



# **Inteligencia Artificial Aplicada para la Economía**



## **Profesores**

**Profesor Magistral**

Camilo Vega Barbosa

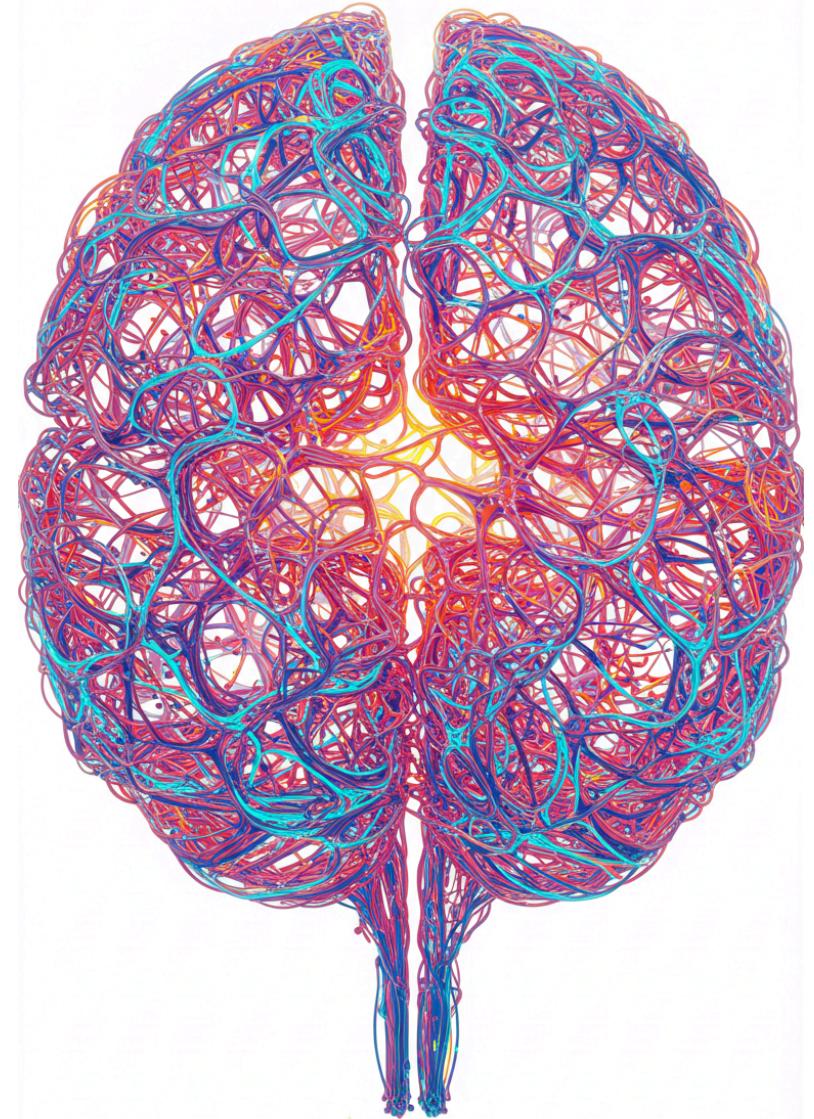
**Asistente de Docencia**

Sergio Julian Zona Moreno



# Deep Learning

## Fundamentos de las Redes Neuronales Profundas



# ¿Qué es el Deep Learning?

El Deep Learning es una rama del Machine Learning que usa **redes neuronales profundas** para aprender representaciones complejas de datos. Funciona en aprendizaje **supervisado** (datos etiquetados) y **no supervisado** (sin etiquetas).

A diferencia de métodos tradicionales, extrae patrones automáticamente mediante **entrenamiento iterativo y backpropagation**, ajustando millones de parámetros para mejorar el rendimiento.

## 🎯 Aplicaciones en Economía

-  Predicción financiera y detección de anomalías.
-  Evaluación de riesgo crediticio.
-  Análisis de tendencias macroeconómicas.



## Comparación de enfoques:

Enfoque	Aplicación	Modelo	Ejemplo
Modelo Tradicional	Series de tiempo	ARIMA	Inflación
Machine Learning	Modelos no lineales	Random Forest, SVM	Segmentación de clientes
Deep Learning	Representaciones complejas	Redes Neuronales	Recomendación de precios/productos



## ¿En qué nos enfocaremos?

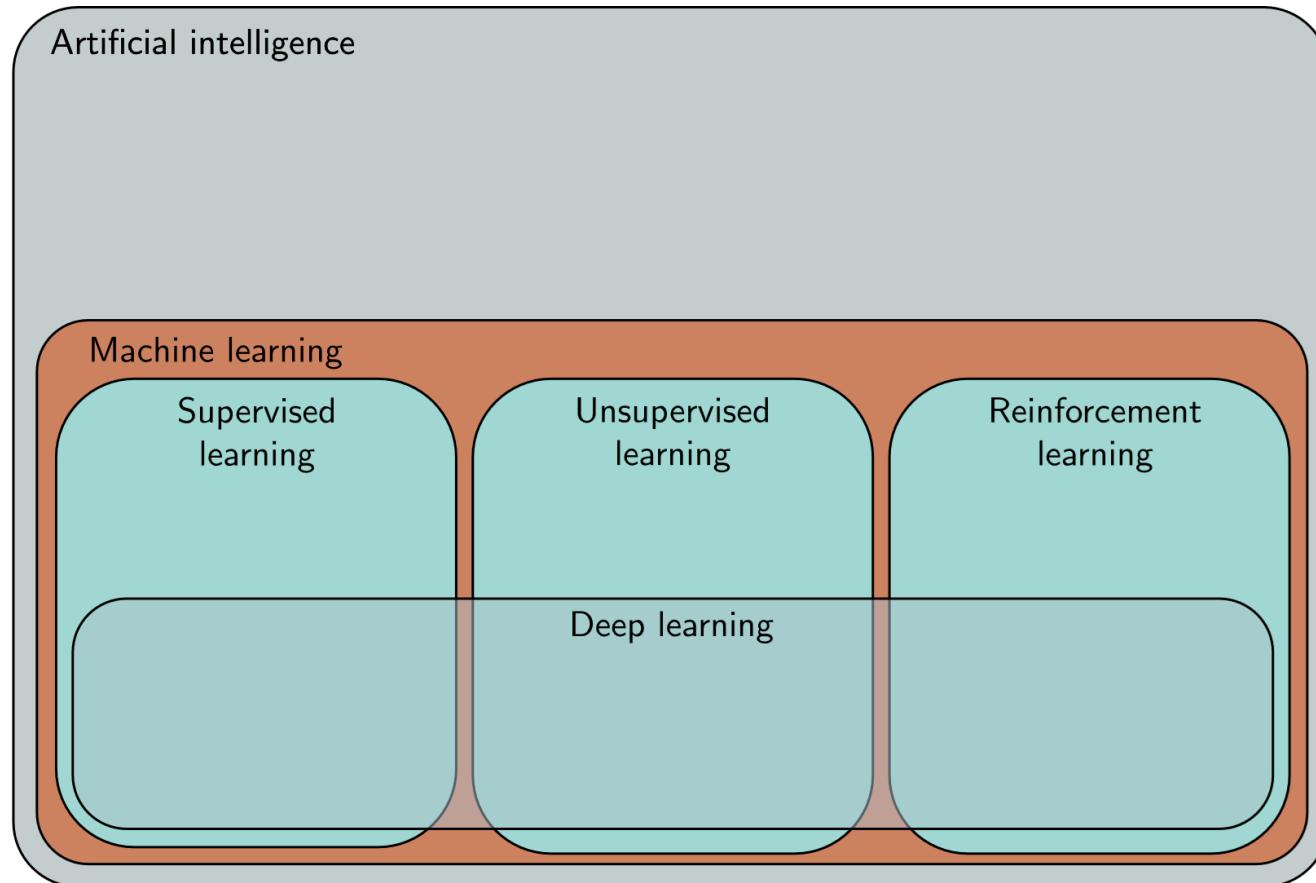
Nos centraremos en Deep Learning aplicado al aprendizaje supervisado. En esta rama:

- Conocemos  $\mathbf{x}$  y sus salidas esperadas  $y$ .
- Buscamos una función  $f(\mathbf{x}; \phi)$  que modele su relación.

 **Nota:**  $\phi$  representa los parámetros del modelo, que se ajustan para minimizar el error. Más adelante hablaremos de ellos.



# IA: ¿Dónde encaja el Deep Learning?



*Imagen tomada de [Understanding Deep Learning](#).*



# IA: ¿Dónde encaja el Deep Learning?

Supervised learning

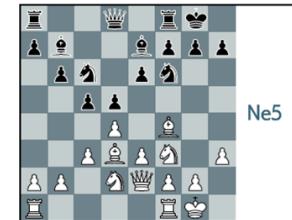


Bicycle

Unsupervised learning



Reinforcement learning



Reward = 0

Apple



Aardvark



Reward = -1

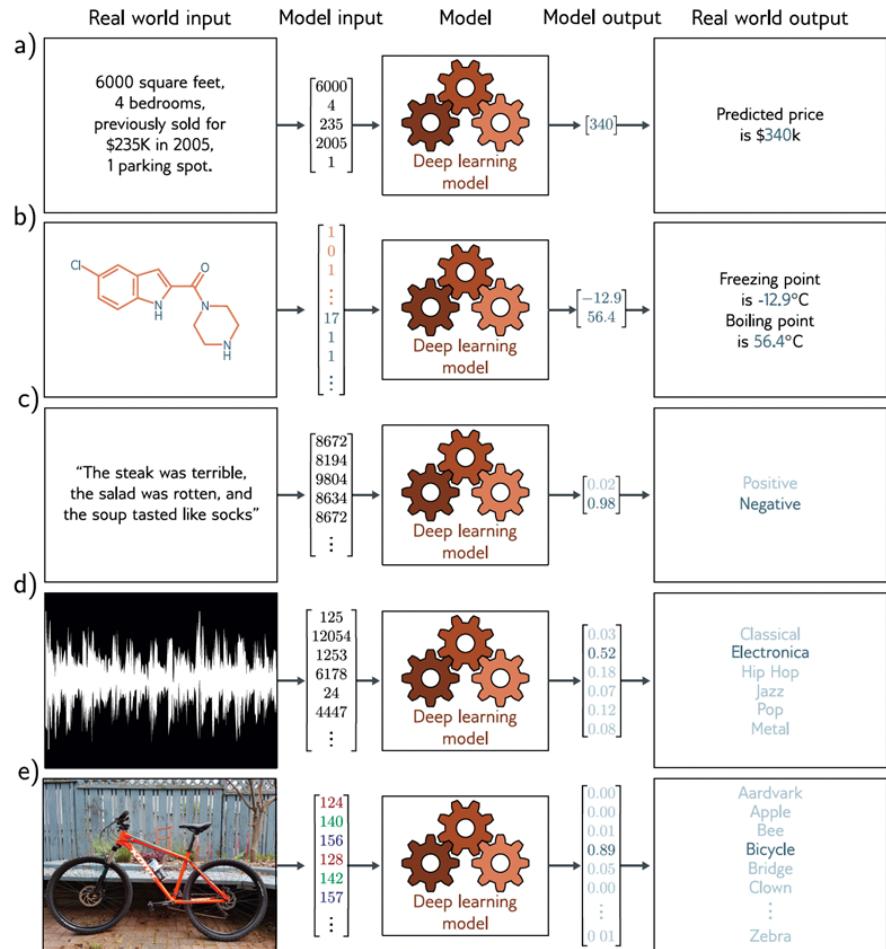
cxd4



Reward = +1

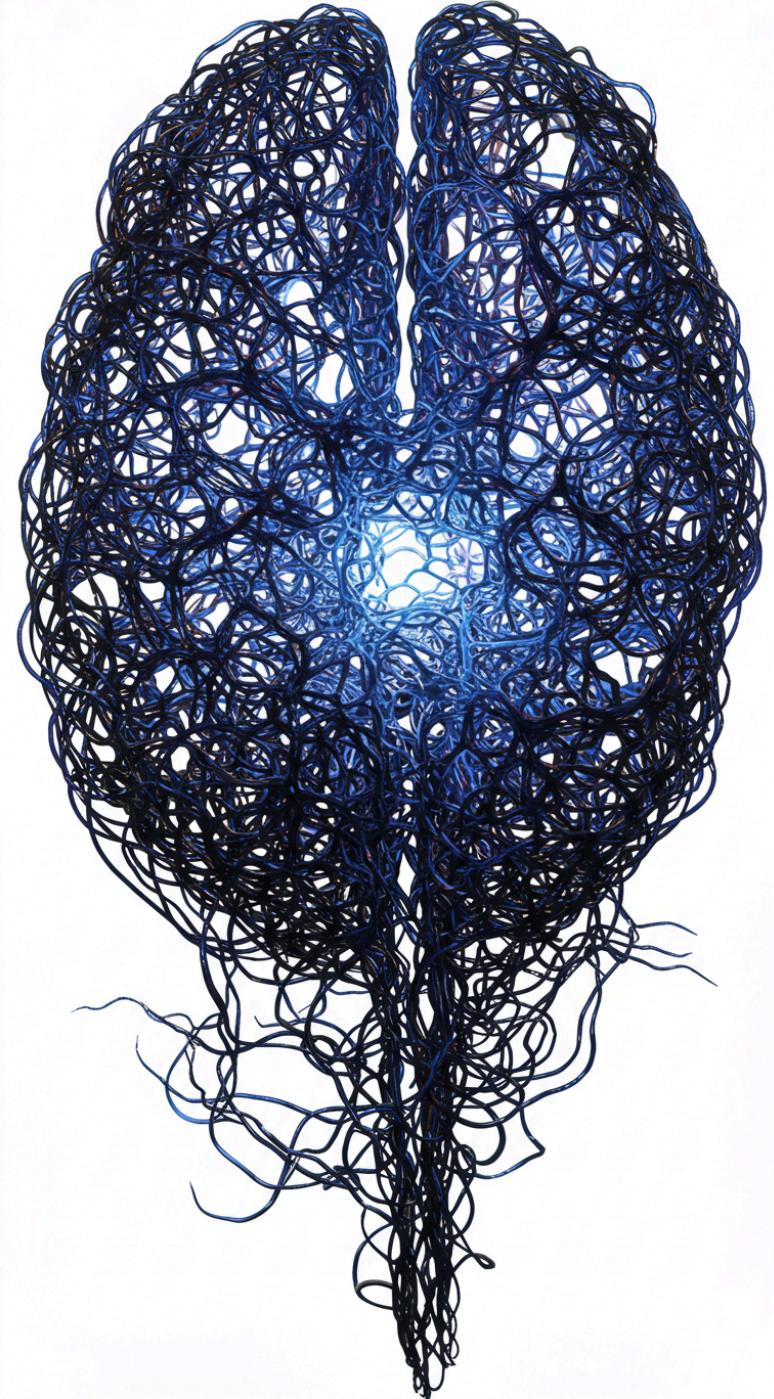


Imagen tomada de [Understanding Deep Learning](#).



**Figure 1.2** Regression and classification problems. a) This *regression* model takes a vector of numbers that characterize a property and predicts its price. b) This *multivariate regression* model takes the structure of a chemical molecule and predicts its melting and boiling points. c) This *binary classification* model takes a restaurant review and classifies it as either positive or negative. d) This *multiclass classification* problem assigns a snippet of audio to one of  $N$  genres. e) A second *multiclass classification* problem in which the model classifies an image according to which of  $N$  possible objects that it might contain.

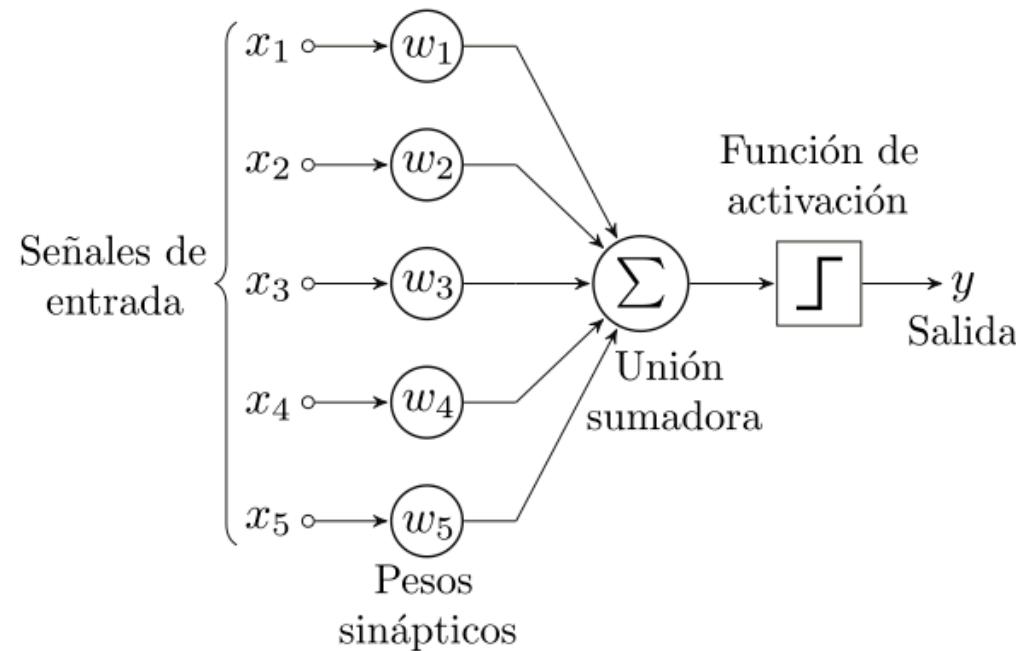
# La primera red neuronal



# El Perceptrón: Inspirado en el Cerebro Humano

El perceptrón, creado por **Frank Rosenblatt en 1957**, fue el primer modelo de **red neuronal artificial**, inspirado en el funcionamiento de las **neuronas biológicas**.

Rosenblatt buscaba replicar cómo las neuronas procesan información en el cerebro: **recibiendo señales, ponderándolas según su importancia y produciendo una respuesta cuando se alcanza cierto umbral de activación.**

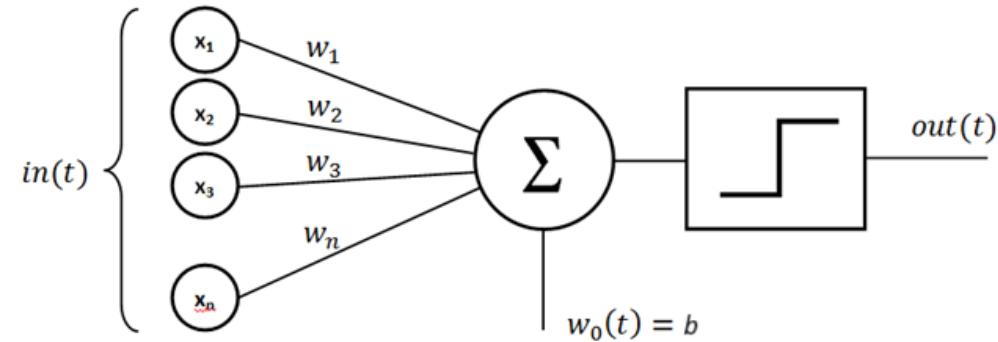




# Anatomía del Perceptrón

## Entradas y Pesos

- **Entradas ( $x$ ):** Señales numéricas que ingresan a la neurona.
- **Pesos ( $w$ ):** Determinan la influencia de cada entrada.
- **Umbral ( $b$ ):** Define el punto de activación de la neurona. También se suele llamar *bias*.

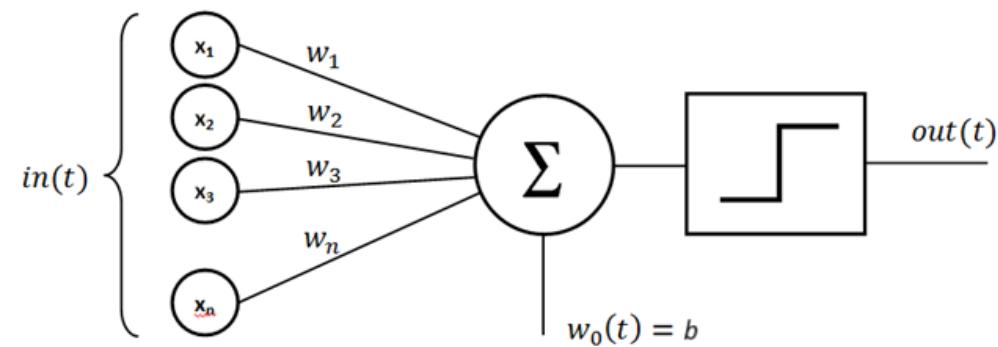




# Anatomía del Perceptrón

## ⚡ Procesamiento

- 1 Suma Ponderada:** Calcula la combinación lineal de entradas y pesos.
- 2 Función de Activación:** Decide si la neurona se activa.
- 3 Salida ( $y$ ):** Genera un resultado binario (0 o 1) según el umbral.





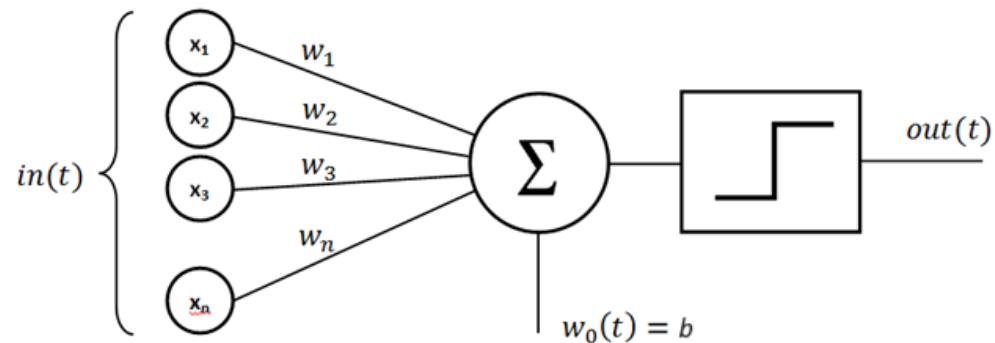
# Anatomía del Perceptrón

## 💡 Fórmula Matemática

$$y = f(w_1x_1 + w_2x_2 + \dots + w_nx_n - b)$$

Donde  $f$  es la función de activación (Escalón):

$$f(z) = \begin{cases} 1, & \text{si } z \geq 0 \\ 0, & \text{en otro caso} \end{cases}$$





## Funciones de Activación comunes

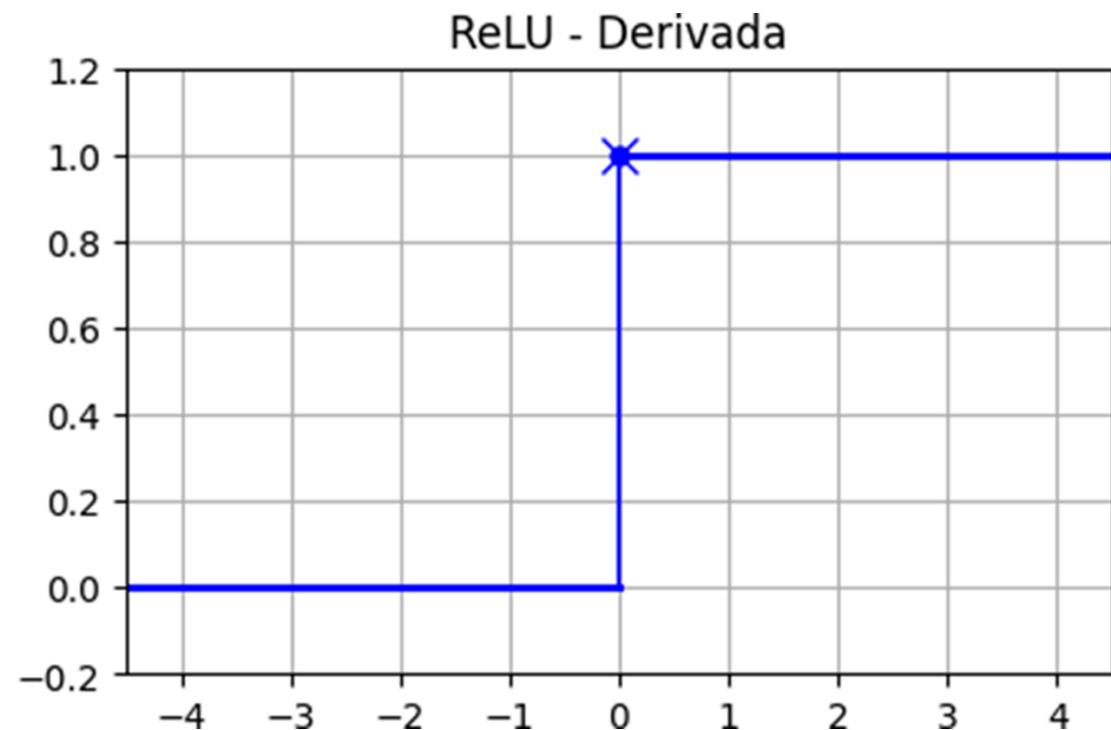
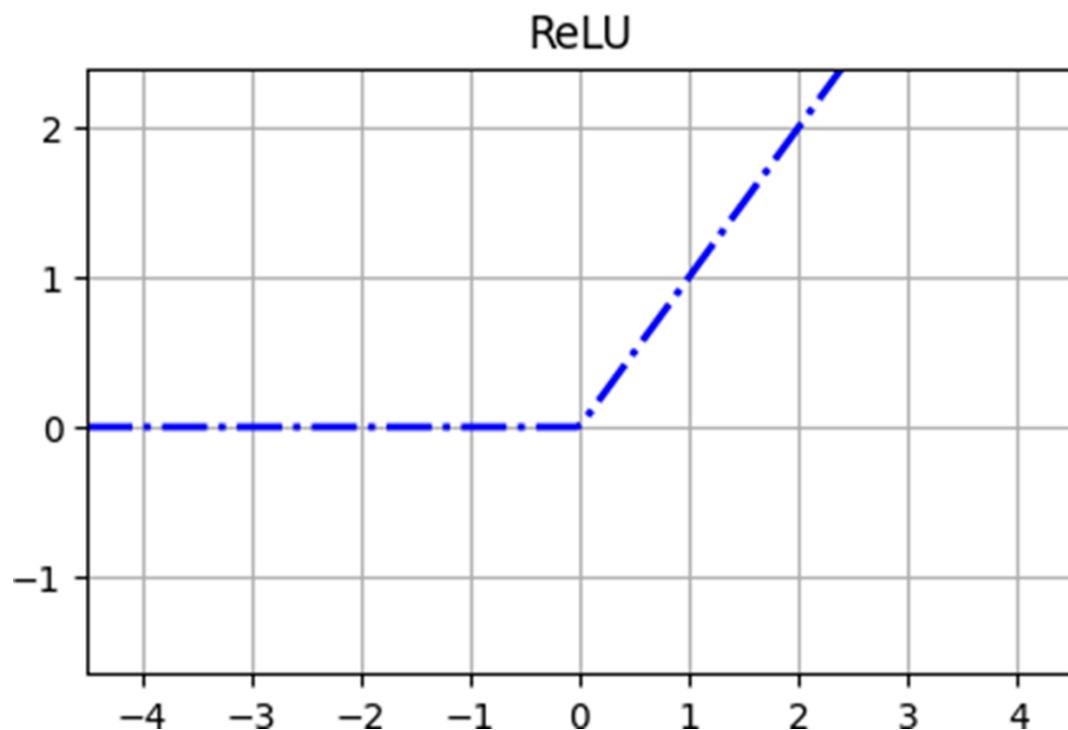
$$\text{ReLU}(x) = \max(0, x)$$

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

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

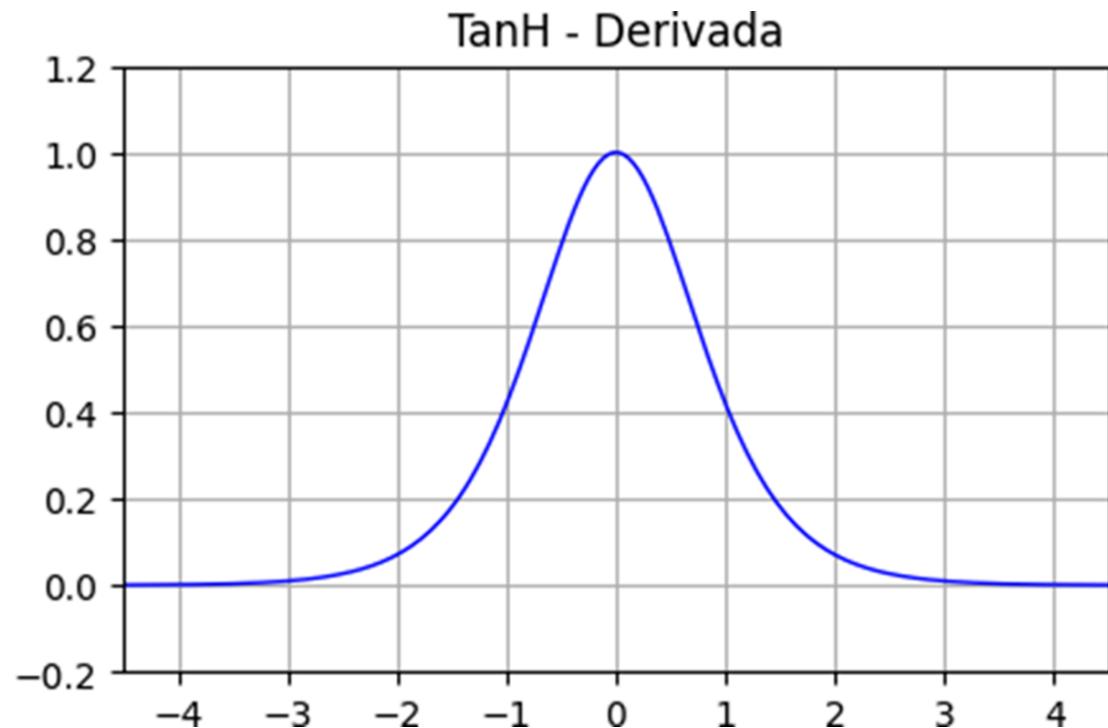
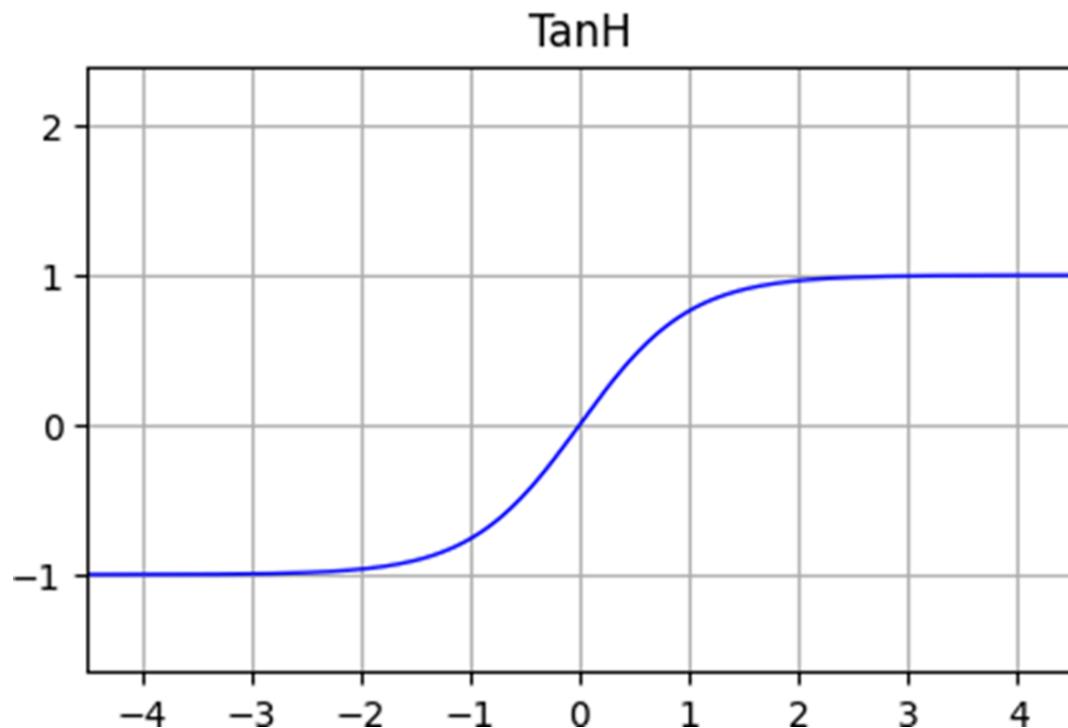


# Gráficamente



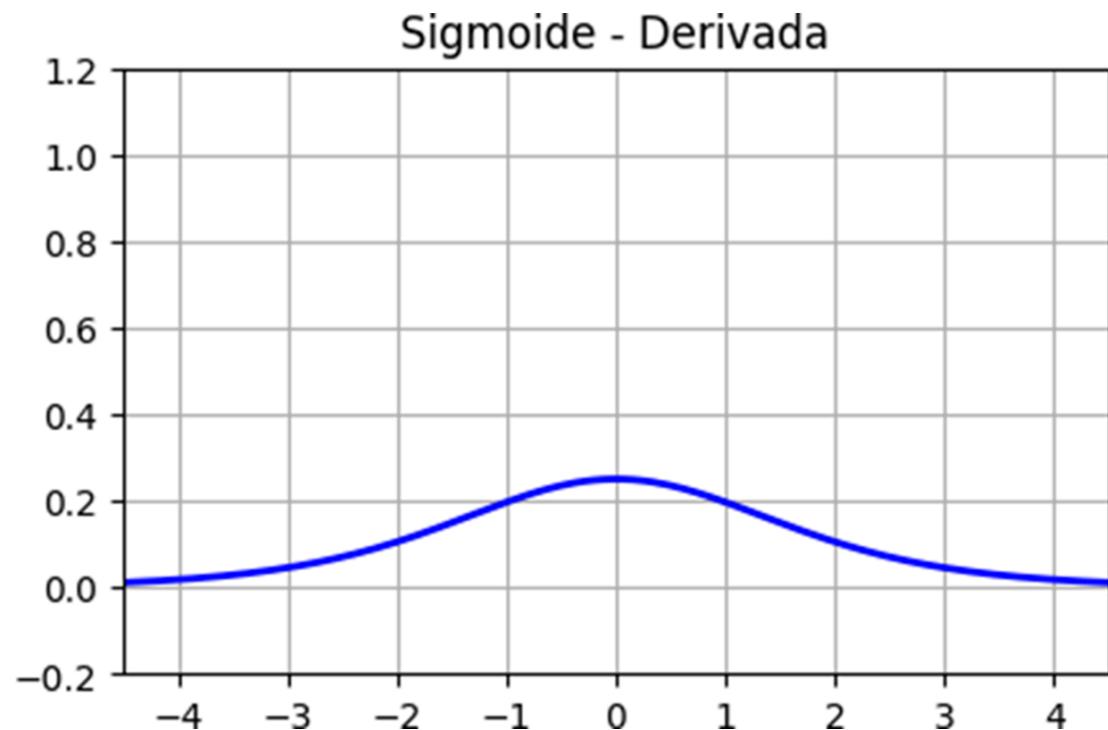
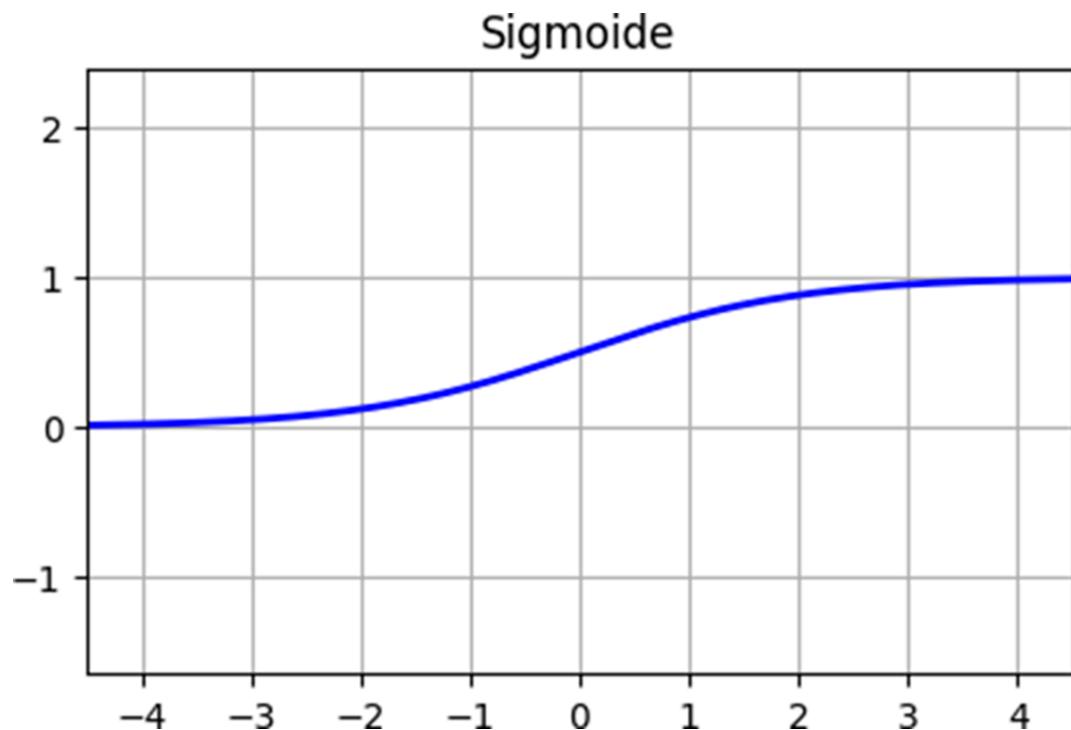


# Gráficamente





# Gráficamente





## Ejemplo de cálculo en un Perceptrón Básico

Dado un perceptrón con **dos entradas** y una salida binaria, queremos calcular su activación:

- Entradas:  $x_1, x_2$
- Pesos:  $w_1, w_2$
- Umbral:  $b$
- Función de activación: Escalón unitario



## Solución del Perceptrón:

### 1 Datos

- Pesos:  $w_1 = 0.6, w_2 = -0.8, b = 0.2$
- Entrada:  $x_1 = 1, x_2 = 0$

### 2 Fórmula General

$$z = w_1x_1 + w_2x_2 - b$$

$$y = f(z) = \begin{cases} 1, & \text{si } z \geq 0 \\ 0, & \text{si } z < 0 \end{cases}$$

### 3 Cálculo

$$z = (0.6)(1) + (-0.8)(0) - 0.2 = 0.6 - 0.2 = 0.4$$

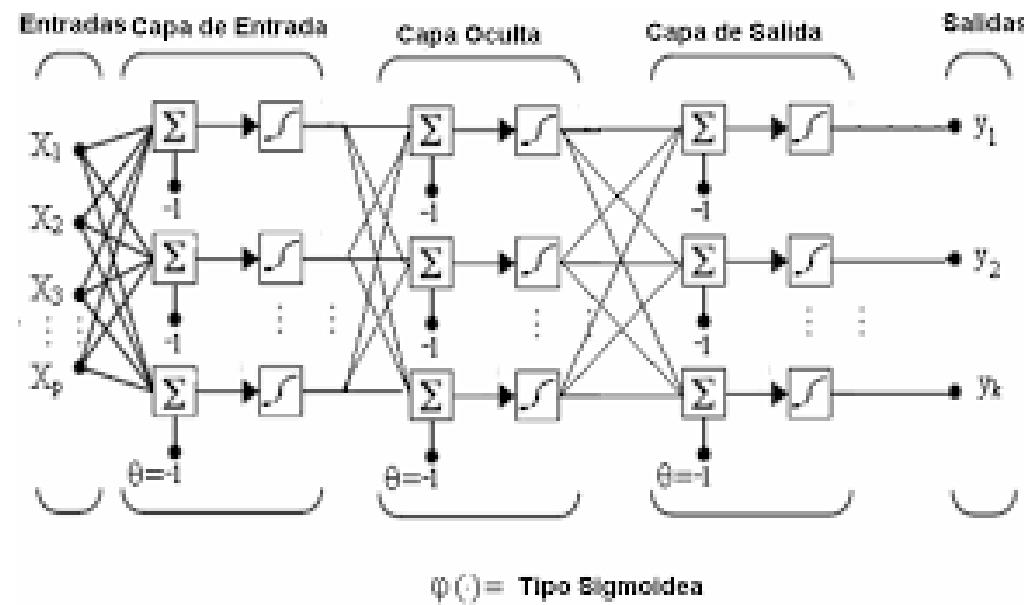
Como  $z > 0$ , la neurona **se activa** y la salida es  $y = 1$

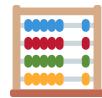
# Perceptrón Multicapa (MLP)

El perceptrón multicapa extiende el concepto básico agregando **capas intermedias** (ocultas) entre la entrada y la salida. Esto permite aprender representaciones más complejas.

## ⟳ Estructura del MLP

- **Capa de Entrada:** Recibe los datos originales.
- **Capas Ocultas:** Procesan y transforman la información.
- **Capa de Salida:** Produce el resultado final.





# Matemática del MLP

## 💡 Cálculo en Cada Neurona

Para cada neurona en cualquier capa, el cálculo es:

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

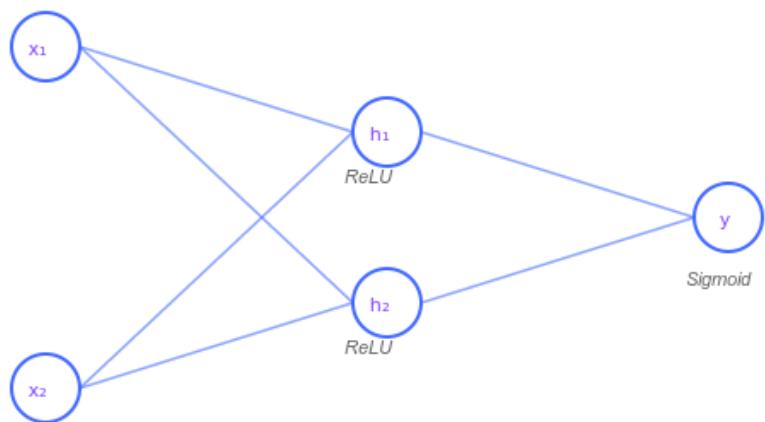
Donde ahora:

- $f$  es una función de activación no lineal (ReLU, sigmoid, tanh).
- $b$  es el bias.
- La salida  $y$  es continua, no binaria.



# Ejemplo de MLP

Capa de Entrada      Capa Oculta (ReLU)      Capa de Salida (Sigmoid)



## Estructura de la Red

- Entrada: 2 neuronas ( $x_1, x_2$ )
- Capa Oculta: 2 neuronas con ReLU
- Salida: 1 neurona con sigmoid



## Datos de Entrada

$$x = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$



# Parámetros de la Red

## 12 34 Capa Oculta

$$W_1 = \begin{bmatrix} 0.5 & -0.2 \\ 0.1 & 0.8 \end{bmatrix}$$

$$b_1 = \begin{bmatrix} 0.3 \\ -0.4 \end{bmatrix}$$

## Capa de Salida

$$W_2 = \begin{bmatrix} 0.7 & 0.3 \end{bmatrix}$$

$$b_2 = 0.2$$



# Cálculo en la Red Neural

## 1 Capa Oculta (ReLU)

$$\begin{aligned} h &= \text{ReLU}(W_1x + b_1) \\ &= \text{ReLU} \left( \begin{bmatrix} 0.5 & -0.2 \\ 0.1 & 0.8 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 0.3 \\ -0.4 \end{bmatrix} \right) \\ &= \text{ReLU} \begin{bmatrix} 0.1 \\ 1.3 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 1.3 \end{bmatrix} \end{aligned}$$



# Cálculo en la Red Neural

## 2 Capa de Salida (Sigmoid)

$$\begin{aligned}y &= \sigma(W_2 h + b_2) \\&= \sigma((0.7 \cdot 0.1 + 0.3 \cdot 1.3) + 0.2) \\&= \sigma(0.07 + 0.39 + 0.2) \\&= \sigma(0.66) \\&= 0.659\end{aligned}$$

## 🔍 Conclusión

- La red se activó con una confianza del 65.9%.
- Este valor sugiere una predicción positiva moderada.



**IMPORTANT**

## ? Preguntas Clave

- ¿Cómo se ajustan los **parámetros** en cada red neuronal?
- ¿Cómo sabemos si una red neuronal está **aprendiendo bien**?
- ¿Qué es el **entrenamiento**?
- ¿Cómo se **inicializan los pesos** en una red neuronal?



## Recapitulemos y formalicemos matemáticamente

Hemos discutido que en el **aprendizaje supervisado**:

- Conocemos las entradas  $\mathbf{x}$  y sus salidas esperadas  $y$ .
- Buscamos una función  $f(\mathbf{x}; \phi)$  que modele su relación.

Donde:

- $\mathbf{x}$ : Vector de características de entrada.
- $y$ : Valor o etiqueta esperada.
- $\phi$ : Parámetros del modelo que ajustamos durante el entrenamiento.

**Objetivo:** Encontrar  $\phi$  que minimice la diferencia entre  $f(\mathbf{x}; \phi)$  y  $y$ .



# Definimos matemáticamente

Input

$$\mathbf{x} = \begin{bmatrix} \text{edad} \\ \text{kilometraje} \end{bmatrix}$$

Output

$$y = [\text{precio}]$$

Modelo

$$y = f(\mathbf{x})$$

## Definimos matemáticamente

El dataset de entrenamiento de tamaño  $I$  estará compuesto por tuplas de la siguiente forma:

$$\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^I$$

La función de pérdida o función de costos mide qué tan mal performa el modelo.

$$L \left[ \underbrace{\phi, f(\mathbf{x}, \phi)}_{\text{model}}, \underbrace{\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^I}_{\text{train data}} \right] = L[\phi]$$

 **Nota:** Devuelve un escalar que es más pequeño cuando el modelo asigna mejor las entradas a las salidas.



# En el Entrenamiento

Función de pérdida:

$$L[\phi]$$

Durante el **entrenamiento**, buscamos los **parámetros** que minimicen la función de pérdida.



## ¿Qué son los parámetros?

Los **parámetros** ( $\phi$ ) son los valores internos del modelo que se ajustan durante el entrenamiento para mejorar la predicción. Incluyen:

- **Pesos**: Determinan la influencia de cada **conexión** entre neuronas.
- **Sesgos (biases)**: Ajustan la **activación** de cada neurona para mejorar el aprendizaje.



## En el Entrenamiento



### Optimización:

El objetivo es encontrar los mejores parámetros  $\hat{\phi}$  que minimicen la pérdida:

$$\hat{\phi} = \arg \min_{\phi} L[\phi]$$

# Parámetros vs. Hiperparámetros

## Ejemplo: Regresión Lineal

En **regresión lineal**, los **parámetros** del modelo son:

- **Pesos ( $w$ )**: Representan la relación entre la entrada y la salida.
- **Bias ( $b$ )**: Ajusta la línea para mejorar la precisión.

El modelo aprende estos parámetros a partir de los datos:

$$\hat{y} = wx + b$$

 **Los parámetros son aprendidos** durante el entrenamiento optimizando la función de pérdida.



## Ejemplo KNN: Hiperparámetros, no Parámetros

A diferencia de regresión lineal, el algoritmo **k**-Nearest Neighbors (KNN) no aprende parámetros internos.

- ◆ KNN clasifica basándose en la distancia a los vecinos más cercanos.
- ◆ Su principal hiperparámetro es  $k$  (número de vecinos considerados).

### 📌 Diferencia clave:

- Parámetros ( $w, b$ ) → Aprendidos durante el entrenamiento.
- Hiperparámetros ( $k$  en KNN) → Definidos antes del entrenamiento y ajustados manualmente.



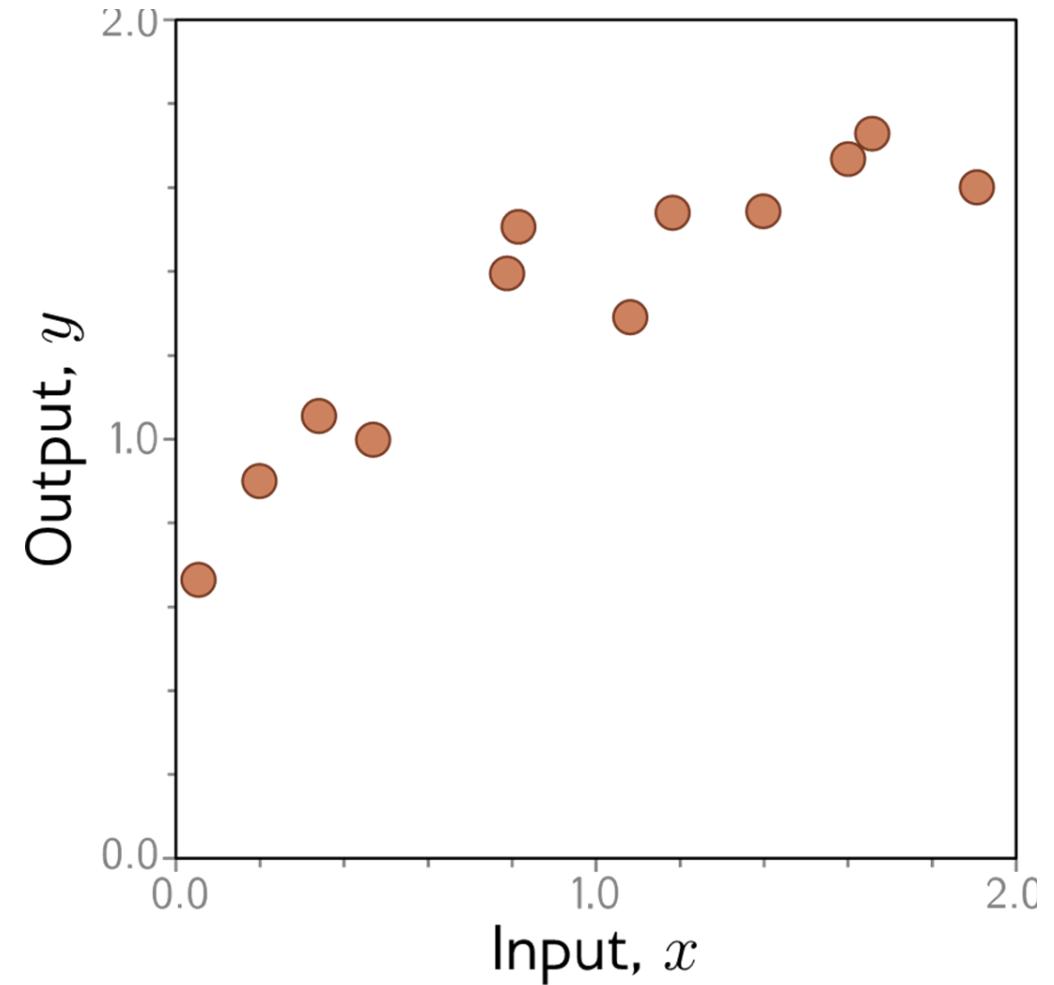
# Modelo de regresión de una dimensión

- ◆ **Modelo:**

$$y = f[x, \phi] = \phi_0 + \phi_1 x$$

- ◆ **Parámetros:**

$$\phi = \begin{bmatrix} \phi_0 \\ \phi_1 \end{bmatrix}$$

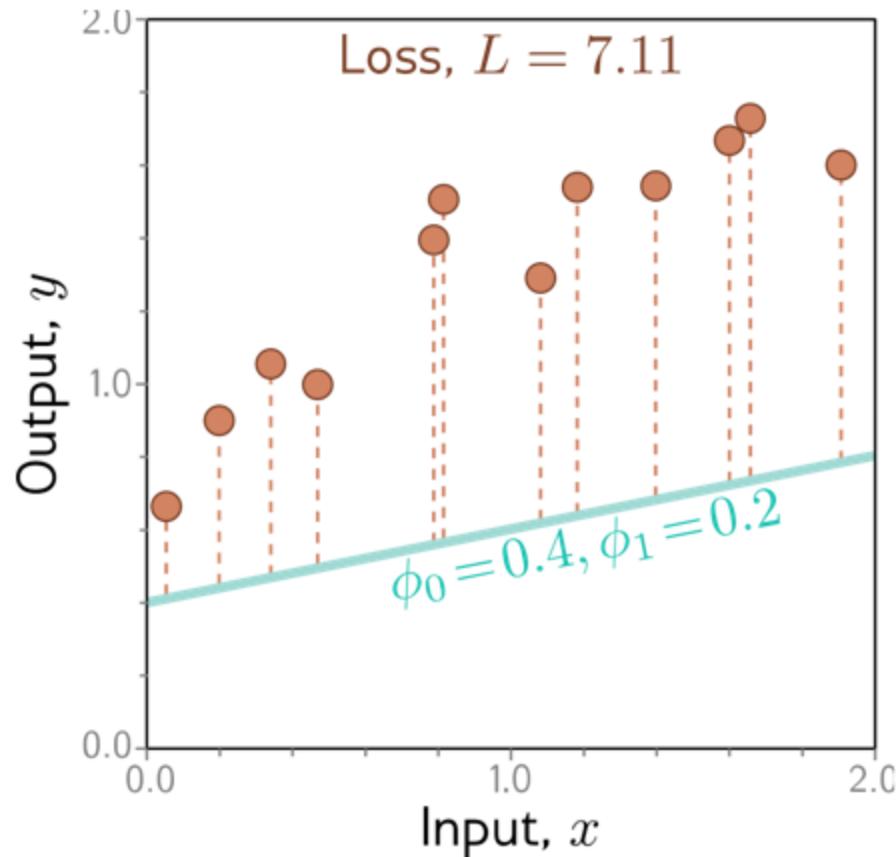




# Modelo de regresión de una dimensión

## ◆ Función de Pérdida:

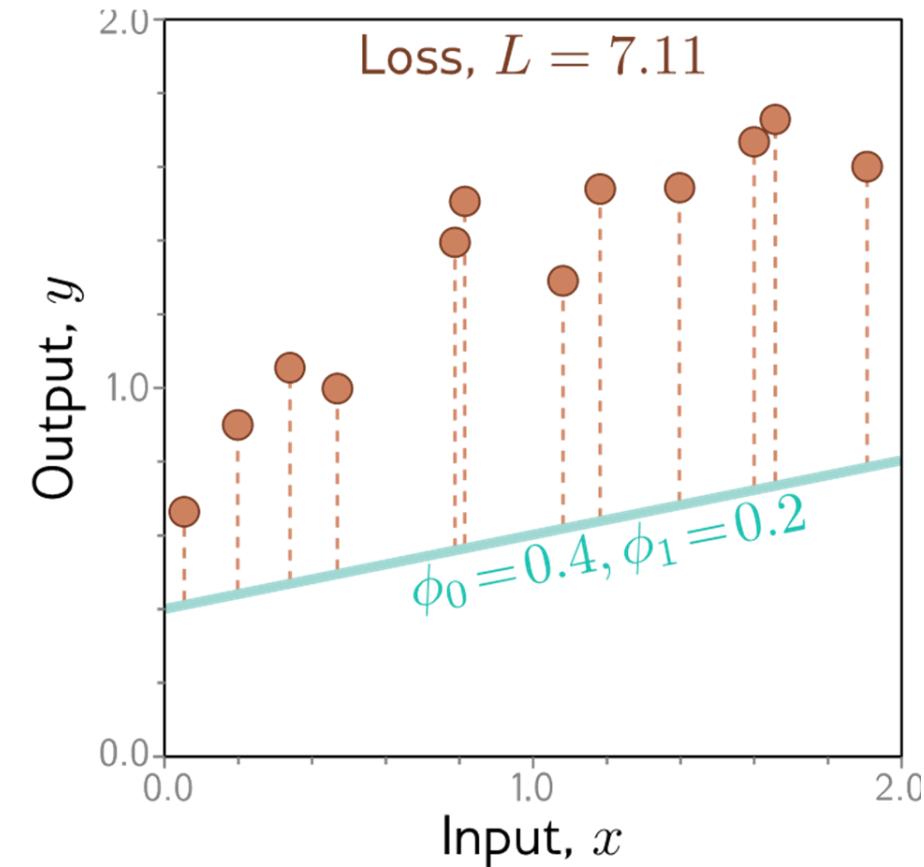
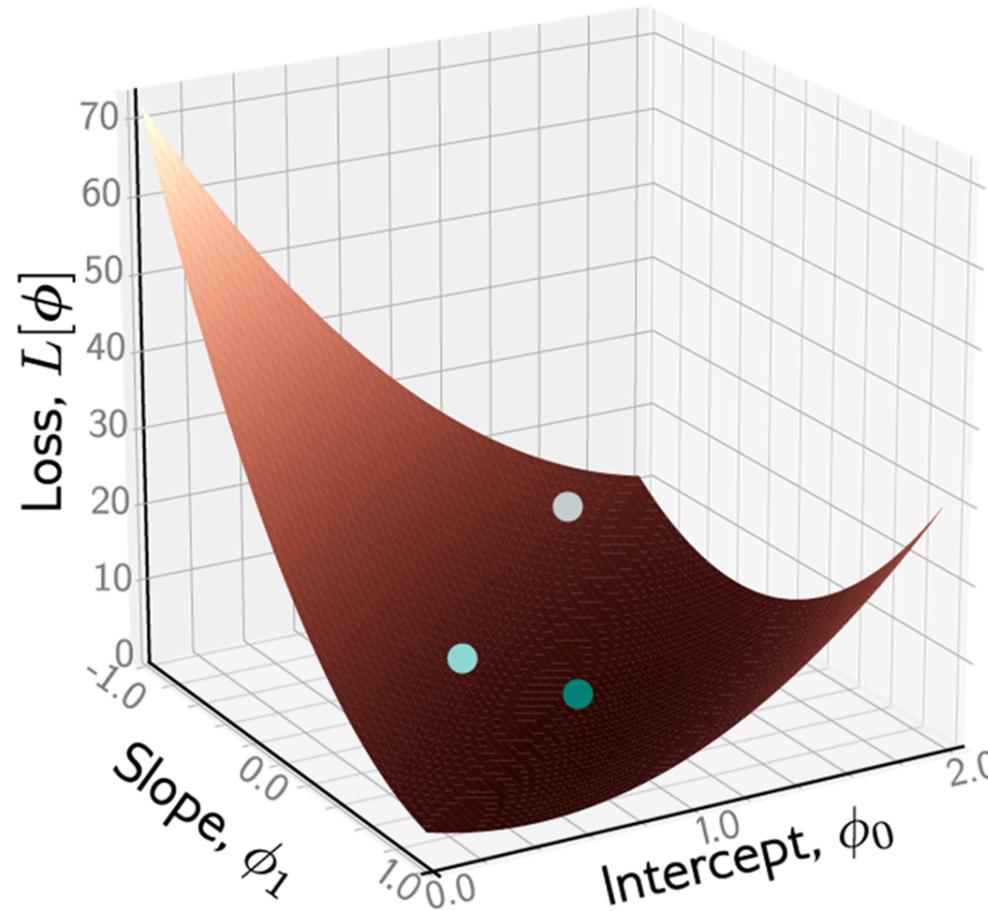
$$\begin{aligned} L[\phi] &= \sum_{i=1}^I (f[x_i, \phi] - y_i)^2 \\ &= \sum_{i=1}^I (\phi_0 + \phi_1 x_i - y_i)^2 \end{aligned}$$



En este caso, la función de pérdida es el **error cuadrático**.

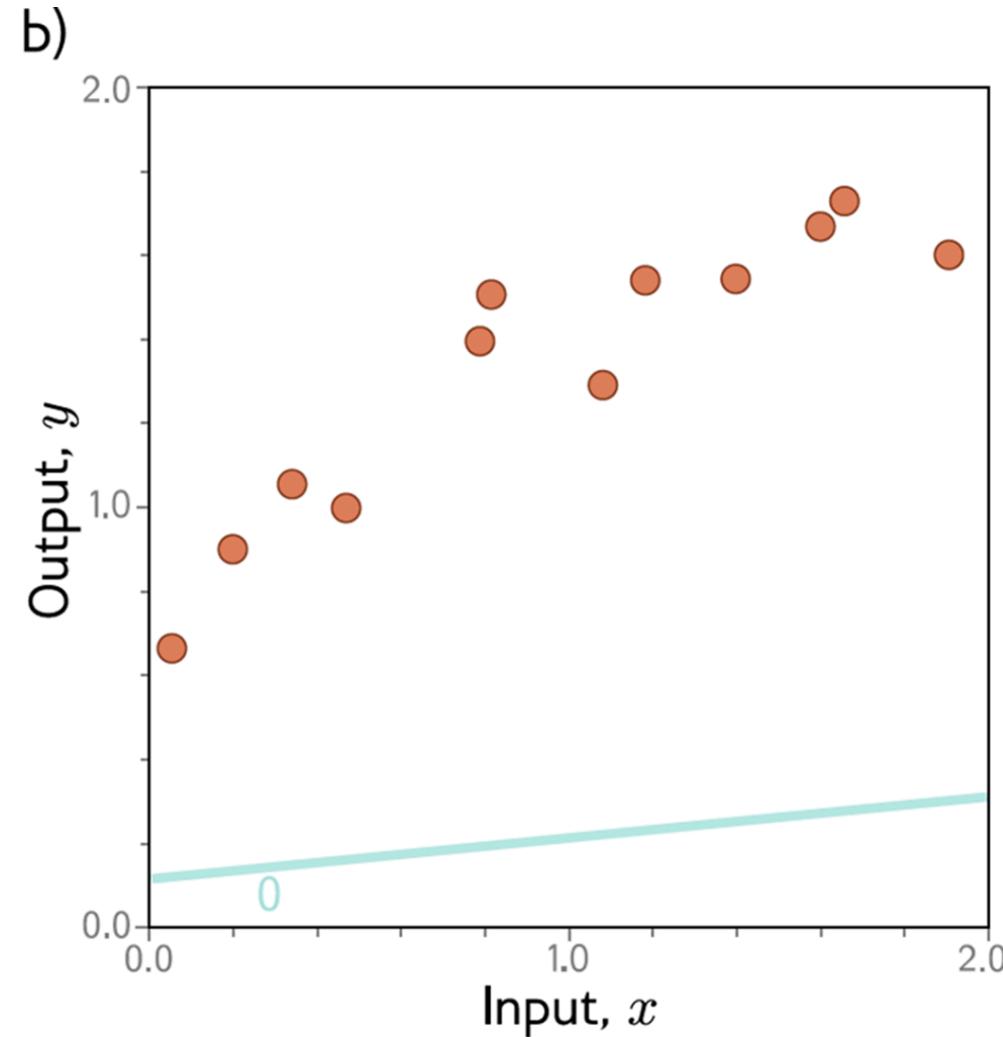
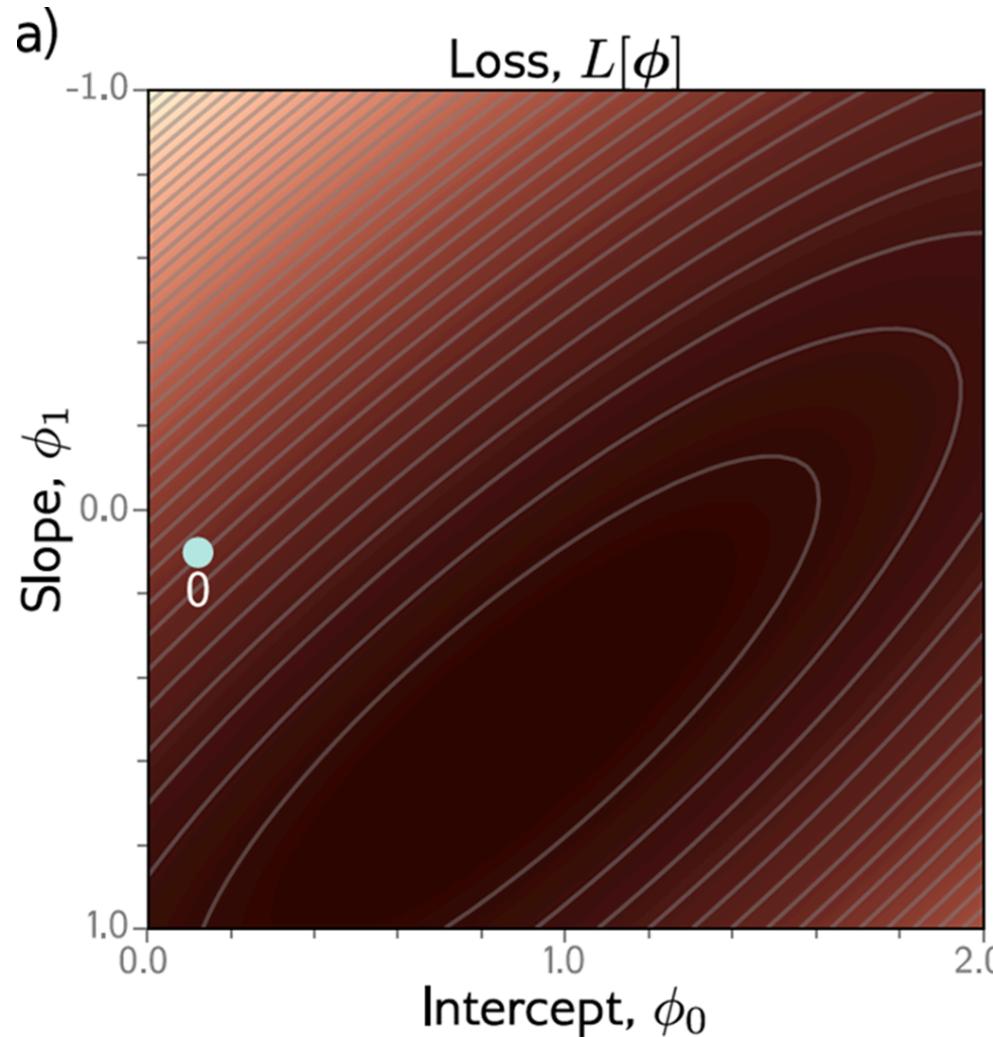


# Modelo de regresión de una dimensión



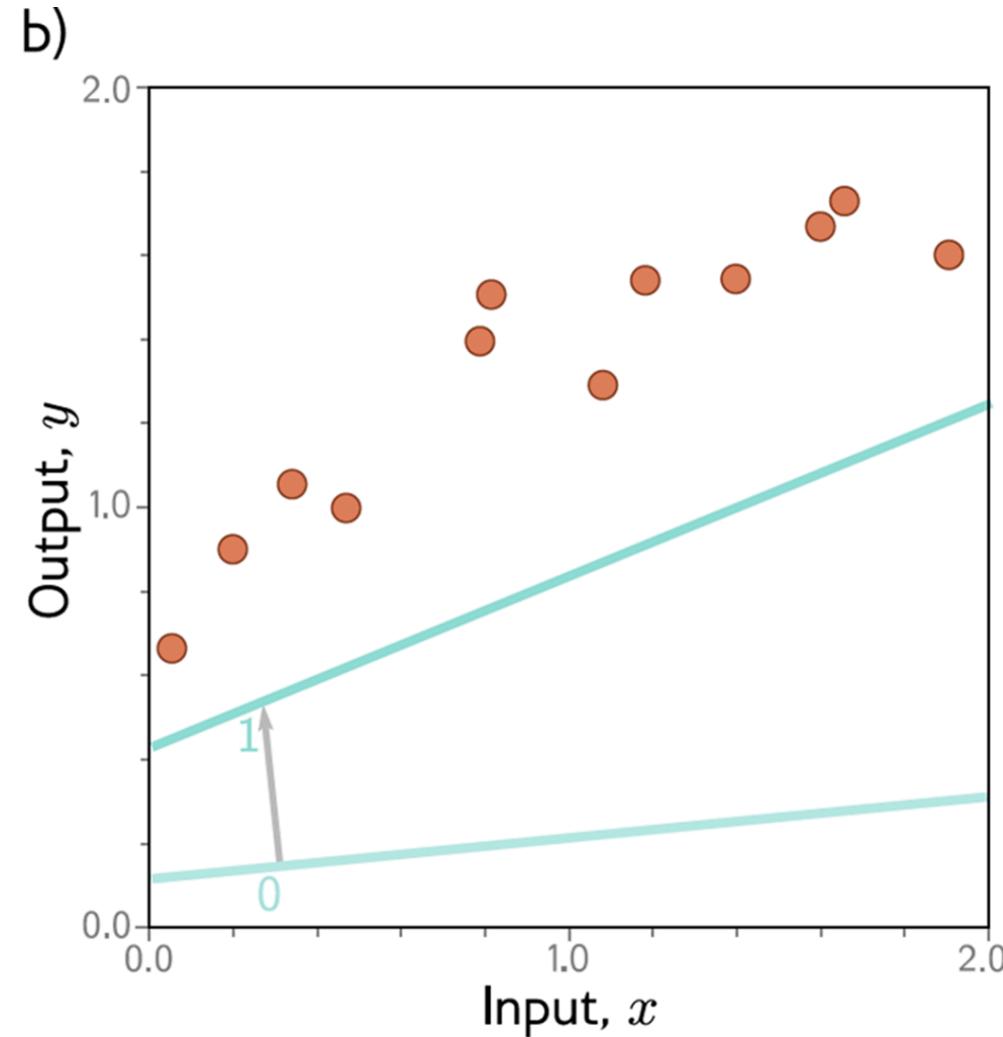
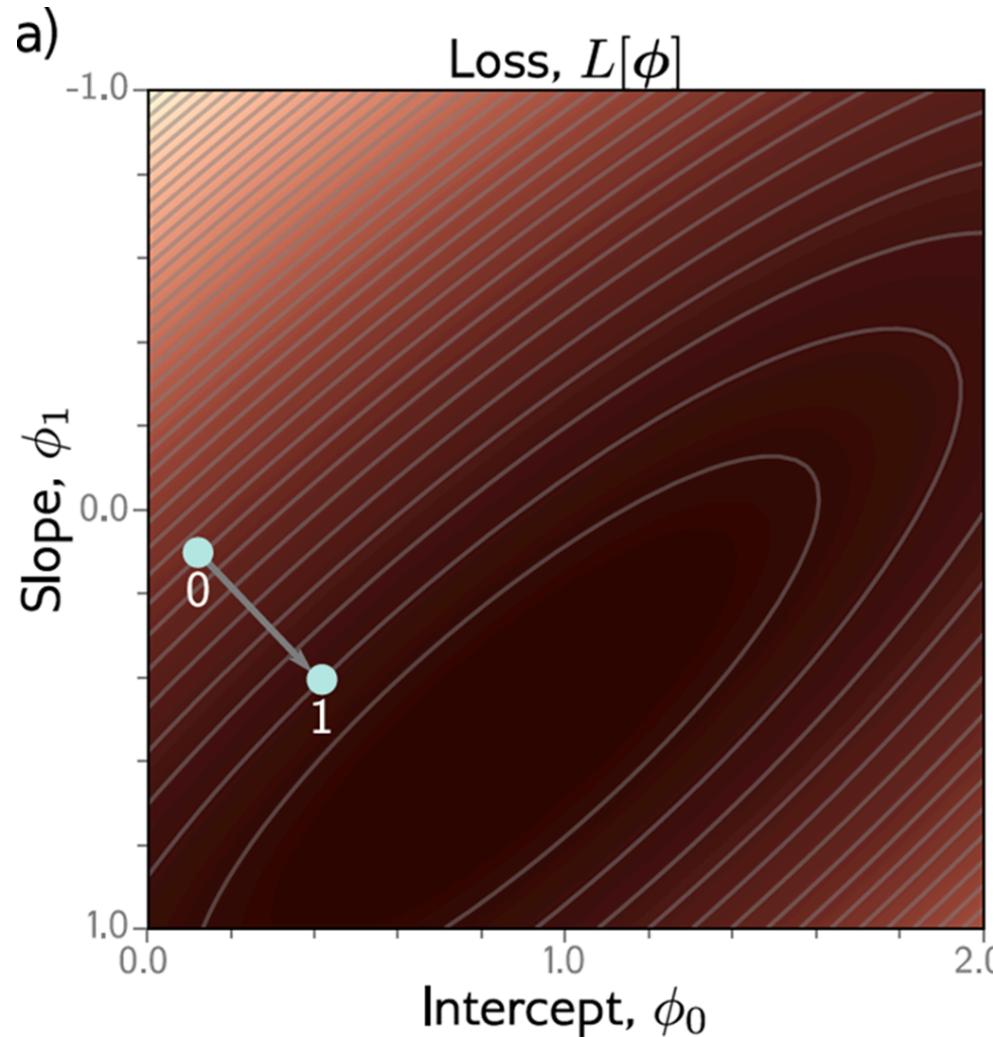


# Modelo de regresión de una dimensión



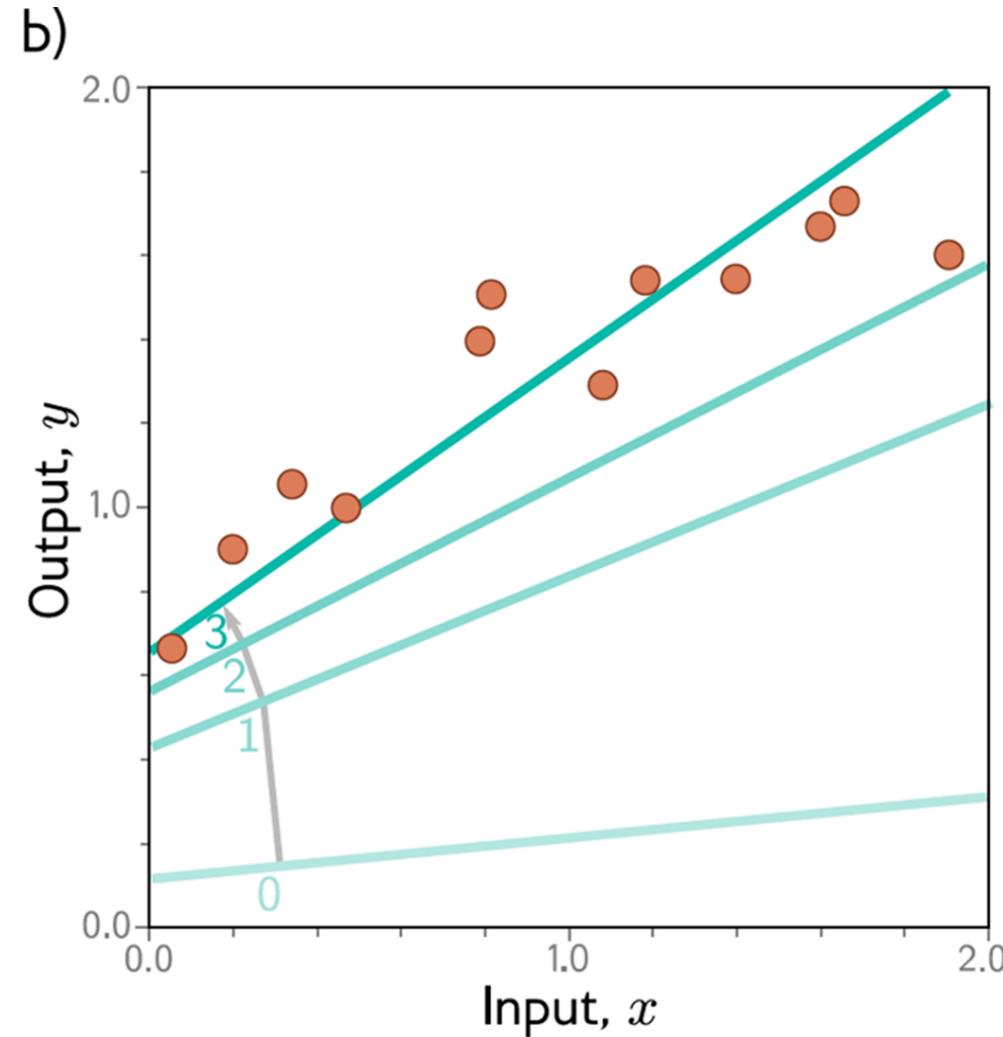
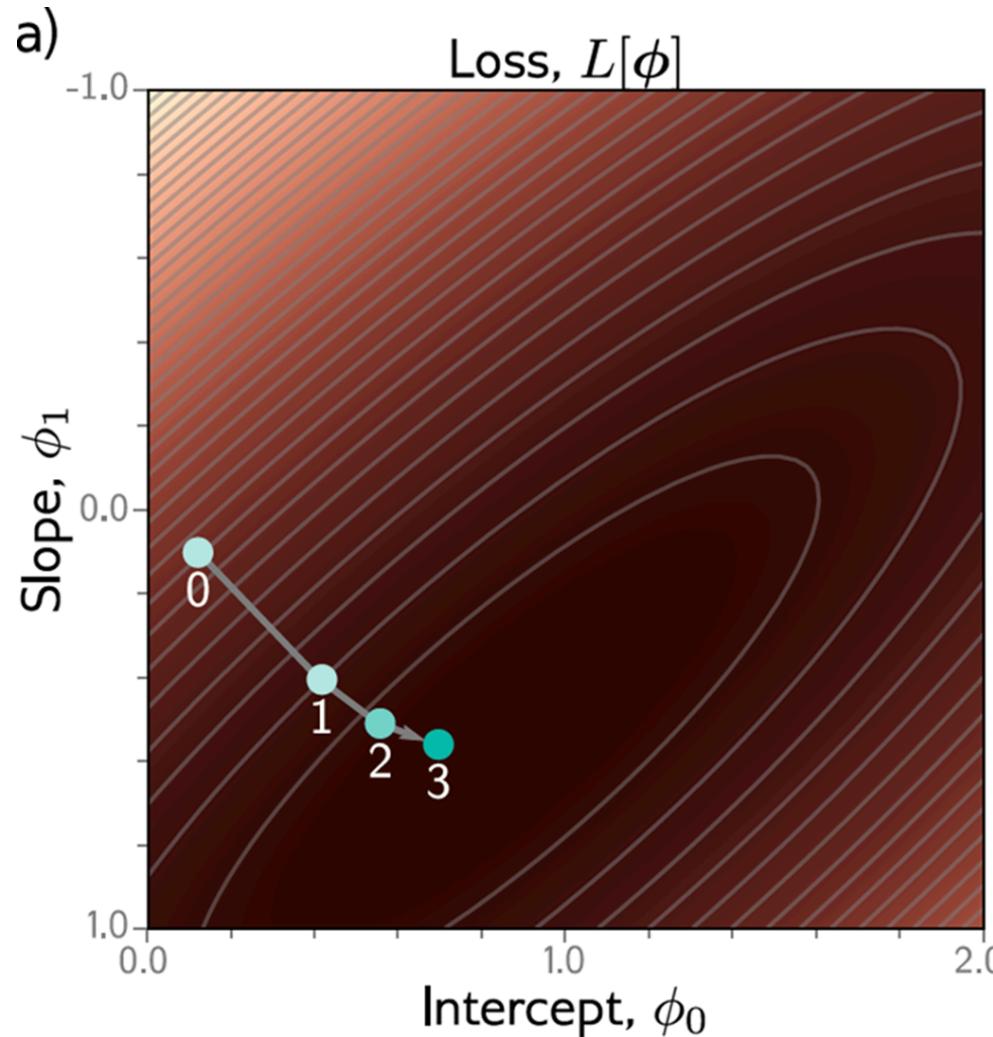


# Modelo de regresión de una dimensión



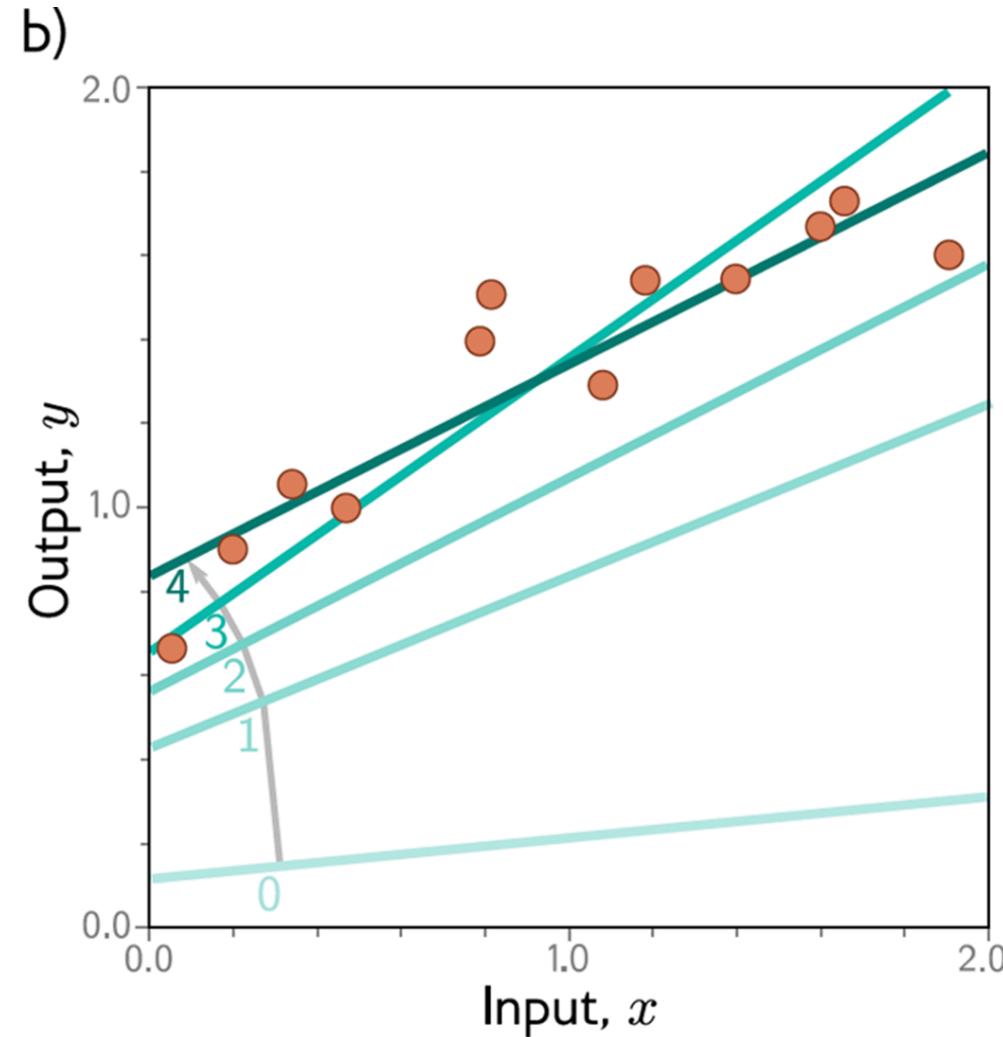
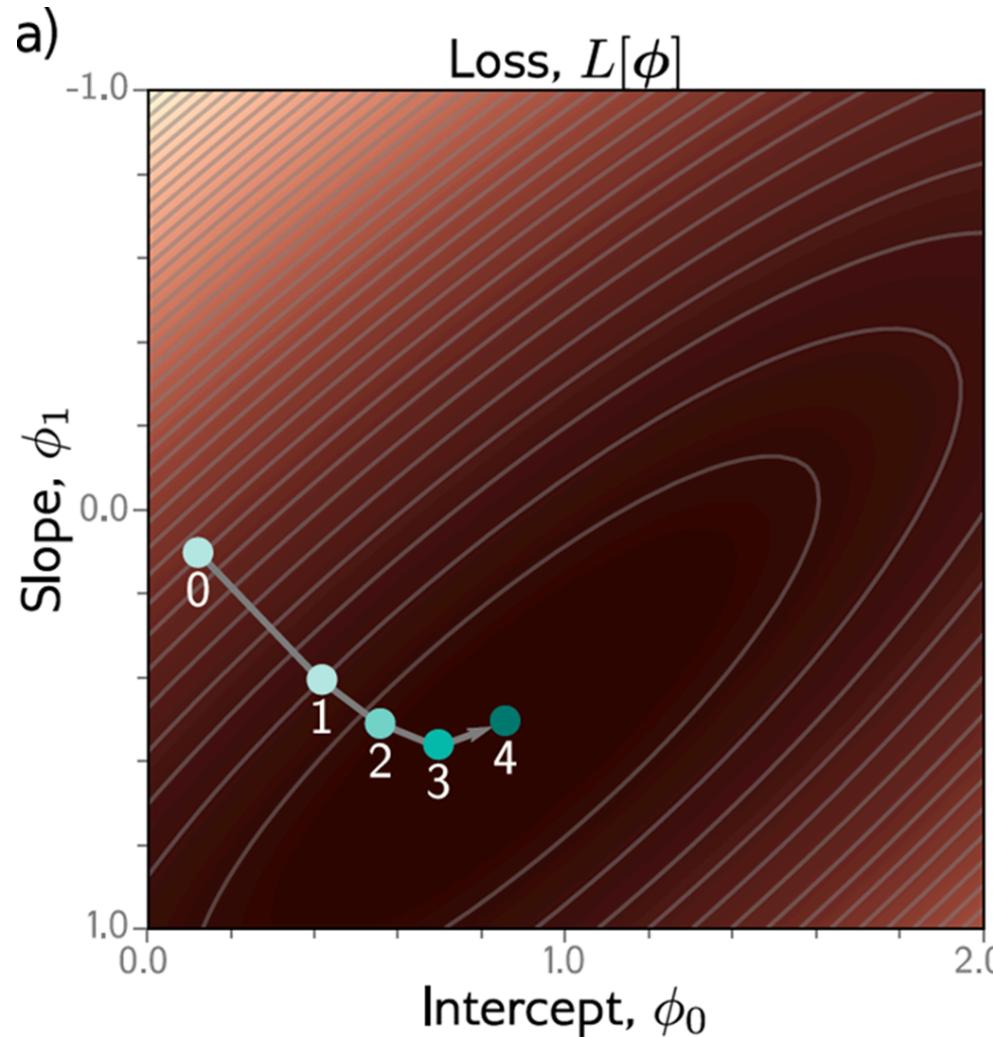


# Modelo de regresión de una dimensión



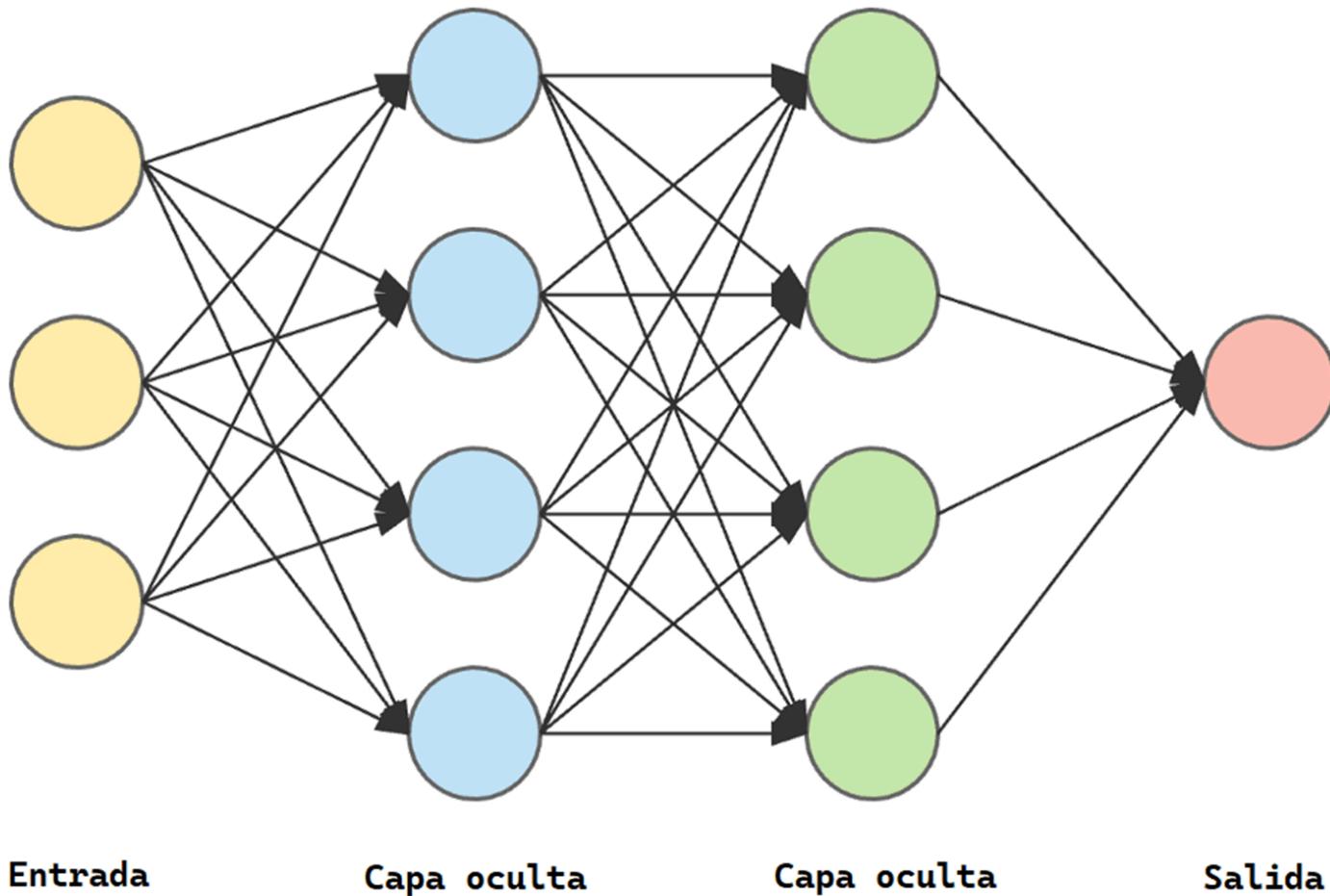


# Modelo de regresión de una dimensión





# Red neuronal perceptrón multicapa (MLP)

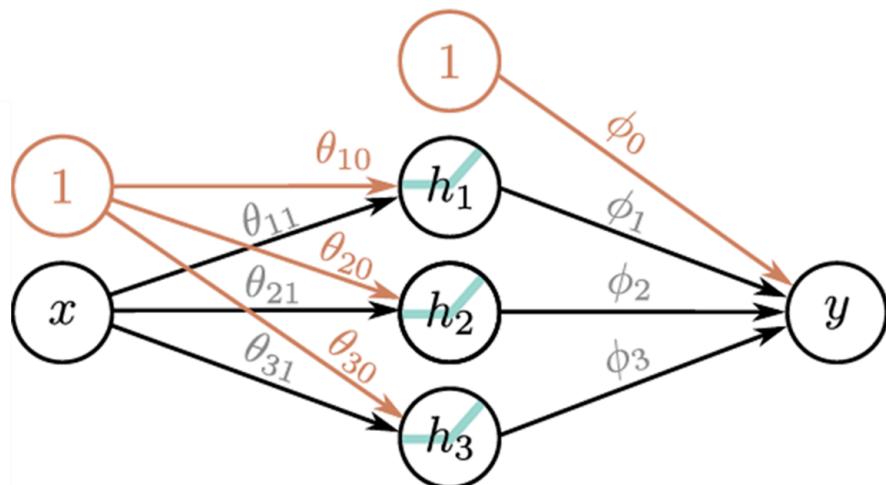


# Definimos Matemáticamente

## ◆ Ecuaciones de la Red Neuronal:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

$$h_1 = a[\theta_{10} + \theta_{11}x]$$



? ¿Qué tiene esta red neuronal?

- # Entradas → 1
- # Salidas → 1
- # Unidades ocultas → 3
- # Capas ocultas → 1
- # Bias → 4 (una por cada unidad en capa oculta y salida)

# Cálculo del Número de Parámetros

📌 Fórmula General para MLP con  $L$  capas ocultas:

$$\text{Parámetros} = \sum_{l=1}^L (\text{Neuronas}_{l-1} \times \text{Neuronas}_l + \text{Bias}_l) + (\text{Neuronas}_L \times \text{Salidas}) + \text{Bias}_{\text{Salida}}$$

📊 Aplicación a nuestra red (1 capa oculta, 3 neuronas):

$$(1 \times 3) + 3 + (3 \times 1) + 1 = 8$$

📊 Ejemplo con 2 capas ocultas (3 y 4 neuronas):

$$(1 \times 3) + 3 + (3 \times 4) + 4 + (4 \times 1) + 1 = 22$$

📌 Nota: Los **parámetros** son el conjunto de **pesos** y **sesgos**, y su ajuste define el aprendizaje de la red. Más capas añaden más parámetros y complejidad al modelo.

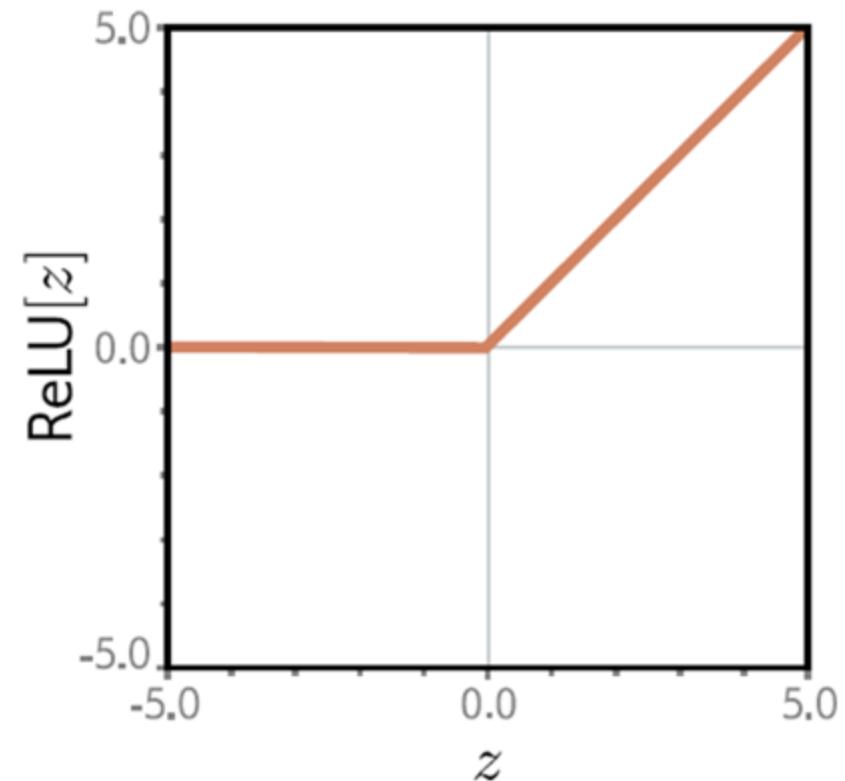
## Definimos Matemáticamente

- ◆ Ecuación de la Red Neuronal:

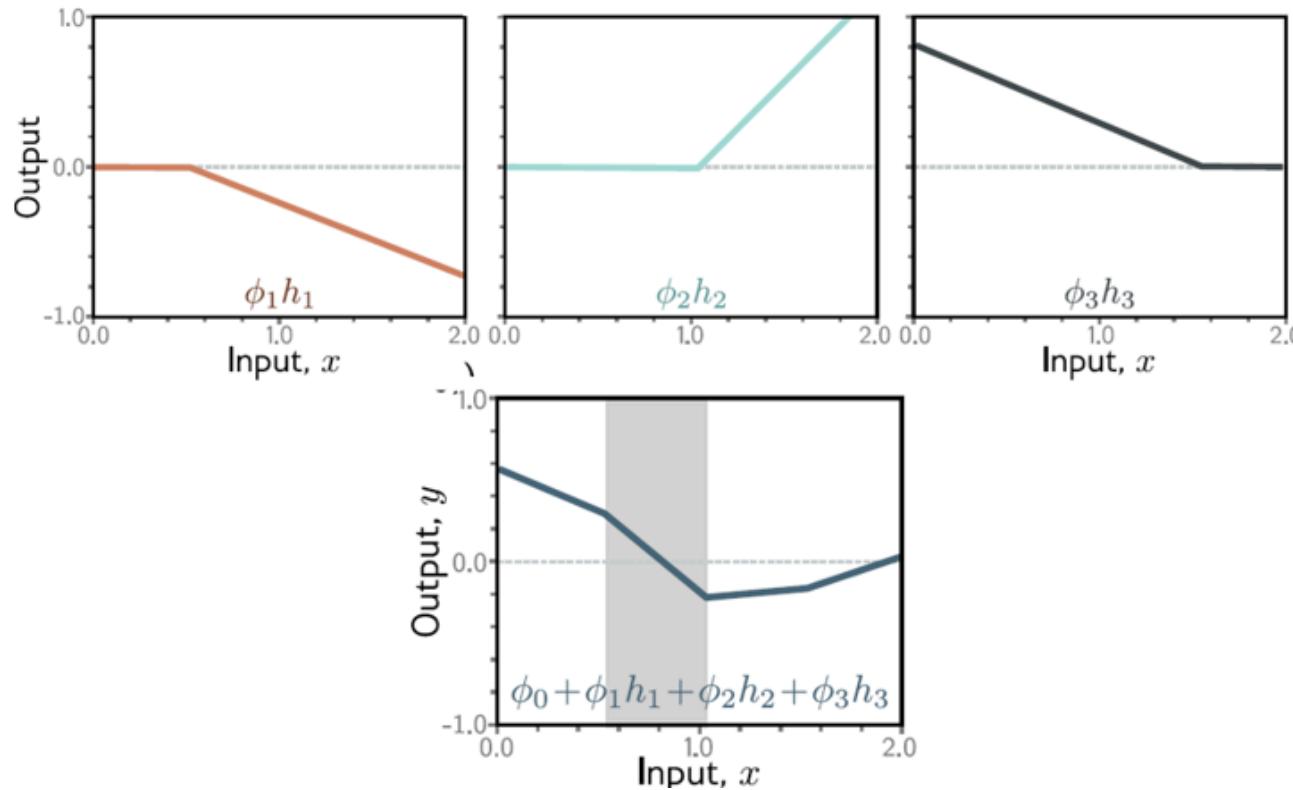
$$y = \phi_0 + \phi_1 a[\theta_{10} + \theta_{11}x] + \phi_2 a[\theta_{20} + \theta_{21}x] + \phi_3 a[\theta_{30} + \theta_{31}x]$$

- ◆ Función de Activación:

$$a[z] = \text{ReLU}[z] = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$



# Definimos Matemáticamente



$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

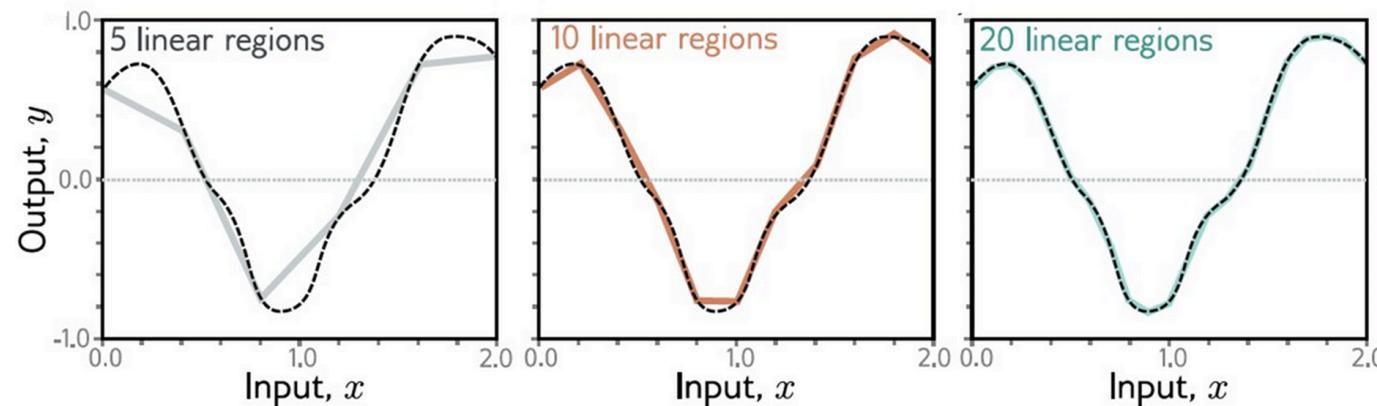
$$h_1 = a[\theta_{10} + \theta_{11}x]$$

## ◆ Generalizando...

$$h_d = a[\theta_{d0} + \theta_{d1}x] \quad y = \phi_0 + \sum_{d=1}^D \phi_d h_d$$

Por el teorema de aproximación general:

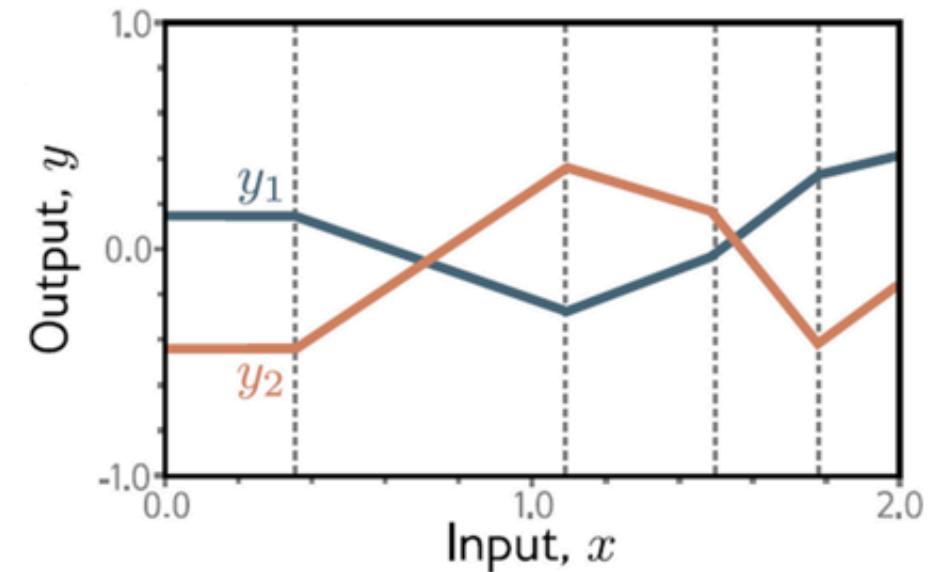
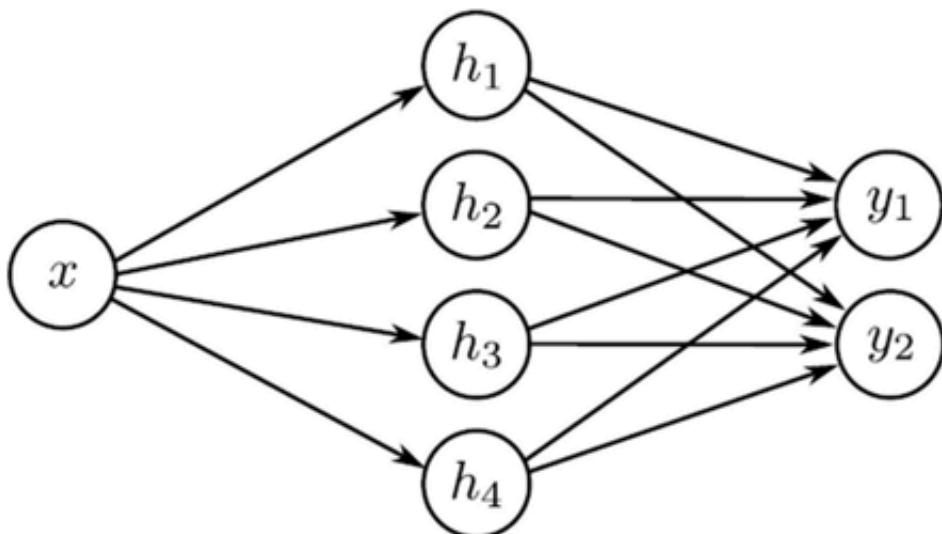
Se puede aproximar cualquier función continua con una composición de funciones lineales.



## Red Neuronal con Varias Salidas

$$y_1 = \phi_{10} + \phi_{11}h_1 + \phi_{12}h_2 + \phi_{13}h_3 + \phi_{14}h_4$$

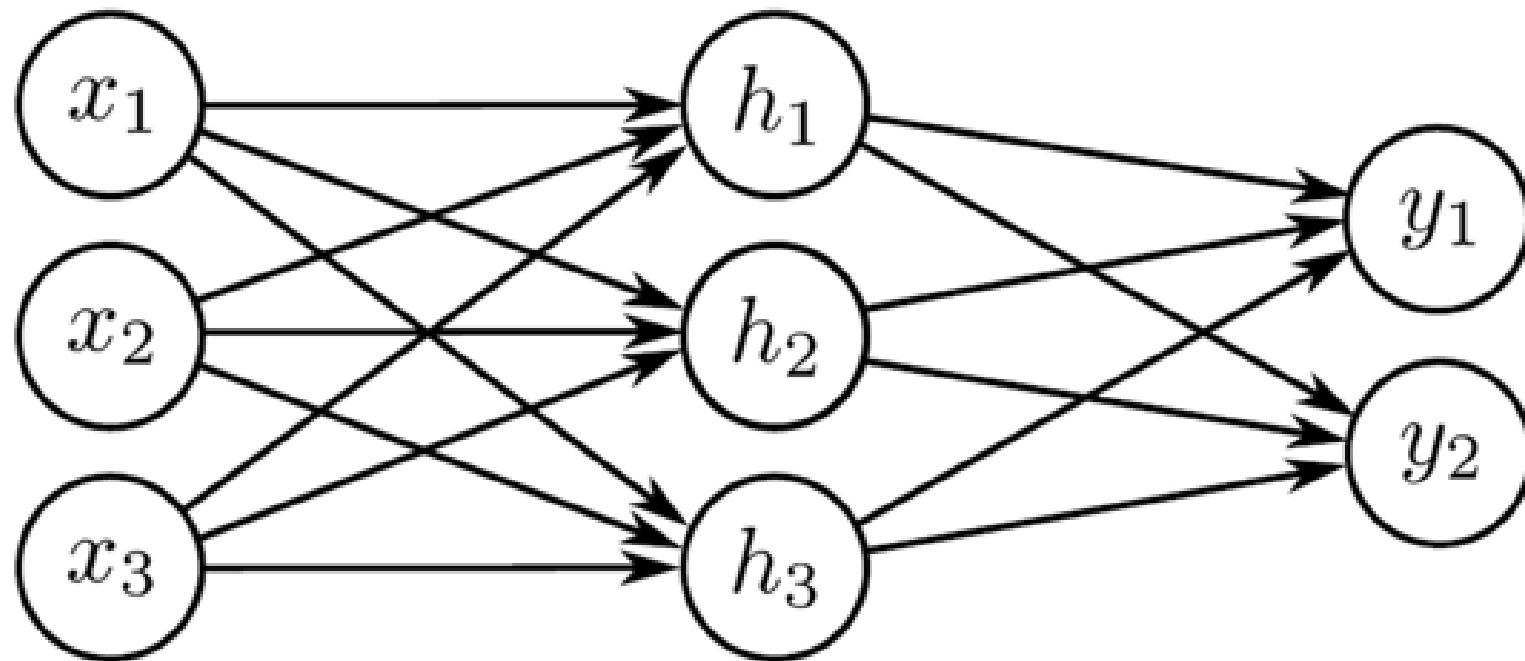
$$y_2 = \phi_{20} + \phi_{21}h_1 + \phi_{22}h_2 + \phi_{23}h_3 + \phi_{24}h_4$$



## Red Neuronal con Varias Salidas y Entradas

$$h_d = a \left[ \theta_{d0} + \sum_{i=1}^{D_i} \theta_{di} x_i \right]$$

$$y_j = \phi_{j0} + \sum_{d=1}^D \phi_{jd} h_d$$





# El Proceso/Arquitectura Feed Forward

## ⚡ ¿Cómo Funciona?

El término "feed forward" significa que la información fluye hacia adelante, desde la entrada hasta la salida, sin retroalimentación:

- 1** Los datos entran por la primera capa.
- 2** Cada neurona calcula su activación y la pasa a la siguiente capa.
- 3** La información solo avanza, nunca retrocede.
- 4** La última capa produce la predicción final.

# Inicialización de Pesos en Redes Neuronales

## ¿Por qué es importante?

Una mala inicialización de los pesos puede hacer que la red:

- Aprenda lentamente.
- Se quede atascada en valores malos.
- No aprenda en absoluto.

## Inicialización Aleatoria Pequeña

- Se asignan valores pequeños y aleatorios cercanos a 0.
- Evita que todas las neuronas aprendan lo mismo.

$$\theta \sim \mathcal{N}(0, 1)$$

## Xavier (Glorot)

- Diseñada para activaciones **Sigmoide** y ✓ **Tanh**.
- Evita que los gradientes desaparezcan o exploten.

$$\theta \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}\right)$$

## He Initialization

- Óptima para **ReLU**.
- Usa una varianza mayor para evitar la desaparición del gradiente.

$$\theta \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right)$$



## Preguntas Clave:

- 1** ¿Cómo se entran estos parámetros?
- 2** ¿Qué son las iteraciones?
- 3** ¿Cuál era el problema hace años y cómo se solucionó?



## Próximo Paso:

 Ahora veremos cómo funciona el proceso de entrenamiento.

# El inicio del Deep Learning



# Del MLP al Deep Learning 🚀

El Deep Learning revoluciona las redes neuronales al utilizar **4 o más capas ocultas** 🧠, superando la estructura básica del MLP tradicional. Esta profundidad permite que la red procese información en niveles cada vez más abstractos ⚡, transformando su capacidad de aprendizaje.

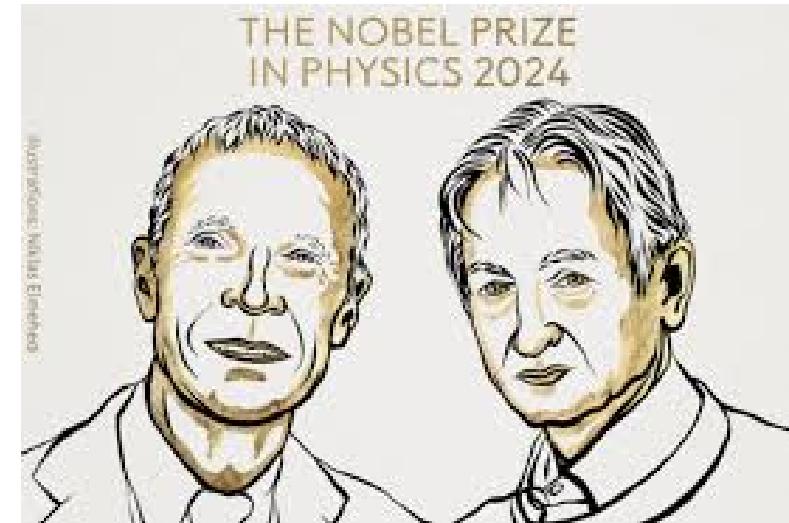
Cada capa construye sobre las anteriores de manera progresiva 🎯: desde **características básicas** hasta **patrones complejos** 🔍, permitiendo abordar tareas sofisticadas como **visión por computador**, **procesamiento de lenguaje** y **análisis financiero** 💰.

# Los pioneros del Deep Learning



Los años 80 sentaron las bases teóricas que harían posible el Deep Learning moderno:

- 🎯 1982: **John Hopfield** revoluciona el campo introduciendo las **redes recurrentes**, demostrando que las redes neuronales podían tener memoria y procesar secuencias temporales.
- 💫 1986: **Geoffrey Hinton** desarrolla el algoritmo de **backpropagation**, resolviendo el problema fundamental de cómo entrenar redes profundas de manera eficiente.





# Backpropagation: El Motor del Aprendizaje

El **backpropagation** es el algoritmo que permite a las redes neuronales **aprender** de sus errores, ajustando sus pesos para mejorar sus predicciones.

## Conceptos Clave

- Calcula cómo cada peso contribuye al error total.
- Propaga el error desde la salida hacia atrás.
- Actualiza los pesos para minimizar el error.



# Matemática del Backpropagation

**1** Forward Pass: Calcular la predicción y el error

$$E = \frac{1}{2}(y_{real} - y_{pred})^2$$

**2** Backward Pass: Calcular gradientes

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_{pred}} \cdot \frac{\partial y_{pred}}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}}$$

**3** Actualización: Ajustar pesos

$$w_{nuevo} = w_{viejo} - \eta \frac{\partial E}{\partial w}$$

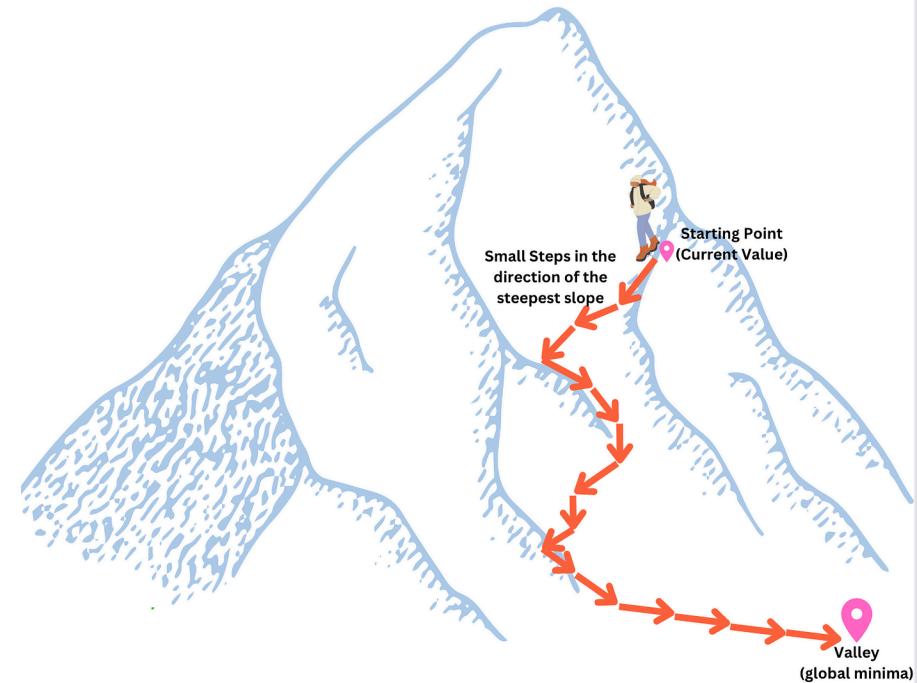
Donde  $\eta$  es la tasa de aprendizaje (learning rate).

# La Caída del Gradiente



Imagina que el error de nuestra red neuronal es como una montaña con muchos valles 🌫️. Cuanto más alto estamos, más grande es el error. Nuestro objetivo es llegar al punto más bajo posible, donde el error sea mínimo.

La **caída del gradiente** funciona como una pelota que soltamos en esta montaña ⏪: naturalmente rodará hacia abajo, tomando el camino más empinado en cada punto. El **learning rate** determina qué tan rápido rueda esta pelota: si va muy rápido podría saltar el valle más profundo, si va muy lento tardará demasiado en llegar.



# Backpropagation y Gradiente: El Dúo del Aprendizaje

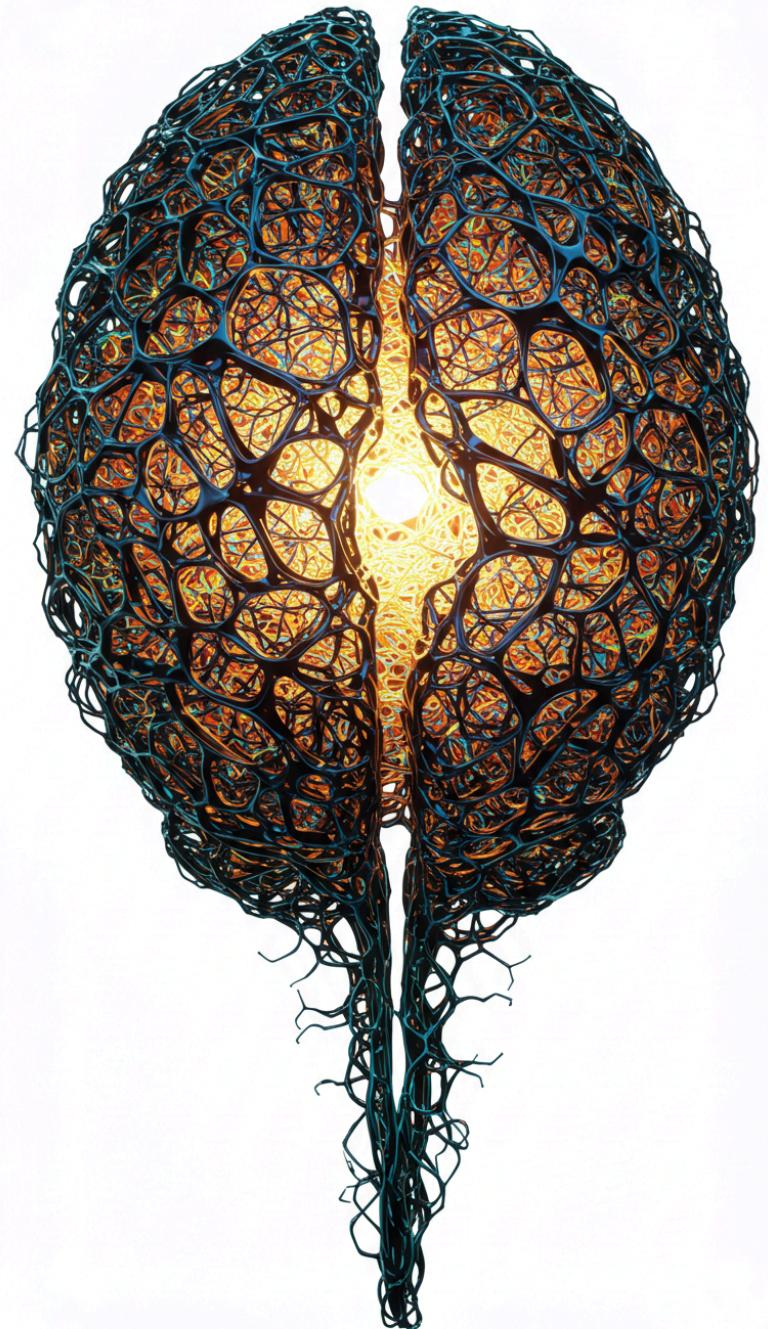


El **backpropagation** y la **caída del gradiente** trabajan juntos como un equipo perfecto en el Deep Learning 🤝. Mientras el backpropagation calcula cómo cada neurona contribuyó al error, propagando esta información desde la última capa hasta la primera, la caída del gradiente usa esta información para ajustar los pesos en la dirección correcta.

## La Fórmula del Éxito 📊

1. **Backpropagation:** Calcula  $\frac{\partial E}{\partial w}$  (qué tanto afecta cada peso al error).
2. **Gradiente:** Ajusta  $w = w' - \eta \frac{\partial E}{\partial w}$  (actualiza los pesos).
3. **Repetir:** Itera hasta encontrar los pesos óptimos.

# Problemas y soluciones para entrenar redes neuronales





# Desafíos en el Entrenamiento

El entrenamiento de redes neuronales profundas presenta desafíos únicos que pueden afectar significativamente su rendimiento. El más crítico es el problema del **gradiente desvaneciente**.



## Vanishing Gradient

Los gradientes son la clave del aprendizaje en redes profundas. Cuando una red tiene muchas capas, estos gradientes pueden volverse extremadamente pequeños, causando que:

- Las capas más cercanas a la entrada **dejan de aprender**.
- El entrenamiento se **estanca**.
- La red no logra capturar patrones complejos.



# Desafíos en el Entrenamiento

## Exploding Gradient

Mientras que el vanishing gradient hace que el aprendizaje se detenga, el exploding gradient causa el efecto contrario. Los valores crecen exponencialmente, generando un efecto dominó que:

- Causa **inestabilidad** en el entrenamiento.
- Puede llevar a **desbordamiento numérico**.
- Resulta en **pesos no óptimos**.



# Soluciones a los Problemas de Gradientes

A lo largo de los años, la comunidad de deep learning ha desarrollado técnicas innovadoras para abordar estos desafíos. Cada solución ataca el problema desde un ángulo diferente.



## Técnicas Sugeridas

Las herramientas más efectivas en nuestro arsenal incluyen:

- **ReLU**: Evita la saturación que causa el vanishing gradient.
- **Batch Normalization**: Mantiene las señales fuertes y estables.
- **Skip Connections**: Crean autopistas para la información.



# El Problema del Overfitting

Imagina un estudiante que memoriza perfectamente las respuestas de exámenes anteriores, pero no puede resolver problemas nuevos. Así funciona el **overfitting**: el modelo se vuelve demasiado específico a los datos de entrenamiento, perdiendo su capacidad de generalización.



## Señales de Overfitting

Las señales de advertencia suelen ser claras y consistentes:

- Rendimiento perfecto en entrenamiento.
- Pobre generalización en validación.
- Predicciones "demasiado confiadas".



## Estrategias de Prevención del Overfitting

La prevención del overfitting es un arte que combina diferentes técnicas. Como un sistema de control y balance, cada estrategia cumple un papel específico:

- El **Dropout** "apaga" neuronas aleatoriamente durante el entrenamiento, forzando a la red a ser más robusta.
- La **Data Augmentation** expone el modelo a más variaciones de los datos.
- El **Early Stopping** nos ayuda a encontrar el punto óptimo de generalización.



# Regularización

La regularización modifica la función de pérdida de la red neuronal añadiendo términos que penalizan la complejidad del modelo. Matemáticamente, si  $L$  es nuestra pérdida original, la regularización añade un término adicional:

$$L_{reg} = L + \lambda R(w)$$



## Tipos Principales

- **L1 (Lasso)**  $\lambda \sum |w_i|$ : Promueve la esparcidad, llevando algunos pesos a cero
- **L2 (Ridge)**  $\lambda \sum w_i^2$ : Penaliza pesos grandes de manera uniforme
- **ElasticNet**  $\lambda_1 \sum |w_i| + \lambda_2 \sum w_i^2$ : Combina las ventajas de L1 y L2

El parámetro  $\lambda$  controla la fuerza de la regularización.



# Hiperparámetros: El Arte del Ajuste

Como un chef ajustando los ingredientes de una receta, el diseño de redes neuronales requiere un delicado balance de hiperparámetros. Su correcta configuración determina el éxito del modelo.



## Hiperparámetros Críticos

Los parámetros fundamentales que debemos considerar son:

- **Learning Rate:** Controla la velocidad de aprendizaje.
- **Batch Size:** Balancea velocidad y estabilidad.
- **Arquitectura:** Define la capacidad del modelo.
- **Regularización:** Previene el sobreajuste.

# Métodos de Selección de Hiperparámetros

La selección de hiperparámetros es un proceso sistemático que requiere un enfoque metodológico riguroso. Existen varios métodos establecidos, cada uno con características distintivas.

## Estrategias Principales

El enfoque manual requiere experiencia previa y comprensión profunda del modelo. Se basa en ajustes iterativos fundamentados en resultados empíricos.

El **Grid Search** implementa una búsqueda sistemática evaluando todas las combinaciones posibles de hiperparámetros en un espacio discreto predefinido. Este método garantiza encontrar el óptimo global dentro del espacio de búsqueda especificado.

# Métodos de Selección de Hiperparámetros

El **Random Search** proporciona una alternativa eficiente al grid search, especialmente en espacios de alta dimensionalidad:

- Explora el espacio de manera más diversa.
- Puede encontrar soluciones óptimas con menos evaluaciones.
- Requiere menos recursos computacionales.

La **Optimización Bayesiana** utiliza un enfoque probabilístico:

- Construye un modelo del espacio de hiperparámetros.
- Optimiza basándose en resultados anteriores.
- Especialmente efectiva para espacios complejos.

# Recursos del Curso

## Plataformas y Enlaces Principales

 GitHub del curso

 [github.com/CamiloVga/IA\\_Aplicada](https://github.com/CamiloVga/IA_Aplicada)

 Asistente IA para el curso

 [Google Notebook LLM](#)