Cómo entrenar a tu perceptrón

Andrés Gil Vicente
202304626@alu.icai.comillas.edu

Jorge Carnicero Príncipe
202311417@alu.comillas.edu

Jorge González Pérez
202304501@alu.comillas.edu

Liam Esgueva González
202301264@alu.comillas.edu

Sergio Fernández Cordero
202305407@alu.comillas.edu

Alejandro Alcázar Mendoza
202302063.alu.comillas.edu

Abstract

En este proyecto se estudia el funcionamiento y entrenamiento de un perceptrón, un modelo básico dentro del aprendizaje automático supervisado. Se parte del problema de clasificación binaria y se propone una aproximación continua mediante funciones de activación diferenciables, como la sigmoide. A través de una serie de ejercicios, se exploran aspectos teóricos y prácticos del modelo, incluyendo el cálculo de derivadas, la definición de funciones de pérdida y la implementación del algoritmo de descenso por gradiente. Se entrena el perceptrón para imitar el comportamiento de puertas lógicas simples (AND, OR) y se analiza su limitación frente al caso XOR. Finalmente, se introduce el perceptrón multicapa como solución a estas limitaciones, extendiendo la capacidad del modelo para representar funciones más complejas.

1 Introducción

El proyecto comienza presentando el problema de clasificación binaria, donde el objetivo es asignar a un objeto una categoría (0 o 1) en función de sus características. Se señala que, aunque esta forma de clasificar es sencilla, no permite estudiar con claridad la influencia de cada característica sobre la decisión, ya que la función de salida es discreta. Para superar esta limitación, se propone reemplazar la función binaria por una función continua que permita aplicar herramientas del cálculo diferencial. De esta forma, es posible analizar cómo variaciones en las características del objeto afectan el resultado de la clasificación.

Descrito de manera más formal, el perceptrón es un modelo que busca aproximar una función de clasificación

$$C : \mathbb{R}^n \to \{0, 1\},$$

que asigna el valor 1 si el objeto posee las características adecuadas para pertenecer a una categoría de interés, y 0 en caso contrario.

Esta aproximación se realiza mediante la composición de dos funciones diferenciables:

- Una **función de nivel** $u : \mathbb{R}^n \to \mathbb{R}$, que representa una combinación lineal de las características del objeto.
- Una **función de activación** $\phi : \mathbb{R} \to \mathbb{R}$, que transforma la salida de u y valora su relevancia. Su resultado es el que será luego comparado contra el umbral de decisión.

Así, el modelo resultante puede escribirse como:

$$C(x) \approx (\phi \circ u)(x),$$

y se decide que el objeto pertenece a la categoría de interés si el valor obtenido supera un umbral prefijado τ_0, es decir, si

$$(\phi \circ u)(x) > \tau_0.$$

2 El perceptrón

La función de nivel empleada en en el perceptrón es la función

$$u(x_1, \ldots, x_n) = b + w_1 x_1 + \ldots + w_n x_n \equiv b + w^t x \tag{1}$$

donde se ha considerado $w = (w_1, \ldots, w_n)$ y $x = (x_1, \ldots, x_n)$ como vectores columna, y se ha introducido un sesgo $b \in \mathbb{R}$.

La propuesta histórica más relevante para la función de activación es la sigmoide:

$$\phi(z) = \frac{1}{1 + e^{-z}} \tag{2}$$

que es diferenciable y sus valores cambian suavemente entre cero y uno, y en z = 0 está entre ambos valores

Se usará la función $F(x_1, \ldots, x_n) = (\phi \circ u)(x_1, \ldots, x_n)$ para aproximar el clasificador $C(x_1, \ldots, x_n)$.La función u (y por tanto F) depende también de los parámetros $\theta = (b, w_1, \ldots, w_n)$.

Estrictamente hablando, $u : \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$, $F : \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$ y evaluamos

$$u(x; \theta) \equiv u(x_1, \ldots, x_n; b, w_1, \ldots, w_n)$$
$$F(x; \theta) = \phi(u(x; \theta)) \equiv \phi(u(x_1, \ldots, x_n; b, w_1, \ldots, w_n)).$$

Finalmente, la función de pérdida a minimizar es

$$L(b, w_1, \ldots, w_n) = \sum_{k=1}^{N} \left(y_k - F(x^k)\right)^2 \tag{3}$$

2.1 Ejercicios propuestos

2.1.1 Escribe las derivadas parciales de la función u de la ecuación (1) respecto a los parámetros $b, w_1, \ldots, w_n$.

Recordemos que:

$$u(x_1, \ldots, x_n) = b + w_1 x_1 + \ldots + w_n x_n$$

Entonces, las derivadas parciales son:

$$\frac{\partial u}{\partial b} = 1, \quad \frac{\partial u}{\partial w_i} = x_i \quad \forall i = 1, \ldots, n.$$

2.1.2 Calcula las mismas derivadas parciales, pero ahora para la función $F = \phi \circ u$, donde ϕ es la función de activación definida en la ecuación (2). La derivada de ϕ puede dejarse indicada como ϕ'.

Aplicando la regla de la cadena:

$$F(x; \theta) = (\phi \circ u)(x; \theta) = \phi(u(x; \theta))$$

$$\frac{\partial F}{\partial b} = \frac{\partial F}{\partial u} \cdot \frac{\partial u}{\partial b} = \phi'(u(x; \theta)) \cdot 1 = \phi'(u(x; \theta))$$

$$\frac{\partial F}{\partial w_j} = \frac{\partial F}{\partial u} \cdot \frac{\partial u}{\partial w_j} = \phi'(u(x; \theta)) \cdot w_j$$

2.1.3 Calcula las derivadas parciales de la función de pérdida L, dada por la ecuación (3), respecto a los parámetros $b, w_1, \ldots, w_n$. Se puede dejar indicada la derivada de ϕ.

Sabiendo que:

$$L(b, \vec{w}) = \sum_{k=1}^{N} \left(y_k - F(x^{(k)}) \right)^2, \quad \text{donde } F(x^{(k)}) = \phi(u(x^{(k)}, b, w_1, \ldots, w_n)).$$

Aplicando la regla de la cadena, obtenemos:

$$\frac{\partial L}{\partial b} = \sum_{k=1}^{N} 2 \left(y_k - \phi(u(x^{(k)})) \right) \cdot \left(-\phi'(u(x^{(k)})) \right) = -2 \sum_{k=1}^{N} \left(y_k - \phi(u(x^{(k)})) \right) \cdot \phi'(u(x^{(k)}))$$

$$\frac{\partial L}{\partial w_i} = \sum_{k=1}^{N} 2 \left(y_k - \phi(u(x^{(k)})) \right) \cdot \left(-\phi'(u(x^{(k)})) \cdot x_i^{(k)} \right) = -2 \sum_{k=1}^{N} x_i^{(k)} \left(y_k - \phi(u(x^{(k)})) \right) \cdot \phi'(u(x^{(k)})),$$
$$\forall i = 1, \ldots, n.$$

Un caso sencillo de clasificación binaria viene dado por las puertas lógicas AND y OR. Estas se pueden considerar como funciones $\mathbb{R}^2 \to \mathbb{R}$, y para cada una de ellas disponemos únicamente de cuatro datos, que se recogen en las siguientes tablas:

Table 1: Valores de la puerta lógica AND

(x_1, x_2)	y
(0, 0)	0
(0, 1)	0
(1, 0)	0
(1, 1)	1

Table 2: Valores de la puerta lógica OR

(x_1, x_2)	y
(0, 0)	0
(0, 1)	1
(1, 0)	1
(1, 1)	1

El objetivo de los siguientes ejercicios es obtener funciones $F = \phi \circ u$ que aproximen el comportamiento de estas puertas lógicas, utilizando para ello el método del descenso por gradiente aplicado a la función de pérdida (3).

2.1.4 Calcula formalmente $\phi'(z)$, y escribe funciones de Matlab para evaluar $\phi(z)$ y $\phi'(z)$.

Calculamos

$$\phi'(z) = (-1) \cdot (1 + e^{-z})^{-2} \cdot (-1) \cdot e^{-z} = \frac{e^{-z}}{(1 + e^{-z})^2} = \frac{1}{1 + e^{-z}} \cdot \frac{e^{-z}}{1 + e^{-z}} =$$
$$= \phi(z) \cdot \frac{1 + e^{-z} - 1}{1 + e^{-z}} = \phi(z) \cdot \left(1 - \frac{1}{1 + e^{-z}} \right) = \phi(z) \cdot (1 - \phi(z))$$

Es decir,

$$\phi'(z) = \phi(z) \cdot (1 - \phi(z)) \tag{4}$$

Las funciones implementadas que se utilizan son *sigmoid.m* y *sigmoid_diff.m*

```
function sigmoid_value = sigmoid(z)
    % Función sigmoide, calcula la sigmoide de un punto o un vector
        punto a punto
    sigmoid_value = 1 ./ (1 + exp(-z));
end
```

Listing 1: Función sigmoid

```
function sigmoid_derivative = sigmoid_diff(z)
    % Derivada de la función sigmoide.
    % Calcula la derivada de la sigmoide elemento a elemento

    s = sigmoid(z);                    % Calculamos la sigmoide una sola
        vez
    sigmoid_derivative = s .* (1 - s);  % Aplicamos la fórmula de la
        derivada
end
```

Listing 2: Función sigmoid_diff

Se incluye también las representaciones de las funciones implementadas:

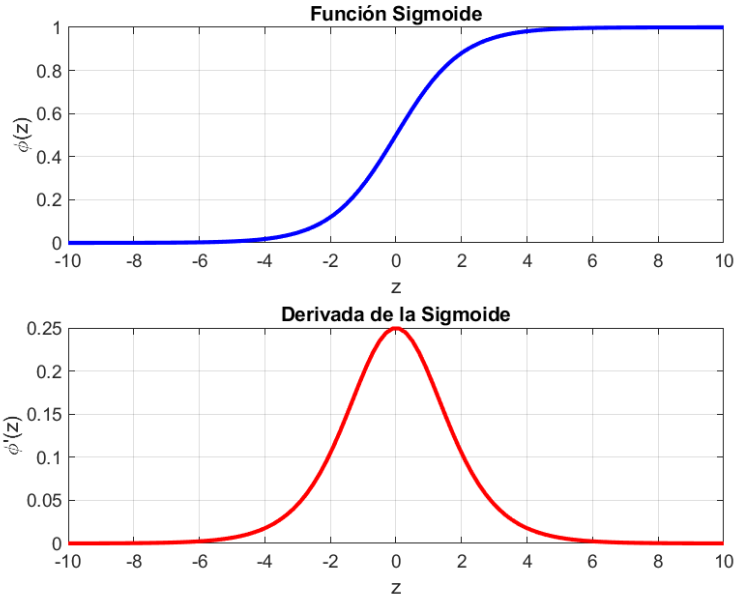


Figure 1: Funciones Sigmoide y Derivada de la Sigmoide

2.1.5 Considerando que $u : \mathbb{R}^2 \to \mathbb{R}$, escribe una función de Matlab que, dados b, $w = (w_1, w_2)$ y $x = (x_1, x_2)$ evalúe $u(x_1, x_2)$.

La función implementada es *funcion_nivel.m*

```
% Calcula la función afín u(x, b, w1, w2).
%   u(x) = b + w1*x1 + w2*x2
%   Si x contiene múltiples puntos, se calculará u(x) para cada punto.
u_value = b + w1 .* x(1,:) + w2 .* x(2,:);
end
```

Listing 3: Función función_nivel

2.1.6 Dado que $F = \phi \circ u$, escribe escribe funciones de Matlab que, b, $w = (w_1, w_2)$ y $x = (x_1, x_2)$ evalúe $F(x_1, x_2)$.

La función implementada es *funcion_perceptron.m*

```
function F_value = funcion_perceptron(b, w1, w2, x)
% Calcula la función F(x, b, w1, w2).
%
%   F(x) = sigma(b + w1*x1 + w2*x2)
%   Si x contiene múltiples puntos, se calculará u(x) para cada punto.
    u_val = funcion_nivel(b, w1, w2, x);
    F_value = sigmoid(u_val);
end
```

Listing 4: Función funcion_perceptron

2.1.7 Aprovechando las funciones escritas previamente, escribe una función que evalúe el gradiente de F respecto a los parámetros. Es decir, (F_b, F_{w_1}, F_{w_2})

Con la derivación de F en función de los parámetros y la ecuación (4), se implementa la función *gradiente_2.m*

```
function [dif_b, dif_w1, dif_w2] = gradiente_2(b, w1, w2, x)
    % Calculamos el nivel de activación del modelo
    nivel = funcion_nivel(b, w1, w2, x);

    % Calculamos la derivada de la función sigmoide evaluada en el
        nivel de activación
    diff_sigmoide_nivel = sigmoid_diff(nivel);

    % Calculamos los gradientes de la función de activación respecto a
        cada parámetro:
    % - La derivada respecto a b es simplemente la derivada de la
        sigmoide
    dif_b = diff_sigmoide_nivel;

    % - La derivada respecto a w1 se obtiene multiplicando la derivada
        de la sigmoide por x1
    dif_w1 = diff_sigmoide_nivel .* x(1,:);

    % - La derivada respecto a w2 se obtiene multiplicando la derivada
        de la sigmoide por x2
    dif_w2 = diff_sigmoide_nivel .* x(2,:);
end
```

Listing 5: Función gradiente_2

2.1.8 Puesto que $L : \mathbb{R}^3 \to \mathbb{R}$, escribe a mano su expresión para los dos casos de puerta lógica, teniendo en cuenta los datos de las tablas

Tabla AND: Precalculamos $F(x^k)$ para $k = \{1, 2, 3, 4\}$ sobre los datos de la tabla 1:

$$F(x^k) = \phi(u(x^k; \theta)) = \phi(w_1 x_1 + w_2 x_2 + b)$$

$$\begin{aligned} F(x^1) &= \phi(b) \\ F(x^2) &= \phi(w_2 + b) \\ F(x^3) &= \phi(w_1 + b) \\ F(x^4) &= \phi(w_1 + w_2 + b) \end{aligned}$$

Entonces, de la ecuación (3),

$$L(x_{AND}) = (y_1 - F(x^1))^2 + (y_2 - F(x^2))^2 + (y_3 - F(x^3))^2 + (y_4 - F(x^4))^2$$

Sustituyendo $y_1 = 0, y_2 = 0, y_3 = 0, y_4 = 1$ y los valores de $F(x^k)$ anteriores:

$$L(x_{AND}) = (\phi(b))^2 + (\phi(w_2 + b))^2 + (\phi(w_1 + b))^2 + (\phi(w_1 + w_2 + b))^2$$

Tabla OR: Los datos x^k son los mismos que en la tabla AND, por lo que solo cambian los y^k

$$L(x_{OR}) = (y_1 - F(x^1))^2 + (y_2 - F(x^2))^2 + (y_3 - F(x^3))^2 + (y_4 - F(x^4))^2$$

Sustituyendo $y_1 = 0, y_2 = 1, y_3 = 1, y_4 = 1$ de la tabla 2, y los valores de $F(x^k)$ anteriores:

$$L(x_{OR}) = (\phi(b))^2 + (1 - \phi(w_2 + b))^2 + (1 - \phi(w_1 + b))^2 + (\phi(w_1 + w_2 + b))^2$$

2.1.9 Calcula formalmente $\nabla L = (L_b, L_{w_1}, L_{w_2})$ para los dos casos anteriores. Se pueden dejar indicadaslas derivadas de ϕ

Aprovechando los cálculos del ejercicio anterior y las derivadas de L en función de los parámetros,

Tabla AND:

$$L_b = -2\sum_{k=1}^{4} \big(y_k - \phi(u(x^k))\big) \cdot \phi'(u(x^k)) =$$
$$= -2(-\phi(b) \cdot \phi'(b) - \phi(w_2 + b) \cdot \phi'(w_2 + b) - \phi(w_1 + b) \cdot \phi'(w_1 + b) +$$
$$+ (1 - \phi(w_1 + w_2 + b)) \cdot \phi'(w_1 + w_2 + b))$$

En la derivada respecto a w_1, se multiplica cada sumando por x_1^k, que es 0 para $k = 1, k = 2$:

$$L_{w_1} = -2\sum_{k=1}^{4} x_1^k \big(y_k - \phi(u(x^k))\big) \cdot \phi'(u(x^k)) =$$
$$= -2\left(\big(y_3 - \phi(u(x^3))\big) \cdot \phi'(u(x^3)) + \big(y_4 - \phi(u(x^4))\big) \cdot \phi'(u(x^4))\right)$$
$$= -2\left(-\phi(w_1 + b) \cdot \phi'(w_1 + b) + (1 - \phi(w_1 + w_2 + b)) \cdot \phi'(w_1 + w_2 + b)\right)$$

En la derivada respecto a w_2, se multiplica cada sumando por x_2^k, que es 0 para $k = 1, k = 3$:

$$L_{w_2} = -2\sum_{k=1}^{4} x_2^k \big(y_k - \phi(u(x^k))\big) \cdot \phi'(u(x^k)) =$$
$$= -2\left(\big(y_2 - \phi(u(x^2))\big) \cdot \phi'(u(x^3)) + \big(y_4 - \phi(u(x^4))\big) \cdot \phi'(u(x^4))\right)$$
$$= -2\left(-\phi(w_2 + b) \cdot \phi'(w_2 + b) + (1 - \phi(w_1 + w_2 + b)) \cdot \phi'(w_1 + w_2 + b)\right)$$

Tabla OR:

$$L_b = -2\sum_{k=1}^{4} \big(y_k - \phi(u(x^k))\big) \cdot \phi'(u(x^k)) =$$
$$= -2(-\phi(b) \cdot \phi'(b) + (1 - \phi(w_2 + b)) \cdot \phi'(w_2 + b)) + (1 - \phi(w_1 + b)) \cdot \phi'(w_1 + b))$$
$$+ (1 - \phi(w_1 + w_2 + b)) \cdot \phi'(w_1 + w_2 + b))$$

En la derivada respecto a w_1, se multiplica cada sumando por x_1^k, que es 0 para $k = 1, k = 2$:

$$L_{w_1} = -2\sum_{k=1}^{4} x_1^k \big(y_k - \phi(u(x^k))\big) \cdot \phi'(u(x^k)) =$$
$$= -2\left(\big(y_3 - \phi(u(x^3))\big) \cdot \phi'(u(x^3)) + \big(y_4 - \phi(u(x^4))\big) \cdot \phi'(u(x^4))\right)$$
$$= -2\left((1 - \phi(w_1 + b)) \cdot \phi'(w_1 + b) + (1 - \phi(w_1 + w_2 + b)) \cdot \phi'(w_1 + w_2 + b)\right)$$

En la derivada respecto a w_2, se multiplica cada sumando por x_2^k, que es 0 para $k = 1, k = 3$:

$$L_{w_2} = -2\sum_{k=1}^{4} x_2^k \big(y_k - \phi(u(x^k))\big) \cdot \phi'(u(x^k)) =$$
$$= -2\left(\big(y_2 - \phi(u(x^2))\big) \cdot \phi'(u(x^3)) + \big(y_4 - \phi(u(x^4))\big) \cdot \phi'(u(x^4))\right)$$
$$= -2\left((1 - \phi(w_2 + b)) \cdot \phi'(w_2 + b) + (1 - \phi(w_1 + w_2 + b)) \cdot \phi'(w_1 + w_2 + b)\right)$$ [1]

[1] Al desarrollar ϕ', las expresiones quedan en función de ϕ y $1 - \phi$.

2.1.10 Implementa en Matlab funciones que evalúen ∇L para cada una de las puertas lógicas

La función implementada es *gradiente_puertas_logicas*. La matriz de datos X se brinda como argumento de la función

```
function [grad_w0, grad_w1, grad_w2] = gradiente_puertas_logicas(X, w0, w1, w2)
    % Esta función calcula el gradiente de F respecto a los parámetros w0, w1 y w2.
    % Entradas:
    %   - X: Matriz 3xN donde la primera fila es x1, la segunda es x2, y la tercera es y.
    %   - w0, w1, w2: Parámetros actuales del modelo.
    %
    % Salidas:
    %   - grad_w0: Derivada de F respecto a w0.
    %   - grad_w1: Derivada de F respecto a w1.
    %   - grad_w2: Derivada de F respecto a w2.

    % Extraemos x1 y x2 como entradas y la última fila como salidas esperadas
    x = X(1:2, :);

    % Calculamos el gradiente de F respecto a (w0, w1, w2)
    [grad_w0, grad_w1, grad_w2] = gradiente_2(w0, w1, w2, x);
end
```

Listing 6: Función gradiente_puertas_logicas

2.1.11 Utilizando las funciones anteriores, implementa en Matlab el método del descenso por gradiente para optimizar los parámetros b, w_1, w_2 por medio de la minimización de L.

La función implementada es *descenso_gradiente*. Se necesitan tanto un learning rate $alpha$ como un umbral como condición de parada (aunque se puede reescribir el umbral en función de los datos).

```
function [grad_opt_w0, grad_opt_w1, grad_opt_w2] = descenso_gradiente(X, w0, w1, w2, umbral, alpha)
    % Extraemos las características de entrada (x1 y x2) y las etiquetas (y)
    x = X(1:2,:);
    y = X(3,:);

    % Inicializamos los valores de los pesos
    grads_t0 = [0, 0, 0];
    grads_t1 = [w0, w1, w2] + umbral + 1; % Garantizamos la entrada al bucle

    % Ejecutamos el bucle hasta que la diferencia entre iteraciones sea menor que el umbral
    while max(abs(grads_t1 - grads_t0)) > umbral
        % Guardamos los valores actuales antes de actualizarlos
        grads_t0 = grads_t1;

        % Calculamos la predicción del modelo
        nivel = funcion_perceptron(w0, w1, w2, x);

        % Calculamos el error
        error = y - nivel;  % Diferencia entre real y predicho

        % Calculamos los gradientes de la función de activación
        [grad_b, grad_w1, grad_w2] = gradiente_2(w0, w1, w2, x);

        % Aplicamos el error a los gradientes (derivada de la función de pérdida)
```

```
        grad_b =  (-2) * sum(error .* grad_b);
        grad_w1 = (-2) * sum(error .* grad_w1);
        grad_w2 = (-2) * sum(error .* grad_w2);

        % Actualizamos los parámetros usando descenso de gradiente
        w0 = w0 - alpha * grad_b;
        w1 = w1 - alpha * grad_w1;
        w2 = w2 - alpha * grad_w2;

        % Guardamos los nuevos valores de los pesos
        grads_t1 = [w0, w1, w2];
    end

    % Retornamos los valores optimizados de los parámetros
    grad_opt_w0 = w0;
    grad_opt_w1 = w1;
    grad_opt_w2 = w2;
end
```

Listing 7: Función descenso_gradiente

2.1.12 Obtén parámetros que ajusten adecuadamente el comportamiento de F para ambas puertas lógicas.

Para trabajar con el perceptrón cómodamente, se define una clase Perceptron_1 en MATLAB:

```
classdef Perceptron_1
    properties
        b   % Sesgo
        w1  % Peso para x1
        w2  % Peso para x2
    end

    methods
        % Constructor
        function obj = Perceptron_1()
            % Pesos a cero por defecto
            obj.b = 0;
            obj.w1 = 0;
            obj.w2 = 0;
        end

        % Método de FIT
        function obj = fit(obj, X, umbral, alpha)
            [b, w1, w2] = descenso_gradiente(X, obj.b, obj.w1, obj.w2,
                umbral, alpha);
            obj.b = b;
            obj.w1 = w1;
            obj.w2 = w2;
        end

        % Método de PREDICT
        function z = predict(obj, x)
            z = funcion_perceptron(obj.b, obj.w1, obj.w2, x);
        end

         % Método de PREDICTED_CLASS
        function class = predict_class(obj, x, tau)
            predicted = obj.predict(x);
            if predicted < tau
                class = 0;
            else
                class = 1;
            end
```

```
        end
    end
end
```

Listing 8: Clase Perceptron_1

El perceptrón toma como valores iniciales 0 en todos sus parámetros en el momento de crear el objeto.

Con el método *fit* se entrena dada una matriz de datos X de tamaño $3 \times N$ cuyas filas son los valores de x_1, x_2, y. Es decir, cada columna contiene los valores x_1, x_2, y de una entrada.

$$X = \begin{bmatrix} x_1^1 & x_1^2 & \dots & x_1^N \\ x_2^1 & x_2^2 & \dots & x_2^N \\ y^1 & y^2 & \dots & y^N \end{bmatrix}$$

El método *predict* obtiene un valor p comprendido entre 0 y 1. Finalmente, el método *predict_class* compara p con un umbral τ_0 y asigna a un dato nuevo $\tilde{x}$ su clase predecida $\hat{y}$:

$$\hat{y} = \begin{cases} 0 & \text{si } p < \tau_0 \\ 1 & \text{si } p \geq \tau_0 \end{cases}$$

Tabla AND: Para la puerta lógica AND, se define una matriz X_{AND} y se entrena al perceptrón

```
% Para la puerta lógica AND: creamos un objeto perceptrón
% y lo entrenamos
Perceptron_AND = Perceptron_1();
X_AND = [0,0,0;
         0,1,0;
         1,0,0;
         1,1,1].'
Perceptron_AND = Perceptron_AND.fit(X_AND, 10^(-7), 0.1)
```

Listing 9: Perceptrón entrenado para la puerta lógica AND

Obteniendo como resultado $b = -20.8241, w_1 = 13.8266, w_2 = 13.8266$:

```
Perceptron_AND =
  Perceptron_1 with properties:
     b: -20.8241
    w1: 13.8266
    w2: 13.8266
```

Se puede comprobar cómo predice resultados con algunos ejemplos

```
prediccion_1_0 = Perceptron_AND.predict([1;0])
prediccion_1_1 = Perceptron_AND.predict([1;1])
clase_1_0 = Perceptron_AND.predict_class([1;0], 0.5)
```

Listing 10: Predicciones del perceptrón AND

El resultado de estas 3 variables es el esperado:

```
prediccion_1_0 = 9.1330e-04
prediccion_1_1 = 0.9989
clase_1_0 = 0
```

Tabla OR: Para la puerta lógica OR, se define una matriz X_{OR} y se entrena al perceptrón

```
% Para la puerta lógica OR: creamos un objeto perceptrón
% y lo entrenamos
Perceptron_OR = Perceptron_1();
```

```
X_OR = [0,0,0;
        0,1,1;
        1,0,1;
        1,1,1].'
Perceptron_OR = Perceptron_OR.fit(X_OR, 10^(-7), 0.1)
```

Listing 11: Perceptrón entrenado para la puerta lógica OR

Obteniendo como parámetros $b = -6.7945, w_1 = 14.0477, w_2 = 14.0477$:

```
Perceptron_OR =
  Perceptron_1 with properties:
     b: -6.7945
    w1: 14.0477
    w2: 14.0477
```

De nuevo, se comprueban casos sencillos

```
prediccion_0_0 = Perceptron_OR.predict([0;0])
prediccion_0_1 = Perceptron_OR.predict([0;1])
clase_0_1 = Perceptron_OR.predict_class([0;1], 0.5)
```

Listing 12: Predicciones del perceptrón OR

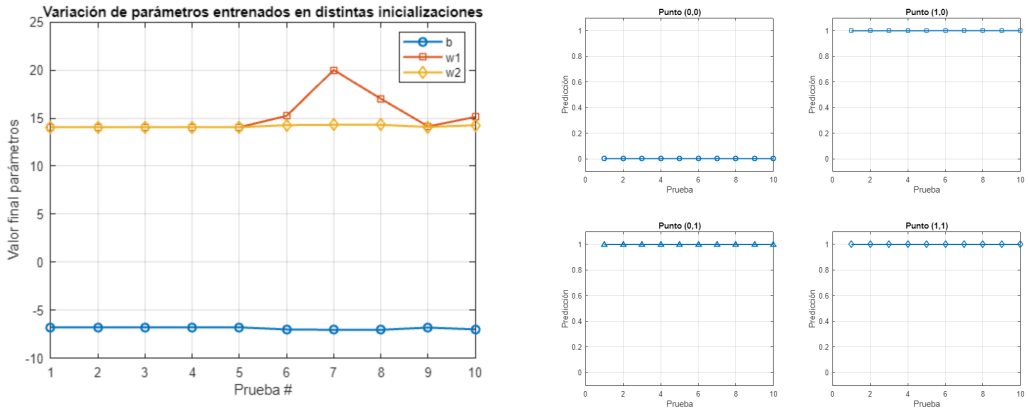
Los resultados son los esperados:

```
prediccion_0_0 = 0.0011
prediccion_0_1 = 0.9993
clase_0_1 = 1
```

2.1.13 ¿Hay mucha dependencia respecto a los candidatos iniciales para los parámetros?

Se inicializa el perceptrón con distintos parámetros inciales, analizando cúales son los valores obtenidos para dichos parámetros una vez se termina el descenso de gradiente.

Se inicializa el perceptrón con distintos parámetros iniciales, analizando cuáles son los valores obtenidos para dichos parámetros una vez se termina el descenso de gradiente.

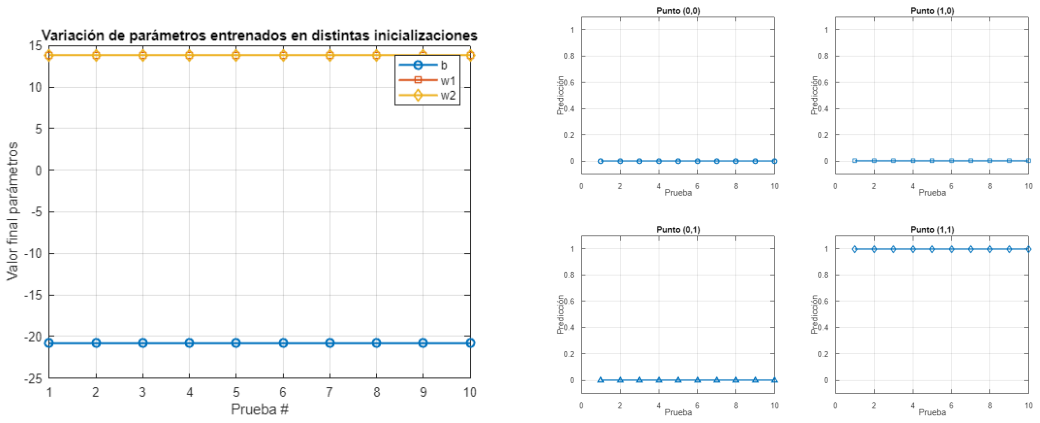


(a) Parámetros obtenidos

(b) Predicciones obtenidas

Figure 2: Resultados para la puerta lógica OR

Perceptrón OR (inicialización entre -5 y 5)



(a) Parámetros obtenidos (b) Predicciones obtenidas

Figure 3: Resultados para la puerta lógica AND

Perceptrón AND (inicialización entre -5 y 5)

Como se puede apreciar en las figuras 2 y 3, se observa que, aunque se inicialicen los parámetros b, w_1 y w_2 con valores distintos (aleatorios en cada una de las pruebas), tras entrenar el perceptrón usando su método `fit`, el descenso de gradiente proporciona siempre los mismos, o muy similares, valores para los parámetros óptimos, es decir, aquellos que minimizan la función de pérdida. Esto indica que los parámetros tienden a estabilizarse en esos valores independientemente del punto de partida.

2.1.14 En cada caso, representa la curva en $\mathbb{R}^2$ de los puntos (x_1, x_2) que cumplen la ecuación $w_1x_1 + w_2x_2 + b = \frac{1}{2}$.

Es decir, representa la curva de nivel $1/2$ de la función u obtenida, y comprueba que efectivamente separa los cuatro puntos de interés en el plano, igual que las puertas lógicas.

Se calcula primero la curva de nivel $L_{1/2} = \{(x_1, x_2) : w_1x_1 + w_2x_2 + b = \frac{1}{2}\}$ Despejando x_2 en función de x_1,

$$w_1x_1 + w_2x_2 + b = \frac{1}{2}$$

$$w_2x_2 = \frac{1}{2} - w_1x_1 - b$$

$$x_2 = \frac{1}{2w_2} - \frac{w_1}{w_2}x_1 - \frac{b}{w_2}$$

Tabla AND:

```
% Para la puerta AND
figure
hold on
grid on

w1 = Perceptron_AND.w1;
w2 = Perceptron_AND.w2;
b = Perceptron_AND.b;

% Dibujamos los puntos de entrenamiento
for i = 1:4
    x = [X_AND(1,i); X_AND(2,i)];
    predicted_class = Perceptron_AND.predict_class(x, 0.5)
    if predicted_class == 1
```

```
        plot(X_AND(1,i), X_AND(2,i), 'o', 'MarkerSize', 10, '
            MarkerFaceColor', 'g'); % Clase 1
    else
        plot(X_AND(1,i), X_AND(2,i), 'o', 'MarkerSize', 10, '
            MarkerFaceColor', 'r'); % Clase 0
    end
end

xx = linspace(-0.5, 1.5, 100);
yy = (1/2-b-w1*xx)/w2;  % curva de nivel 1/2
plot(xx, yy, 'b-', 'LineWidth', 2);

% Etiquetas
xlabel('x1');
ylabel('x2');
title('Puntos en la puerta AND');
axis([-0.5, 1.5, -0.5, 1.5]);
hold off;
```

Listing 13: Código para clasificar de puntos de la puerta AND

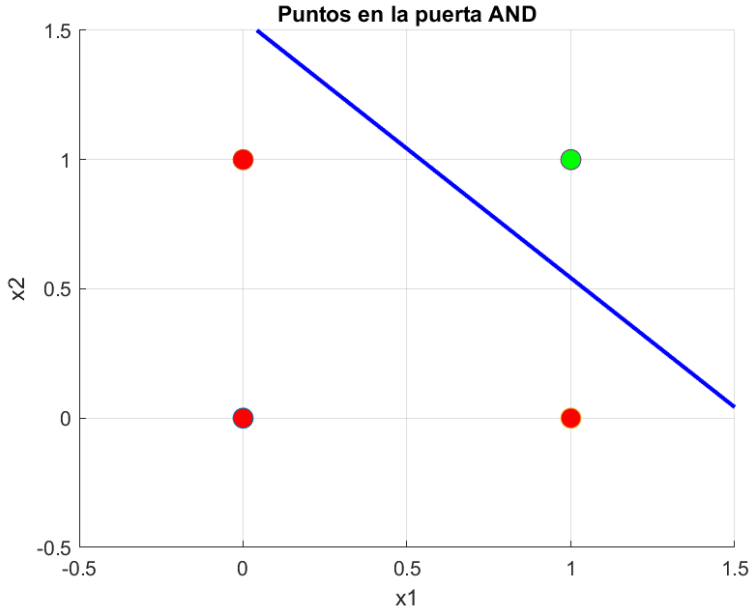


Figure 4: Clasificación de los puntos de la puerta lógica AND

La recta ajustada es $13.8266 \cdot x_1 + 13.8266 \cdot x_2 - 20.8241 = \frac{1}{2}$

Tabla OR:

```
% Para la puerta OR
figure
hold on
grid on

w1 = Perceptron_OR.w1;
w2 = Perceptron_OR.w2;
b = Perceptron_OR.b;

% Dibujamos los puntos de entrenamiento
for i = 1:4
    x = [X_OR(1,i); X_OR(2,i)];
    predicted_class = Perceptron_OR.predict_class(x, 0.5);
```

```
    if predicted_class == 1
        plot(X_OR(1,i), X_OR(2,i), 'o', 'MarkerSize', 10, '
            MarkerFaceColor', 'g'); % Clase 1
    else
        plot(X_OR(1,i), X_OR(2,i), 'o', 'MarkerSize', 10, '
            MarkerFaceColor', 'r'); % Clase 0
    end
end

xx = linspace(-0.5, 1.5, 100);
yy = (0.5-b-w1*xx)/w2;
plot(xx, yy, 'b-', 'LineWidth', 2);

% Etiquetas
xlabel('x1');
ylabel('x2');
title('Puntos en la puerta OR');
axis([-0.5, 1.5, -0.5, 1.5]);
hold off;
```

Listing 14: Código para clasificar de puntos de la puerta OR

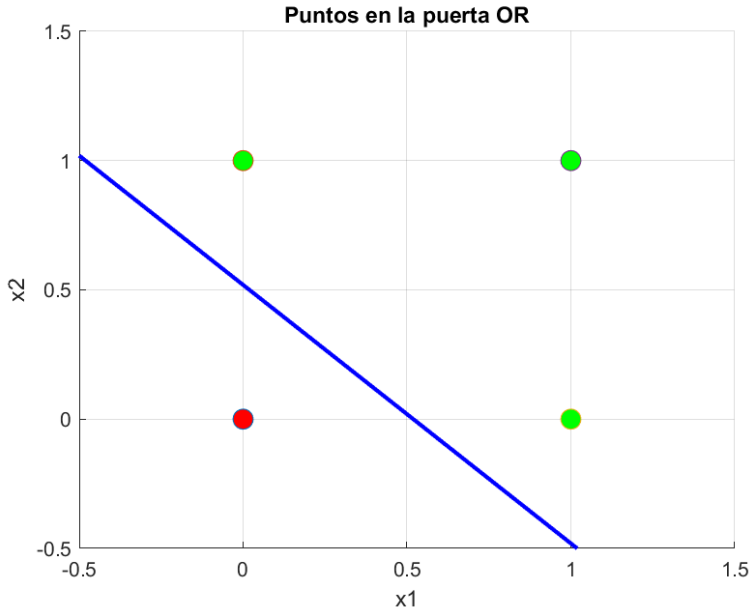


Figure 5: Clasificación de los puntos de la puerta lógica OR

La recta ajustada es $14.0477 \cdot x_1 + 14.0477 \cdot x_2 - 6.7945 = \frac{1}{2}$.

Ambas rectas ajustadas separan correctamente los puntos clasificados como clase 0 o 1

2.1.15 Intenta entrenar F para que clasifique los puntos como una puerta lógica XOR

Como era de esperar, el perceptrón no es capaz de clasificar correctamente los datos correspondientes a la puerta lógica XOR, ya que estos no son linealmente separables.

Dado que el modelo implementado corresponde a un perceptrón de una sola capa, resulta insuficiente para resolver este tipo de problemas. Aunque el entrenamiento se realice correctamente, la arquitectura por sí sola no tiene la capacidad de representar la complejidad del patrón XOR.

Esto se refleja claramente en la visualización de la función aprendida, donde se observa que la línea de nivel obtenida no separa adecuadamente los cuatro puntos del conjunto. En particular, los puntos

que pertenecen a la misma clase se encuentran en lados opuestos de la frontera de decisión. Esta observación confirma la necesidad de incorporar más capas (arquitectura multicapa) para poder abordar correctamente problemas no linealmente separables como este. Los datos no son linealmente separables.

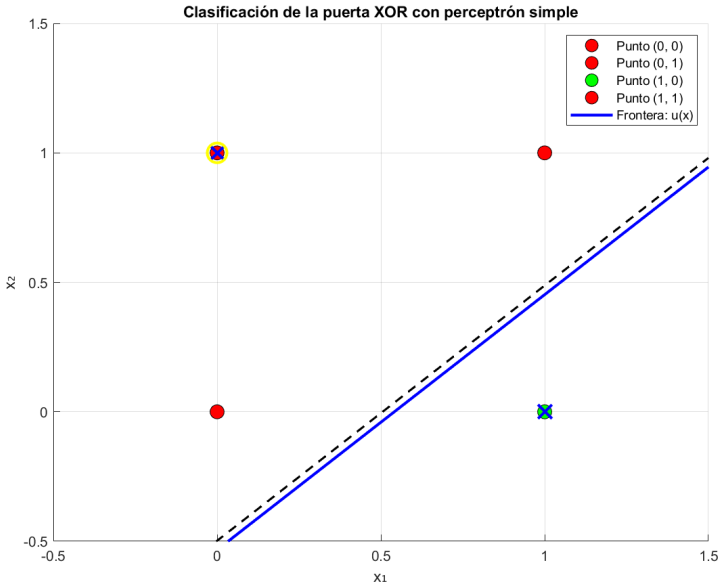


Figure 6: Clasificación de los puntos de la puerta lógica XOR

3 Perceptrón multicapa

El perceptrón de una única capa sólo puede hacer clasificaciones mediante rectas. La clasificación como una puerta XOR no es posible así. Menos aún separar los puntos de la circunferencia unidad de los de fuera. Esta limitación se puede solventar añadiendo capas al perceptrón. De hecho, con suficientes capas, un perceptrón con función de activación sigmoide puede aproximar cualquier función continua (se dice que es un aproximador universal).

Vamos a trabajar con una capa adicional. Una forma de interpretar esto es usar n perceptrones de una única capa, con lo cual las características $(x_1, \ldots, x_n)$ originales, se transforman en unas nuevas características $(\tilde{x}_1, \ldots, \tilde{x}_n)$, a las cuales se les aplica un único perceptrón para obtener, finalmente, un único valor escalar con el que discriminar la pertenencia del objeto a la categoría de interés.

$$U(x) = Wx + b,$$

donde $W = (w_{ij})_{i,j=1}^{n} \in \mathbb{R}^{n\times n}$ y $b = (b_1, \ldots, b_n) \in \mathbb{R}^n$. El resultado de esto es un vector del mismo tamaño que x, al que tendremos que aplicar múltiples funciones de activación. Esto lo podemos plantear como una función

$$\Phi(z_1, \ldots, z_n) = (\phi(z_1), \ldots, \phi(z_n)).$$

El resultado de $(\Phi \circ U)(x)$ es un nuevo vector $\tilde{x} \in \mathbb{R}^n$, al que aplicaremos las funciones de la sección anterior. Es decir, el perceptrón de dos capas podemos escribirlo como la siguiente composición

$$F(x) = (\phi \circ u \circ \Phi \circ U)(x) = [(\phi \circ u) \circ \phi \circ U](x) = (\phi \circ u)(\tilde{x}). \tag{5}$$

Sin embargo, cabe recordar que estamos omitiendo la dependencia de los parámetros. Es decir, en términos generales, las funciones con las que estamos trabajando funcionan del siguiente modo:

$$U : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^{n\times n} \to \mathbb{R}^n,$$
$$\Phi : \mathbb{R}^n \to \mathbb{R}^n,$$
$$u : \mathbb{R}^n \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n,$$
$$\phi : \mathbb{R} \to \mathbb{R}\,.$$

Esto hace que, términos generales, la composición de la ecuación (5) sea incorrecta. No obstante, en el caso de tener los parámetros ya establecidos, la ecuación (5) simplemente representa la cadena de composición que hace falta para evaluar la función en $x = (x_1, \ldots, x_n)$ y la podemos dar por correcta a esos efectos.

Por otra parte, dado que a la hora de entrenar la red neuronal vamos a considerar que las características $x = (x_1, \ldots, x_n)$ están fijas (por los datos observacionales), puede ser útil establecer algo de notación para referirse al gradiente o la matriz jacobiana de una función respecto a los parámetros. Eso se suele denotar por ∇_θ o D_θ

3.1 Ejercicios propuestos

3.1.1 Siguiendo la estela de los ejercicios anteriores, escribe en Matlab una función para evaluar la función U(x) dados unos datos x, una matriz W y un vector de sesgos b.

```
function u_value = funcion_oculta_matriz(b, W, x)
% FUNCION_OCULTA Calcula la activación lineal de varias neuronas
    ocultas.
%    u(x) = W * x + b
%    Args:
%        b  (kx1 vector)   : Vector de sesgos.
%        W  (kxn matrix)   : Matriz de pesos. Cada fila corresponde a
    una neurona.
%        x  (nxm matrix)   : Entradas (n características, m patrones).
%
%    Returns:
%        u_value (kxm matrix): Salida de cada neurona para cada patrón.

```

```
    u_value = W * x + b;
end
```

Listing 15: Funcion funcion_oculta_matriz

3.1.2 Escribe una función que calcule $\Phi(z_1, ..., z_n)$, así como su matriz jacobiana $D\Phi(z_1, ..., z_n)$

Si $\vec{F} : \mathbb{R}^n \to \mathbb{R}^m$ está dada por

$$\vec{F} = [f_1(x_1, x_2, \ldots, x_n) \quad f_2(x_1, x_2, \ldots, x_n) \quad \ldots \quad f_m(x_1, x_2, \ldots, x_n)]$$

La matriz jacobiana J es

$$DF(x_1, \ldots, x_n) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Se puede interpretar Φ como una función $\vec{\Phi} : \mathbb{R}^n \to \mathbb{R}^n$ tal que $\vec{\Phi}(\vec{z}) = (\phi_1(\vec{z}), \ldots, \phi_n(\vec{z}))$.
Definiendo cada $\phi_i : \mathbb{R}^n \to \mathbb{R}$,

$$\phi_i(\vec{z}) = \phi_i(z_1, \ldots, z_n) = \phi(z_i) = \frac{1}{1 + e^{-z_i}} \tag{6}$$

De esta forma,

$$\Phi(z_1, \ldots, z_n) = (\phi_1(z_1, \ldots, z_n), \ldots, \phi_n(z_1, \ldots, z_n)) = (\phi(z_1), \ldots, \phi(z_n))$$

Reescribiendo F, f_i en función de Φ, ϕ_i,

$$D\Phi(z_1, \ldots, z_n) = \begin{bmatrix} \frac{\partial \phi_1}{\partial z_1} & \frac{\partial \phi_1}{\partial z_2} & \cdots & \frac{\partial \phi_1}{\partial z_n} \\ \frac{\partial \phi_2}{\partial z_1} & \frac{\partial \phi_2}{\partial z_2} & \cdots & \frac{\partial \phi_2}{\partial z_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \phi_n}{\partial z_1} & \frac{\partial \phi_n}{\partial z_2} & \cdots & \frac{\partial \phi_n}{\partial z_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial \phi_1}{\partial z_1} & 0 & \cdots & 0 \\ 0 & \frac{\partial \phi_2}{\partial z_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\partial \phi_n}{\partial z_n} \end{bmatrix} = \begin{bmatrix} \phi'(z_1) & 0 & \cdots & 0 \\ 0 & \phi'(z_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \phi'(z_n) \end{bmatrix},$$

porque cada función ϕ_i depende únicamente de su z_i correspondiente:

$$\frac{\partial \phi_i}{\partial z_j} = \begin{cases} \phi'(z_i) & \text{si } i = j, \\ 0 & \text{si } i \neq j. \end{cases}$$

3.1.3 Asumiendo que los datos x están fijos, escribe cuál sería formalmente la derivada parcial $\partial_b F$, y la derivada parcial $\partial_{w_1} F$

Para distinguir mejor los parámetros de la primera y segunda capa, los parámetros de la segunda se indican con un [2]: $b^{[2]}$ para el sesgo, $w^{[2]} = (w_1^{[2]}, \ldots, w_n^{[2]})$ para los pesos. Asímismo, los de la primera capa vienen dados por $w_{ij}^{[1]}, b_i^{[1]}$

La derivada parcial de F con respecto a $b^{[2]}$ es:

$$\frac{\partial F}{\partial b^{[2]}} = \frac{\partial}{\partial b^{[2]}} \left(\phi(u(\Phi(U(x, W, \mathbf{b})), b^{[2]}, w^{[2]}) \right).$$

Simplificando la expresión, $\Phi(U(x, W, \mathbf{b})) = z$. Buscamos entonces calcular

$$\frac{\partial}{\partial b^{[2]}} \left(\phi(u(z, b^{[2]}, w^{[2]})) \right)$$

Aplicando la regla de la cadena:

$$\frac{\partial F}{\partial b^{[2]}} = \phi'(u(z, b^{[2]}, w^{[2]})) \cdot \frac{\partial (u(z, b^{[2]}, w^{[2]}))}{\partial b^{[2]}}$$

Como $u(z, b^{[2]}, w^{[2]}) = {w^{[2]}}^t z + b^{[2]}$, se tiene que $\frac{\partial u}{\partial b^{[2]}} = 1$

Entonces,

$$\frac{\partial F}{\partial b^{[2]}} = \phi'(u(z, b^{[2]}, w^{[2]}))$$

De manera similar, la derivada parcial de F con respecto a $w_1^{[2]}$ es:

$$\frac{\partial F}{\partial w_1^{[2]}} = \frac{\partial}{\partial w_1^{[2]}} \left(\phi(u(z, b^{[2]}, w^{[2]})) \right)$$

Aplicando la regla de la cadena:

$$\frac{\partial F}{\partial w_1^{[2]}} = \phi'(u(z, b^{[2]}, w^{[2]})) \cdot \frac{\partial (u(z, b^{[2]}, w^{[2]}))}{\partial w_1^{[2]}}$$

Como $u(z, b^{[2]}, w^{[2]}) = {w^{[2]}}^t \tilde{x} + b^{[2]} = w_1^{[2]} z_1 + \cdots + w_n^{[2]} z_n + b^{[2]}$, entonces $\frac{\partial u}{\partial w_1^{[2]}} = z_1$

Finalmente,

$$\frac{\partial F}{\partial w_1^{[2]}} = z_1 \cdot \phi'(u(z, b^{[2]}, w^{[2]}))$$

De manera general,

$$\frac{\partial F}{\partial w_i^{[2]}} = z_i \cdot \phi'(u(z, b^{[2]}, w^{[2]})),\ \forall i = \{1, \ldots, n\}$$

3.1.4 Escribe formalemente cuál es $\partial_{b_1} U$ y $\partial_{w_{11}} U$

$$U(x, W, \mathbf{b^{[1]}}) = Wx + \mathbf{b^{[1]}} = \begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} & \ldots & w_{1n}^{[1]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1}^{[1]} & w_{n2}^{[1]} & \ldots & w_{nn}^{[1]} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ \vdots \\ b_n^{[1]} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^n (w_{1j}^{[1]} \cdot x_j) + b_1^{[1]} \\ \vdots \\ \sum_{j=1}^n (w_{nj}^{[1]} \cdot x_j) + b_n^{[1]} \end{bmatrix}$$

$$\partial_{b_1} U = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

$$\partial_{w_{11}} U = \begin{bmatrix} x_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

3.1.5 Escribe formalmente (con la regla de la cadena) cuál sería $\partial_{b_1} F$ y $\partial_{w_{11}} F$

Para poder derivar parcialmente la composición de las funciones, se muestra el árbol de recurrencia en función de $b^{[1]}$.

Teniendo en cuenta la ecuación (5) y partiendo del desarrollo de $U(x, W, \mathbf{b^{[1]}})$, en la primera capa:

$$U(x, W, \mathbf{b^{[1]}}) = \begin{bmatrix} \sum_{j=1}^n (w_{1i}^{[1]} \cdot x_j) + b_1^{[1]} \\ \vdots \\ \sum_{j=1}^n (w_{ni}^{[1]} \cdot x_j) + b_n^{[1]} \end{bmatrix} = \begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix} = s$$

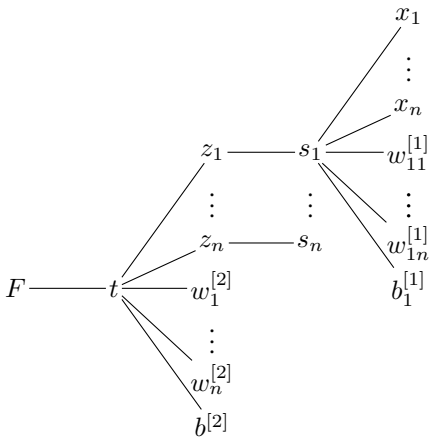
$$\Phi(s) = \Phi\left(\begin{bmatrix} s_1 \\ \vdots \\ s_n \end{bmatrix}\right) = \begin{bmatrix} \phi(s_1) \\ \vdots \\ \phi(s_n) \end{bmatrix} = \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix} = z$$

En la segunda capa,

$$u(z, w^{[2]}, b^{[2]}) = [w_1^{[2]}, \dots w_n^{[2]}] \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix} + b^{[2]} = \sum_{i=1}^{n} (w_i^{[2]} \cdot z_i) + b^{[2]} = t$$

$$F = \phi(t)$$

El árbol de recurrencia es el siguiente:

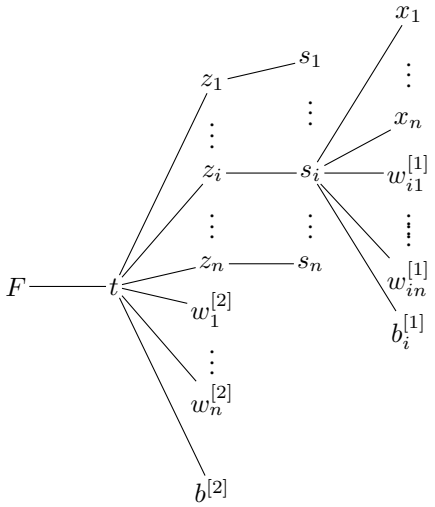


Se han omitido las ramas de $s_2, \dots s_n$ al estar derivando unicamente en función de $b_1^{[1]}$ y $w_{11}^{[1]}$
Aplicando la regla de la cadena,

$$\frac{\partial F}{\partial b_1^{[1]}} = \frac{\partial F}{\partial t} \cdot \frac{\partial t}{\partial z_1} \cdot \frac{\partial z_1}{\partial s_1} \cdot \frac{\partial s_1}{\partial b_1^{[1]}} = \phi'(t) \cdot w_1^{[2]} \cdot \phi'(s_1)$$

$$\frac{\partial F}{\partial w_{11}^{[1]}} = \frac{\partial F}{\partial t} \cdot \frac{\partial t}{\partial z_1} \cdot \frac{\partial z_1}{\partial s_1} \cdot \frac{\partial s_1}{\partial w_{11}^{[1]}} = \phi'(t) \cdot w_1^{[2]} \cdot \phi'(s_1) \cdot x_1$$

De manera general, para un $b_i^{[1]}$, $w_{ij}^{[1]}$:



Recordando que

$$s_i = \sum_{j=1}^{n} w_{ij}^{[1]} \cdot x_j + b_i^{[1]}$$

$$z_i = \phi(s_i)$$

$$t = \sum_{i=1}^{n} w_i^{[2]} \cdot z_i + b^{[2]}$$

$$F = \phi(t),$$

De nuevo, se tiene que

$$\frac{\partial F}{\partial b_i^{[1]}} = \frac{\partial F}{\partial t} \cdot \frac{\partial t}{\partial z_i} \cdot \frac{\partial z_i}{\partial s_i} \cdot \frac{\partial s_i}{\partial b_i^{[1]}}$$

$$\frac{\partial F}{\partial w_{ij}^{[1]}} = \frac{\partial F}{\partial t} \cdot \frac{\partial t}{\partial z_i} \cdot \frac{\partial z_i}{\partial s_i} \cdot \frac{\partial s_i}{\partial w_{ij}^{[1]}}$$

Calculamos cada producto explícitamente, será útil para implementarlo en MATLAB.

$$\frac{\partial F}{\partial t} = \frac{\partial \phi(t)}{\partial t} = \phi'(t)$$

$$\frac{\partial t}{\partial z_i} = \frac{\partial \sum_{i=1}^{n} w_i^{[2]} \cdot z_i + b^{[2]}}{\partial z_i} = w_i^{[2]}$$

$$\frac{\partial z_i}{\partial s_i} = \frac{\partial \phi(s_i)}{\partial t} = \phi'(s_i)$$

$$\frac{\partial s_i}{\partial b_i^{[1]}} = \frac{\partial \sum_{j=1}^{n} w_{ij}^{[1]} \cdot x_j + b_i^{[1]}}{\partial b_i^{[1]}} = 1$$

$$\frac{\partial s_i}{\partial w_{ij}^{[1]}} = \frac{\partial \sum_{j=1}^{n} w_{ij}^{[1]} \cdot x_j + b_i^{[1]}}{\partial w_{ij}^{[1]}} = x_j$$

Sustituyendo,

$$\frac{\partial F}{\partial b_i^{[1]}} = \phi'(t) \cdot w_i^{[2]} \cdot \phi'(s_i)$$

$$\frac{\partial F}{\partial w_{ij}^{[1]}} = \phi'(t) \cdot w_i^{[2]} \cdot \phi'(s_i) \cdot x_j$$

Por último, desarrollando completamente las expresiones:

$$\frac{\partial F}{\partial b_i^{[1]}} = \phi'\left(\sum_{i=1}^{n} w_i^{[2]} \cdot \phi\left(\sum_{j=1}^{n} w_{ij}^{[1]} \cdot x_j + b_i^{[1]}\right) + b^{[2]}\right) \cdot w_i^{[2]} \cdot \phi'\left(\sum_{j=1}^{n} w_{ij}^{[1]} \cdot x_j + b_i^{[1]}\right)$$

$$\frac{\partial F}{\partial w_{ij}^{[1]}} = \phi'\left(\sum_{i=1}^{n} w_i^{[2]} \cdot \phi\left(\sum_{j=1}^{n} w_{ij}^{[1]} \cdot x_j + b_i^{[1]}\right) + b^{[2]}\right) \cdot w_i^{[2]} \cdot \phi'\left(\sum_{j=1}^{n} w_{ij}^{[1]} \cdot x_j + b_i^{[1]}\right) \cdot x_j$$

$$\frac{\partial F}{\partial b^{[2]}} = \phi'\left(\sum_{i=1}^{n} w_i^{[2]} \cdot \phi\left(\sum_{j=1}^{n} w_{ij}^{[1]} \cdot x_j + b_i^{[1]}\right) + b^{[2]}\right)$$

$$\frac{\partial F}{\partial w_i^{[2]}} = \phi'\left(\sum_{i=1}^{n} w_i^{[2]} \cdot \phi\left(\sum_{j=1}^{n} w_{ij}^{[1]} \cdot x_j + b_i^{[1]}\right) + b^{[2]}\right) \cdot \phi\left(\sum_{j=1}^{n} w_{ij}^{[1]} \cdot x_j + b_i^{[1]}\right)$$

3.1.6 ¿Por qué variar un parámetros como w_{11} hace que aparezcan las derivadas de u respecto a sus variables x?

A medida que se avanza en las capas de la red, los inputs que recibe cada nueva capa corresponden a los outputs generados por la capa anterior, los cuales están multiplicados por los pesos correspondientes. En este contexto, el parámetro w_{11} influye directamente en los outputs de la primera capa, y estos, a su vez, pasan a formar parte del input de la siguiente capa. Como consecuencia, al derivar la función final u respecto a sus variables de entrada, aparecen términos dependientes de w_{11}, lo que explica por qué su variación afecta a dichas derivadas.

3.1.7 Para el caso del estudio de las puertas lógicas de dos entradas y una salida, ¿cuántos parámetros hay que establecer en el perceptrón de dos capas?

Si hablamos de dimensiones nos encontramos con lo siguiente:

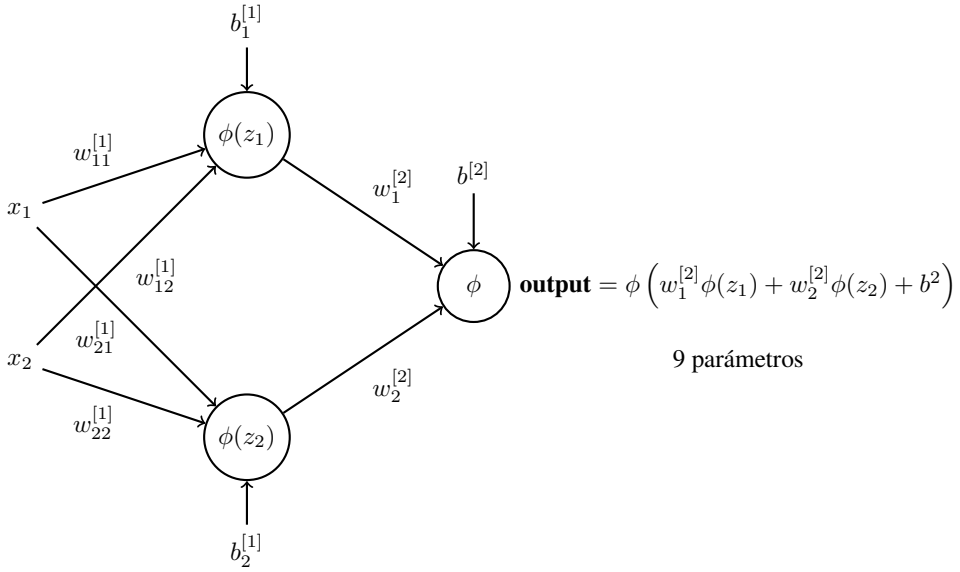
$$w^{[2]^T}\left(Wx + \mathbf{b^{[1]}}\right) + b^{[2]}$$

Por tanto, si contamos las dimensiones de las matrices relacionadas con los parámetros, nos encontramos con:

$$\begin{aligned} W: &\quad n \times n = n^2 \\ \mathbf{b^{[1]}}: &\quad n \\ w^{[2]}: &\quad n \\ b^{[2]}: &\quad 1 \end{aligned}$$

Entonces, sumando todos los parámetros nos da $n^2 + 2n + 1$ parámetros. Sustituyendo ahora que tenemos 2 inputs nos da 4 + 4 + 1 = **9 parámetros.**

Otra opción es representar gráficamente cómo se establecen las conexiones y calcular el número total de parámetros, obteniendo así la siguiente red:



3.1.8 Para este caso, escribe en Matlab un código que evalúe las derivadas de F respecto a cada uno de los parámetros

Se implementa en MATLAB la clase Perceptron_2 con los métodos de entrenamiento y clasificación, así como el cálculo de las derivadas necesarias

```
classdef Perceptron_2
    properties
        % Parámetros de la red
```

```
        W1  % (n x n), pesos de la capa oculta
        b1  % (n x 1), bias   de la capa oculta
        W2  % (n x 1), pesos de la capa de salida
        b2  % escalar,  bias de la capa de salida

        n   % número de neuronas en la capa oculta y dimensión de la
            entrada
    end

    methods
        %-------------------------------------------------------------
        % Constructor
        %-------------------------------------------------------------
        function obj = Perceptron_2(n)
            if nargin > 0
                obj.n = n;

                % 1) Cambiamos la semilla para que no inicie siempre
                    igual
                rng('shuffle');

                % 2) Aumentar la dispersión de la inicialización
                obj.W1 = 0.3 * randn(n,n);
                obj.b1 = 0.3 * randn(n,1);
                obj.W2 = 0.3 * randn(n,1);
                obj.b2 = 0.3 * randn;
            end
        end

        %-------------------------------------------------------------
        % Método FIT
        %-------------------------------------------------------------
        function obj = fit(obj, X, Y, alpha, maxIter, tol)
            if nargin < 6
                tol = 1e-3; % Por defecto si no se especifica
            end

            N = size(X, 2);
            iter = 0;
            cost = inf;

            while (iter < maxIter) && (cost > tol)
                iter = iter + 1;

                % Barajar el orden de los patrones en cada epoch
                idx_perm = randperm(N);

                % Inicializamos acumuladores de gradiente
                accum_dFb1 = zeros(obj.n,1);
                accum_dFw1 = zeros(obj.n,obj.n);
                accum_dFb2 = 0;
                accum_dFw2 = zeros(obj.n,1);

                cost = 0;

                % Recorremos los patrones en orden aleatorio
                for kk = 1:N
                    k = idx_perm(kk);
                    xk = X(:,k);
                    yk = Y(k);

                    % 1) Forward y derivadas
                    [F_k, dFb1_k, dFw1_k, dFb2_k, dFw2_k] = ...
                        obj.gradienteMulticapa(xk);

```

```
                    % 2) Cálculo de coste (MSE en un patrón)
                    e_k = (F_k - yk);
                    cost_k = 0.5 * e_k^2;
                    cost = cost + cost_k;

                    % 3) Regla de la cadena: derivada del coste
                    %    d(Cost) = (F_k - yk) * dF_k
                    chain = e_k;
                    accum_dFb1 = accum_dFb1 + chain * dFb1_k;
                    accum_dFw1 = accum_dFw1 + chain * dFw1_k;
                    accum_dFb2 = accum_dFb2 + chain * dFb2_k;
                    accum_dFw2 = accum_dFw2 + chain * dFw2_k;
                end

                % 4) Actualizar parámetros
                obj.W1 = obj.W1 - alpha * accum_dFw1;
                obj.b1 = obj.b1 - alpha * accum_dFb1;
                obj.W2 = obj.W2 - alpha * accum_dFw2;
                obj.b2 = obj.b2 - alpha * accum_dFb2;

                % (Opcional) ver progresión
                if mod(iter, 500) == 0
                    fprintf('Iter %d, Coste total = %.6f\n', iter,
                        cost);
                end
            end

            fprintf('\nTerminó fit() en %d iteraciones, Coste final
                =%.4f\n',...
                    iter, cost);
        end

        %-------------------------------------------------------------
        % Predicción
        %-------------------------------------------------------------
        function F = predict(obj, x)
            if size(x,2) == 1
                F = obj.forwardOneSample(x);
            else
                M = size(x,2);
                F = zeros(1,M);
                for m = 1:M
                    F(m) = obj.forwardOneSample(x(:,m));
                end
            end
        end

        %-------------------------------------------------------------
        % Clasificación binaria (umbral tau)
        %-------------------------------------------------------------
        function class = predict_class(obj, x, tau)
            val = obj.predict(x);
            class = double(val >= tau);
        end
    end

    methods (Access = private)
        %-------------------------------------------------------------
        % forwardOneSample: salida para un patrón
        %-------------------------------------------------------------
        function F = forwardOneSample(obj, x)
            s = zeros(obj.n,1);
            for i = 1:obj.n
                s(i) = obj.W1(i,:) * x + obj.b1(i);
            end
```

```
130             z = obj.sigmoid(s);
131
132             t = obj.b2 + obj.W2' * z;
133             F = obj.sigmoid(t);
134         end
135
136         %---------------------------------------------------------------
137         % gradienteMulticapa: F y derivadas para un patrón
138         %---------------------------------------------------------------
139         function [F, dFb1, dFw1, dFb2, dFw2] = gradienteMulticapa(obj,
                x)
140             % 1) Capa oculta
141             s = zeros(obj.n,1);
142             for i = 1:obj.n
143                 s(i) = obj.W1(i,:)*x + obj.b1(i);
144             end
145             z = obj.sigmoid(s);
146
147             % 2) Capa salida
148             t = obj.b2 + obj.W2' * z;
149             F = obj.sigmoid(t);
150
151             % 3) Derivadas: dF/dt = sig'(t)
152             dt = obj.sigmoidDiff(t);
153
154             %     dF/db2 = dt
155             dFb2 = dt;
156
157             %     dF/dw2_i = dt * z_i
158             dFw2 = zeros(obj.n,1);
159             for i = 1:obj.n
160                 dFw2(i) = dt * z(i);
161             end
162
163             %     dF/db1_i = dt * w2_i * sig'(s_i)
164             dFb1 = zeros(obj.n,1);
165             for i = 1:obj.n
166                 dFb1(i) = dt * obj.W2(i) * obj.sigmoidDiff(s(i));
167             end
168
169             %     dF/dw1_{i,j} = dt * w2_i * sig'(s_i)* x_j
170             dFw1 = zeros(obj.n,obj.n);
171             for i = 1:obj.n
172                 for j = 1:obj.n
173                     dFw1(i,j) = dt * obj.W2(i) * obj.sigmoidDiff(s(i))
                            * x(j);
174                 end
175             end
176         end
177
178         %---------------------------------------------------------------
179         % Funciones de activación
180         %---------------------------------------------------------------
181         function y = sigmoid(~, u)
182             y = 1./(1+exp(-u));
183         end
184
185         function y = sigmoidDiff(obj, u)
186             su = obj.sigmoid(u);
187             y = su.*(1 - su);
188         end
189     end
190 end
```

Listing 16: Clase Perceptron_2

3.1.9 Utilizando la función anterior, entrena esta red neuronal para aproximar la puerta lógica XOR

```
%% Pruebas de Red Neuronal Multicapa para la Puerta XOR
% Este Live Script entrena una red neuronal de dos capas (una oculta y una de salida)
% utilizando la clase 'Perceptron_2'. El objetivo es aprender el comportamiento de la función XOR.

clc; clear; close all;

%% Definir los datos de la puerta XOR
X = [0 0 1 1;
     0 1 0 1];    % (2 x 4) columnas = patrones
Y = [0 1 1 0];    % (1 x 4) salidas deseadas

disp("Datos de entrada (X):");
disp(X);

disp("Etiquetas deseadas (Y):");
disp(Y);

%% Crear la red neuronal
% Se utiliza una red con:
% - 2 entradas
% - 2 neuronas en la capa oculta
% - Función de activación: sigmoide
% - Activación definida dentro de la clase

n = 2;  % número de entradas y de neuronas ocultas
redXOR = Perceptron_2(n);

%% Entrenar la red
% Parámetros del entrenamiento:
alpha = 0.5;        % tasa de aprendizaje
maxIter = 60000;     % máximo de iteraciones
tol = 1e-4;         % tolerancia (umbral) para el coste mínimo

disp("Iniciando entrenamiento de la red...");
redXOR = redXOR.fit(X, Y, alpha, maxIter, tol);

%% Evaluar la red entrenada
% Mostramos la salida continua y la clase binaria para cada entrada XOR
fprintf('\n--- RESULTADOS FINALES ---\n');
for k = 1:4
    xk = X(:,k);
    yk = Y(k);
    salida = redXOR.predict(xk);
    clase = redXOR.predict_class(xk, 0.5);  % umbral = 0.5
    fprintf('XOR(%d, %d) -> Salida = %.4f, Clase = %d, Esperado = %d\n', ...
        xk(1), xk(2), salida, clase, yk);
end

%% Visualizar la función lógica aprendida
% Se visualiza la salida de la red para una malla de puntos en [0,1]x[0,1]

[X1, X2] = meshgrid(linspace(0,1,100), linspace(0,1,100));
Z = zeros(size(X1));


for i = 1:size(X1,1)
    for j = 1:size(X1,2)
```

```
        punto = [X1(i,j); X2(i,j)];
        Z(i,j) = redXOR.predict(punto);
    end
end

figure;
surf(X1, X2, Z);
title('Superficie aprendida por la red neuronal (XOR)');
xlabel('x1'); ylabel('x2'); zlabel('Salida');
colormap turbo;
colorbar;
shading interp;
```

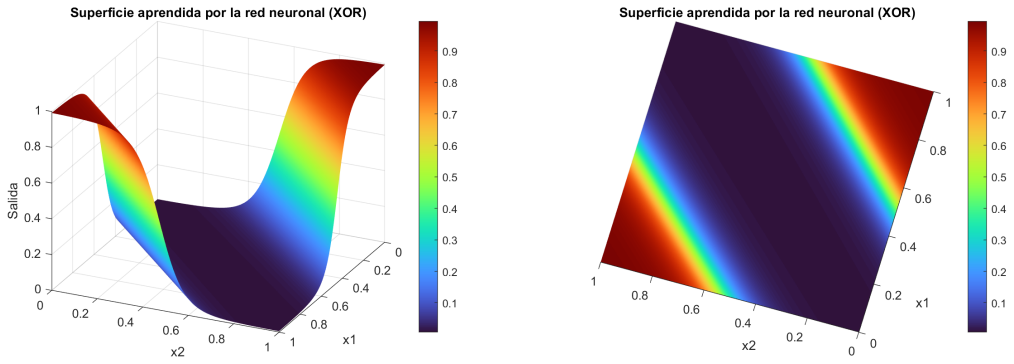
Listing 17: Entrenamiento Perceptrón XOR

Los resultados obtenidos con la nueva clase son los siguientes:

Entrada 1	Entrada 2	Salida	Clase	Esperado	XOR
0	0	0.0065	0	0	XOR(0, 0)
0	1	0.9926	1	1	XOR(0, 1)
1	0	0.9926	1	1	XOR(1, 0)
1	1	0.0069	0	0	XOR(1, 1)

Table 3: Resultados obtenidos para la puerta XOR tras el entrenamiento.

También se grafica la superficie aprendida por la red neuronal:



(a) Vista 3D de la superficie (b) Vista en planta (proyección)

Figure 7: Superficie aprendida por la red neuronal desde diferentes perspectivas.

Con esto concluyen los apartados obligatorios del proyecto.

4 Apartados Opcionales

4.1 Estudia cómo modificar las funciones y el algoritmo para considerar una capa intermedia más grande que la cantidad de características $(x_1, \ldots, x_n)$ utilizadas para describir un objeto. Comprueba si es útil para aproximar mejor el comportamiento de la puerta lógica XOR.

Para **ensanchar la capa oculta**, es decir, hacerla de dimensión mayor que el número de características de entrada n, se deben modificar los siguientes aspectos del modelo:

1. Dimensiones de pesos y sesgos

- Como se desean usar $M > n$ neuronas en la capa oculta, la matriz de pesos de la primera capa $W^{[1]}$ será de dimensión $M \times n$.
- Los sesgos correspondientes a cada perceptrón de la primera capa serán $b_i^{[1]} \in \mathbb{R}$, para $i = 1, \ldots, M$.

2. Propagación hacia adelante (Forward)

$$
\begin{aligned}
s_i &= \sum_{j=1}^{n} w_{ij}^{[1]} x_j + b_i^{[1]} \\
z_i &= \phi(s_i) \\
t &= \sum_{i=1}^{M} w_i^{[2]} z_i + b^{[2]} \\
F &= \phi(t)
\end{aligned}
$$

3. Cálculo de derivadas (Backpropagation)

Aplicando la regla de la cadena para los nuevos índices:

$$
\begin{aligned}
\frac{\partial F}{\partial b_i^{[1]}} &= \frac{\partial F}{\partial t} \cdot \frac{\partial t}{\partial z_i} \cdot \frac{\partial z_i}{\partial s_i} \cdot \frac{\partial s_i}{\partial b_i^{[1]}} = \phi'(t) \cdot w_i^{[2]} \cdot \phi'(s_i) \\
\frac{\partial F}{\partial w_{ij}^{[1]}} &= \frac{\partial F}{\partial t} \cdot \frac{\partial t}{\partial z_i} \cdot \frac{\partial z_i}{\partial s_i} \cdot \frac{\partial s_i}{\partial w_{ij}^{[1]}} = \phi'(t) \cdot w_i^{[2]} \cdot \phi'(s_i) \cdot x_j \\
\frac{\partial F}{\partial b^{[2]}} &= \phi'(t) \\
\frac{\partial F}{\partial w_i^{[2]}} &= \phi'(t) \cdot z_i
\end{aligned}
$$

4. Utilidad para XOR

Para el problema XOR, con dos entradas x_1, x_2, normalmente se necesita al menos una capa oculta con $M = 2$ neuronas. Sin embargo, usar más neuronas puede:

- Mejorar la capacidad de aproximación de la red.
- Facilitar la convergencia durante el entrenamiento.
- Proveer redundancia útil frente a datos ruidosos o variaciones del problema.

Por tanto, sí puede ser útil considerar una capa intermedia más grande para aproximar el comportamiento de la puerta lógica XOR, especialmente si se pretende generalizar mejor o acelerar el entrenamiento.