;

         //Va de neurona en neurona de la capa 1
         for (int p = 0; p < NeuronasCapa1; p++) {
            double InternoAcumula = 0;

            //Va de neurona en neurona de la capa 2
            for (int i = 0; i < NeuronasCapa2; i++) {
               double Yi = Capas[2].Salidas[i];
               double Si = SalidaEsperada[i];
               double W2pi = Capas[2].Neuronas[i].Pesos[p];
               InternoAcumula += W2pi * (Yi - Si) * Yi * (1 - Yi);
            }
            double W1kp = Capas[1].Neuronas[p].Pesos[k];
            double a1p = Capas[1].Salidas[p];
            Acumula += W1kp * a1p * (1 - a1p) * InternoAcumula;
         }
         double a0k = Capas[0].Salidas[k];
         double dE0 = a0k * (1 - a0k) * Acumula;
         double Nuevo = Capas[0].Neuronas[k].Umbral - Alpha * dE0;
         Capas[0].Neuronas[k].NuevoUmbral = Nuevo;
      }

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

}

class Capa {
   //Las neuronas que tendrá la capa
   public List<Neurona> Neuronas;

   //Almacena las salidas de cada neurona
   public List<double> Salidas;

   public Capa(Random Azar, int TotalNeuronas, int TotalEntradas) {
```

```
      Neuronas = [];
      Salidas = [];

      //Genera las neuronas
      for (int Contador = 0; Contador < TotalNeuronas; Contador++) {
         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 Contador = 0; Contador < Neuronas.Count; Contador++)
         Salidas[Contador] = Neuronas[Contador].CalculaSalida(Entradas);
   }

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

class Neurona {
   //Los pesos para cada entrada
   public List<double> Pesos;

   //Nuevos pesos dados por el algoritmo de "backpropagation"
   public List<double> NuevosPesos;

   //El peso del umbral
   public double Umbral;

   //Nuevo umbral dado por el algoritmo de "backpropagation"
   public double NuevoUmbral;

   //Inicializa los pesos y umbral con un valor al azar
   public Neurona(Random Azar, int TotalEntradas) {
      Pesos = [];
      NuevosPesos = [];
      for (int Contador = 0; Contador < TotalEntradas; Contador++) {
         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 Contador = 0; Contador < Pesos.Count; Contador++)
            Valor += Entradas[Contador] * Pesos[Contador];
         Valor += Umbral;
         return 1 / (1 + Math.Exp(-Valor));
      }

      //Reemplaza viejos pesos por nuevos
      public void Actualiza() {
         for (int Contador = 0; Contador < Pesos.Count; Contador++)
            Pesos[Contador] = NuevosPesos[Contador];
         Umbral = NuevoUmbral;
      }
   }
}
```

Un ejemplo de cómo la red neuronal se adapta a la serie:

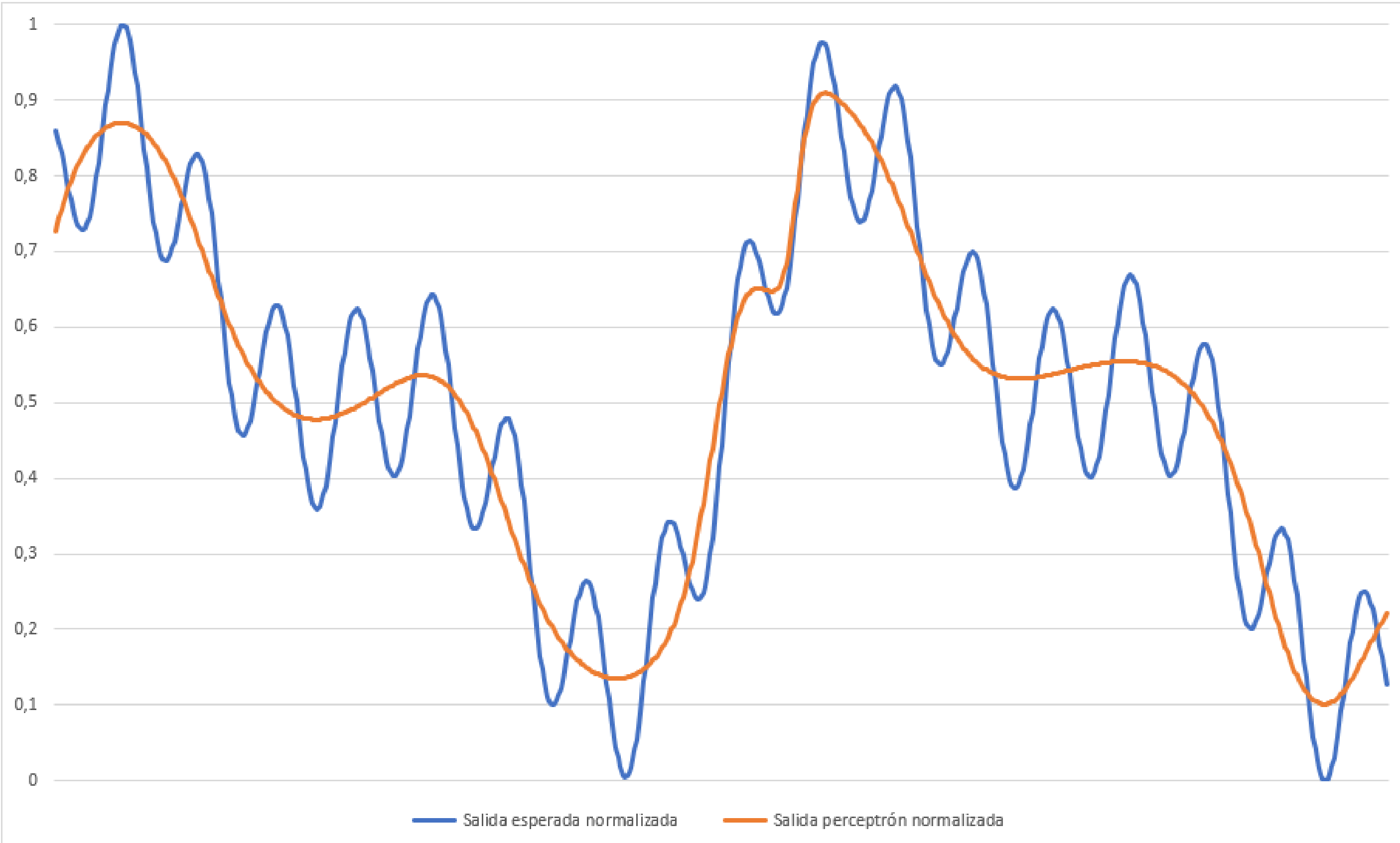


Ilustración 56: Red neuronal adaptándose a una serie temporal

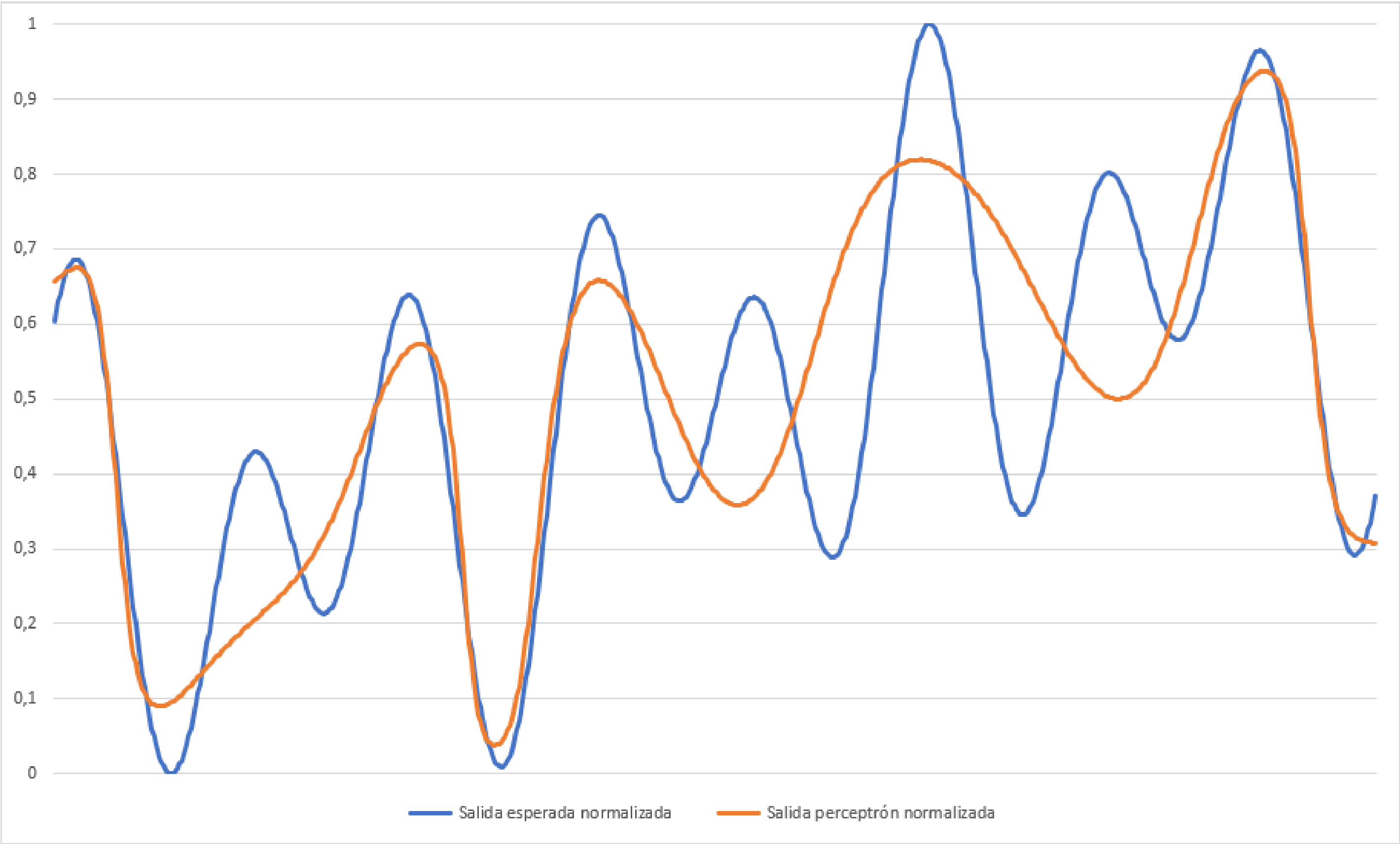


Ilustración 57: Red neuronal adaptándose a una serie temporal

Comparativa de red neuronal vs algoritmo evolutivo

Llegó el momento de poner a prueba los dos métodos de detección de patrones. Se usarán datos reales.

El algoritmo evolutivo ha cambiado, este es el nuevo:

```
Algoritmo Evolutivo
Inicio
    Leer archivo CSV con dos campos X,Y (donde X no se repite)
    Normalizar los datos X,Y leídos (quedando entre 0 y 1 ambos)
    Crear población al azar. Cada individuo del tipo Y = a*seno(b*X+c) + … + p*seno(q*X+r)
    Mientras tiempo de evaluación no se agote hacer
        Tomar individuo de la población y comparar su ajuste a los datos leídos normalizados
        Modificar algún coeficiente del individuo al azar
        Si la modificación mejora, se conserva, de lo contrario, restaura el valor anterior
    Fin mientras
    Revisar en la población, cuál individuo generó el mejor ajuste (el más cercano a cero)
Fin
```

Debido a la complejidad del código se opta por dividirlo en varios archivos .CS, todos quedan bajo la carpeta K/018. Estos son los programas:

K/018/Program.cs

```
/* Algoritmo evolutivo para detección de patrones
 * (uso de series de Fourier)
 * Autor: Rafael Alberto Moreno Parra
 *
 * Problema:
 * Dado un conjunto de datos del tipo X, Y
 * donde X es la variable independiente y
 * Y la variable dependiente,
 * hallar la función Y = F(X) que más
 * coincida con ese conjunto de datos.
 * */

using System.Text;

namespace Ejemplo {

   class Program {
      static void Main(string[] args) {
         //Generador de números aleatorios único
```

```
Random Azar = new();

//=========================================
//Configuración de la operación de ajuste
//=========================================

//Cuántos milisegundos dará a cada algoritmo
long TiempoParaOperar = 30000;

//=========================================
//Configuración del algoritmo evolutivo
//=========================================
int TamanoPoblacion = 200;

//==================================
//Configuración de la red neuronal
//==================================
int NeuronasCapa0 = 7; //Total neuronas en la capa 0
int NeuronasCapa1 = 7; //Total neuronas en la capa 1

//=====================================
//Imprime los datos de la comparativa
//=====================================
Console.OutputEncoding = Encoding.UTF8;
Console.WriteLine("Algoritmo Evolutivo vs Red Neuronal\r\n");
Console.WriteLine("Series de tiempo.\r\n");
Console.WriteLine("Milisegundos: " + TiempoParaOperar);
Console.WriteLine("\r\n\r\nAlgoritmo Evolutivo");
Console.WriteLine("Tamaño de la población: " + TamanoPoblacion);
Console.WriteLine("\r\n\r\nRed Neuronal: Perceptrón multicapa");
Console.WriteLine("Neuronas en Capa 1 oculta: " + NeuronasCapa0);
Console.WriteLine("Neuronas en Capa 2 oculta: " + NeuronasCapa1);

//================================
//Leer los datos del archivo CSV
//================================
DatosArchivo Datos = new();
Datos.LeeXYdeCSV(args[0]);

//===================================
//Algoritmo Evolutivo
//===================================

//Configura la población que se
//adaptará al conjunto de datos
Poblacion poblacion = new(Azar, TamanoPoblacion);

//Proceso de buscar el mejor individuo
```

```
        //al conjunto de datos generado
        List<double> ResultadoEvolutivo = [];
        poblacion.Proceso(Azar, Datos.XentradaN,
                      Datos.YsalidasN,
                      TiempoParaOperar,
                      ResultadoEvolutivo);

        //==================================
        //Red Neuronal
        //==================================
        RedesNeuronales Redes = new();
        List<double> ResultadoNeuronal = [];
        Redes.Proceso(Azar, TiempoParaOperar,
                   NeuronasCapa0, NeuronasCapa1,
                   Datos, ResultadoNeuronal);

        //==================================
        //Imprime la comparativa
        //==================================
        double AjusteEvolutivo = 0;
        double AjusteNeuronal = 0;
        Console.Write("\r\nEntrada;Salida Esperada;");
        Console.WriteLine("Salida Evolutivo;Salida Red Neuronal");

        for (int Cont = 0; Cont < Datos.Xentrada.Count; Cont++) {
           double valE = Datos.Xentrada[Cont];
           double valS = Datos.Ysalidas[Cont];
           Console.Write( valE + ";" + valS + ";");

           double valA = ResultadoEvolutivo[Cont] / 1000;
           double valB = Datos.Ysalidas.Max();
           double valC = Datos.Ysalidas.Min();
           double Evolutivo = valA * (valB - valC) + valC;

           double valN = ResultadoNeuronal[Cont];
           Console.WriteLine(Evolutivo + ";" + valN);

           AjusteEvolutivo += Math.Abs(valS - Evolutivo);
           AjusteNeuronal += Math.Abs(valS - valN);
        }
        Console.WriteLine("\r\nAjuste Evolutivo: " +  AjusteEvolutivo);
        Console.WriteLine("Ajuste Neuronal: " + AjusteNeuronal);
        Console.WriteLine("\r\nFINAL\r\n");
      }
   }
}
```

```
using System.Globalization;

namespace Ejemplo {
   internal class DatosArchivo {
      //Los datos seleccionados para
      //encontrar patrón de comportamiento
      public List<double> Xentrada;
      public List<double> Ysalidas;

      //Los datos normalizados para la red neuronal
      public List<double> XentradaN;
      public List<double> YsalidasN;

      //Lee los valores X y Y.
      public void LeeXYdeCSV(string urlArchivo) {
         //Empieza a leer el archivo
         var Archivo = new StreamReader(urlArchivo);

         //Inicializa las listas
         Xentrada = [];
         Ysalidas = [];
         XentradaN = [];
         YsalidasN = [];

         //Lee la linea de los dos datos numéricos
         string LineaDato;
         double valX, valY;
         while ((LineaDato = Archivo.ReadLine()) != null) {
            int Coma = LineaDato.IndexOf(',');
            string Xc = LineaDato[..Coma];
            string Yc = LineaDato[(Coma + 1)..];
            valX = double.Parse(Xc, CultureInfo.InvariantCulture);
            valY = double.Parse(Yc, CultureInfo.InvariantCulture);
            Xentrada.Add(valX);
            Ysalidas.Add(valY);
         }

         //Normaliza los datos para la red neuronal
         double MaximoX = Xentrada.Max();
         double MinimoX = Xentrada.Min();
         double MaximoY = Ysalidas.Max();
         double MinimoY = Ysalidas.Min();
         double DividirX = MaximoX - MinimoX;
         double DividirY = MaximoY - MinimoY;
         for (int Cont = 0; Cont < Xentrada.Count; Cont++) {
            XentradaN.Add((Xentrada[Cont] - MinimoX) / DividirX);
            YsalidasN.Add((Ysalidas[Cont] - MinimoY) / DividirY);
```

```
            }
        }
    }
}
```

K/018/RedesNeuronales.cs

```
using System.Diagnostics;

namespace Ejemplo {
   internal class RedesNeuronales {
      public void Proceso(Random Azar, long TiempoParaOperar,
                     int NeuronasCapa0, int NeuronasCapa1,
                     DatosArchivo Datos,
                     List<double> ResultadoNeuronal) {
         //======================================
         //Red Neuronal
         //======================================
         int TotalEntradas = 1; //Número de entradas
         int NeuronasCapa2 = 1; //Total neuronas en la capa 2
         Perceptron RedNeuronal = new(Azar, TotalEntradas,
                                   NeuronasCapa0,
                                   NeuronasCapa1, NeuronasCapa2);

         //Esta será la única entrada externa
         //al perceptrón, es decir, X
         List<double> Entrada = [0];

         //Esta será la salida esperada externa
         //al perceptrón, es decir, Y
         List<double> SalidaEsperada = [0];

         //Medidor de tiempos
         Stopwatch cronometro = new();
         cronometro.Reset();
         cronometro.Start();

         //Tiempo que repetirá el proceso evolutivo
         while (cronometro.ElapsedMilliseconds < TiempoParaOperar) {

            //Por cada iteración, se entrena el
            //perceptrón con toda la tabla
            for (int Cnj = 0; Cnj < Datos.XentradaN.Count; Cnj++) {

               //Entrada y salida esperadas
               Entrada[0] = Datos.XentradaN[Cnj];
               SalidaEsperada[0] = Datos.YsalidasN[Cnj];
```

```
                //Primero calcula la salida del
                //perceptrón con esa entrada
                RedNeuronal.CalculaSalida(Entrada);

                //Luego entrena el perceptrón para
                //ajustar los pesos y umbrales
                RedNeuronal.Entrena(Entrada, SalidaEsperada);
            }
        }

        //Muestra el ajuste de la red neuronal
        //a los datos dados.
        ResultadoNeuronal.Clear();
        for (int Cnj = 0; Cnj < Datos.XentradaN.Count; Cnj++) {
            //Entradas y salidas esperadas
            Entrada[0] = Datos.XentradaN[Cnj];
            SalidaEsperada[0] = Datos.YsalidasN[Cnj];

            //Calcula la salida del perceptrón con esas entradas
            RedNeuronal.CalculaSalida(Entrada);

            //Las salidas de la red desnormalizadas
            double valA = RedNeuronal.Capas[2].Salidas[0];
            double valB = Datos.Ysalidas.Max();
            double valC = Datos.Ysalidas.Min();
            double Salir = valA * (valB - valC) + valC;
            ResultadoNeuronal.Add(Salir);
        }
      }
   }
}
```

K/018/Perceptron.cs

```
namespace Ejemplo {
   internal class Perceptron {
      public List<Capa> Capas;

      //Imprime los datos de las diferentes capas
      public void SalidaPerceptron(List<double> Entradas,
                              List<double> SalidaEsperada) {
         //Personalizado para series de tiempo:
         //Una entrada y una salida
         Console.Write(Entradas[0] + ";");
         Console.Write(SalidaEsperada[0] + ";");
         Console.WriteLine(Capas[2].Salidas[0]);
      }
```

```
    //Crea las diversas capas
    public Perceptron(Random Azar, int TotalEntradas, int NeuronasCapa0,
                 int NeuronasCapa1, int NeuronasCapa2) {
       Capas =
       [
          new Capa(Azar, NeuronasCapa0, TotalEntradas), //Crea la capa 0
          new Capa(Azar, NeuronasCapa1, NeuronasCapa0), //Crea la capa 1
          new Capa(Azar, NeuronasCapa2, NeuronasCapa1), //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 NeuronasCapa0 = Capas[0].Neuronas.Count;
       int NeuronasCapa1 = Capas[1].Neuronas.Count;
       int NeuronasCapa2 = Capas[2].Neuronas.Count;

       //Factor de aprendizaje
       double Alpha = 0.4;

       //====================
       //Procesa pesos capa 2
       //====================

       //Va de neurona en neurona de la capa 1
       for (int j = 0; j < NeuronasCapa1; j++)

          //Va de neurona en neurona de la capa de salida (capa 2)
          for (int i = 0; i < NeuronasCapa2; i++) {

             //Salida de la neurona de la capa de salida
             double Yi = Capas[2].Salidas[i];

             //Salida esperada
```

```
            double Si = SalidaEsperada[i];

            //Salida de la capa 1
            double a1j = Capas[1].Salidas[j];

            //La fórmula del error
            double dE2 = a1j * (Yi - Si) * Yi * (1 - Yi);

            //Ajusta el nuevo peso
            double Nuevo = Capas[2].Neuronas[i].Pesos[j] - Alpha * dE2;
            Capas[2].Neuronas[i].NuevosPesos[j] = Nuevo;
         }

      //======================
      //Procesa pesos capa 1
      //======================

      //Va de neurona en neurona de la capa 0
      for (int j = 0; j < NeuronasCapa0; j++)

         //Va de neurona en neurona de la capa 1
         for (int k = 0; k < NeuronasCapa1; k++) {
            double Acumula = 0;

            //Va de neurona en neurona de la capa 2
            for (int i = 0; i < NeuronasCapa2; i++) {

               //Salida de la capa 2
               double Yi = Capas[2].Salidas[i];

               //Salida esperada
               double Si = SalidaEsperada[i];
               double W2ki = Capas[2].Neuronas[i].Pesos[k];
               Acumula += 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) * Acumula;
            double Nuevo = Capas[1].Neuronas[k].Pesos[j] - Alpha * dE1;
            Capas[1].Neuronas[k].NuevosPesos[j] = Nuevo;
         }

      //======================
      //Procesa pesos capa 0
      //======================

      //Va de entrada en entrada
      for (int j = 0; j < Entradas.Count; j++)
```

```
    //Va de neurona en neurona de la capa 0
    for (int k = 0; k < NeuronasCapa0; k++) {
       double Acumula = 0;

       //Va de neurona en neurona de la capa 1
       for (int p = 0; p < NeuronasCapa1; p++) {
          double InternoAcumula = 0;

          //Va de neurona en neurona de la capa 2
          for (int i = 0; i < NeuronasCapa2; i++) {
             double Yi = Capas[2].Salidas[i];
             double Si = SalidaEsperada[i]; //Salida esperada
             double W2pi = Capas[2].Neuronas[i].Pesos[p];

             //Sumatoria interna
             InternoAcumula += W2pi * (Yi - Si) * Yi * (1 - Yi);
          }
          double W1kp = Capas[1].Neuronas[p].Pesos[k];
          double a1p = Capas[1].Salidas[p];

          //Sumatoria externa
          Acumula += W1kp * a1p * (1 - a1p) * InternoAcumula;
       }
       double xj = Entradas[j];
       double a0k = Capas[0].Salidas[k];
       double dE0 = xj * a0k * (1 - a0k) * Acumula;
       double W0jk = Capas[0].Neuronas[k].Pesos[j];
       Capas[0].Neuronas[k].NuevosPesos[j] = W0jk - Alpha * dE0;
    }

 //========================
 //Procesa umbrales capa 2
 //========================

 //Va de neurona en neurona de la capa de salida (capa 2)
 for (int i = 0; i < NeuronasCapa2; i++) {

    //Salida de la neurona de la capa de salida
    double Yi = Capas[2].Salidas[i];

    //Salida esperada
    double Si = SalidaEsperada[i];
    double dE2 = (Yi - Si) * Yi * (1 - Yi);
    double Nuevo = Capas[2].Neuronas[i].Umbral - Alpha * dE2;
    Capas[2].Neuronas[i].NuevoUmbral = Nuevo;
 }
```

```
//========================
//Procesa umbrales capa 1
//========================

//Va de neurona en neurona de la capa 1
for (int k = 0; k < NeuronasCapa1; k++) {
   double Acumula = 0;

   //Va de neurona en neurona de la capa 2
   for (int i = 0; i < NeuronasCapa2; i++) {

      //Salida de la capa 2
      double Yi = Capas[2].Salidas[i];
      double Si = SalidaEsperada[i];
      double W2ki = Capas[2].Neuronas[i].Pesos[k];
      Acumula += W2ki * (Yi - Si) * Yi * (1 - Yi);
   }
   double a1k = Capas[1].Salidas[k];
   double dE1 = a1k * (1 - a1k) * Acumula;
   double Nuevo = Capas[1].Neuronas[k].Umbral - Alpha * dE1;
   Capas[1].Neuronas[k].NuevoUmbral = Nuevo;
}

//========================
//Procesa umbrales capa 0
//========================

//Va de neurona en neurona de la capa 0
for (int k = 0; k < NeuronasCapa0; k++) {
   double Acumula = 0;

   //Va de neurona en neurona de la capa 1
   for (int p = 0; p < NeuronasCapa1; p++) {
      double InternoAcumula = 0;

      //Va de neurona en neurona de la capa 2
      for (int i = 0; i < NeuronasCapa2; i++) {
         double Yi = Capas[2].Salidas[i];
         double Si = SalidaEsperada[i];
         double W2pi = Capas[2].Neuronas[i].Pesos[p];
         InternoAcumula += W2pi * (Yi - Si) * Yi * (1 - Yi);
      }
      double W1kp = Capas[1].Neuronas[p].Pesos[k];
      double a1p = Capas[1].Salidas[p];
      Acumula += W1kp * a1p * (1 - a1p) * InternoAcumula;
   }
   double a0k = Capas[0].Salidas[k];
   double dE0 = a0k * (1 - a0k) * Acumula;
```

```
            double Nuevo = Capas[0].Neuronas[k].Umbral - Alpha * dE0;
            Capas[0].Neuronas[k].NuevoUmbral = Nuevo;
         }

         //Actualiza los pesos
         Capas[0].Actualiza();
         Capas[1].Actualiza();
         Capas[2].Actualiza();
      }
   }
}
```

K/018/Capa.cs

```
namespace Ejemplo {
   internal class Capa {
      //Las neuronas que tendrá la capa
      public List<Neurona> Neuronas;

      //Almacena las salidas de cada neurona
      public List<double> Salidas;

      public Capa(Random Azar, int TotalNeuronas, int TotalEntradas) {
         Neuronas = [];
         Salidas = [];

         //Genera las neuronas
         for (int Contador = 0; Contador < TotalNeuronas; Contador++) {
            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 Contador = 0; Contador < Neuronas.Count; Contador++)
            Salidas[Contador] = Neuronas[Contador].CalculaSalida(Entradas);
      }

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

```
namespace Ejemplo {
   internal class Neurona {
      //Los pesos para cada entrada
      public List<double> Pesos;

      //Nuevos pesos dados por el algoritmo de "backpropagation"
      public List<double> NuevosPesos;

      //El peso del umbral
      public double Umbral;

      //Nuevo umbral dado por el algoritmo de "backpropagation"
      public double NuevoUmbral;

      //Inicializa los pesos y umbral con un valor al azar
      public Neurona(Random Azar, int TotalEntradas) {
         Pesos = [];
         NuevosPesos = [];
         for (int Contador = 0; Contador < TotalEntradas; Contador++) {
            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 Contador = 0; Contador < Pesos.Count; Contador++)
            Valor += Entradas[Contador] * Pesos[Contador];
         Valor += Umbral;
         return 1 / (1 + Math.Exp(-Valor));
      }

      //Reemplaza viejos pesos por nuevos
      public void Actualiza() {
         for (int Contador = 0; Contador < Pesos.Count; Contador++)
            Pesos[Contador] = NuevosPesos[Contador];
         Umbral = NuevoUmbral;
      }
   }
}
```

```
using System.Diagnostics;

namespace Ejemplo {
   internal class Poblacion {
      //Para agilizar cálculos
      static private double Radian;

      //Los individuos
      double[][] Indiv;
      double[] Ajuste;

      //Inicializa la población con los individuos
      public Poblacion(Random Azar, int TamanoPoblacion) {
         Indiv = new double[TamanoPoblacion][];
         Ajuste = new double[TamanoPoblacion];

         Radian = Math.PI / 180;
         for (int cont = 0; cont < Indiv.Length; cont++) {
            Ajuste[cont] = double.MaxValue;
            Indiv[cont] = new double[30];
            for (int valor = 0; valor < 30; valor++) {
               Indiv[cont][valor] = Azar.NextDouble() * 2 - 1;
            }
         }
      }

      //Proceso evolutivo.
      public void Proceso(Random Azar, List<double> Entradas,
                   List<double> SalidaEsperada,
                   long TiempoParaOperar,
                   List<double> ResultadoEvolutivo) {

         //Medidor de tiempos
         Stopwatch cronometro = new();
         cronometro.Reset();
         cronometro.Start();

         //Tiempo que repetirá el proceso evolutivo
         int Turno = 0;
         while (cronometro.ElapsedMilliseconds <= TiempoParaOperar) {

            //Guarda los valores anteriores
            int CualMuta = Azar.Next(30);
            double ValorAnterior = Indiv[Turno][CualMuta];
            Indiv[Turno][CualMuta] += Azar.NextDouble() * 2 - 1;

            //Deduce el ajuste
```

```
    double NuevoAjuste = 0;
    for (int Xval = 0; Xval < Entradas.Count; Xval++) {

        //Entrada del ambiente
        double X = Entradas[Xval] * 1000;

        //Deduce el valor Y del individuo con
        //esa entrada X del ambiente
        double Y = 0;
        for (int cont=0; cont <= 27; cont += 3) {
            double valA = Indiv[Turno][cont];
            double valB = Indiv[Turno][cont + 1] * X;
            double valC = Indiv[Turno][cont + 2];
            Y += valA * Math.Sin((valB + valC) * Radian);
        }

        //Diferencia entre la salida calculada y la esperada
        NuevoAjuste += Math.Abs(Y - SalidaEsperada[Xval] * 1000);

        //Si el nuevo ajuste supera al
        //ajuste anterior, sale del ciclo
        if (NuevoAjuste > Ajuste[Turno]) break;
    }

    //Si el cambio mejora al individuo se conserva,
    //caso contrario se restaura el valor anterior
    if (NuevoAjuste > Ajuste[Turno])
        Indiv[Turno][CualMuta] = ValorAnterior;
    else
        Ajuste[Turno] = NuevoAjuste;

    if (++Turno >= Indiv.Length) Turno = 0;
}

//Imprime el mejor individuo
int Mejor = -1;
double MejorAjuste = double.MaxValue;
for (int cont = 0; cont < Indiv.Length; cont++)
    if (Ajuste[cont] < MejorAjuste) {
        MejorAjuste = Ajuste[cont];
        Mejor = cont;
    }

for (int Xval = 0; Xval < Entradas.Count; Xval++) {
    //Entrada del ambiente
    double X = Entradas[Xval] * 1000;

    //Deduce el valor Y del individuo con
```

```
            //esa entrada X del ambiente
            double Y = 0;
            for (int cont = 0; cont <= 27; cont += 3) {
               double valA = Indiv[Mejor][cont];
               double valB = Indiv[Mejor][cont + 1] * X;
               double valC = Indiv[Mejor][cont + 2];
               Y += valA * Math.Sin((valB + valC) * Radian);
            }
            ResultadoEvolutivo.Add(Y);
         }
      }
   }
}
```

El algoritmo evolutivo está en el archivo Poblacion.cs , estas son sus características:

1. Cada individuo es del tipo Y = a*seno(b*X+c) + ….. p*seno(q*X+r) dónde se considera como bloque cada subexpresión: a*seno(b*X+c), por ahora está fijo a 10 bloques, que serían 30 coeficientes, los cuáles son los que varían.
2. Los datos normalizados están entre 0 y 1, por ese motivo se multiplica por 1000 el valor de cada X normalizado. ¿Por qué? para que la expresión seno() pueda tener un claro comportamiento sinusoidal. El ajuste se calcula multiplicando el Y normalizado por 1000 y comparándolo (diferencia absoluta) con el Y obtenido por el individuo.

Prueba #1: Números primos

Se busca el patrón de los primeros 100 números primos. Este es el archivo de entrada:

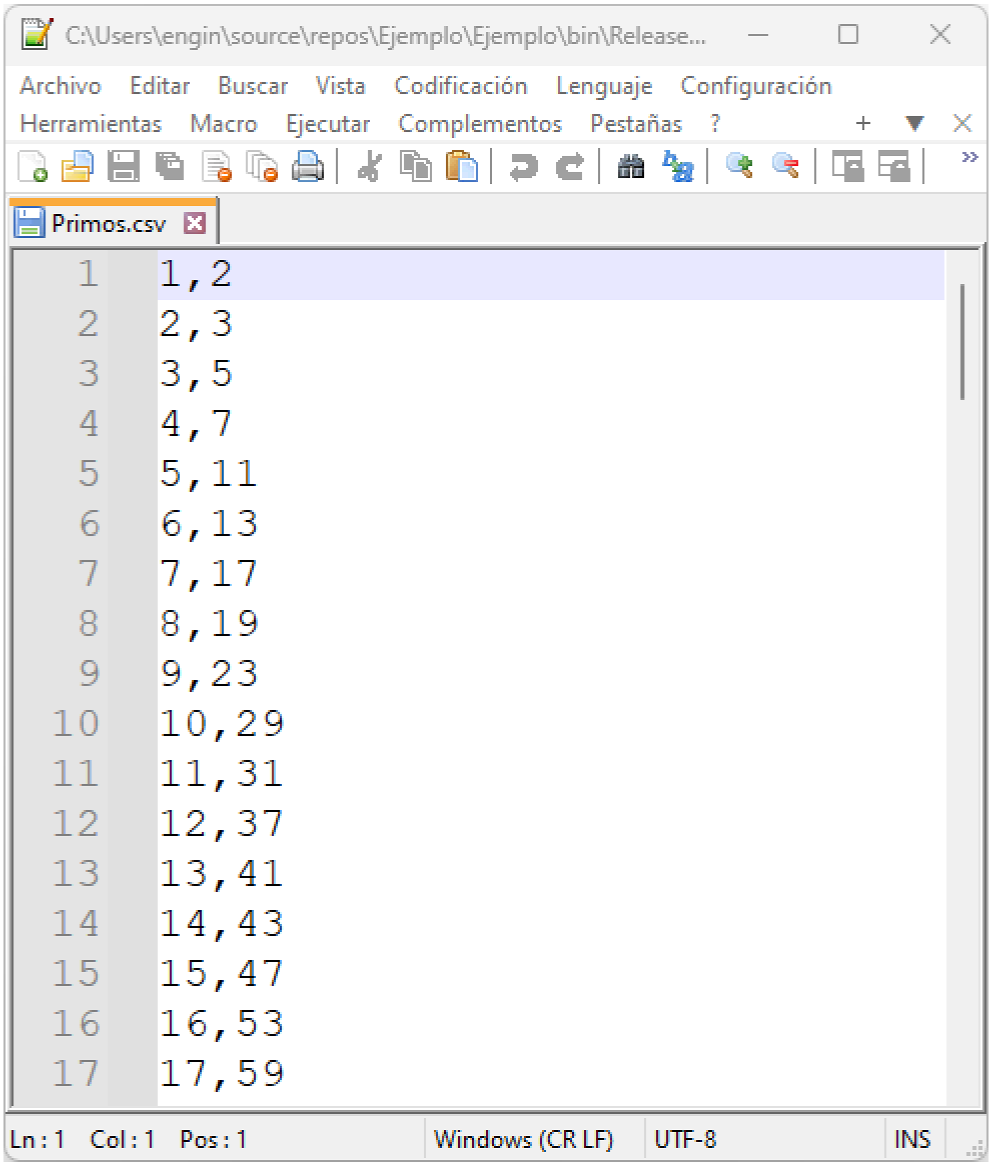


Ilustración 58: Serie de primos

```
Algoritmo Evolutivo vs Red Neuronal. Series de tiempo.

Tiempo en milisegundos para operar: 60000


Algoritmo Evolutivo
Tamaño de la población: 200


Red Neuronal: Perceptrón multicapa
Número de neuronas en Capa 1 oculta: 5
Número de neuronas en Capa 2 oculta: 5
```

Ilustración 59: Inicio de la comparativa

Este es el resultado dando un minuto a cada algoritmo:

```
92;479;480,52193157887115;477,0474499315329
93;487;485,9737207277953;483,286494274874
94;491;491,31078773977043;490,0021446135504
95;499;496,525105367195;497,23714448955127
96;503;501,6086887716638;504,9244387534105
97;509;506,5536262378394 7;512,8219528205645
98;521;511,35210975847696;520,4788888415304
99;523;515,996465314769;527,2976033805028
100;541;520,4791826778848;532,7276413462417

Ajuste Evolutivo: 232,3382169815784
Ajuste Neuronal: 200,30440829288548

FINAL
```

Ilustración 60: Resultado al comparar

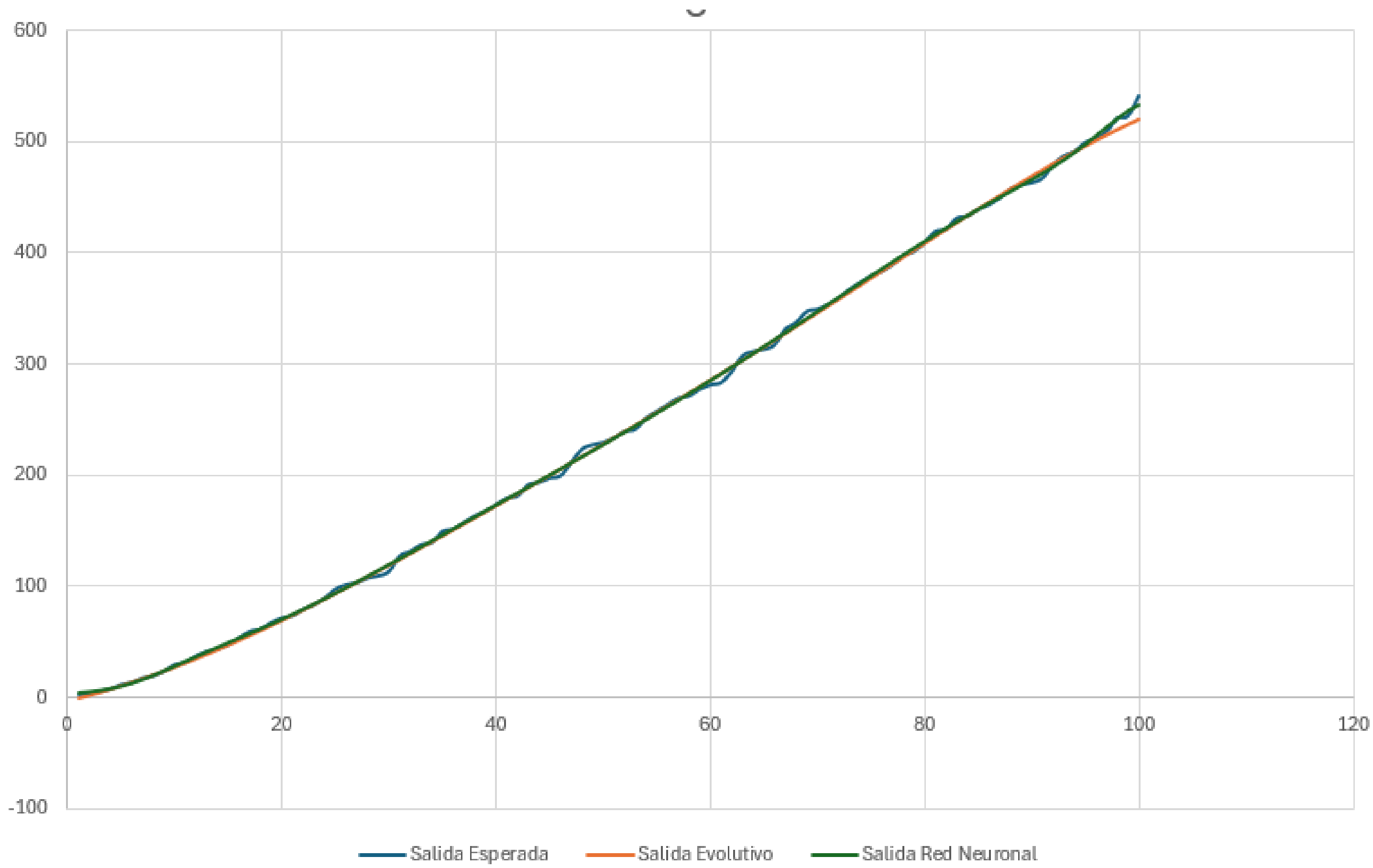


Ilustración 61: Gráfico lineal comparativo

La red neuronal tuvo el mejor ajuste, el algoritmo evolutivo quedó muy cerca.

Prueba #2: Mareas

En https://www.datosabiertos.gob.ec/dataset/altura-horaria-predicha-de-mareas/resource/788b326c-657c-4fc0-975d-0b9cd8690e30 se puede descargar el archivo CSV con el histórico de mareas.

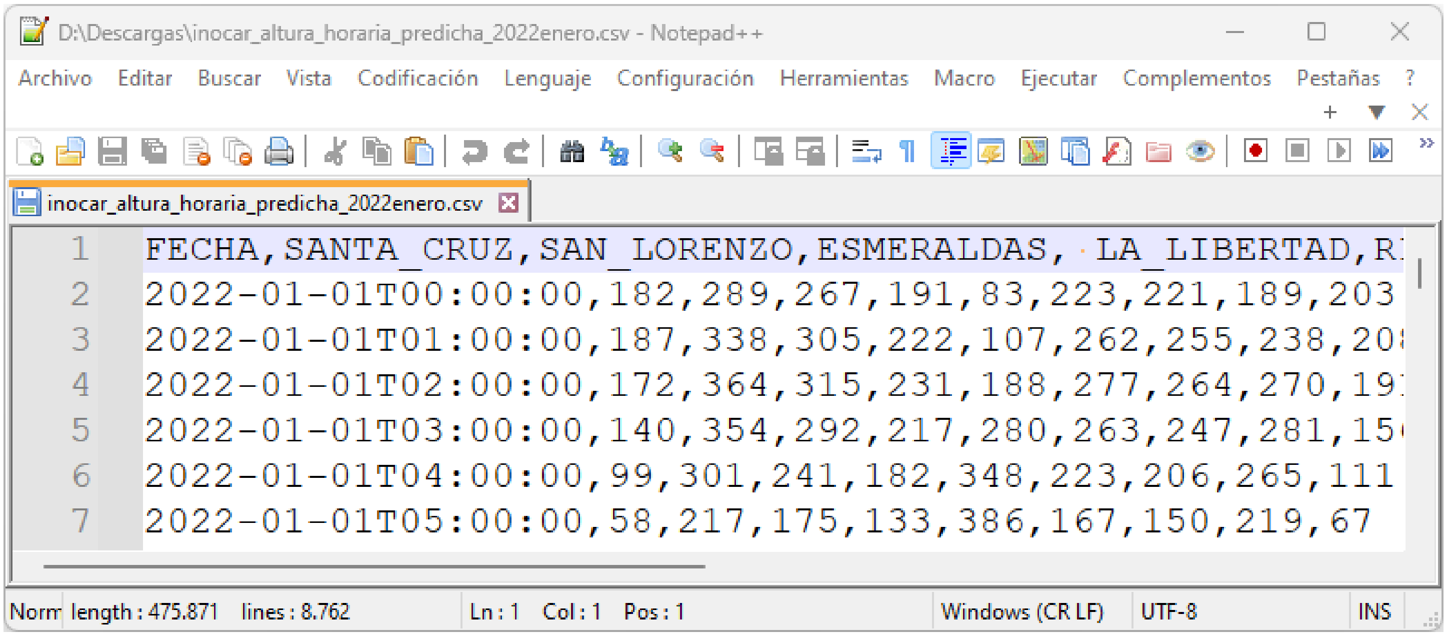


Ilustración 62: Archivo original CSV

La columna fecha se puede simplificar a valores enteros consecutivos y la segunda columna, una localidad, quedando así:

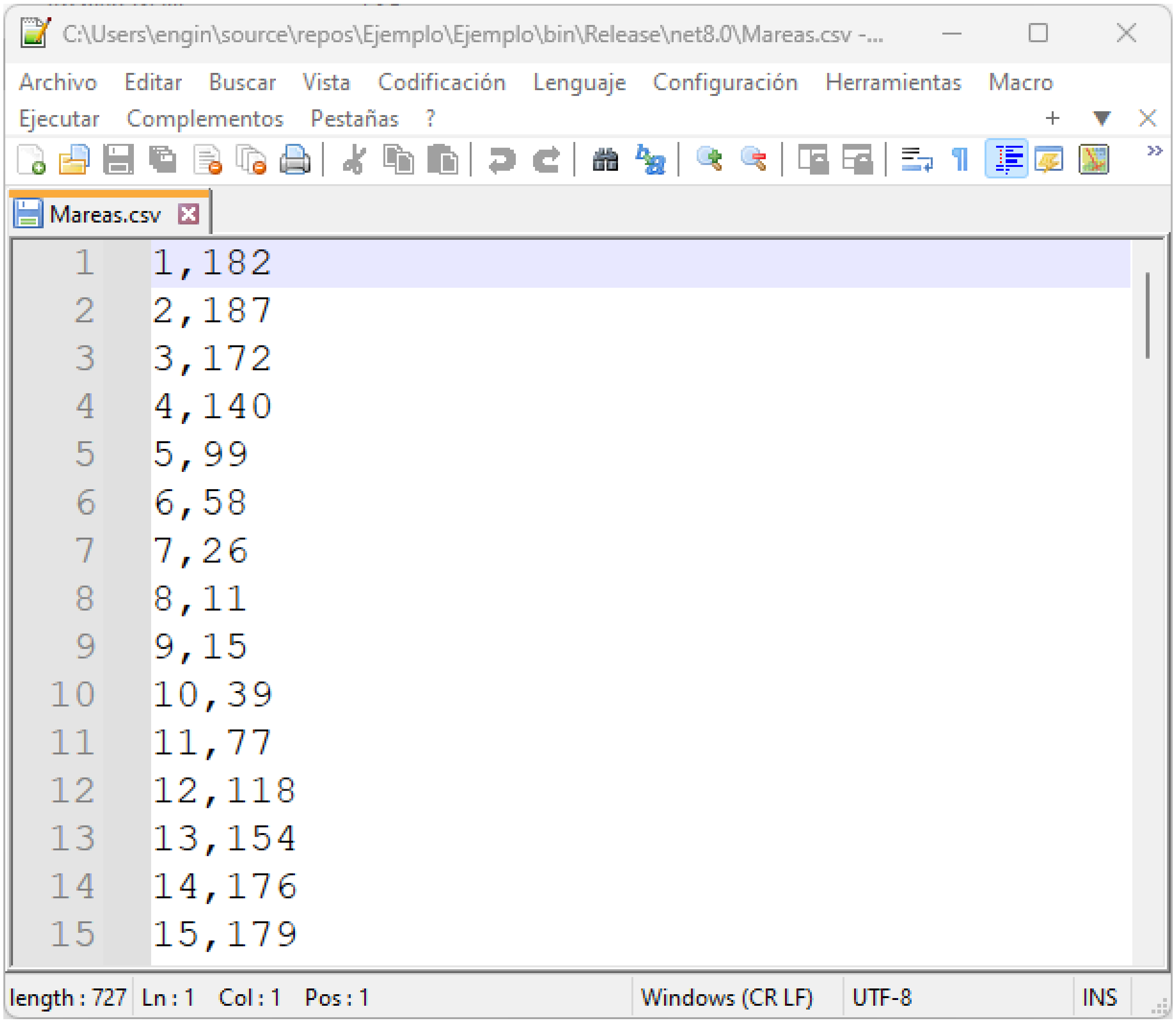


Ilustración 63: Conversión de datos

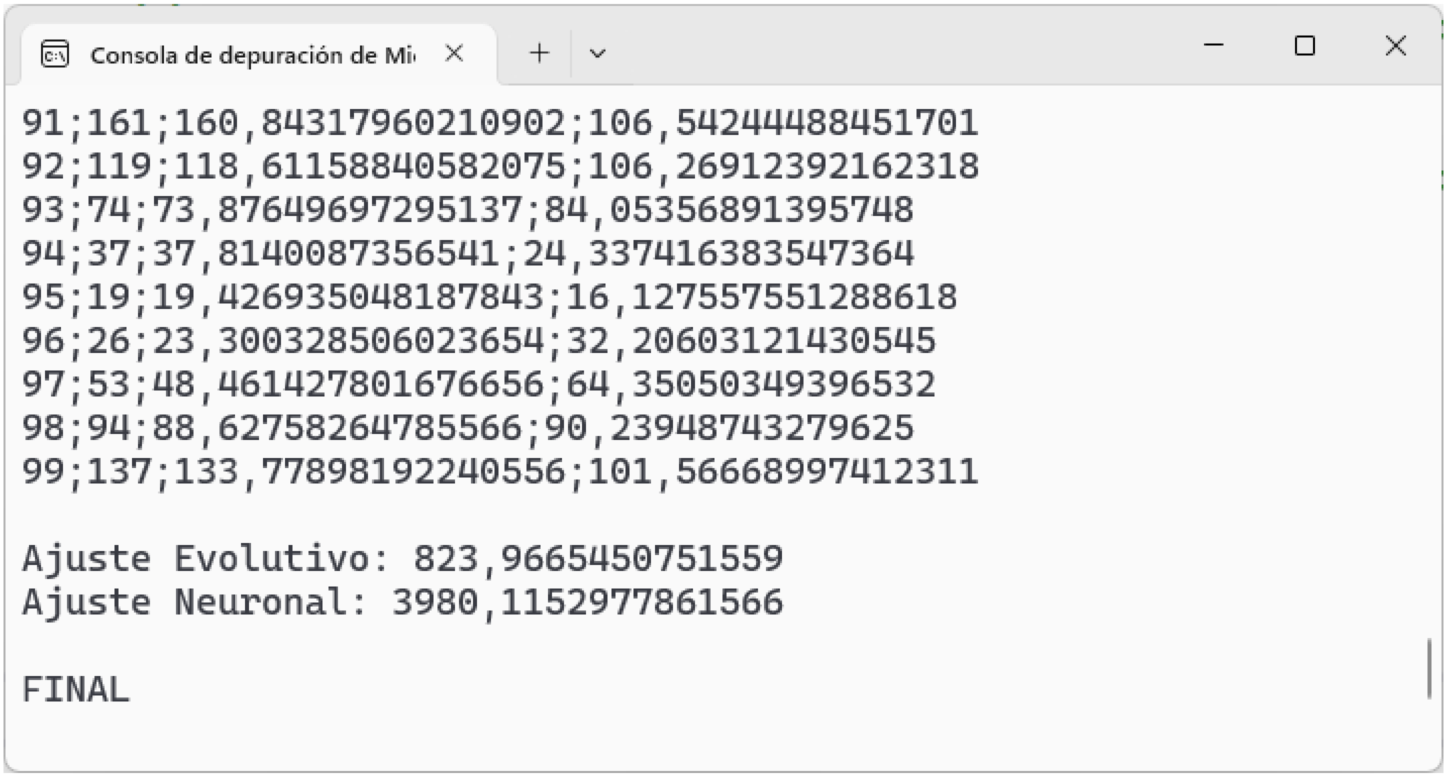


Ilustración 64: Ejecución del programa

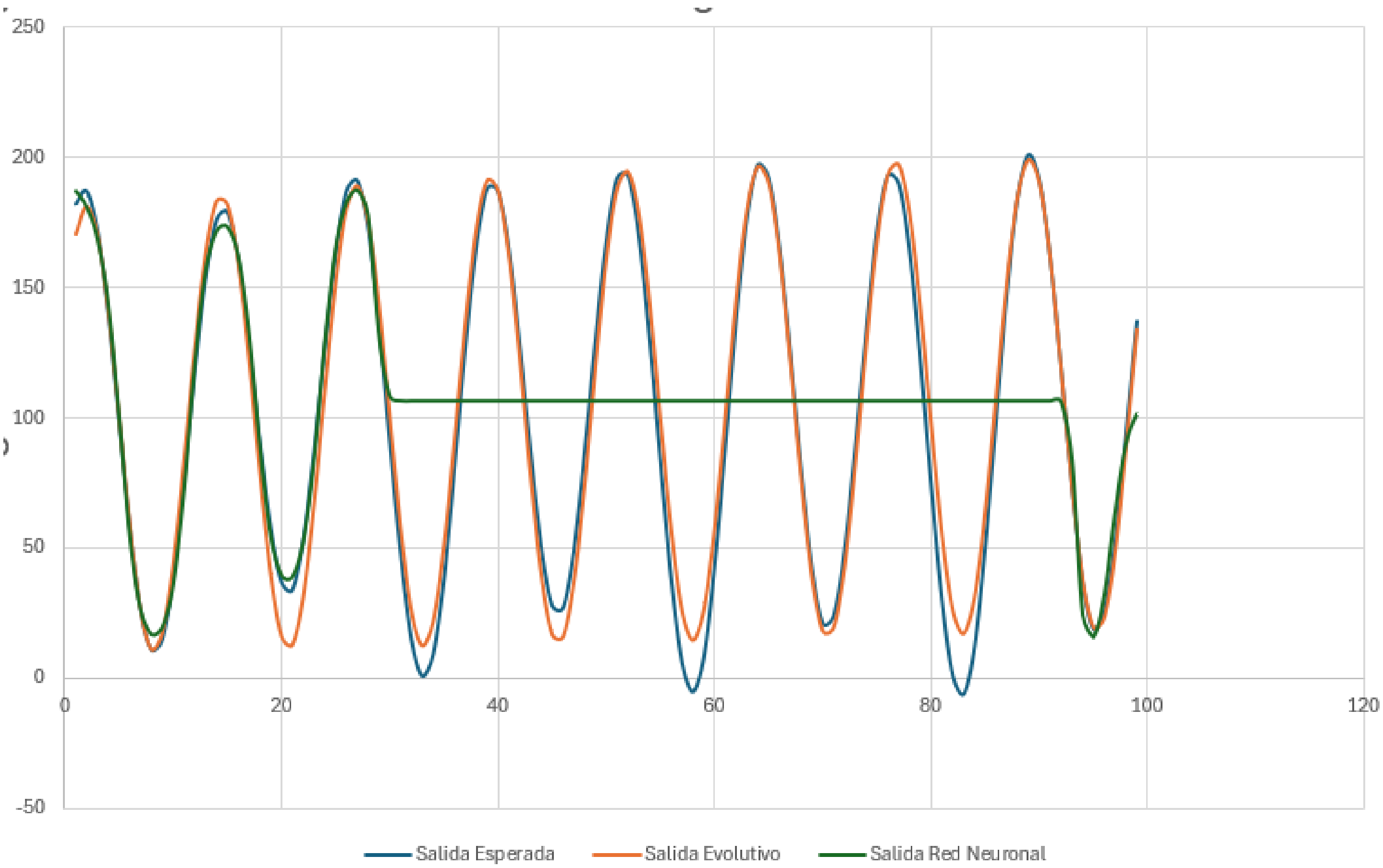
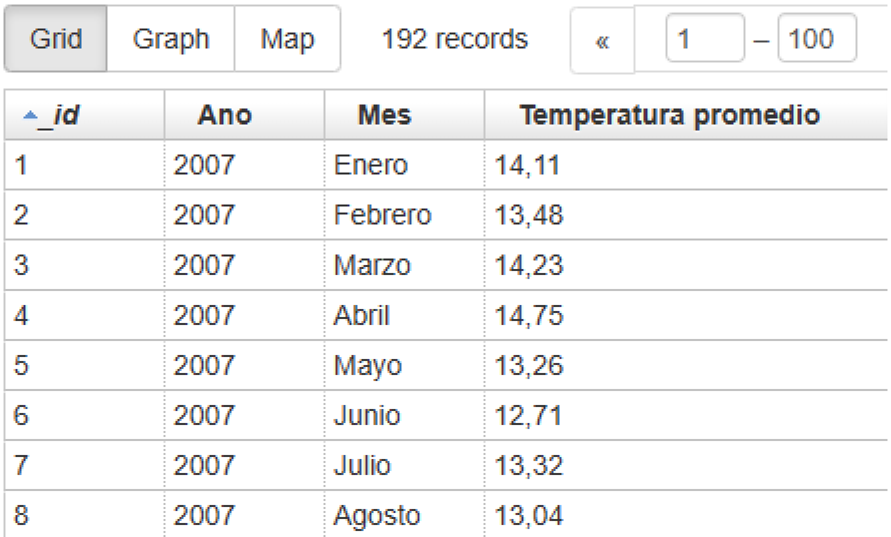


Ilustración 65: Comparativa en gráfico

El algoritmo evolutivo fue notablemente más preciso que la red neuronal. Esto puede indicar que, ante eventos con comportamientos cíclicos o sinusoidales, el algoritmo evolutivo lo va a hacer mucho mejor.

Prueba #3: Temperaturas

En https://datosabiertos.bogota.gov.co/dataset/temperaturas-en-bogota-d-c/resource/c3867459-1aa7-4e16-8ae8-ca41693f7261 se descargan los datos de temperatura en Bogotá.



_id	Ano	Mes	Temperatura promedio
1	2007	Enero	14,11
2	2007	Febrero	13,48
3	2007	Marzo	14,23
4	2007	Abril	14,75
5	2007	Mayo	13,26
6	2007	Junio	12,71
7	2007	Julio	13,32
8	2007	Agosto	13,04

Ilustración 66: Archivo original CSV

Sólo quedan las columnas _id y temperatura promedio

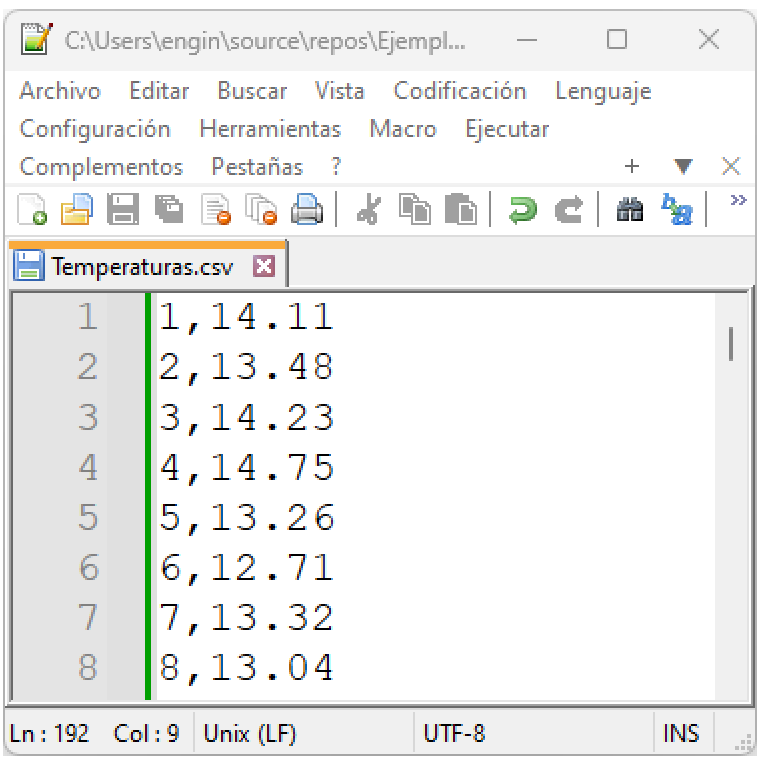


Ilustración 67: Archivo de datos a examinar

```
Consola de depuración de Mi

176;14,52;14,550233593409475;14,776423679628515
177;14,47;14,614665826945512;14,757951262189238
178;14,9;14,685294564796616;14,739582432268975
179;14,82;14,754648023003757;14,721379058233627
180;15,49;14,81540769673251;14,703400187481162
181;14,94;14,86121104522314;14,685701620201451
182;14,77;14,887346214964108;14,668335514116276
183;15,06;14,891258494405356;14,65135003098088
184;14,92;14,872810261494429;14,634789033762711
185;15,06;14,834265104737689;14,61869184146247
186;13,95;14,779999149238613;14,603093046538781
187;14,7;14,715974575522557;14,58802239788738 6
188;14,39;14,649038102974883;14,573504750362385
189;14,21;14,586127538907078;14,559560079965257
190;14,61;14,533479970501762;14,546203562119125
191;14,56;14,49593457044189;14,53344570893676
192;14,3;14,476411382035312;14,521292560114988

Ajuste Evolutivo: 60,879630807314044
Ajuste Neuronal: 54,305106030667105

FINAL
```

Ilustración 68: Ejecución del programa

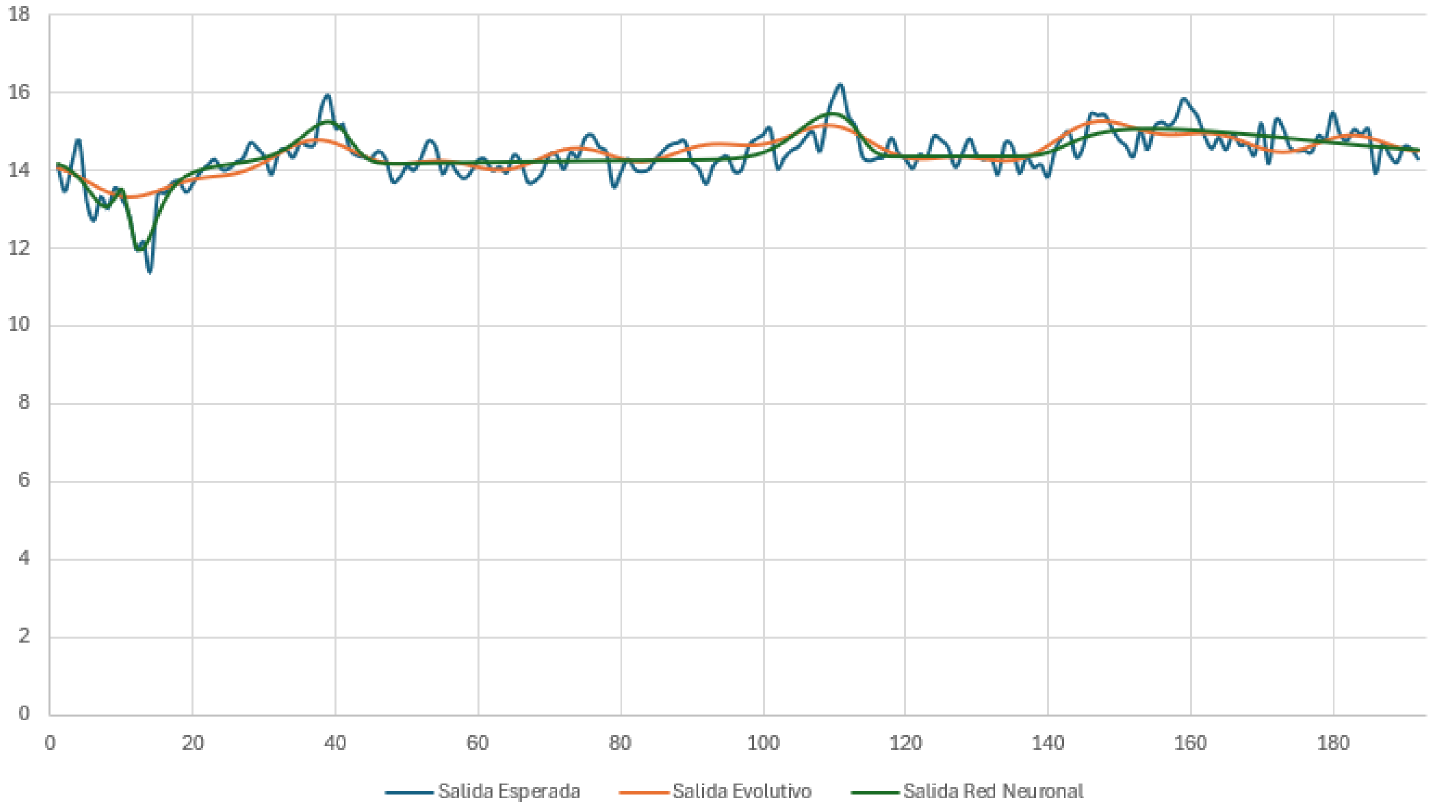
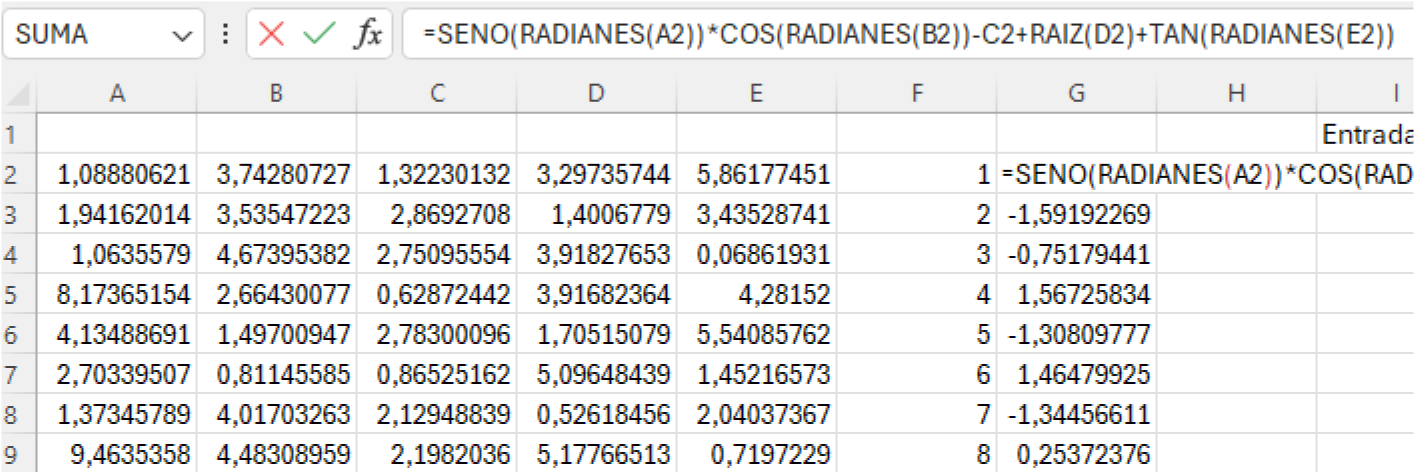


Ilustración 69: Gráfico comparativo

La red neuronal tuvo un mejor ajuste por poco.

El problema de la causalidad y las series de tiempo

En la siguiente hoja en Excel se observa que el valor de la columna G es el resultado de una ecuación donde participan las columnas A, B, C, D y E, pero no participa la columna F. Sin embargo, en el archivo CSV generado, se pone la columna F y la G para buscar un patrón, algo que no tiene sentido. Luego si se pone a competir el algoritmo evolutivo y la red neuronal, estos dos algoritmos intentan encontrar un patrón donde simplemente no lo hay.


SUMA
=SENO(RADIANES(A2))*COS(RADIANES(B2))-C2+RAIZ(D2)+TAN(RADIANES(E2))

	A	B	C	D	E	F	G	H	I
1									Entrada
2	1,08880621	3,74280727	1,32230132	3,29735744	5,86177451	1	=SENO(RADIANES(A2))*COS(RAD		
3	1,94162014	3,53547223	2,8692708	1,4006779	3,43528741	2	-1,59192269		
4	1,0635579	4,67395382	2,75095554	3,91827653	0,06861931	3	-0,75179441		
5	8,17365154	2,66430077	0,62872442	3,91682364	4,28152	4	1,56725834		
6	4,13488691	1,49700947	2,78300096	1,70515079	5,54085762	5	-1,30809777		
7	2,70339507	0,81145585	0,86525162	5,09648439	1,45216573	6	1,46479925		
8	1,37345789	4,01703263	2,12948839	0,52618456	2,04037367	7	-1,34456611		
9	9,4635358	4,48308959	2,1982036	5,17766513	0,7197229	8	0,25372376		

Ilustración 70: La columna G producto de una ecuación de varias variables

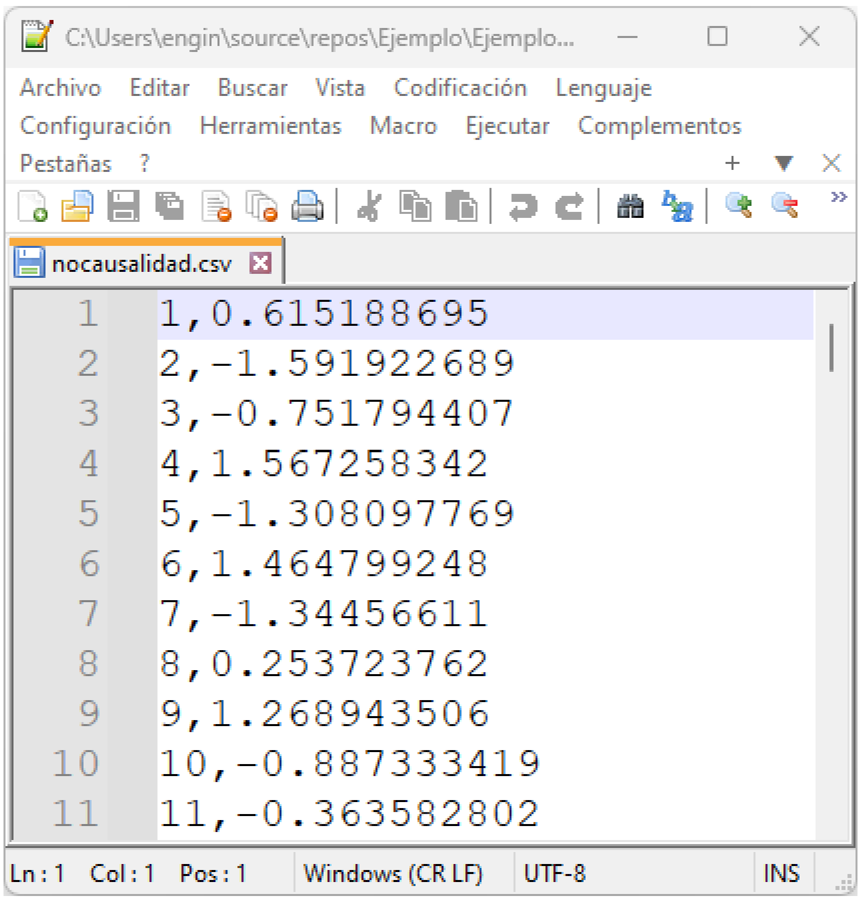


Ilustración 71: Archivo CSV con dos columnas sin relación alguna

I	J	K	L
Entrada	Salida Esper	Salida Evolutivo	Salida Neuronal
1	0,6151887	-0,039241004	-0,268727432
2	-1,59192269	-0,145178138	-0,238309977
3	-0,75179441	-0,288506854	-0,200534233
4	1,56725834	-0,445893683	-0,155886878
5	-1,30809777	-0,592088377	-0,104347379
6	1,46479925	-0,703422263	-0,042748481
7	-1,34456611	-0,761087507	0,035619856
8	0,25372376	-0,753743014	0,13371855
9	1,26894351	-0,679085998	0,241251188
10	-0,88733342	-0,5441715	0,337308662
11	-0,3635828	-0,364435601	0,403559913
12	2,17429576	-0,161557643	0,432613206
13	1,44296053	0,039542408	0,426796288
14	-1,06090569	0,214155522	0,393339336
15	1,82593704	0,341205937	0,340631723
16	-2,38273002	0,40617021	0,276326367
17	0,23699854	0,403088598	0,206624141
18	0,90296564	0,335390158	0,136106499
19	1,17208317	0,21541726	0,067773398
20	-0,01517071	0,06271538	0,003130827
21	-0,23912519	-0,098674909	-0,057751831
22	-1,19097608	-0,243548637	-0,116294947

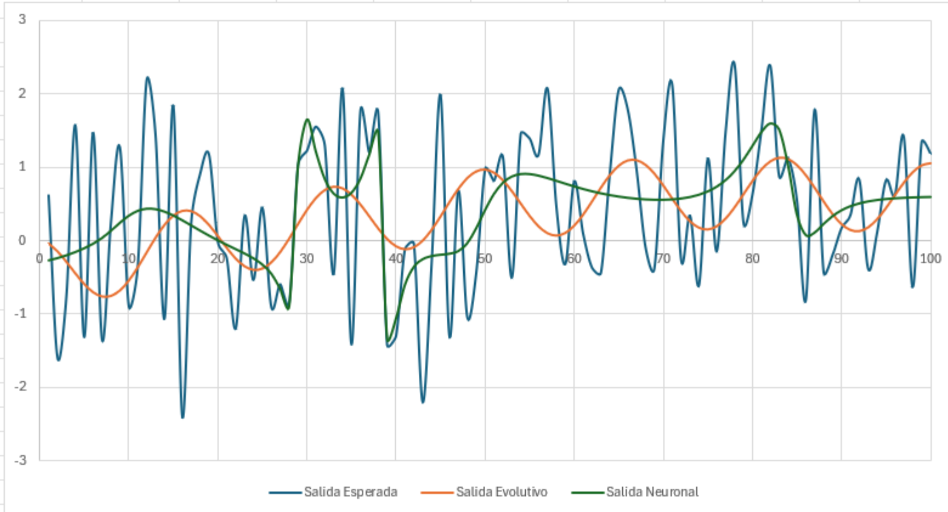


Ilustración 72: Curvas para buscar un "patrón" en los datos

Luego la comparativa entre algoritmo evolutivo y red neuronal midiendo series de tiempo no es válida si la serie de tiempo tiene varias variables de entrada, como sucede con el ejemplo de las Temperaturas, que puede ser el resultado de muchísimas variables, no de una sola (un tiempo secuencial).