

Más allá de PyTorch: construyendo un motor de aprendizaje automático con NumPy y CuPy

Mitchell Mirano

Facultad de Ciencias Matemáticas – UNMSM
Área de Computación Científica

12 de noviembre de 2025
Semana de Computación Científica

Contenido

- 1 Introducción
- 2 Fundamentos de Machine Learning (ML)
- 3 Fundamentos de redes neuronales
- 4 Funciones de Activación
- 5 Capas de una Red Neuronal
- 6 Introducción a sorix
- 7 Autograd
- 8 Ejecución en GPU
- 9 Machine Learning con Sorix

- **Necesidad de Control y Explicabilidad Profunda**
 - Las arquitecturas de “caja negra” limitan la trazabilidad y el **control total sobre el flujo computacional**.
 - La transparencia es esencial para una **explicabilidad rigurosa** de las predicciones y los gradientes.
- **Ineficiencia y Costos Operacionales**
 - Los *frameworks* monolíticos (TensorFlow, PyTorch) suelen ser sobredimensionados para problemas específicos.
 - Producen **sobreconsumo de memoria y recursos**, elevando los costos en despliegues de producción.
- **Necesidad de Flexibilidad e Innovación**
 - Los sistemas existentes restringen la **experimentación con nuevas funciones de activación, gradientes o arquitecturas**.
 - Un **framework propio** permite adaptar la estructura matemática y computacional al objetivo de investigación o producción.

Aprendizaje Automático (ML)

El Machine Learning es una rama de la Inteligencia Artificial que permite a los sistemas **aprender de datos**, identificar patrones y tomar decisiones con **mínima o nula intervención humana explícita**.

- Se basa en modelos matemáticos capaces de ajustar sus parámetros internos a medida que se exponen a nueva información.
- El objetivo es la **generalización**: obtener un rendimiento útil en datos no vistos previamente.
- Los algoritmos de aprendizaje automático se dividen en los siguiente paradigmas: **aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo**.

Aprendizaje Supervisado: Regresión

Definición

Objetivo: Predecir una **variable continua**, es decir, un número real dentro de un rango.

Aplicaciones Comunes

- **Finanzas:** Estimación de precios de vivienda o acciones.
- **Medio Ambiente:** Predicción de la temperatura o niveles de polución.
- **Comercio:** Proyecciones detalladas de ventas futuras.

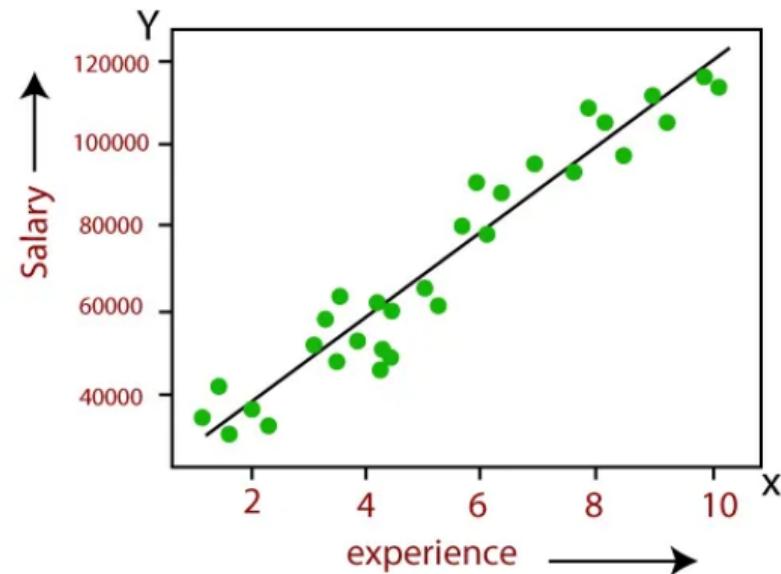


Figure: Modelo prediciendo valores continuos.

Aprendizaje Supervisado: Clasificación

Definición

Objetivo: Asignar una instancia a una **categoría o clase discreta** predefinida.

Aplicaciones Comunes

- **Seguridad:** Detección de *spam* (Clase A) vs. *No-spam* (Clase B).
- **Medicina:** Diagnóstico binario (enfermo/sano).
- **Visión:** Reconocimiento de dígitos o de objetos en imágenes.

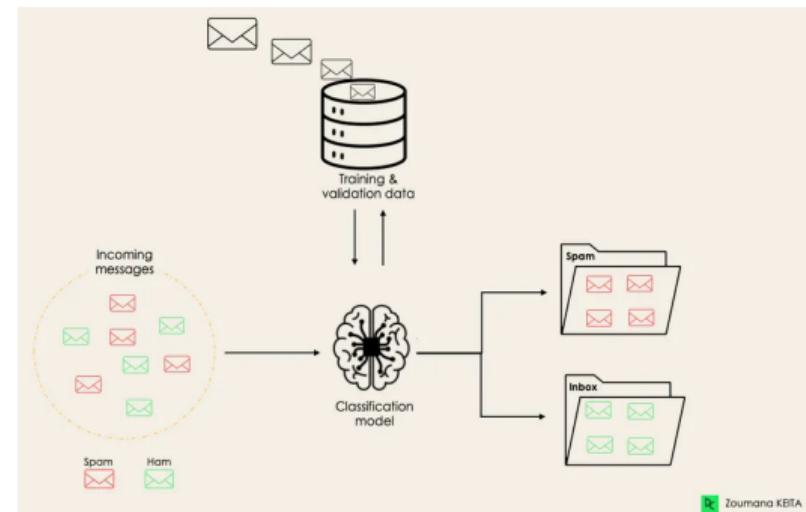


Figure: Separación de datos en clases discretas.

Aprendizaje No Supervisado: Clustering (Agrupamiento)

Definición

Objetivo: Identificar **grupos o estructuras naturales** en los datos. Es un problema de Aprendizaje **No Supervisado**.

Aplicaciones Comunes

- **Marketing:** Segmentación de clientes basada en comportamiento de compra.
- **Investigación:** Agrupamiento automático de documentos o artículos científicos.
- **Análisis de Datos:** Detección de anomalías o *outliers*.

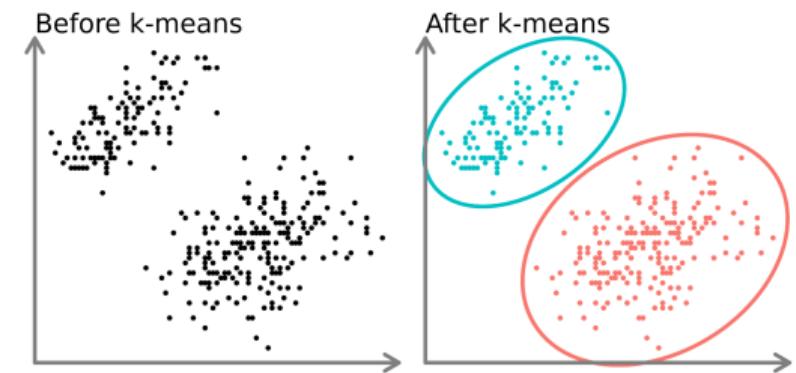


Figure: Identificación de estructuras sin etiquetas previas.

Aprendizaje por Refuerzo (RL)

Definición

Objetivo: Entrenar un **agente** para tomar decisiones secuenciales en un entorno, maximizando una **recompensa** acumulada.

Aplicaciones Comunes

- **Robótica:** Control y navegación autónoma.
- **Finanzas:** Optimización de carteras de inversión.
- **Juegos:** Desarrollo de IA que supera a humanos (e.g., AlphaGo).

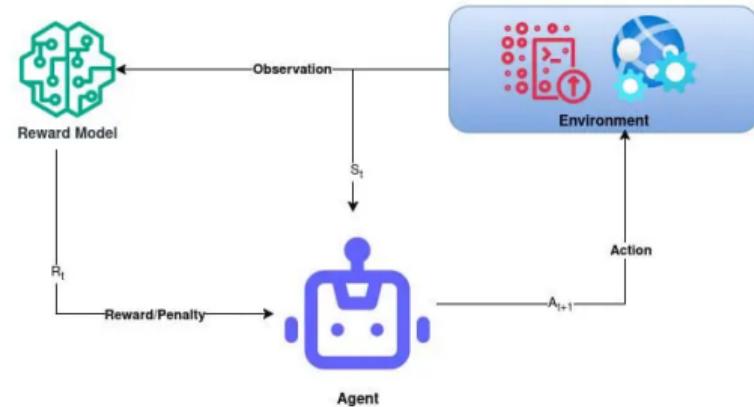


Figure: Diagrama del ciclo de interacción en RL.

Fundamentos de Redes Neuronales (NN)

Definición de Red Neuronal

Un modelo computacional que imita la estructura del cerebro, compuesto por capas de unidades simples (**neuronas artificiales**) interconectadas para aprender patrones complejos.

Componentes Clave

- **Entradas:** Datos iniciales (*features*).
- **Capas Ocultas:** Donde ocurre la transformación de los datos y la extracción de características.
- **Salida:** La predicción o resultado final del modelo.

Arquitectura y Flujo de Datos

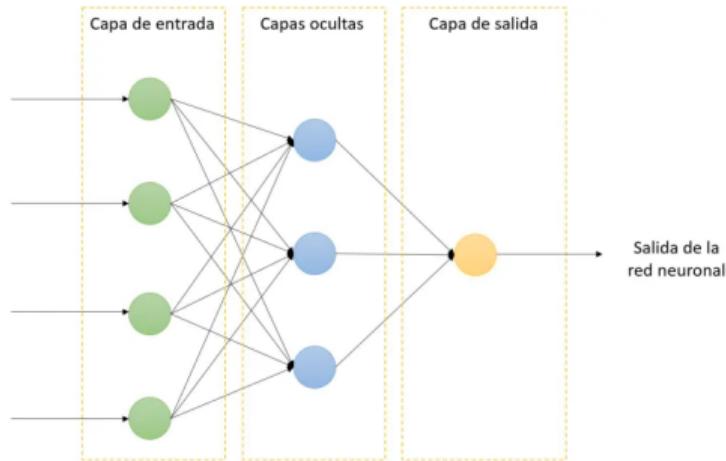


Figure: Estructura de una Red Neuronal básica.

Ubicación de las Redes Neuronales en el Campo de la IA

La Jerarquía de la Inteligencia Artificial

1. Inteligencia Artificial (IA)

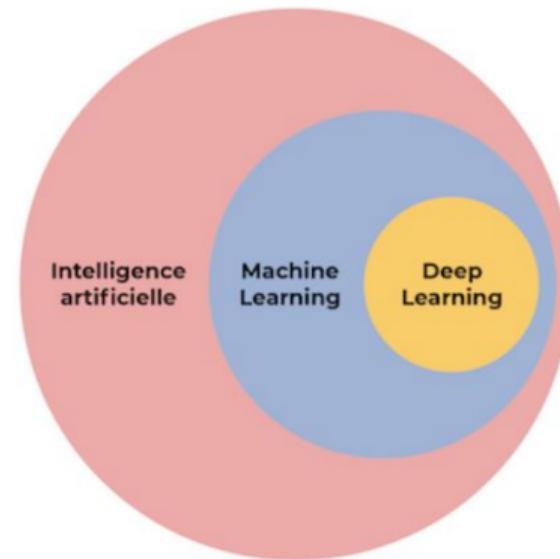
- Es el campo más amplio. Su meta es crear sistemas que imiten la inteligencia humana para realizar tareas.
- Ejemplos: Planificación, resolución de problemas, y razonamiento.

2. Machine Learning (ML)

- Se centra en el desarrollo de algoritmos que permiten a las computadoras **aprender de los datos** sin programación explícita.

3. Redes Neuronales(NN)/Deep Learning(DL)

- Utiliza modelos complejos basados en capas (**Deep Learning**) para tareas de abstracción y predicción de alto nivel.



Le Deep Learning est une sous-discipline du Machine Learning

Figure: Relación jerárquica entre IA, ML y DL.

Neurona Artificial: Definición e Inspiración

Definición

Es la unidad fundamental de las Redes Neuronales, diseñada para simular el proceso de **activación e inhibición** de las neuronas biológicas. Su función principal es recibir señales y producir una única salida.

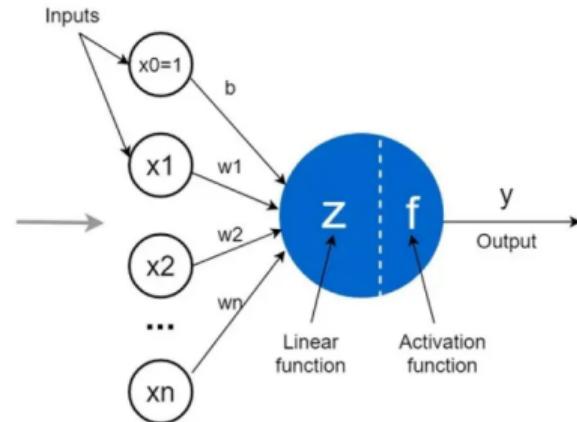


Figure: Paralelo entre neurona biológica y unidad artificial.

Suma Ponderada (z): Fundamento de la Neurona

Definición Vectorial

La activación interna o señal integrada (z) se calcula como el **producto escalar** entre el vector de pesos $\mathbf{w} \in \mathbb{R}^n$ y el vector de entrada $\mathbf{x} \in \mathbb{R}^n$, más un término escalar de sesgo ($b \in \mathbb{R}$):

$$\hat{y} = z = \mathbf{w}^\top \mathbf{x} + b$$

Interpretación Matemática:

- z representa una **combinación lineal** de las entradas ponderadas por sus pesos.
- El sesgo b introduce un desplazamiento adicional que evita la restricción de pasar por el origen.
- Este modelo constituye la base de toda red neuronal feed-forward.

Expansión Escalar:

$$z = \sum_{i=1}^n w_i x_i + b$$

Interpretación Geométrica:

- La ecuación define un **hiperplano** en \mathbb{R}^n :

$$\mathbf{w}^\top \mathbf{x} + b = 0$$

- El vector \mathbf{w} es **normal** al hiperplano.
- El término b determina su desplazamiento respecto al origen:

$$\text{distancia} = -\frac{b}{\|\mathbf{w}\|}$$

Regresión Lineal: Aplicación de la Neurona Artificial (I)

Regresión Lineal Simple:

- Se modela una variable de salida $y \in \mathbb{R}$ a partir de una única variable de entrada $x \in \mathbb{R}$.
- La relación se expresa como:

$$\hat{y} = z = w_1 x + b$$

donde w_1 representa la pendiente y b el intercepto.

Regresión Lineal Múltiple:

- Se extiende el modelo a n variables de entrada:

$$\hat{y} = z = \sum_{i=1}^n w_i x_i + b = \mathbf{w}^\top \mathbf{x} + b$$

- Cada w_i representa la contribución lineal de la característica x_i a la variable objetivo y .

Regresión Lineal: Formulación Matricial (II)

El modelo lineal para n observaciones y m características se expresa como:

$$\hat{\mathbf{y}} = \mathbf{z} = \mathbf{X}\mathbf{w} + \mathbf{b}$$

donde:

- $\mathbf{X} \in \mathbb{R}^{n \times m}$: matriz de diseño o entradas.
- $\mathbf{w} \in \mathbb{R}^m$: vector de parámetros o pesos.
- $\mathbf{b} \in \mathbb{R}^n$: vector de sesgos o bias.
- $\hat{\mathbf{y}} \in \mathbb{R}^n$: vector de valores predichos.

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nm} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} + \begin{bmatrix} b \\ b \\ \vdots \\ b \end{bmatrix}$$

Regresión Lineal: Función de Costo (III-A)

Definición del Error Cuadrático Medio (MSE):

El objetivo es estimar los parámetros \mathbf{w} y b que minimicen la discrepancia entre los valores observados \mathbf{y} y los predichos $\hat{\mathbf{y}}$:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

Sustituyendo el modelo lineal:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n (\mathbf{w}^\top \mathbf{X}_i + b - y_i)^2$$

Interpretación:

- Mide el **error promedio cuadrático** entre la salida real y la predicha.
- Penaliza más fuertemente los errores grandes.

Regresión Lineal: Función de Costo (III-B)

Forma Matricial:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{n} \|\mathbf{X}\mathbf{w} + \mathbf{b} - \mathbf{y}\|_2^2$$

Donde:

- $\mathbf{X} \in \mathbb{R}^{n \times m}$ es la matriz de diseño.
- \mathbf{y} es el vector de salidas reales.

Regresión Lineal: Derivadas del MSE (IV-A)

Función de costo:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n (\mathbf{w}^\top \mathbf{X}_i + b - y_i)^2$$

Sea el error para el ejemplo i -ésimo:

$$e_i = \hat{y}_i - y_i = \mathbf{w}^\top \mathbf{X}_i + b - y_i$$

Entonces:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n e_i^2$$

Derivadas parciales:

$$\frac{\partial \mathcal{L}}{\partial w_j} = \frac{2}{n} \sum_{i=1}^n e_i \mathbf{X}_{ij} \quad \frac{\partial \mathcal{L}}{\partial b} = \frac{2}{n} \sum_{i=1}^n e_i$$

Regresión Lineal: Derivadas del MSE (IV-B)

Recordando la forma matricial:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{n} \|\mathbf{X}\mathbf{w} + \mathbf{b} - \mathbf{y}\|_2^2$$

Gradientes vectoriales:

$$\nabla_{\mathbf{w}} \mathcal{L} = \frac{2}{n} \mathbf{X}^\top (\mathbf{X}\mathbf{w} + \mathbf{b} - \mathbf{y})$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{2}{n} \mathbf{1}_n^\top (\mathbf{X}\mathbf{w} + \mathbf{b} - \mathbf{y})$$

Donde:

- $\mathbf{1}_n$ es el vector de unos en \mathbb{R}^n .

Regresión Lineal: Condiciones del Mínimo Analítico (V-A)

Condición de óptimo (mínimo global):

$$\nabla_{\mathbf{w}} \mathcal{L} = 0, \quad \frac{\partial \mathcal{L}}{\partial b} = 0$$

Sustituyendo y simplificando:

$$\mathbf{X}^T(\mathbf{X}\mathbf{w} + \mathbf{b} - \mathbf{y}) = 0$$

$$\mathbf{1}_n^T(\mathbf{X}\mathbf{w} + \mathbf{b} - \mathbf{y}) = 0$$

Estas dos ecuaciones conforman el **sistema normal de mínimos cuadrados**.

Regresión Lineal: Sistema Normal y Despeje de b (V-B1)

Expandiendo el sistema:

$$\begin{cases} \mathbf{X}^\top \mathbf{Xw} + b\mathbf{X}^\top = \mathbf{X}^\top \mathbf{y} \\ \mathbf{1}_n^\top \mathbf{Xw} + b\mathbf{1}_n^\top = \mathbf{1}_n^\top \mathbf{y} \end{cases}$$

Solución cerrada para los parámetros:

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{H} \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{H} \mathbf{y}$$

$$b^* = \bar{y} - (\mathbf{w}^*)^\top \bar{\mathbf{x}}$$

Donde:

$$\mathbf{H} = I_n - \frac{1}{n} \mathbf{1}_n \mathbf{1}_n^\top, \quad \bar{\mathbf{x}} = \frac{1}{n} \mathbf{X}^\top \mathbf{1}_n \in \mathbb{R}^m, \quad \bar{y} = \frac{1}{n} \mathbf{1}_n^\top \mathbf{y} \in \mathbb{R}$$

Interpretación:

- \mathbf{w}^* es la solución de **mínimos cuadrados ordinarios (OLS)**.
- b^* compensa el desplazamiento medio de los datos.
- Esta solución coincide con el mínimo global del error cuadrático.

Descenso del Gradiente: Principio General (VI-A)

Objetivo: minimizar una función de costo diferenciable $\mathcal{L}(\mathbf{w}, b)$ ajustando iterativamente los parámetros.

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}^{(t)}, b^{(t)})$$

$$b^{(t+1)} = b^{(t)} - \eta \frac{\partial \mathcal{L}(\mathbf{w}^{(t)}, b^{(t)})}{\partial b}$$

donde:

- $\eta > 0$ es la **tasa de aprendizaje** usualmente $\eta = 0.01$.
- $\nabla_{\mathbf{w}} \mathcal{L}$ y $\frac{\partial \mathcal{L}}{\partial b}$ indican la dirección de máximo incremento de J .
- Restarlas implementa el **descenso** hacia el mínimo.

Criterio de parada:

$$\|\nabla \mathcal{L}(\mathbf{w}^{(t)}, b^{(t)})\|_2 < \varepsilon \quad \text{o} \quad t \geq t_{\max}$$

Descenso del Gradiente: Minimización del Error Cuadrático Medio (VI-B)

Función de costo:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{n} \|\mathbf{X}\mathbf{w} + \mathbf{b} - \mathbf{y}\|_2^2$$

Gradientes:

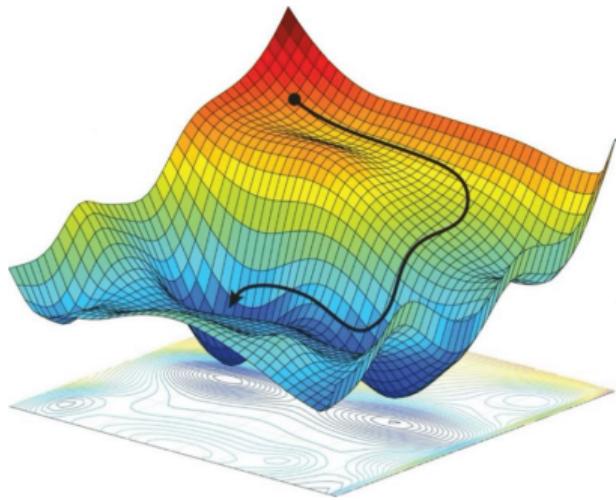
$$\nabla_{\mathbf{w}} \mathcal{L} = \frac{2}{n} \mathbf{X}^\top (\mathbf{X}\mathbf{w} + \mathbf{b} - \mathbf{y})$$

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{2}{n} \mathbf{1}_n^\top (\mathbf{X}\mathbf{w} + \mathbf{b} - \mathbf{y})$$

Actualización iterativa:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \frac{2\eta}{n} \mathbf{X}^\top (\mathbf{X}\mathbf{w}^{(t)} + \mathbf{b}^{(t)} - \mathbf{y})$$

$$b^{(t+1)} = b^{(t)} - \frac{2\eta}{n} \mathbf{1}_n^\top (\mathbf{X}\mathbf{w}^{(t)} + \mathbf{b}^{(t)} - \mathbf{y})$$



Evaluación del Modelo: Métricas en Regresión (VIII)

Error Absoluto Medio (MAE)

$$MAE = \frac{1}{n} \sum_{i=1}^n |\hat{y}_i - y_i|$$

Error Porcentual Absoluto Medio (MAPE)

$$MAPE = \frac{100}{n} \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right|$$

Error Cuadrático Medio (MSE)

$$MSE = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

Coeficiente de Determinación (R^2)

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

Función de Activación: Sigmoide (IX-A)

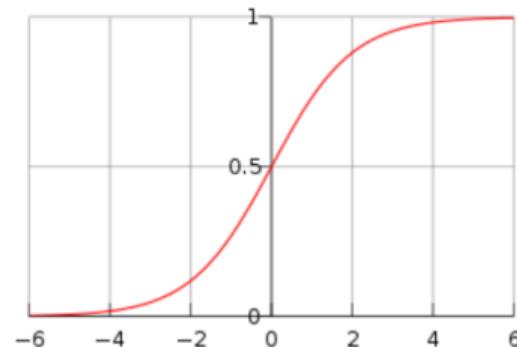
Definición: Función no lineal que comprime la entrada en el rango $(0, 1)$, comúnmente usada en modelos probabilísticos y neuronas binarias.

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

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

Aplicaciones:

- Clasificación binaria $p(y = 1|x)$.
- Capa de salida en regresión logística.
- Modelos donde se requiere una interpretación probabilística.



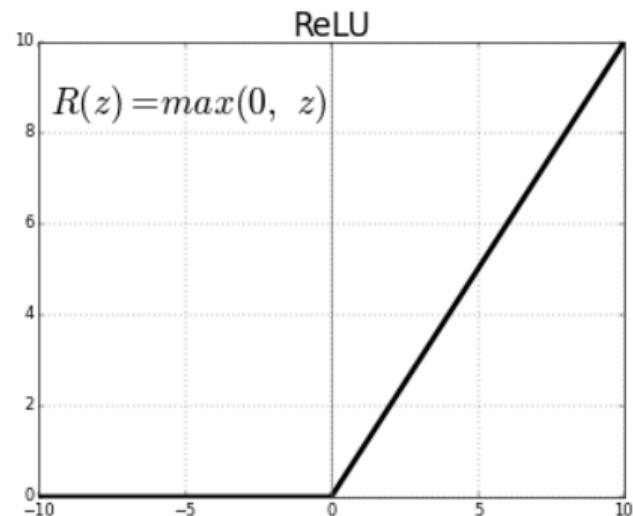
Función de Activación: ReLU (IX-B)

Definición: Activa solo valores positivos de la entrada, anulando los negativos. Introduce no linealidad con bajo costo computacional.

$$\text{ReLU}(x) = \max(0, x)$$

Aplicaciones:

- Capas ocultas en redes neuronales profundas (DNN, CNN).
- Mejora la convergencia del entrenamiento.
- Evita el problema de saturación del gradiente.



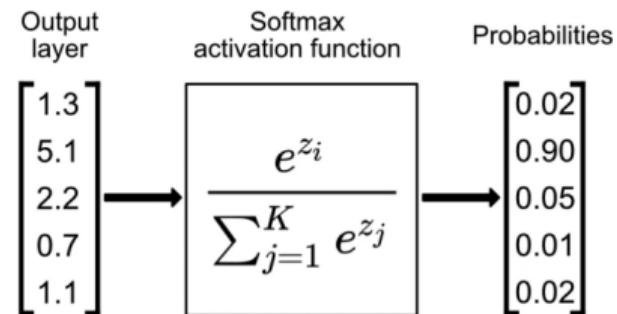
Función de Activación: Softmax (IX-D)

Definición: Convierte un vector de valores reales en una distribución de probabilidad sobre K clases.

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

Aplicaciones:

- Capa de salida en clasificación multiclas.
- Modelos probabilísticos como redes neuronales bayesianas.
- Permite interpretar la salida como $p(y = k|x)$.



Capas de una Red Neuronal: Formulación Matricial (I)

En una neurona individual, la operación lineal se define como:

$$\hat{\mathbf{y}} = \mathbf{z} = \mathbf{X}\mathbf{w} + \mathbf{b}$$

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nm} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} + \begin{bmatrix} b \\ b \\ \vdots \\ b \end{bmatrix}$$

Capas de una Red Neuronal: Formulación Matricial (II)

Consideremos una capa completamente conectada que recibe como entrada una matriz de datos:

$$\mathbf{X} \in \mathbb{R}^{n \times m},$$

donde:

- n : número de observaciones (filas),
- m : número de características (columnas).

La capa posee k neuronas, cada una con un conjunto de m pesos y un sesgo. Por tanto, los parámetros de la capa se representan como:

$$\mathbf{W} \in \mathbb{R}^{m \times k}, \quad \mathbf{B} \in \mathbb{R}^{n \times k}.$$

El modelo lineal de la capa se expresa matricialmente como:

$$\mathbf{Z} = \mathbf{X}\mathbf{W} + \mathbf{B},$$

donde $\mathbf{Z} \in \mathbb{R}^{n \times k}$ contiene las salidas (pre-activaciones) de las k neuronas para las n observaciones.

Capas de una Red Neuronal: Formulación Matricial (III)

$$\begin{bmatrix} z_{11} & z_{12} & \dots & z_{1k} \\ z_{21} & z_{22} & \dots & z_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \dots & z_{nk} \end{bmatrix} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nm} \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1k} \\ w_{21} & w_{22} & \dots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \dots & w_{mk} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 & \dots & b_k \\ b_1 & b_2 & \dots & b_k \\ \vdots & \vdots & \ddots & \vdots \\ b_1 & b_2 & \dots & b_k \end{bmatrix}$$

Ahora pasamos \mathbf{Z} por una función de activación $a(\mathbf{Z})$ para obtener la salida final de la capa.

$$\hat{\mathbf{Y}} = a(\mathbf{Z})$$

Red Neuronal como Composición de Funciones (I)

Una red neuronal puede interpretarse matemáticamente como una **composición de funciones**. Cada capa transforma su entrada mediante una combinación lineal seguida de una función de activación no lineal.

Para una red con L capas, cada una definida por:

$$\mathbf{X}^{(l)} = a^{(l)}(\mathbf{Z}^{(l)}), \quad \mathbf{Z}^{(l)} = \mathbf{X}^{(l-1)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}$$

donde:

- $\mathbf{W}^{(l)} \in \mathbb{R}^{m_{l-1} \times m_l}$: pesos de la capa l ,
- $\mathbf{b}^{(l)} \in \mathbb{R}^{m_l}$: sesgos,
- $a^{(l)}(\cdot)$: función de activación de la capa l .
- $\mathbf{X}^{(l)} \in \mathbb{R}^{n \times m_l}$: salida de la capa l .

Red Neuronal como Composición de Funciones (II)

En forma más explícita, la salida de la red para una entrada \mathbf{X} se obtiene aplicando sucesivamente las transformaciones de cada capa:

$$\begin{aligned}\mathbf{x}^{(1)} &= a^{(1)}(\mathbf{z}^{(1)}) = a^{(1)}(\mathbf{XW}^{(1)} + \mathbf{b}^{(1)}), \\ &\vdots \\ \mathbf{x}^{(L-1)} &= a^{(L-1)}(\mathbf{z}^{(L-2)}) = a^{(L-1)}(\mathbf{x}^{(L-2)}\mathbf{W}^{(L-2)} + \mathbf{b}^{(L-2)}), \\ \hat{\mathbf{Y}} &= a^{(L)}(\mathbf{z}^{(L)}) = a^{(L)}(\mathbf{x}^{(L-1)}\mathbf{W}^{(L)} + \mathbf{b}^{(L)}).\end{aligned}$$

De esta forma, una red neuronal profunda implementa una función compuesta:

$$\hat{\mathbf{Y}} = f(\mathbf{X}; \Theta) = (a^{(L)} \circ \mathbf{Z}^{(L)} \circ \dots \circ a^{(1)} \circ \mathbf{Z}^{(1)})(\mathbf{X}),$$

donde $\Theta = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^L$ representa el conjunto total de parámetros del modelo.

Retropropagación: Cálculo de Gradientes (I)

El objetivo del entrenamiento es minimizar una función de pérdida $\mathcal{L}(\hat{\mathbf{Y}}, \mathbf{Y})$ respecto a los parámetros $\Theta = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^L$.

Dado que la red es una composición de funciones,

$$\hat{\mathbf{Y}} = f(\mathbf{X}; \Theta) = a^{(L)}(\mathbf{Z}^{(L)}) = a^{(L)}(\mathbf{X}^{(L-1)}\mathbf{W}^{(L)} + \mathbf{b}^{(L)}),$$

aplicamos la **regla de la cadena** para propagar los gradientes desde la salida hacia las capas anteriores.

Para la capa L :

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{Z}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{Y}}} \odot a'^{(L)}(\mathbf{Z}^{(L)}),$$

donde \odot denota el producto elemento a elemento.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = (\mathbf{X}^{(L-1)})^\top \delta^{(L)}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(L)}} = \sum_i \delta_i^{(L)}.$$

Retropropagación: Cálculo de Gradientes (II)

Para las capas ocultas $l = L - 1, L - 2, \dots, 1$, los gradientes se propagan hacia atrás utilizando:

$$\delta^{(l)} = (\delta^{(l+1)}(\mathbf{W}^{(l+1)})^\top) \odot a'^{(l)}(\mathbf{Z}^{(l)}),$$
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = (\mathbf{X}^{(l-1)})^\top \delta^{(l)}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \sum_i \delta_i^{(l)}.$$

Este procedimiento se repite recursivamente desde la capa de salida hasta la capa de entrada, actualizando los parámetros según descenso del gradiente:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}, \quad \mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}},$$

donde η es la tasa de aprendizaje.

Retos para construir un Framework Minimalista

Principales retos identificados:

- **Vincular matemáticas y programación:** llevar la formulación teórica de la composición de funciones y la retropropagación a código ejecutable.
- **Eficiencia numérica:** garantizar operaciones vectorizadas y cálculo de gradientes sin pérdida de rendimiento.
- **Portabilidad y escalabilidad:** permitir la ejecución del mismo código tanto en CPU como en GPU.
- **Modularidad:** diseñar una arquitectura flexible que permita incorporar nuevas funciones y capas sin modificar el núcleo.
- **Usabilidad y claridad:** ofrecer una interfaz de programación simple, cercana a los paradigmas de bibliotecas modernas de ML.

Solución: Sorix, Un Motor de ML Minimalista

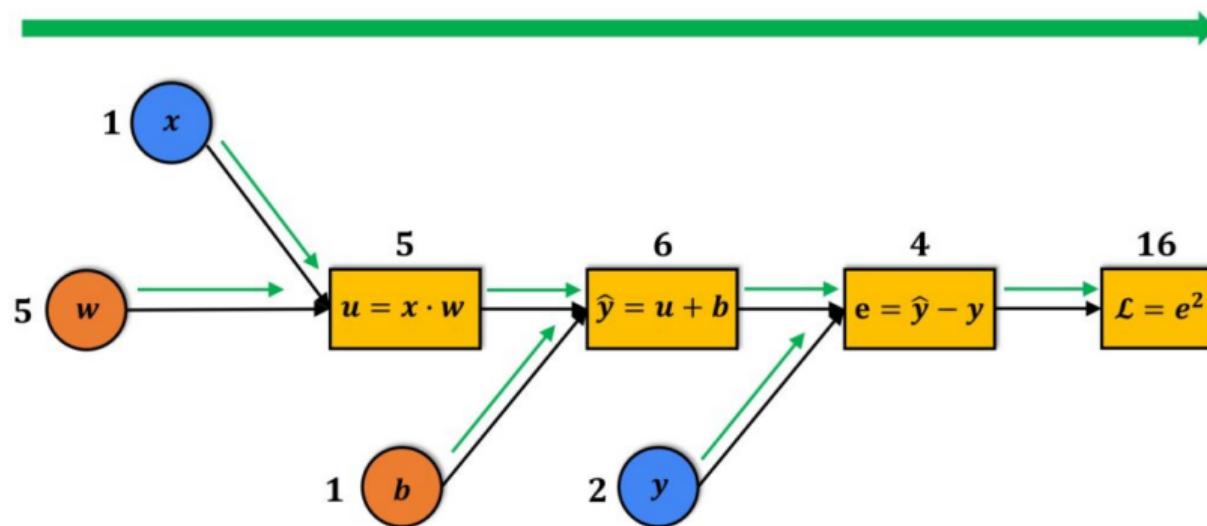
- **Sorix: Biblioteca de Aprendizaje Automático desde Cero**
 - Diseñada para ofrecer **control de bajo nivel** y máxima eficiencia.
 - Disponible públicamente a través de **PyPI**(pip install sorix).
- **Arquitectura de Alto Rendimiento**
 - **Core:** Implementación propia y limpia de **Autograd** (Diferenciación Automática).
 - **Backend CPU:** Construido sobre **NumPy**.
 - **Backend GPU:** Soporte nativo mediante **CuPy** para aceleración masiva.
- **Usabilidad**
 - Proporciona una **API similar a PyTorch**, lo que facilita el prototipado rápido y la migración de código existente.

Concepto de Autodiferenciación

Definición

La autodiferenciación calcula derivadas exactas mediante la aplicación sistemática de la **regla de la cadena** sobre un grafo computacional dinámico.

Forward propagation



Estructura de la clase tensor

Cada instancia de tensor constituye un nodo en el grafo computacional y contiene:

- **data**: arreglo numérico definido sobre \mathbb{R} , \mathbb{R}^N o $\mathbb{R}^{M \times N}$.
- **requires_grad**: Determina si el tensor participa en el cálculo de gradientes.
- **grad**: tensor del mismo orden que data, donde se acumulan los gradientes durante la retropropagación.
- **operaciones**: conjunto de transformaciones matemáticas definidas sobre el tensor (suma, producto, activaciones, etc.).
- **_op**: La operación que se utilizó para generarlo.
- **_prev**: referencias a los tensores padres utilizados para generar el nodo actual.
- **backward**: función local que define la regla de derivación asociada a la operación realizada.

clase tensor en Sorix

```
class tensor:

    Windsurf: Refactor | Explain | Generate Docstring | x
    def __init__(self, data, _children=[], _op='', device='cpu', requires_grad=False):

        if device == 'gpu' and not _cupy_available:

            raise Exception('Cupy is not available')

        xp = cp if (device == 'gpu' and _cupy_available) else np

        if isinstance(data, (list, tuple, np.ndarray, pd.DataFrame, pd.Series)):

            data = xp.array(data)

        self.data = data

        self.device = device
        self.requires_grad = requires_grad
        self.grad = xp.zeros_like(self.data) if requires_grad else None
        self._backward = _noop
        global _autograd_enabled
        self._prev = _children if (_autograd_enabled and requires_grad) else []
        self._op = _op if _autograd_enabled else ''
```

Operaciones con Autograd en Sorix(foward)

```
def __add__(self, other):
    other = other if isinstance(other, tensor) else tensor(other)

    global _autograd_enabled

    if not _autograd_enabled:
        return tensor(self.data + other.data, device=self.device)

    requires_grad = self.requires_grad or other.requires_grad

    out = tensor(self.data + other.data, [self, other], '+', device=self.device, requires_grad=requires_grad)

Windsurf: Refactor | Explain | Generate Docstring | ×
def __backward__():
    if self.requires_grad:
        grad_self = tensor._match_shape(out.grad, self.data.shape)
        self.grad += grad_self
    if other.requires_grad:
        grad_other = tensor._match_shape(out.grad, other.data.shape)
        other.grad += grad_other

    out.__backward__ = __backward__
    return out
```

Retropropagación Automática en Sorix(backward)

```
def backward(self):
    topo = []
    visited = set()

windsurf: Refactor | Explain | Generate Docstring | ×
def build_topo(t):
    if id(t) not in visited:
        visited.add(id(t))
        for child in t._prev:
            build_topo(child)
        topo.append(t)

build_topo(self)
self.grad = cp.ones_like(self.data) if self.device == 'gpu' else np.ones_like(self.data)

for node in reversed(topo):
    node._backward()
```

Control del Grafo: sorix.no_grad()

Modo sin gradientes

Durante inferencia o validación, el grafo no se construye:

```
with sorix.no_grad():
    pred = model(x_val)
```

- **sorix.no_grad()**: controla la variable global **_autograd_enabled**.
- Ahorra memoria y acelera la ejecución.
- Equivalente a `torch.no_grad()` en PyTorch.

Usando la GPU: NumPy vs CuPy

CPU – NumPy

```
import numpy as np  
x = np.random.randn(1000000)  
y = np.exp(x)
```

GPU – CuPy

```
import cupy as cp  
x = cp.random.randn(1000000)  
y = cp.exp(x)
```

cambiando np y cp por xp

```
xp = cp if (device == 'gpu' and _cupy_available) else np
```

Entrenamiento de una Red Neuronal en Sorix



Ejemplo: Entrenamiento de una red neuronal en Sorix. [Abrir en Google Colab](#)

- Explore more examples of **Machine Learning** with Sorix.
- Deepen your use of **Autograd** for neural networks.
- Contribute actively to the development of the project.



Scan the QR code to access the repository.

¡Gracias por su atención!

Preguntas o comentarios son bienvenidos.