PRESENTACIÓN 4: REDES NEURONALES

Machine Learning II

Jose Alexander Fuentes Montoya Ph.D(c)

REDES NEURONALES

OBJETIVOS

Después de esta presentación, el estudiante será capaz de:

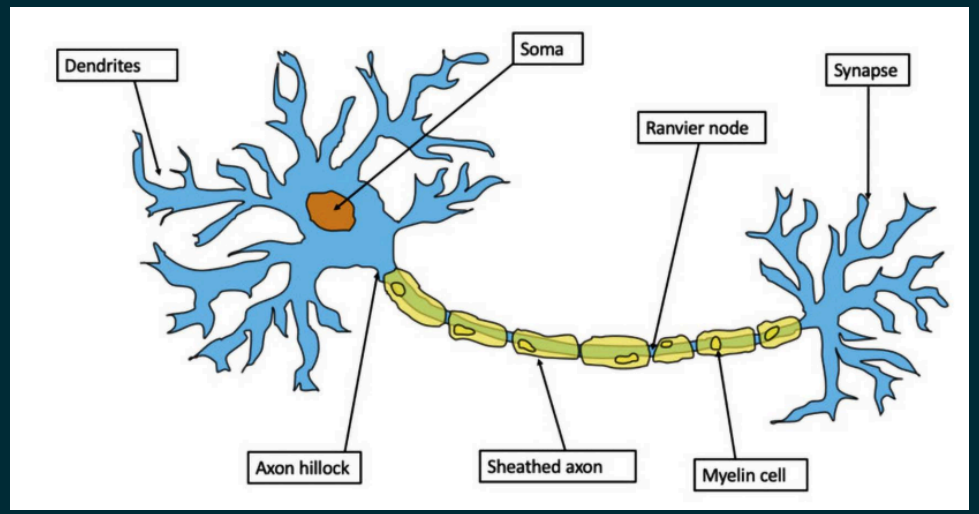
- Comprender el Perceptrón de Capa Única (Single-Layer Perceptron).
- Entender el problema XOR.
- Conocer diversas funciones de activación.
- Explorar la arquitectura de redes profundas y su optimización mediante gradiente descendente.

- Profundizar en el concepto, algoritmo e implementación de Perceptrón Multicapa (Multi-layer Perceptron, MLP).
- Comprender la transformación de espacios mediante activaciones no lineales.
- Analizar el proceso de propagación (forward y backward) en deep learning.

INTRODUCCIÓN

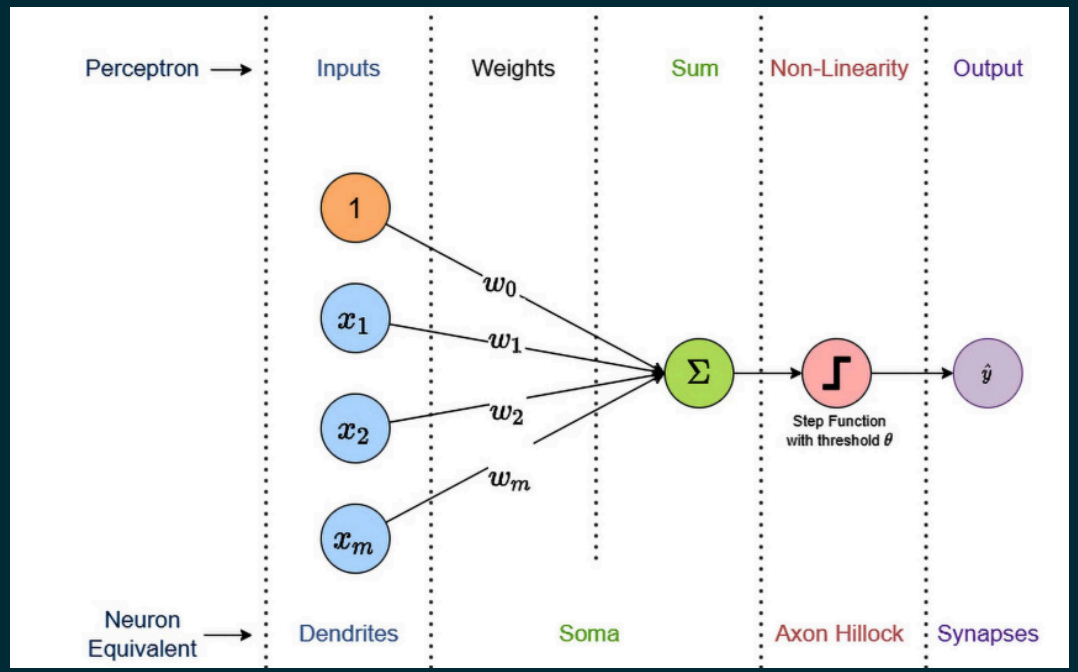
Nuestro cerebro recibe señales a través de las neuronas, las procesa y produce respuestas.
Los receptores envían información a las neuronas, que se comunican mediante sinapsis para coordinar respuestas, tal como describió Cajal [1].
Esta idea inspiró el desarrollo de modelos como el perceptrón, el bloque fundamental en toda red neuronal moderna.

EL ORIGEN BIOLÓGICO Y MATEMÁTICO DEL PERCEPTRÓN



Biológico:

- **Dendritas:** Recogen las señales entrantes.
- **Soma:** Integra la información.
- **Axón y Axón Hillock:** Determinan el disparo de la señal.
- **Sinapsis:** Conectan las neuronas entre sí.



Descripción:

- **Inputs:** Se alimentan valores reales al perceptrón, similar a cómo las dendritas.
- **Suma Ponderada:** Cada entrada se multiplica por un peso que determina su importancia.
- **Función de Activación (No Linealidad):** La suma pasa por una función no lineal (originalmente una función escalón) que decide el disparo del perceptrón.

Matemático:

El perceptrón, introducido inicialmente por McCulloch & Pitts y luego extendido por Rosenblatt, modela este comportamiento de forma simple:

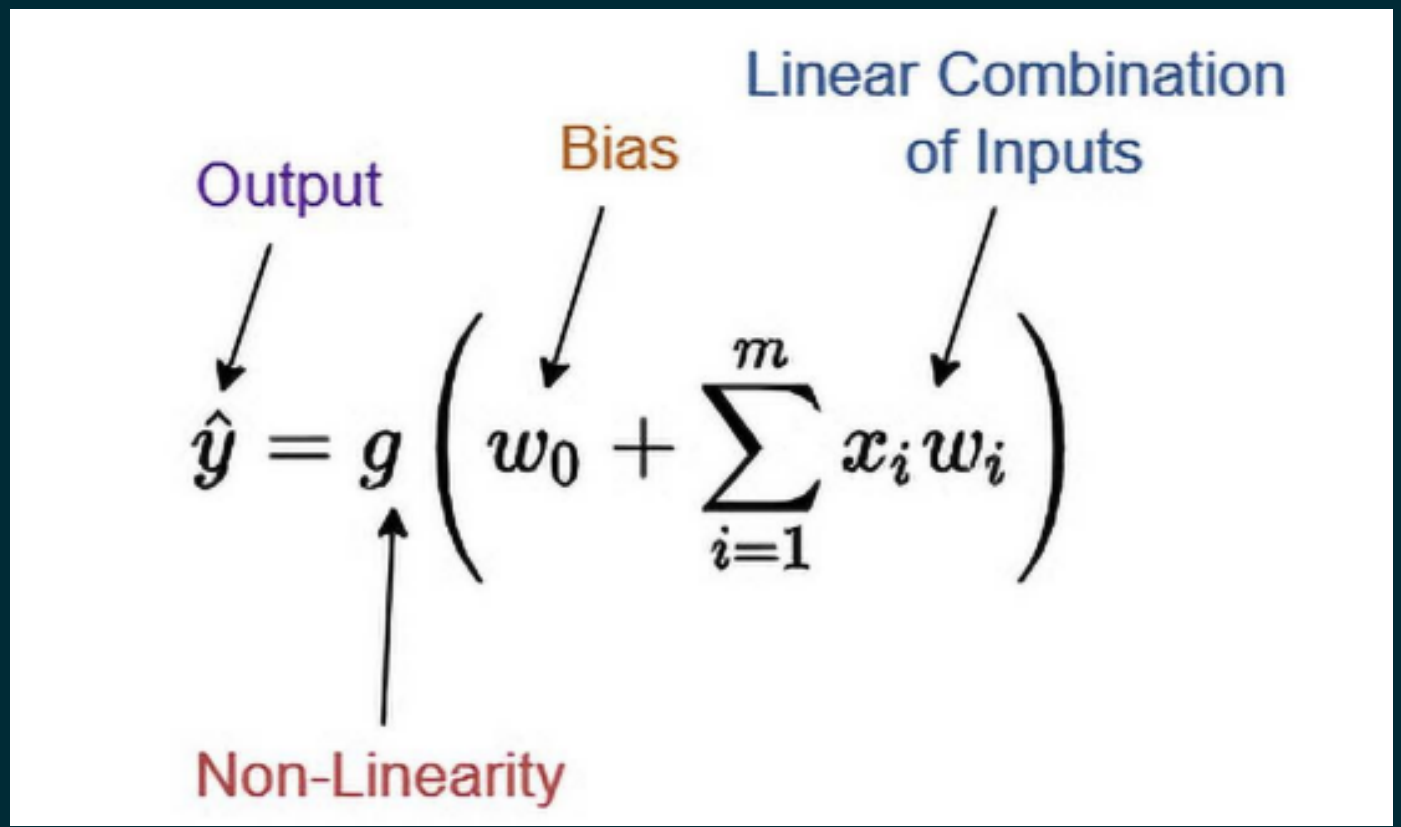


Figura 4.5. Neuro presentación matemática

donde el término incorpora el sesgo (bias).
Esta formulación es la base para aplicar conceptos de álgebra lineal que permiten procesar múltiples entradas de forma eficiente

PERSPECTIVA MATEMÁTICA DEL PERCEPTRÓN

La salida del perceptrón se calcula como una combinación lineal de las entradas, seguida de una función de activación no lineal:

$$U = \mathbf{W}^T \mathbf{X} + b \quad \text{y} \quad y = f(U$$

Ejemplo: Utilizando la función escalón en el perceptrón original, se decide si la salida es 0 o 1 según si supera un umbral. U

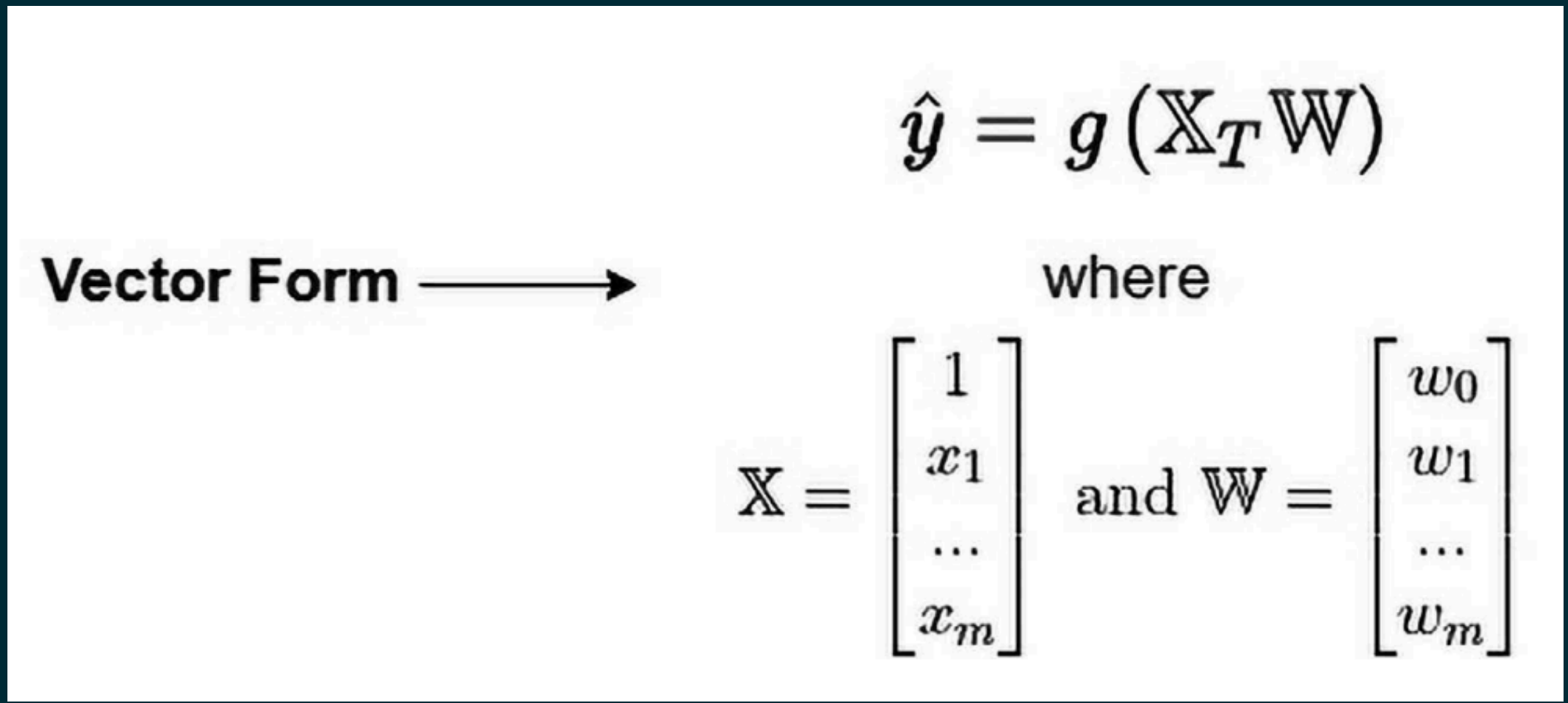


Figura 4.6. forma matricial

INTUICIÓN DE ÁLGEBRA LINEAL EN DEEP LEARNING

Recordemos que:

- Un **vector** es un punto en un espacio n-dimensional.
- Una **matriz** representa una transformación lineal del espacio.

  La transformación de un vector por una matriz se expresa como:

$$\vec{y} = \mathbf{W}\vec{x} + \vec{b}$$

Ver código en GoogleColab Álgebra lineal

PROBLEMA XOR

La tabla clásica XOR:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

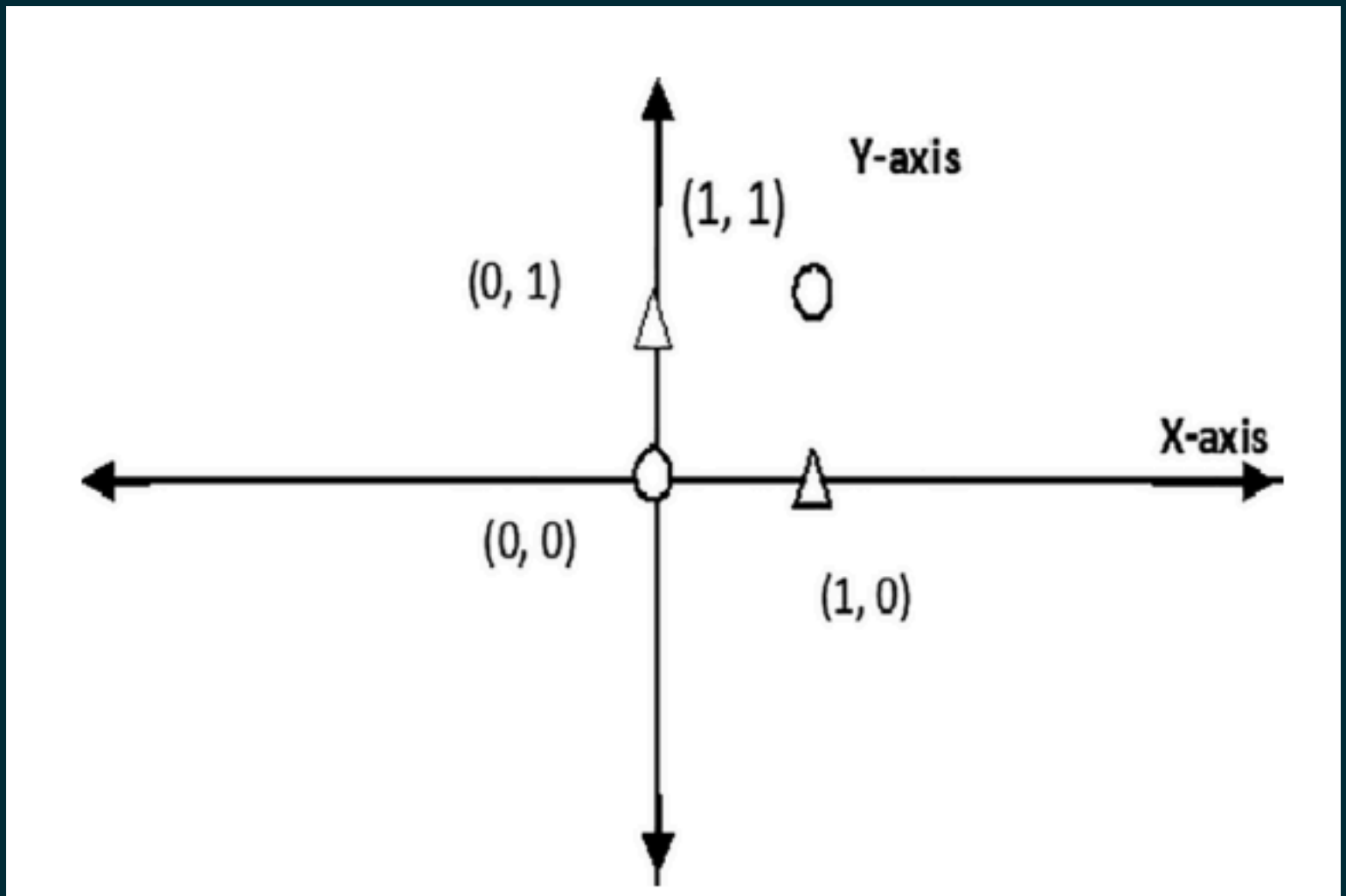


Figura 3-7. Problema XOR

En 2D, **Y=0** (círculo) y **Y=1** (triángulo) no se separan con una línea. El SLP no puede resolverlo, pero el MLP sí lo logra.

FUNCIONES DE ACTIVACIÓN Y TRANSFORMACIÓN DE ESPACIOS

Las funciones de activación introducen la no linealidad necesaria para aprender relaciones complejas. Se describen a continuación:

1. SIGMOIDE

Definición

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

- Salida ((0,1)).
- Ideal para probabilidades.
- Sufre *vanishing gradient* en redes profundas.

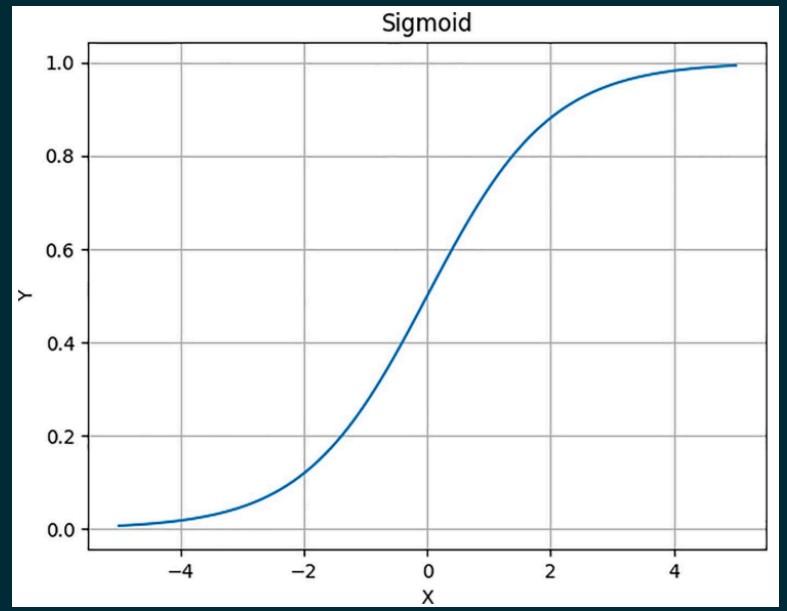


Figura 3-8. Función Sigmoide

Nota de importancia
La función Sigmoide se popularizó en redes neuronales por su **interpretación probabilística** (como si diera la probabilidad de “activación”).

2. TANH

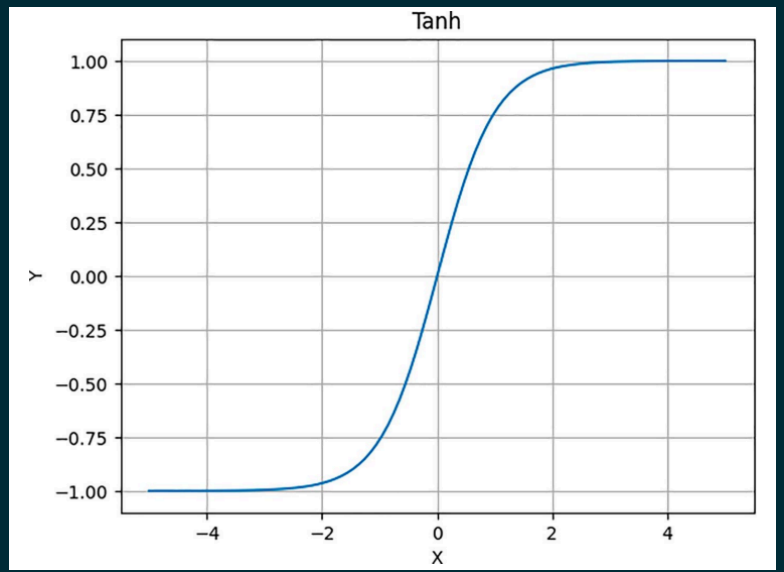


Figura 3-9. Función tanh

Definición

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

- Rango ((-1,1)).
- *Zero-centered*.
- También padece *vanishing gradient*.

Nota de importancia
La función *tanh* se introdujo como **alternativa a la Sigmoide** para que la salida se centre en torno a 0, lo cual en muchos casos facilita el entrenamiento y la convergencia.

3. RELU (RECTIFIED LINEAR UNIT)

4. LEAKY RELU

Definición

$$f(x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$$

- Evita el problema de neuronas muertas (al permitir un gradiente distinto de 0 en la región negativa).
- Mantiene un gradiente continuo para .

$$x < 0$$

- Pierde parte de la “sparsidad” (pues ya no pone en cero todos los valores negativos).

Nota de importancia

La Leaky ReLU se creó para **abordar la desventaja** de la ReLU pura. Al permitir una pequeña pendiente en el dominio negativo, ayuda a reducir la posibilidad de neuronas muertas.

α

TABLA COMPARATIVA DE FUNCIONES DE ACTIVACIÓN

Función	Ecuación	Rango	Ventajas / Desventajas
Sigmoide	$f(x) = \frac{1}{1+e^{-x}}$	(0, 1)	Interpretación probabilística; sufre saturación y vanishing gradient.
tanh	$f(x) = \tanh(x)$	(-1, 1)	Zero-centered; parecido a sigmoide, pero también saturable.
ReLU	$f(x) = \max(0, x)$	[0, ∞)	Eficiente, no satura en la región positiva; puede generar "neurona muerta".
Leaky ReLU	$f(x) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}$	$\mathbb{R}$	Evita neuronas muertas; pierde parte de la "sparsidad".
SoftPlus	$f(x) = \ln(1 + e^x)$	(0, ∞)	Versión "suave" de ReLU; no tiene discontinuidad en ; puede saturarse para valores muy negativos. $x = 0$
GELU	, donde es la CDF de la Normal estándar $f(x) = x\Phi(x)$ $\Phi(x)$	$\mathbb{R}$	Similar a ReLU, pero introduce un componente probabilístico; transición más suave alrededor de . $x = 0$

Ver código en GoogleColab Funciones de activación

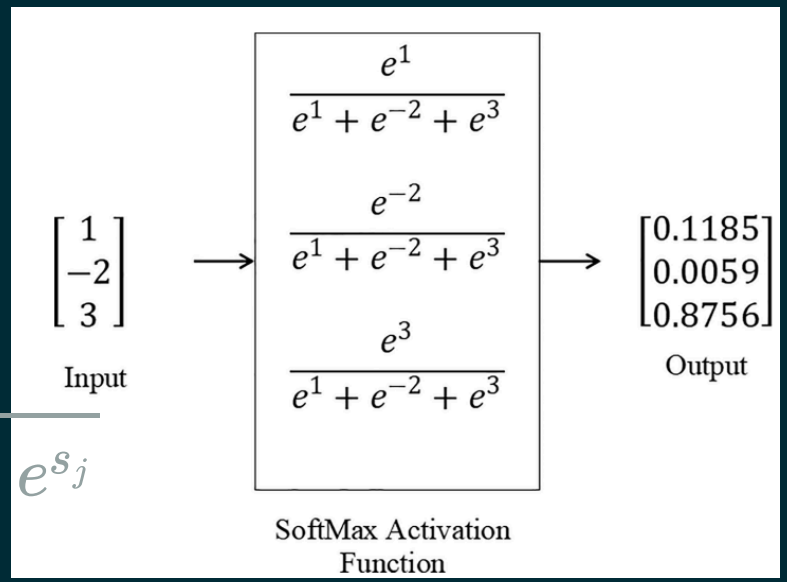
FUNCIONES DE SALIDA: SOFTMAX

FUNCIONES DE SALIDA PARA CLASIFICACIÓN MULTICLASE

- **Uso principal:** Clasificación multiclase (K clases).
- **Definición:**

$$\text{softmax}(s_i) = \frac{e^{s_i}}{\sum_{j=1}^{K} e^{s_j}}$$

- **Interpretación:**
  - Transforma un vector de puntajes $(s_1, s_2, \ldots, s_K)$ en probabilidades que **suman 1**.
  - Resalta la clase de mayor puntaje y "suprime" las restantes, haciendo que una de las probabilidades destaque.



Adicional

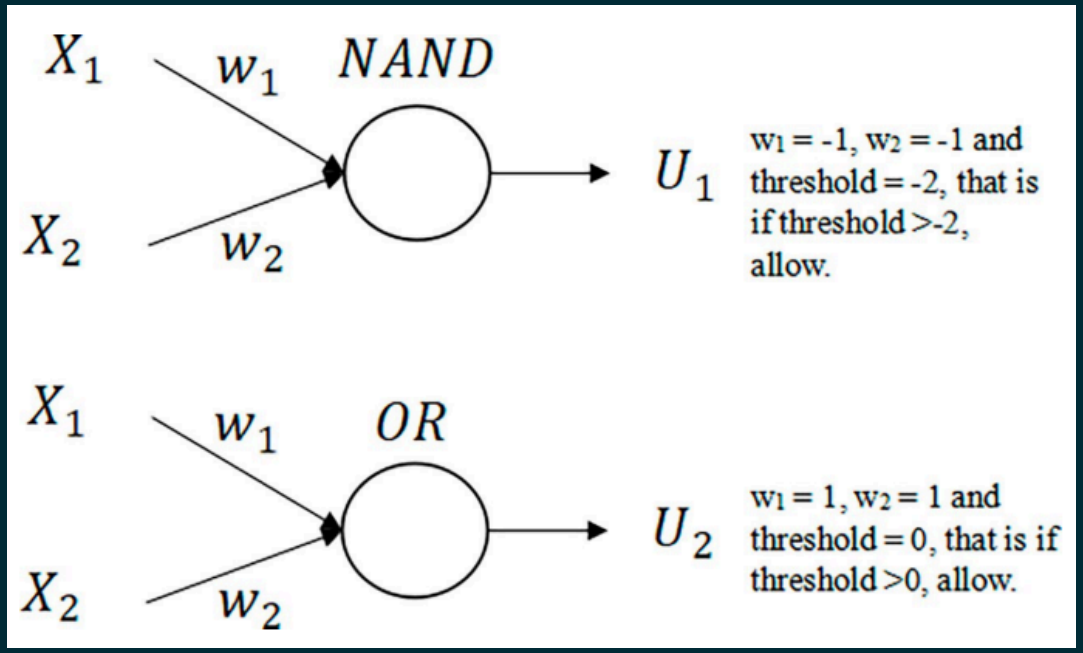
Las probabilidades suman 1 para que haya una distribución de probabilidad coherente sobre las K clases posibles.

MULTI-LAYER PERCEPTRON (MLP)

El Multi-layer Perceptron crea efectivamente una representación de características jerárquica y puede manejar datos no linealmente separables. Empecemos resolviendo el problema XOR usando un MLP.

RESOLVIENDO EL PROBLEMA XOR (I)

Consideremos una compuerta XOR. Ya hemos visto que no puede implementarse con un SLP. También se puede crear una compuerta NAND de la misma manera que una compuerta AND con entradas negadas.



Ahora, consideremos una red con dos entradas X_1 y X_2. Es sencillo crear un SLP para la implementación de las compuertas NAND y OR, tal como se muestra en la Figura.

RESOLVIENDO EL PROBLEMA XOR (II)

ARQUITECTURA DE MLP Y FORWARD PASS (I)

Architecture of MLP and Forward Pass

Un Multi-layer Perceptron tiene una capa de entrada, una de salida y al menos una capa oculta.

Cálculo en la capa oculta

Sea $X_1, X_2, \ldots, X_n$ la entrada y W_{ij} los pesos entre la primera y la segunda capa. Para la neurona p en la capa oculta:

$$U_p = \sum_i X_i W_{ip} + b_p$$

La salida de esa neurona:

$$V_p = f(U_p)$$

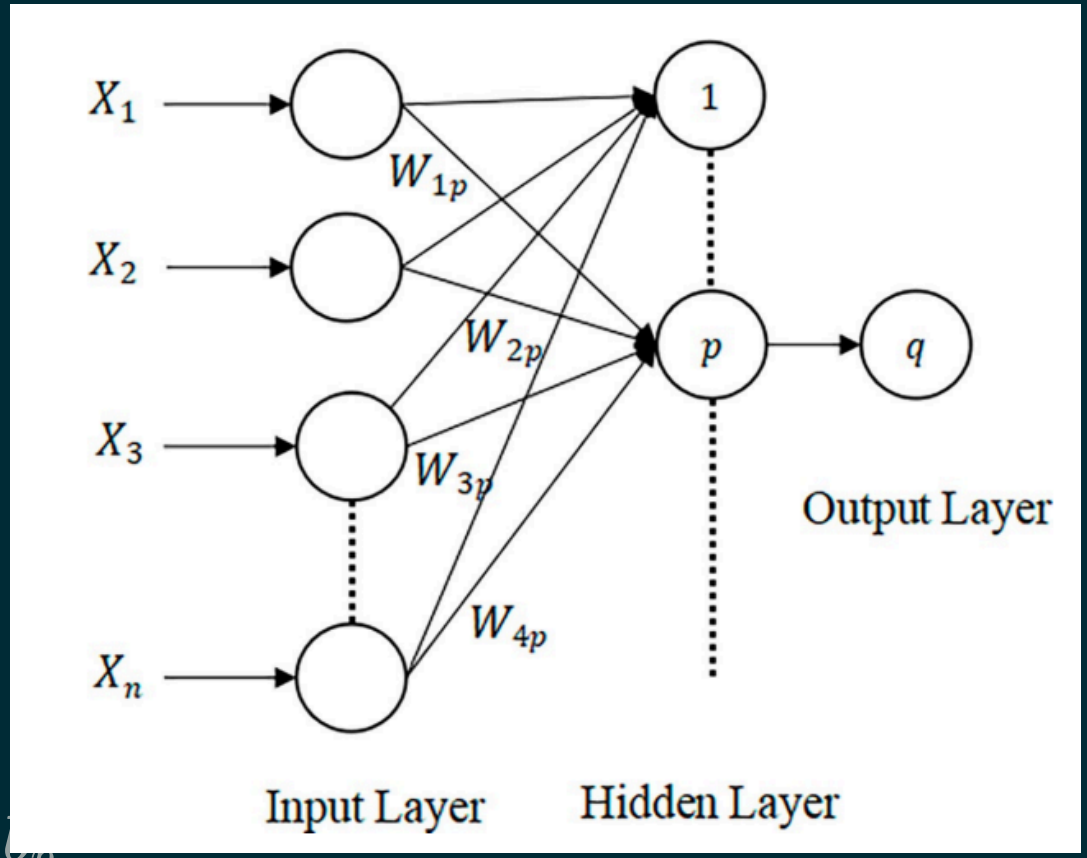


Figura 4.10. Placeholder: MLP con n entradas y 1 salida

ARQUITECTURA DE MLP Y FORWARD PASS

Cálculo en la capa de salida

Sea (q) la neurona en la capa de salida:

$$U_q = \sum_p V_p W_{pq} + b_q$$

$$V_q = f(U_q)$$

En cada capa, calculamos la salida para usarla como entrada de la siguiente capa. Este tipo de red se denomina **Feed-Forward Network**.

Como ejemplo, consideremos la clasificación de un subconjunto del **IRIS dataset** (dos clases: Setosa y Versicolor, cuatro features). La red tiene cuatro neuronas de entrada, dos en la capa oculta y una en la de salida.

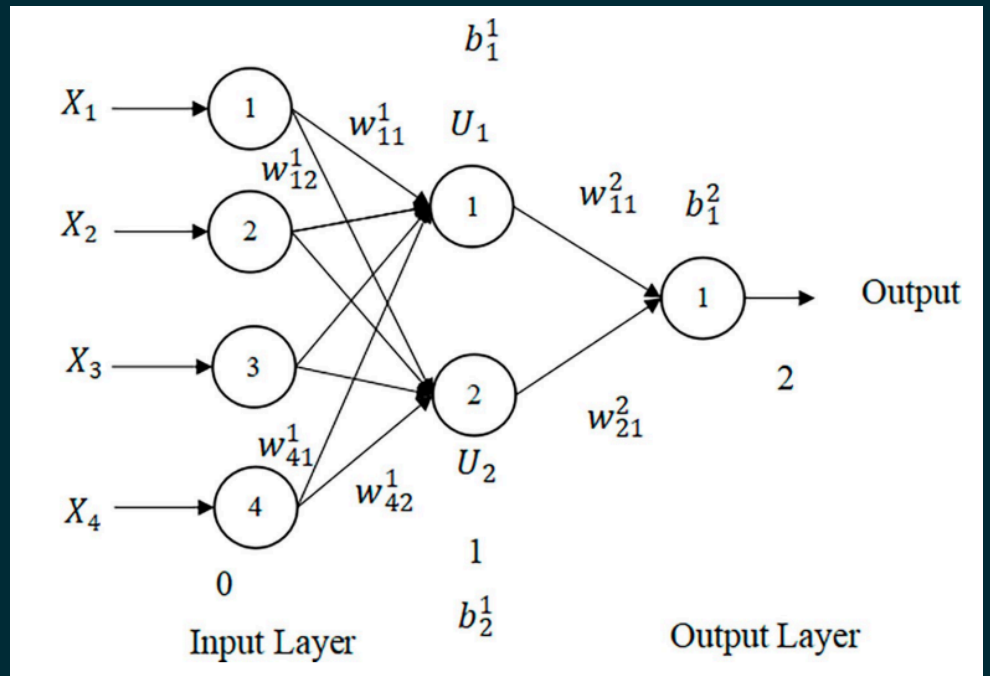


Figura 4.11. Placeholder: MLP con 4 entradas, 2 oculta, 1 salida

- **Forward Pass**
  Se asume que los pesos y bias están inicializados. Para la primera capa oculta:

Feed-Forward

$$U_1 = \sum_i X_i w_{i1}^1 + b_1^1$$

$$V_1 = f\left(\sum_i X_i w_{i1}^1 + b_1^1\right) = f\left(\left(X_1 \times w_{11}^1\right) + \left(X_2 \times w_{21}^1\right) + \left(X_3 \times w_{31}^1\right) + \left(X_4 \times w_{41}^1\right) + b_1^1\right)$$

$$U_2 = \sum_i X_i w_{i2}^1 + b_2^1$$

$$V_2 = f\left(\sum_i X_i w_{i2}^1 + b_2^1\right) = f\left(\left(X_1 \times w_{12}^1\right) + \left(X_2 \times w_{22}^1\right) + \left(X_3 \times w_{32}^1\right) + \left(X_4 \times w_{42}^1\right) + b_2^1\right)$$

La salida de la neurona final:

$$O = f\left(V_1\, w_1^2 + V_2\, w_2^2 + b\right)$$

CÁLCULO DE LA SALIDA (EJEMPLO IRIS)

Comparación y Error

Una vez que se calcula la salida , se compara con la salida deseada (o , etc.). Por ejemplo, si usamos error cuadrático:

$\hat{y}$ y_i O

$$L = \frac{1}{2}(\hat{y} - O)^2.$$

Idea: Minimizar . Para ello, es necesario ajustar los pesos mediante un método de optimización, típicamente **Gradiente Descendiente**.

L

MODULARIDAD EN UN SISTEMA DE DEEP LEARNING

Según la perspectiva presentada en el complemento, un sistema de deep learning se compone de:

- **Bloques de representación:** Secuencias de transformaciones lineales y no lineales que extraen características relevantes.
- **Clasificador lineal:** Una última capa que asigna la etiqueta de clase basándose en las representaciones aprendidas.

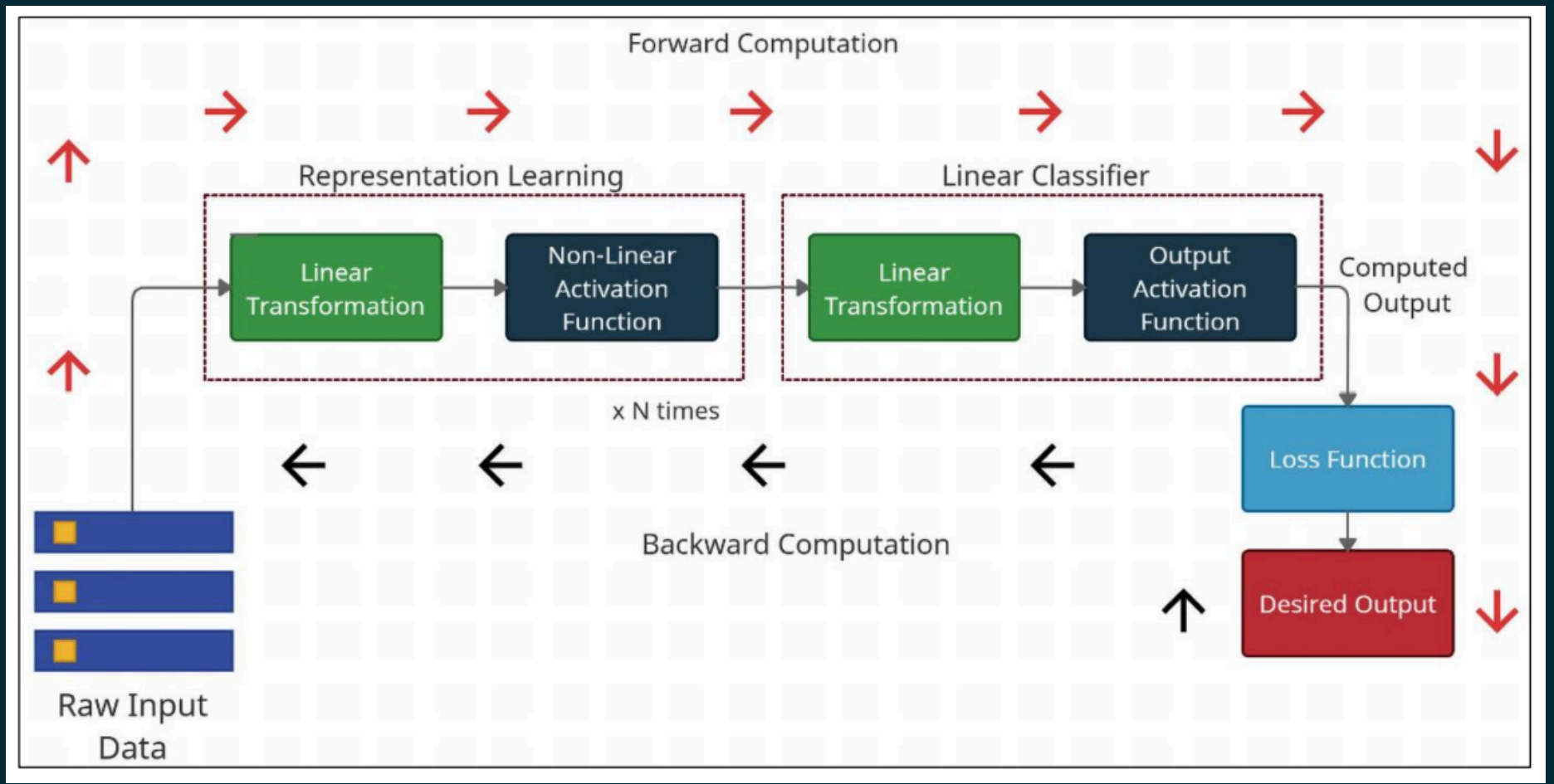


Figura 4.7. Sistema de Deep Learning

> Los módulos se ensamblan en grafos y se optimizan mediante métodos basados en gradiente

OPTIMIZACIÓN: PROPAGACIÓN Y GRADIENTE DESCENDENTE

GRADIENTE DESCENDIENTE

En un pipeline de Machine Learning, se suelen:

1. Preprocesar datos.
2. Extraer y seleccionar características.
3. Hacer predicciones con un modelo.
4. Diseñar una función de pérdida para medir la diferencia entre predicción y valor real.

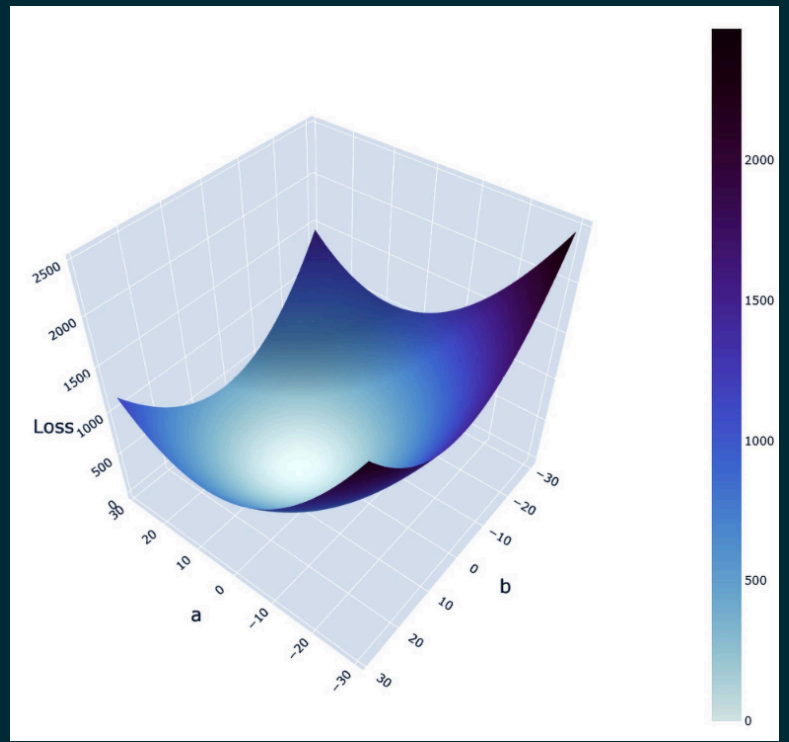


Figura 4.7. gráfico en 3D ilustra cómo se va minimizando la función de pérdida a lo largo de dos parámetros

El proceso de aprendizaje de una red neuronal se divide en dos fases:

1. **Forward Propagation:**
   Se calculan las salidas pasando la entrada a través de todas las capas.
2. **Backward Propagation:**
   Se calcula el gradiente del error para ajustar los parámetros mediante técnicas de optimización **gradiente descendente**.

La regla de actualización de parámetros es:

$$\theta := \theta - \eta \nabla_\theta \mathcal{L}(\theta)$$

donde:

- θ representa los parámetros (pesos y bias).
- η es la tasa de aprendizaje.
- $\nabla_\theta \mathcal{L}(\theta)$ es el gradiente del error.

Consulta el notebook 03-Gradient_Descent.ipynb para un ejemplo interactivo.

DESCENSO POR GRADIENTE

1. **Valor predicho**

$$\hat{y} = f(\mathbf{W}^T\mathbf{X} + b)$$

2. **Función de pérdida** (mitad del error cuadrático)

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial\left(1/2(\hat{y}_i - y_i)^2\right)}{\partial \mathbf{W}}$$

3. **Derivada de la pérdida respecto a (W)**

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial \left(1/2(f(\mathbf{W}^T\mathbf{X} + b) - y_i)^2\right)}{\partial \mathbf{W}}$$

4. **Derivada de la pérdida respecto a (b)**

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\left(f(\mathbf{W}^T\mathbf{X} + b) - y_i\right)\right) \times f(1 -$$

Actualización de los pesos y bias

$$\mathbf{W}_{\text{nuevo}} = \mathbf{W}_{\text{viejo}} - \alpha \, \frac{\partial L}{\partial \mathbf{W}},$$

Comentario:

$$b_{\text{nuevo}} = b_{\text{viejo}} - \alpha(\hat{y}_i - y_i) f(1-f).$$

- α = tasa de aprendizaje
- $\mathbf{X}$ vector de entrada

BACKPROPAGATION (I)

BACKPROPAGATION Y AJUSTE DE PESOS

En redes neuronales multicapa (MLP), el **backpropagation** es el método que ajusta los pesos en **todas** las capas, propagando el error desde la salida hasta la capa de entrada.

EXPLICACIÓN CONCEPTUAL DE BACKPROPAGATION

Backpropagation es el proceso para actualizar los pesos en cada capa de una MLP:

1. **Cálculo de Error** en la capa de salida.
2. **Propagación** de ese error hacia capas anteriores.
3. **Regla de Actualización** de los pesos mediante descenso por gradiente.

   La lógica se basa en la **regla de la cadena (chain rule)**:

- Cada neurona recibe parte de la “responsabilidad” del error total.
- Las capas ocultas se actualizan después de que la capa de salida ya calculó su contribución al error.

ALGORITMO DE BACKPROPAGATION

El **algoritmo en 7 pasos**. En resumen:

1. **Inicializar** los pesos y bias con valores pequeños aleatorios.
2. **Forward Pass**: calcular sumas ponderadas y activaciones hasta la salida.
3. **Calcular el Error** (p. ej., error cuadrático).
4. **Calcular gradiente** en la capa de salida.
5. **Actualizar** los pesos de la última capa.
6. **Propagar** el error a la(s) capa(s) oculta(s) y actualizar sus pesos.
7. **Repetir** (Forward + Backward) hasta convergencia o hasta un máximo de etapas.

A continuación detallamos cada paso con sus ecuaciones.

PASOS 1-2

PASO 1: INICIALIZACIÓN

- Se asignan valores aleatorios pequeños a W_{ij} y a los bias b_j de todas las capas.

PASO 2: FORWARD PASS

- Para cada capa k, calculamos la suma ponderada:

$$U_j^{(k)} = \sum_i \left(\mathbf{X}_i W_{ij}^{(k)} \right) + b_j^{(k)}$$

y luego la salida de cada neurona:

$$O_j^{(k)} = f\left(U_j^{(k)}\right).$$

PASOS 3 AL 4

PASO 3: CALCULAR LA FUNCIÓN DE ERROR

$$L = \frac{1}{2}\left(\hat{y}_i - y_i\right)^2.$$

(La constante $1/2$ facilita la derivada.)

PASO 4: CALCULAR EL GRADIENTE EN LA CAPA DE SALIDA

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial\left(1/2(\hat{y}_i - y_i)^2\right)}{\partial \mathbf{W}}.$$

PASOS 5 AL 6

PASO 5: ACTUALIZAR LOS PESOS DE LA ÚLTIMA CAPA

- La regla de descenso por gradiente:

$$\mathbf{W}_{\text{nuevo}} = \mathbf{W}_{\text{viejo}} - \alpha \frac{\partial L}{\partial \mathbf{W}}$$

PASO 6: ACTUALIZAR LOS PESOS DE LA(S) CAPA(S) OCULTA(S) (BACKPROPAGATION)

1. Para cada neurona p en la capa $(k-1)$:

$$W_{ij}^{(k)} = W_{ij}^{(k)} - \eta\, \delta_i^{(k)}\, O_j^{(k-1)},$$

donde η es la tasa de aprendizaje y $O_i^{(k-1)}$ la salida de la neurona i en la capa anterior.

2. Actualizar pesos:

$$= O_i^k(1 - O_i^k) \sum_{j=1}^{M_{k+1}} \partial_j^{(k+1)} W_{ij}^{(k+1)}$$

PASO 7: REPETIR

$$O_j^{(k+1)}(1 - O_j^{(k+1)})(O_j^{(k+1)} - y_j).$$

- Se repite **forward pass + backward pass** por múltiples etapas, hasta convergencia o hasta el número máximo de iteraciones.

PRESENTACIÓN DEL TALLER SEMANA 2

TALLER PRÁCTICO: ENTRENANDO MLP CON SCIKIT-LEARN Y KERAS

En este taller cubriremos:

1. **Clasificación binaria** usando **Wine Dataset** (scikit-learn).
2. **Clasificación binaria** con **Breast Cancer Dataset** (Keras).
3. Análisis de **número de neuronas**, **capas ocultas** y **tasa de aprendizaje**.

Al final, tendremos una **comprensión práctica** de cómo configurar **Multi-Layer Perceptrons** y cómo los cambios en la arquitectura impactan la precisión y la pérdida.

EJERCICIO 1 - WINE DATASET (SCIKIT-LEARN)

Objetivo: Clasificar las dos primeras clases del dataset Wine (13 características, 130 muestras), y comparar dos MLP:

- Uno con **parámetros por defecto** (100 neuronas en la capa oculta).
- Otro con solo **3 neuronas** en la capa oculta.

PASOS GENERALES

1. **Cargar** el dataset y filtrar las clases a usar.
2. **Separar** datos en entrenamiento y prueba (`train_test_split`).
3. **Definir** y **entrenar** dos modelos MLP (uno por defecto, otro con 3 neuronas).
4. **Comparar** precisión en el set de prueba.

Ver código en GoogleColab Ejercicio 4_1

CONCLUSIONES EJERCICIO 1

1. **Más neuronas** → mayor capacidad de ajuste → tiende a mejor desempeño (aunque no siempre).
2. Un número muy bajo de neuronas puede **subajustar** (underfitting).
3. Elección del tamaño de la capa oculta = **hiperparámetro** que depende del dataset y se suele ajustar de forma **empírica**.

TAREA SUGERIDA:

- Prueba distintos tamaños de capa oculta: (10,), (50,), (200,).
- Observa cómo varía la precisión de prueba y el tiempo de entrenamiento.

EJERCICIO 2 - BREAST CANCER (KERAS)

Objetivo: Utilizar la librería **Keras** para crear un MLP que clasifique el dataset de **Cáncer de Mama** (Breast Cancer).
- Modelo 1: 1 capa oculta (16 neuronas).
- Modelo 2: 2 capas ocultas (16 y 8 neuronas).

Además, **experimentar** con optimizadores (SGD, RMSprop, Adam) y tasas de aprendizaje.

PASOS GENERALES

1. Cargar `load_breast_cancer` (569 muestras, 30 features).
2. Separar en entrenamiento y prueba (`train_test_split`).
3. Definir un **Sequential** con `Dense(...)`.
4. Compilar el modelo (`model.compile(...)`) indicando el optimizador, la pérdida (`binary_crossentropy`) y la métrica (`accuracy`).
5. Entrenar (`model.fit(...)`) y evaluar (`model.evaluate(...)`).

CÓDIGO EJERCICIO 2 (KERAS, 1 CAPA OCULTA)

Ver código en GoogleColab Ejercicio 4_2

VISUALIZACIÓN DE RESULTADOS

Generalmente, se grafican:

- **Training vs. Validation Loss**
- **Training vs. Validation Accuracy**

Podemos ver si el modelo **sobreeentrena** (training accuracy sube y validation no mejora) o si converge adecuadamente.

MÚLTIPLES CAPAS OCULTAS Y OPTIMIZACIÓN

Modelo 2: 2 capas ocultas (16 y 8 neuronas).

- **Ver código en GoogleColab Ejercicio 4_3**
- **Ver código en GoogleColab Ejercicio 4_4**
- **Ver código en GoogleColab Ejercicio 4_5**
- **Ver código en GoogleColab Ejercicio 4_6**

Cambiar optimizador / LR (ej. Adam con lr=0.001):

Observa la variación de desempeño con distintas capas y optimizadores.

TAREA FINAL

1. **Experimentar con más etapas**: aumenta o reduce `epochs` (por ejemplo, 100, 200).
2. **Cambiar la función de activación**: `'relu'` en lugar de `'sigmoid'` entre otras.
3. **Variar el número de capas**:
    - 1 capa oculta (p.ej. 50 neuronas).
    - 2 capas ocultas (p.ej. 25 y 12).
4. **Monitorear** la pérdida y precisión en cada experimento.
5. **Comparar**: ¿Cuál configuración converge más rápido? ¿Cuál obtiene mejor exactitud?

Al final, **documenta** tus hallazgos: - ¿Qué impacto tuvo la tasa de aprendizaje?
- ¿Cambió el problema de *overfitting*?

¡Éxitos en tu exploración con MLPs!

CONCLUSIONES Y DEBATE

- Hemos explorado desde la base del perceptrón y su inspiración biológica hasta la complejidad de sistemas de deep learning.
- Se ha destacado la importancia de las transformaciones no lineales para lograr la separación de datos complejos.
- La optimización mediante propagación hacia atrás y gradiente descendente es fundamental para ajustar los parámetros.
- **Número de neuronas** y **capas ocultas** son hiperparámetros cruciales.
- **Tasa de aprendizaje** y **optimizador** pueden alterar la velocidad y estabilidad de convergencia.
- Para problemas reales, conviene **probar múltiples configuraciones** (grid search o random search), con validación cruzada.

REFERENCIAS

1. Definición informal de ML.
2. Definición de IA.
3. Definición formal de ML (Tom Mitchell).
4. Minsky, M. & Papert, S. (1969). *Perceptrons*. MIT Press.
5. Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). *Learning representations by back-propagating errors*. Nature, 323(6088), 533–536.
6. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). *Gradient-based learning applied to document recognition*. Proceedings of the IEEE, 86(11), 2278–2324.
7. Hubel, D. H., & Wiesel, T. N. (1962). *Receptive fields, binocular interaction and functional architecture in the cat's visual cortex*. Journal of Physiology, 160, 106–154.
8. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). *ImageNet classification with deep convolutional neural networks*. NeurIPS.
9. He, K., Zhang, X., Ren, S., & Sun, J. (2016). *Deep residual learning for image recognition*. CVPR.