# Apuntes sobre inteligencia artificial

Sergio de Mingo

6 de Febrero de 2026

## Índice



## 1. El perceptrón

El Perceptrón (propuesto por Rosenblatt en los 50) es la unidad atómica de la IA. Es básicamente una función matemática que intenta imitar a una neurona biológica. No es más que un clasificador lineal binario. Un clasificador binario es una función que puede decidir si una entrada, representada por un vector de números, pertenece o no a una clase específica.

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

- Entradas ($x$): Los datos que llegan (ej. valores de píxeles).
- Pesos ($w$): La "importancia" que la red le da a cada entrada. Aprender en IA es, literalmente, ajustar estos números.
- Bias ($b$): El sesgo, que permite desplazar la función de activación hacia la izquierda o derecha para ajustarse mejor a los datos. Es como el umbral de un interruptor: Si el bias es muy alto y positivo, la neurona es «entusiasta» y se activa casi siempre. Por contra, si el bias es muy negativo, la neurona es «escéptica» y necesita señales de entrada muy fuertes para activarse.
- Función de activación ($f$): Decide si la neurona "dispara" una señal o no. Antiguamente usábamos la Heaviside (escalón) o la Sigmoide.

La limitación de este algoritmo es que si dibujamos en un gráfico estos elementos, se deben poder separar con un hiperplano únicamente los elementos «deseados» discriminándolos (separándolos) de los «no deseados». Esto lo veremos mejor en el punto 2, donde estudiaremos una Red Neuronal Clásica (Multilayer Perceptron). Este tipo no es más que muchas de estas neuronas organizadas en capas: una de entrada, una o varias ocultas y una de salida.

### 1.1. El aprendizaje de un perceptrón

El proceso de aprendizaje de un perceptrón comienza con una fase de **inicialización**, donde el modelo parte de un estado de desconocimiento absoluto, asignando valores aleatorios a sus pesos y un valor inicial (normalmente cero) a su sesgo o bias. Una vez configurado, **el aprendizaje se convierte en un ciclo iterativo que se repite durante varias épocas**. En cada iteración, el perceptrón realiza primero un forward pass o predicción: toma los datos de entrada, los multiplica por sus respectivos pesos, suma el sesgo y pasa el resultado por una función de activación (como la función escalón) para decidir si la neurona debe activarse o no. Inmediatamente después, el modelo compara su predicción con el valor real ($y_i$) que debería haber obtenido, calculando así el error:

$$Error = y_i - f(w_i \cdot x_i + b)$$

Si existe una discrepancia, entra en juego la regla de aprendizaje, donde el perceptrón ajusta sus pesos internos de forma proporcional a tres factores: la magnitud del error cometido, el valor de la entrada que causó dicho error y, fundamentalmente, la tasa de aprendizaje, que actúa como un regulador de la velocidad del cambio.

$$w_i = w_i + (x_i \times \text{Error} \times \text{Tasa de aprendizaje})$$

Este ajuste busca "mover" la frontera de decisión del perceptrón (la línea recta que separa las clases) para que, en el siguiente intento, la predicción sea más precisa. A través de la repetición constante de este proceso con todos los datos del conjunto de entrenamiento, los pesos convergen gradualmente hacia unos valores óptimos que permiten al modelo clasificar correctamente las entradas, logrando así que la máquina "aprenda" la lógica subyacente de los datos,

### 1.2. Trabajando con el perceptrón

Para empezar a programar un perceptrón necesitamos establecer su estado interno, tanto los pesos $W$ como el sesgo $b$. Normalmente estos se inicializarán con valores pequeños. Tras esto haremos el cálculo de la predicción en sí, donde aplicamos la fórmula inicial del perceptrón con la aplicación de la función de activación. Por último, el aprendizaje o el ajuste de los pesos. En este caso esto se ven el método `init()`. Este método recibe además el número de inputs que tendrá el perceptrón para poder generar un array de pesos adecuado. El método `predict()` es básicamente la aplicación de la función de activación para cada componente del vector de entrada `X`. En este caso `Z` es el producto de cada componente del vector por el peso asignado (y al que al final sumamos el sesgo). Como activación usamos la función escalón por lo que el resultado será 1 si $z > 0$ y 0 en cualquier otro caso. El mecanismo fundamental se encuentra en el método `train()`. En este método es donde se irá probando al perceptrón y ajustando sus pesos hasta que el resultado sea el esperado. En este caso probamos 100 veces. En cada prueba realizamos el proceso de aprendizaje/reajuste, teniendo en cuenta que los resultados esperados los tenemos en el vector `y_AND`:

1. Calculamos la predicción para esa entrada: `prediction = self.predict(X[i])`

2. Calculamos el error comparando el resultado con lo esperado: `error = y[i] - prediction`

3. Recalculamos los pesos: `self.weights += error * self.lr * X[i]`

4. Recalculamos el sesgo: `self.bias += error * self.lr`

La magia se produce cuando tras probar el perceptrón con el vector de aprendizaje `y_AND` y viendo que ha sabido hacer una operación AND cambiamos el vector por `y_OR` y aprende a hacer una OR haciéndola correctamente para las mismas entradas.

Siguiendo este razonamiento nos topamos con el primer gran problema del perceptrón. Al intentar implementar este mismo aprendizaje para XOR usando un vector `y_XOR = np.array([0, 1, 1, 0])` vemos que es imposible. Si visualizamos el espacio de entrada como un plano con dos ejes y cuatro puntos (0,0),(0,1),(1,0),(1,1), vemos que en el AND, solo el punto (1,1) es positivo. Puedes dibujar una línea recta que separe ese punto de los otros tres. Lo mismo pasa en el OR donde tres puntos son positivos y solo el (0,0) es negativo. También puedes trazar una línea recta para separarlos.

A continuación se muestra el código del fichero `src/perceptron.py`:

```
class Perceptron:
    def __init__(self, n_inputs, learning_rate=0.1):
        self.weights = np.random.randn(n_inputs)
        self.bias = 0
        self.lr = learning_rate

    def predict(self, x):
        z = np.dot(x, self.weights) + self.bias
        return 1 if z > 0 else 0

    def train(self, X, y, epochs=100):
        for _ in range(epochs):
            for i in range(len(X)):
                prediction = self.predict(X[i])
                error = y[i] - prediction
                self.weights += error * self.lr * X[i]
                self.bias += error * self.lr


X = np.array([[0,0], [0,1], [1,0], [1,1]])
y_AND = np.array([0, 0, 0, 1])
y_OR = np.array([0, 1, 1, 1])

p = Perceptron(n_inputs=2)
p.train(X, y_AND)

print("Probamos que sepa calcular un AND u OR:")
for t in X:
    print(f"Entrada: {t} -> Prediccion: {p.predict(t)}")
```

## 2. Redes multicapa

Si recordamos el razonamiento del punto anterior donde definíamos en un plano de coordenadas los resultados del perceptrón, si intentamos separar los resultados positivos del XOR vemos que es imposible hacerlo con una sola línea. En el XOR, los puntos positivos (los resultados válidos con los que se obtiene un 1) son (0,1) y (1,0) y los negativos son (0,0) y (1,1). Es imposible separar ambos espacios (de positivos y negativos) con una sola línea recta. Necesitas dos o bien una curva. Esto se explica matemáticamente porque un perceptrón simple es matemáticamente un hiperplano. En 2D,

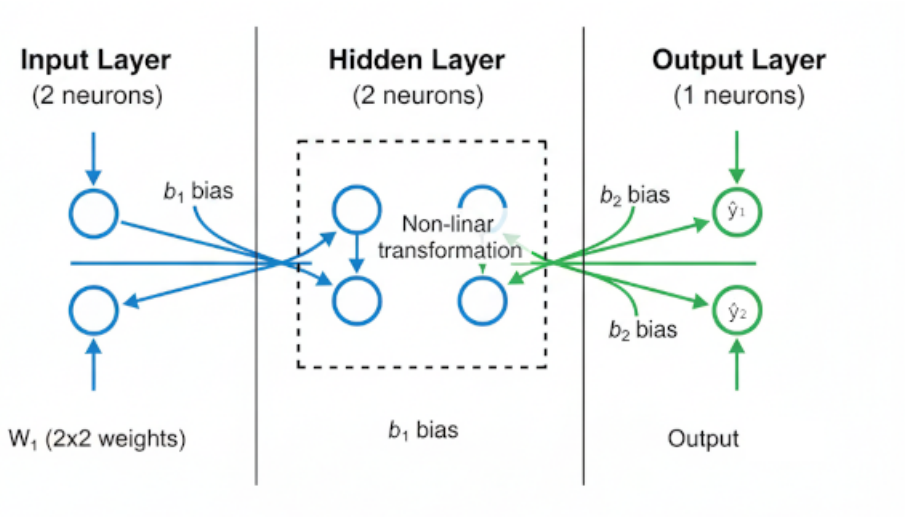


Figura 1: Esquema general de la red multicapa

es una recta. Si tus datos no son "linealmente separables", el perceptrón se quedará oscilando para siempre sin encontrar una solución. Para resolver el XOR, necesitamos añadir hacer una pequeña red multicapa:

1. **Capa de entrada** formada por dos neuronas: Recibirá los valores $x_1$ y $x_2$
2. **Capa oculta** formada por dos o tres neuronas: Aquí es donde ocurre la magia debido a que estas neuronas crearán nuevas dimensiones.
3. **Capa de salida** formada por una neurona que nos da el resultado final de la operación: 0 o 1.

Para realizar el cálculo ya no multiplicaremos el peso por la entrada en bucle, como haciamos antes. Ahora usaremos multiplicación de matrices. Siendo $X$ la matriz de entrada y $W_1$ la matriz de pesos de la capa oculta calcularemos $Z_1 = X \cdot W_1 + b_1$ y luego aplicaremos la función de activación. En este caso usaremos la sigmoide. Para la capa de salida calcularemos $Z_2 = A_1 \cdot W_2 + b_2$ y de nuevo activaremos con la función sigmoide. Como nota indicamos que usamos la función sigmoide debido a que es una función no lineal. Si usáramos una función lineal, por muchas capas que pongamos, la red seguiría siendo una simple combinación lineal (una sola línea recta). La no-linealidad es lo que permite "curvar" el espacio necesario para este caso.

## 2.1. Aprendizaje en varias capas

El concepto que revolucionó la IA en los 80 fue: ¿Cómo el error se calcula al final, en la salida? ¿cómo sabemos cuánto deben cambiar los pesos de la primera capa? El proceso de aprendizaje ahora se basa en estos tres pasos:

1. El *Forward Pass* o la multiplicación de las matrices y la activación
2. El cálculo del error; ¿Cuánto nos queda para el valor real?
3. El *Backpropagation* donde calculamos el «gradiente» de cada capa usando La Regla de la cadena.
4. Actualizamos los pesos restando el gradiente.

El **Gradiente** es lo que nos dice en qué dirección y con qué fuerza debemos mover el peso para que el error total disminuya lo más rápido posible. En una red neuronal, para saber cómo afecta un

peso de la Capa 1 al error final (que se mide en la Salida), tenemos que aplicar **La Regla de La Cadena** a través de todas las capas intermedias. El gradiente que llega a la Capa 1 es el producto de las derivadas de las capas superiores. La derivada es como la velocidad a la que fluye la información del error. Si la derivada es alta (cercana a 1 o mayor): El error viaja con fuerza. El peso $w_1$ recibe un mensaje claro: *"¡Oye! Te has equivocado mucho, cambia tu valor rápido"*. Hay aprendizaje. Si la derivada es pequeña, el error se va atenuando en cada capa. ¿Por qué usamos derivadas? El objetivo de la red es minimizar una función de Error (o Pérdida). Imagina que el error es una montaña y tú estás en la cima, a ciegas. Quieres bajar al valle (error cero). ¿Cómo sabes hacia dónde dar el paso? Tocando el suelo con el pie para ver la pendiente. Esa pendiente es la derivada. Si la derivada es positiva, el terreno sube; vas hacia atrás. Si es negativa, el terreno baja; vas hacia adelante. El uso de la regla de la cadena es una cuestión de esto anterior. En definitiva una red neuronal es una función compuesta gigante. Si tenemos dos capas, una entrada $X$, una salida $Y$ y una función de activación $\sigma$. La función que resumiría la primera fase sería:

$$Y = \sigma_2\left(W_2 \cdot \sigma_1\left(W_1 \cdot X\right)\right)$$

Cuando queremos saber cómo afecta el peso de la primera capa $W_1$ al error final, tenemos que "deshacer" la función de fuera hacia adentro. La Regla de la Cadena nos dice que **la derivada de una función compuesta es el producto de las derivadas de sus componentes**. Usamos la derivada de la activación porque es la única forma matemática de saber cuánta responsabilidad tiene una neurona específica en el error final. Esto es exactamente lo mismo que el análisis de sensibilidad en ingeniería de sistemas. ¿Cómo afecta una pequeña variación en la entrada a la salida del sistema? La respuesta es siempre la derivada. Si definimos el error como $E$ y recordamos que $\sigma'$ es la derivada de la función de activación, la fórmula compacta para calcular el gradiente de la primera capa (la más profunda en el Backpropagation) es:

$$\delta_1 = \underbrace{\left(\overbrace{\underbrace{(A_2 - y)}_{\text{Gradiente Salida } (\delta_2)}}^{\text{Error}} \odot \sigma'(Z_2) \cdot W_2^T\right)}_{\text{Error retropropagado}} \odot \sigma'(Z_1)$$

1. El inicio del error: $(A_2 - y)$ calcula cuánto nos alejamos del objetivo.
2. El filtro de la salida: Al multiplicar por $\odot\sigma'(Z_2)$, decidimos cuánta importancia dar a ese error según el estado de la neurona de salida.
3. El salto al pasado: Al multiplicar por $W_2^T$, proyectamos ese error de la salida hacia las neuronas ocultas usando los mismos pesos (pero transpuestos).
4. El filtro oculto: Finalmente, $\odot\sigma'(Z_1)$ ajusta ese error proyectado según la sensibilidad de la capa oculta.

A continuación se muestra el bucle de aprendizaje de la multicapa implementada de forma completa en `src/multicapa.py`:

```
for epoch in range(epochs):
    # --- FORWARD PASS ---
    hidden_layer_input = np.dot(X, W1) + b1
    hidden_layer_output = sigmoid(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_output, W2) + b2
    predicted_output = sigmoid(output_layer_input)

    # --- BACKPROPAGATION ---
    error = y - predicted_output
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden_layer = d_predicted_output.dot(W2.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    # --- ACTUALIZACION DE PESOS (Gradiente Descendente) ---
    W2 += hidden_layer_output.T.dot(d_predicted_output) * learning_rate
    b2 += np.sum(d_predicted_output, axis=0, keepdims=True) * learning_rate
    W1 += X.T.dot(d_hidden_layer) * learning_rate
    b1 += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate
```

Tras la aplicación del *forward pass* y el cálculo de la predicción vemos como empieza la siguiente fase, el *backpropagation*. En el código, la variable `d_predicted_output` ya representa la derivada del error respecto a la salida. A esta variable la llamaremos también **Delta de salida** ($\delta_{out}$). Es básicamente el error ponderado por la pendiente o derivada de la función de activación. Necesitamos ponderar a la pendiente de la función para ver como de grande o pequeño tiene que ser el ajuste de los pesos. Para seguir propagando este error hacia adentro, ahora calculamos el reajuste en la capa intermedia u «oculta». Para actualizar los pesos de la segunda capa ($W_2$), aplicamos la regla de la cadena igualmente: la variación o delta del error en este caso depende del delta de salida anterior y de los pesos de esta capa. Esto lo guardamos en `error_hidden_layer` e igual que antes lo ponderamos a la derivada de la función de activación de esa capa.

Tras el cálculo de estas deltas comenzaría la fase de *Actualización de pesos*. La actualización de pesos en la salida la estamos haciendo en `W2 += hidden_layer_output.T.dot(...)` .... Aquí estamos calculando el gradiente para la matriz de pesos completa. Usamos la transpuesta (`.T`) porque queremos relacionar cada neurona oculta con cada error de salida, creando una matriz de ajustes que coincida con las dimensiones de $W_2$. La actualización de pesos en la entrada sigue la misma lógica pero un paso más atrás. La hacemos en `W1 += X.T.dot(....)`. Aquí, el gradiente depende de la entrada original X y del error que hemos "retropropagado" hasta la capa oculta (`d_hidden_layer`).

Con los sesgos no tenemos una propagación. Cada sesgo se actualiza en base a su propia neurona ya que el sesgo realmente determina qué tan fácil es que la neurona se active, independientemente de la señal de entrada. Para entender bien su actualización pensemos en una ecuación líneal $y = wx + b$. El peso ($w$) controla la pendiente (la inclinación de la línea) pero el bias ($b$) controla la intersección con el eje Y (desplaza la línea hacia arriba o hacia abajo). Si solo ajustáramos los pesos, todas nuestras líneas de decisión tendrían que pasar obligatoriamente por el origen (0,0). El bias permite que la línea se mueva libremente por el espacio para rodear los datos donde sea que estén.

## 2.2. Representación matricial

Vamos a representar las diferentes matrices que más juego tienen en el ejemplo para visualizar mejor todo esto. Partimos de las dos matrices iniciales: la **matriz de datos de entrada** o $X$, donde la primera columna es la entrada $x_1$ y la segunda es $x_2$ y la **matriz de resultados esperados** o $y$:

$$X = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \qquad y = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

### 2.2.1. Forward Pass

Ahora empezamos con las matrices más importantes y relacionadas con el proceso de aprendizaje. Primeramente tenemos la **matriz de pesos de la capa oculta** o $W_1$. Esta matriz conecta las 2 entradas con las 2 neuronas ocultas. Es una matriz de $2 \times 2$. Cada columna representa los pesos que llegan a una neurona oculta específica.

$$W_1 = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix}$$

Tras esta también tenemos la **matriz de Salida de la Capa Oculta** o $A_1$ o `hidden_layer_output`). Es el resultado de multiplicar la entrada $X$ (de $4 \times 2$) por $W_1$. Con esto obtenemos el resultado parcial $Z_1$ y finalizamos aplicando la función de activación (sigmoide). El resultado es una matriz de 4 filas (ejemplos) y 2 columnas (activaciones de las neuronas ocultas). Cada fila $i$ representa cómo "ve" la capa oculta el ejemplo $i$ del XOR.

$$Z_1 = \left( \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \right) + (b_1, b_1)$$

$$A_1 = \sigma(Z_1) = \begin{pmatrix} \sigma(z_{11}) & \sigma(z_{12}) \\ \sigma(z_{21}) & \sigma(z_{22}) \\ \sigma(z_{31}) & \sigma(z_{32}) \\ \sigma(z_{41}) & \sigma(z_{42}) \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \\ a_{4,1} & a_{4,2} \end{pmatrix}$$

Vamos ahora con el siguiente paso del aprendizaje. Ahora igual que antes obtenemos la salida de esta capa aplicando como entrada $A$ (calculada anteriormente) con los pesos $W_2$ y el sesgo apropiado. A esto lo llamamos $Z_2$ y a esto le aplicaremos la función de activación para obtener $A_2$ o lo que es lo mismo, la **matriz de predicción final**:

$$Z_2 = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} \times \begin{pmatrix} w'_1 \\ w'_2 \end{pmatrix} + \begin{pmatrix} b_2 \\ b_2 \\ b_2 \\ b_2 \end{pmatrix} = \begin{pmatrix} (a_{11} \cdot w'_1 + a_{12} \cdot w'_2) + b_2 \\ (a_{21} \cdot w'_1 + a_{22} \cdot w'_2) + b_2 \\ (a_{31} \cdot w'_1 + a_{32} \cdot w'_2) + b_2 \\ (a_{41} \cdot w'_1 + a_{42} \cdot w'_2) + b_2 \end{pmatrix}$$

$$A_2 = \sigma(Z_2) = \begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \hat{y}_4 \end{pmatrix}$$

Cada $\hat{y}_i$ es ahora una combinación no lineal de las activaciones ocultas. Si la capa oculta ha hecho bien su trabajo, habrá separado los puntos de tal manera que para el caso (0,1) y (1,0), los valores de $a_{i1}$ y $a_{i2}$ activen con fuerza a $W_2$, mientras que para (0,0) y (1,1) la combinación resulte en un valor que la sigmoide convierta en casi 0.

### 2.2.2. Backpropagation o propagación del error

Entramos en la parte más profunda del backpropagation de la IA. Aquí es donde el error viaja hacia el pasado. Primeramente calculamos cuánto error hay en la salida final. Este valor también lo llamaremos **delta de salida** y se representa en código con la variable `d_predicted_output`. Es el producto elemento a elemento (Hadamard product, representado por $\odot$) entre el error residual y la derivada de lo que salió. Esto nos devuelve finalmente una matriz $4 \times 1$:

$$\delta_{out} = (y - A_2) \odot \sigma'(Z_2)$$

$$\delta_{out} = \begin{pmatrix} (y_1 - \hat{y}_1) \cdot \sigma'(z_{2,1}) \\ (y_2 - \hat{y}_2) \cdot \sigma'(z_{2,2}) \\ (y_3 - \hat{y}_3) \cdot \sigma'(z_{2,3}) \\ (y_4 - \hat{y}_4) \cdot \sigma'(z_{2,4}) \end{pmatrix} = \begin{pmatrix} \delta_{out,1} \\ \delta_{out,2} \\ \delta_{out,3} \\ \delta_{out,4} \end{pmatrix}$$

Aquí es donde aplicamos la línea: `error_hidden_layer = d_predicted_output.dot(W2.T)`. Como ingeniero, piensa en esto como una proyección inversa: estamos enviando el error de salida de vuelta a través de los mismos cables ($W_2$) por los que vino.

$$\text{Error}_{hidden} = \delta_{out} \times W_2^T$$

$$\text{Error}_{hidden} = \begin{pmatrix} \delta_{out,1} \\ \delta_{out,2} \\ \delta_{out,3} \\ \delta_{out,4} \end{pmatrix} \times \begin{pmatrix} w_1' & w_2' \end{pmatrix} = \begin{pmatrix} \delta_{out,1} \cdot w_1' & \delta_{out,1} \cdot w_2' \\ \delta_{out,2} \cdot w_1' & \delta_{out,2} \cdot w_2' \\ \delta_{out,3} \cdot w_1' & \delta_{out,3} \cdot w_2' \\ \delta_{out,4} \cdot w_1' & \delta_{out,4} \cdot w_2' \end{pmatrix}$$

El resultado es una matriz de $4 \times 2$. Cada columna representa cuánto ruido o error le llegó a cada una de las 2 neuronas ocultas. Finalmente, para saber cuánto debemos cambiar los pesos $W_1$, necesitamos filtrar ese error por la sensibilidad de la activación de la capa oculta. Es decir, multiplicamos por la derivada de la sigmoide de la capa oculta. Esto se representa en el código con la variable `d_hidden_layer`:

$$\delta_{hidden} = \text{Error}_{hidden} \odot \sigma'(Z_1)$$

$$\delta_{hidden} = \begin{pmatrix} \text{err}_{1,1} \cdot \sigma'(z_{1,1}) & \text{err}_{1,2} \cdot \sigma'(z_{1,2}) \\ \text{err}_{2,1} \cdot \sigma'(z_{2,1}) & \text{err}_{2,2} \cdot \sigma'(z_{2,2}) \\ \text{err}_{3,1} \cdot \sigma'(z_{3,1}) & \text{err}_{3,2} \cdot \sigma'(z_{3,2}) \\ \text{err}_{4,1} \cdot \sigma'(z_{4,1}) & \text{err}_{4,2} \cdot \sigma'(z_{4,2}) \end{pmatrix} = \begin{pmatrix} \delta_{h1,1} & \delta_{h2,1} \\ \delta_{h1,2} & \delta_{h2,2} \\ \delta_{h1,3} & \delta_{h2,3} \\ \delta_{h1,4} & \delta_{h2,4} \end{pmatrix}$$

Mientras que `error_hidden_layer` es el error en bruto que le llega a la capa oculta desde la salida, `d_hidden_layer` es el error ya ponderado por la pendiente de la activación. Si una neurona oculta estaba en la zona plana de la sigmoide (valor muy alto o muy bajo), su derivada será casi 0, y por tanto, su `d_hidden_layer` será casi 0. Dicho de otra manera, si la neurona ya estaba muy convencida de su respuesta (activación saturada), no le importa el error que le mandes porque no va a cambiar sus pesos.

### 2.2.3. Actualización de pesos

Para completar el ciclo, vamos a ver cómo ese error que ha viajado hacia atrás se convierte finalmente en instrucciones de ajuste para los pesos. En el código, estas son las líneas de Gradiente Descendente. La variable en el código que vamos a desarrollar ahora sería: `W2 += hidden_layer_output.T.dot(...) * learning_rate.` En ella se actualizan los valores de $W2$. Para ajustar esos pesos (recordamos que són los que conectan la capa oculta con la salida), multiplicamos la activación que salió de la capa oculta ($A_1^T$) y que calculamos en la fase de *Forward Pass* por el delta de salida ($\delta_{out}$) obtenido en la fase de *backpropagation*. En la formulación usaremos $\eta$ para referirnos al *learning rate*. El resultado es una matriz de $[2 \times 1]$, que encaja perfectamente con las dimensiones de $W_2$:

$$\Delta W_2 = \eta \cdot \left( \begin{pmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \end{pmatrix} \times \begin{pmatrix} \delta_{out,1} \\ \delta_{out,2} \\ \delta_{out,3} \\ \delta_{out,4} \end{pmatrix} \right)$$

$$\Delta W_2 = \eta \cdot \begin{pmatrix} \sum(a_{i1} \cdot \delta_{out,i}) \\ \sum(a_{i2} \cdot \delta_{out,i}) \end{pmatrix}$$

Ahora hacemos algo similar con los pesos iniciales o $W_1$. La variable en el código es: `W1 += X.T.dot(d_hidden_layer) * learning_rate`. Aquí es donde resolvemos el misterio del XOR. Usamos la entrada original ($X^T$) y la multiplicamos por el error que hemos calculado para la capa oculta ($\delta_{hidden}$). El resultado es una matriz de $[2 \times 2]$, lista para sumarse a $W_1$:

$$\Delta W_1 = \eta \cdot \left( \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \delta_{h1,1} & \delta_{h2,1} \\ \delta_{h1,2} & \delta_{h2,2} \\ \delta_{h1,3} & \delta_{h2,3} \\ \delta_{h1,4} & \delta_{h2,4} \end{pmatrix} \right)$$

$$\Delta W_1 = \eta \cdot \begin{pmatrix} \sum(x_{i1} \cdot \delta_{h1,i}) & \sum(x_{i1} \cdot \delta_{h2,i}) \\ \sum(x_{i2} \cdot \delta_{h1,i}) & \sum(x_{i2} \cdot \delta_{h2,i}) \end{pmatrix}$$

Es importante fijarse en la belleza de la operación $X^T \cdot \delta_{hidden}$. Si la entrada $x_{11}$ fue 0, no importa cuán grande sea el error $\delta_{h1,1}$, el producto será 0. Esto tiene todo el sentido lógico: si una entrada fue

cero, no pudo haber contribuido al error final de esa neurona, por lo tanto, no hay razón para cambiar el peso que viene de ella.

Para terminar dejamos aquí una relación entre las variables usadas en el código fuente y su simbología matemática utilizada en el desarrollo:

- `X`: ($X$) Matriz de Entrada: Los 4 casos del XOR ($4 \times 2$).
- `y`: ($y$) Etiquetas Reales: Los resultados deseados ($4 \times 1$).
- `W1, W2`: ($W_1, W_2$) Matrices de Pesos: Las conexiones entre capas.
- `b1, b2`: ($b_1, b_2$) Vectores de Sesgo (Bias): El umbral de activación.
- `hidden_layer_output`: ($A_1$) Activación Oculta: La salida de la capa intermedia.
- `predicted_output`: ($A_2$ o $\hat{y}$) Predicción Final: La salida de la red.
- `error`: ($y - A_2$) Error Residual: Diferencia bruta entre realidad y predicción.
- `d_predicted_output`: ($\delta_2$ o Delta 2) Gradiente de Salida: Error de salida multiplicado por la derivada.
- `error_hidden_layer`: ($E_{hidden}$) Error Retropropagado: El error que "rebota" hacia la capa oculta.
- `d_hidden_layer`: ($\delta_1$ o Delta 1) Gradiente Oculto: El error en la capa oculta listo para ajustar $W_1$.
- `learning_rate`: ($\eta$) Tasa de Aprendizaje: El factor de escala de los ajustes.

## 2.3. Estudio del error y del aprendizaje

En la gráfica que se muestra a continuación se ve un estudio de como evoluciona el error en las diferentes iteraciones del aprendizaje con diferentes tasas de aprendizaje $\eta$. Recordamos que para calcular el error durante el aprendizaje usamos en cada iteración la formula del Error Cuadrático Medio aunque realmente lo que propagábamos era su derivada. Con `square()` elevamos cada miembro del array al cuadrado y con `mean()` calculamos la media de todos los valores del vector de errores. Si se usa una tasa muy grande como $\eta = 10$ el modelo "saltará" tan fuerte que nunca caerá en el valle del error. Verás que los resultados oscilan locamente. Por contra si usamos una tasa muy baja el modelo tardará millones de épocas en aprender. Es como intentar vaciar el océano con una cuchara.

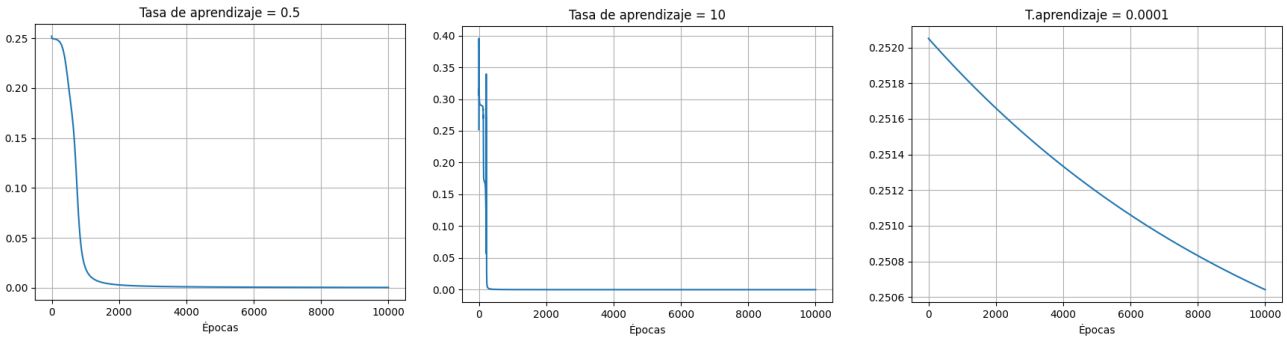


Figura 2: Evolución del error con diferentes tasas de aprendizaje

Usamos una tasa intermedia ($\eta = 0{,}5$). Verás que al principio el error baja muy poco (la red está explorando), luego hay una caída vertical pronunciada (ha encontrado la lógica del XOR) y finalmente se estabiliza cerca de cero. Si el error se queda plano en un valor alto (por ejemplo, 0.25),

significa que la red se ha quedado atrapada en un mínimo local. En el XOR, esto suele significar que la red ha decidido que «la mejor opción es decir siempre 0.5», porque no logra entender la lógica no lineal. En estos casos es posible que necesitemos reiniciar la red debido a la mala asignación de pesos aleatoria al comienzo. Como los pesos iniciales son aleatorios, a veces la red empieza en una posición «desafortunada» del mapa matemático. Se pueden utilizar otras técnicas (como Xavier/Glorot) para realizar una preasignación más eficiente que la simplemente aleatoria.

Si el error colapsa hacia cero alrededor de las 1500 iteraciones (como en el caso de usar $\eta = 0{,}5$) en un problema como el XOR, significa que la tasa de aprendizaje está muy bien balanceada. Que la curva baje muy rápido tampoco es siempre un buen síntoma. Por ejemplo, si cayera a las 10 iteraciones: Sospecharíamos de overfitting (sobreajuste) o de que el problema es demasiado simple para la potencia de la red. A veces también puede ocurrir que una caída rápida se detiene en seco antes de llegar a cero. Eso significa que la red «se ha rendido» en un punto que le parece aceptable pero que no es la solución óptima. Para confirmar que esa tasa de 1500 es perfecta, haz esta prueba: fíjate en el final de la curva. Si es plana y suave al final tu tasa es perfecta. La red ha aterrizado suavemente en el mínimo. Si el final de la curva tiene ruido (pequeños dientes de sierra) tu tasa es un poco alta. La red ha llegado al fondo pero está rebotando un poco en las paredes del valle.