

*Departamento de Física Médica - Centro atómico Bariloche - IB*

# Introducción a las Redes Neuronales

Ariel Hernán Curiale  
[ariel.curiale@cab.cnea.gov.ar](mailto:ariel.curiale@cab.cnea.gov.ar)

Algunos slides fueron adaptados de Fei Fei Li, J.  
Johnson y S. Yeung, cs231n, Stanford 2017.



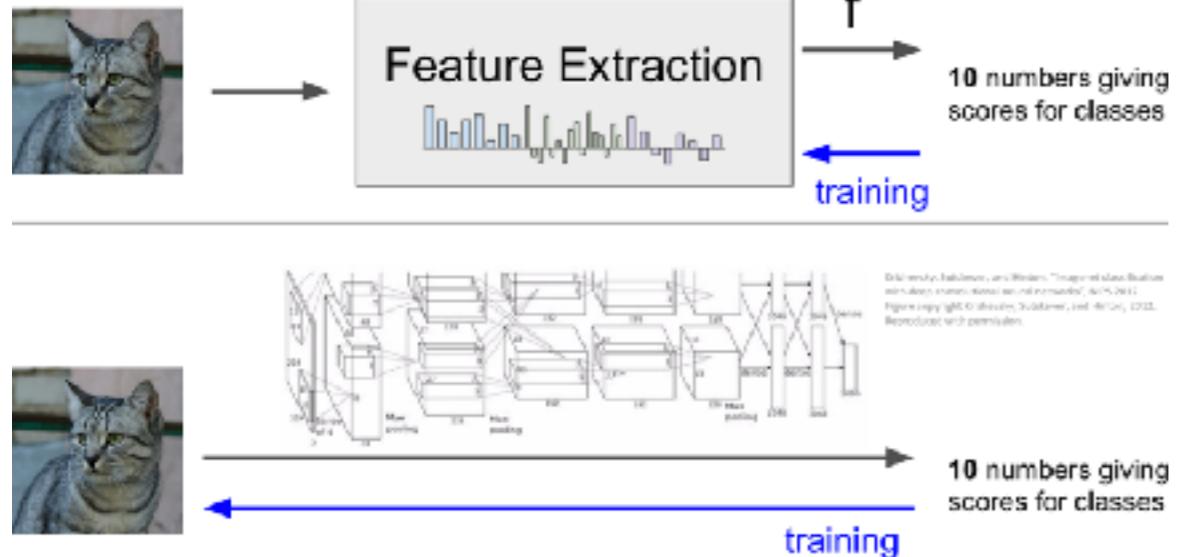
**UNCUYO**  
UNIVERSIDAD  
NACIONAL DE CUYO



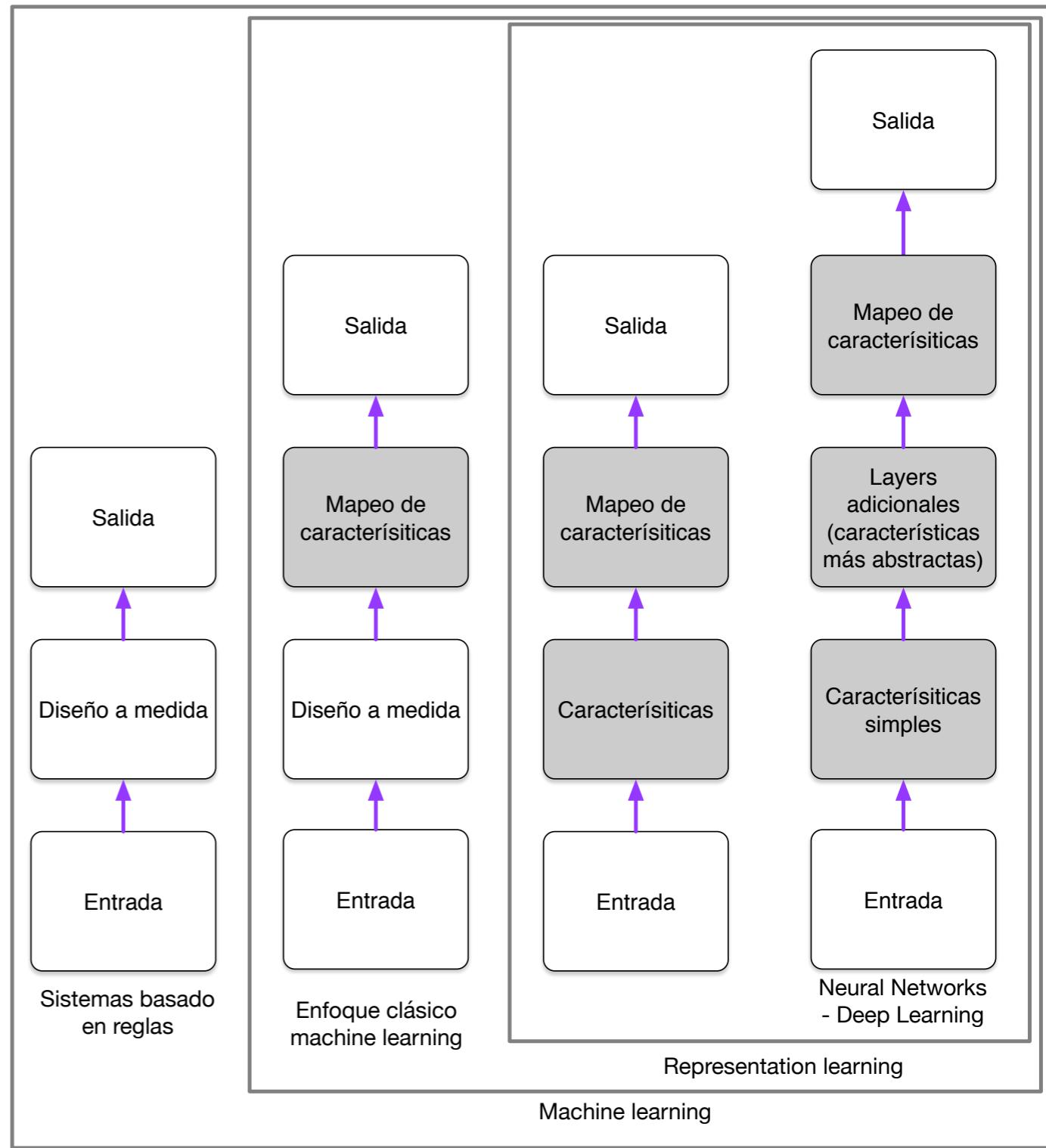
# La última vez vimos

- ❖ Clasificadores lineales  $f(x, W) = Wx$
  - ❖ Distintas funciones de costo que dieron a diferentes clasificadores
    - ❖ SVM  $L = \sum_{j \neq y_i} \max(0, S_j - S_{y_i} + \Delta)$
    - ❖ Softmax  $L = -\log \left( \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \right) = -f_{y_i} + \log \left( \sum_j e^{f_j} \right)$
  - ❖ El rol que cumple el término de regularización
$$\text{loss} = \frac{1}{N} \sum_i L(f(x_i, W), y_i) + \lambda R(W)$$
  - ❖ Optimización: queremos calcular gradientes (BSG - SGD)
$$\nabla_W \text{loss}$$
- Quedo pendiente que ustedes  
calculen el gradiente de Softmax  
para el prob. 1-6 y 2.4
- 

# La última vez vimos



Img. source: Stanford CS231n 2017



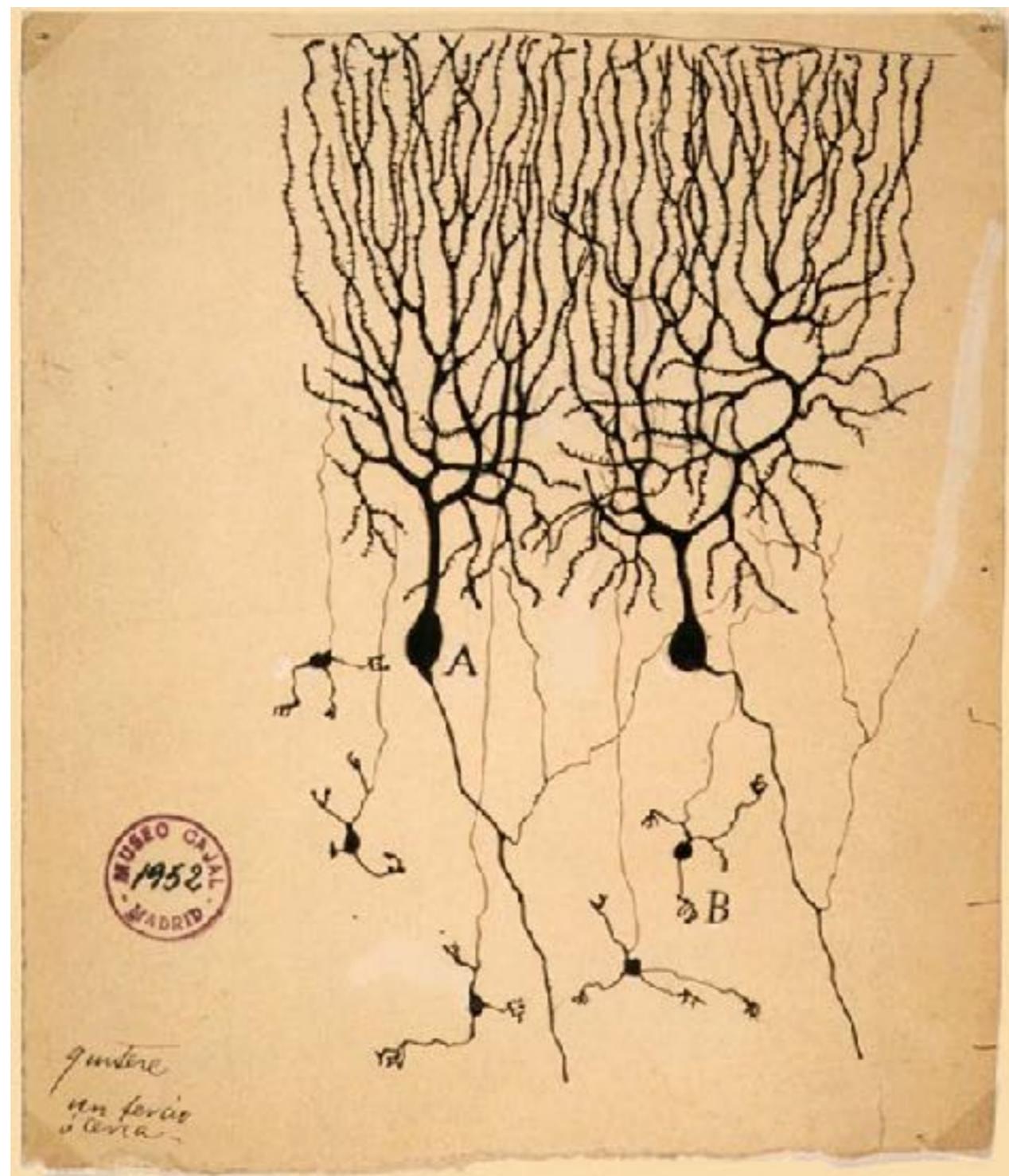
---

# Lo que viene

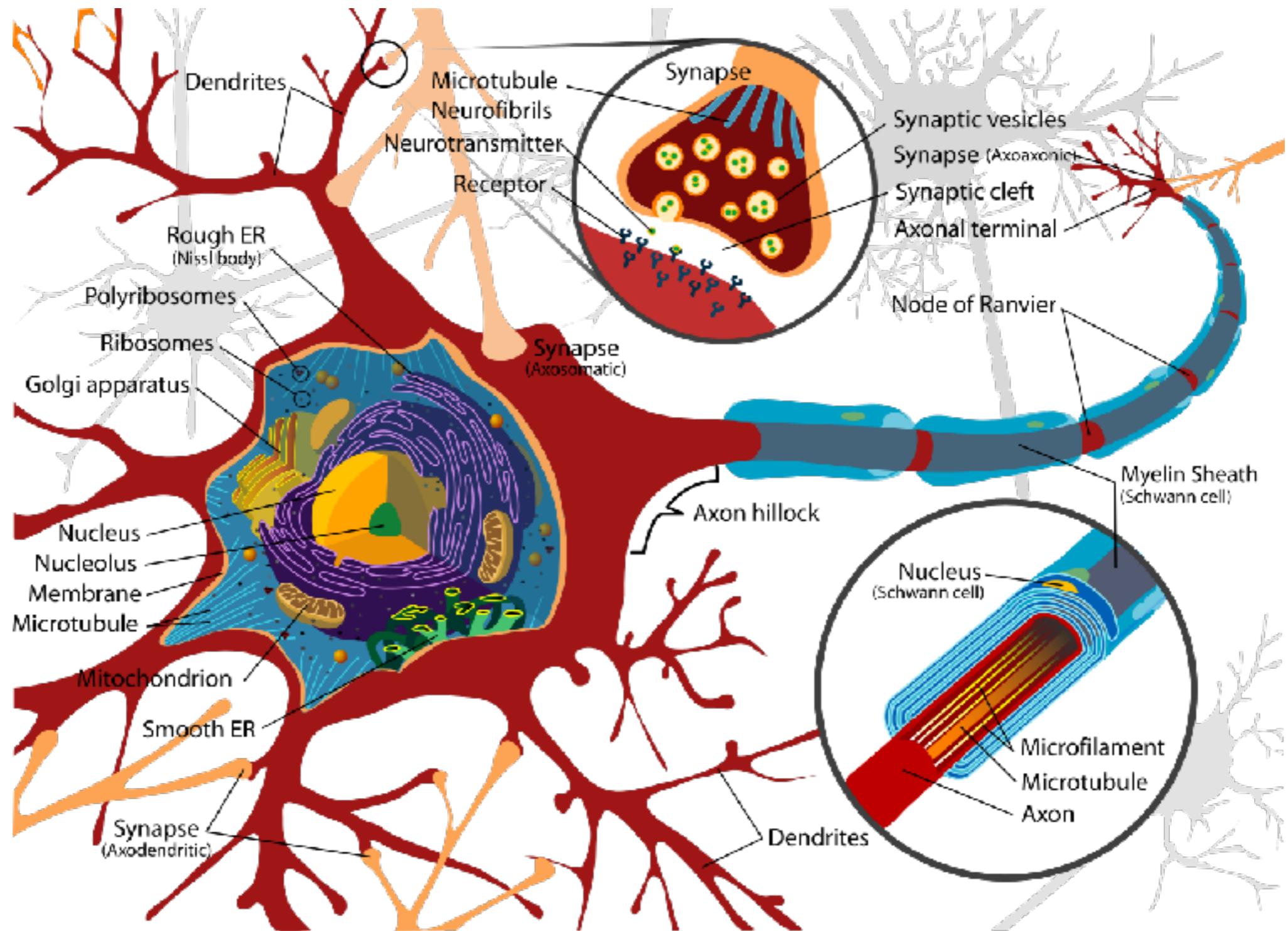
---

- ❖ Perceptron, perceptron multicapa
- ❖ Funciones de activación más comunes (intro)
- ❖ Intuición sobre el algoritmo de aprendizaje
- ❖ Intuición sobre Backpropagation
- ❖ Introducción a Computational Graph
- ❖ Ejemplos

# Redes Neuronales

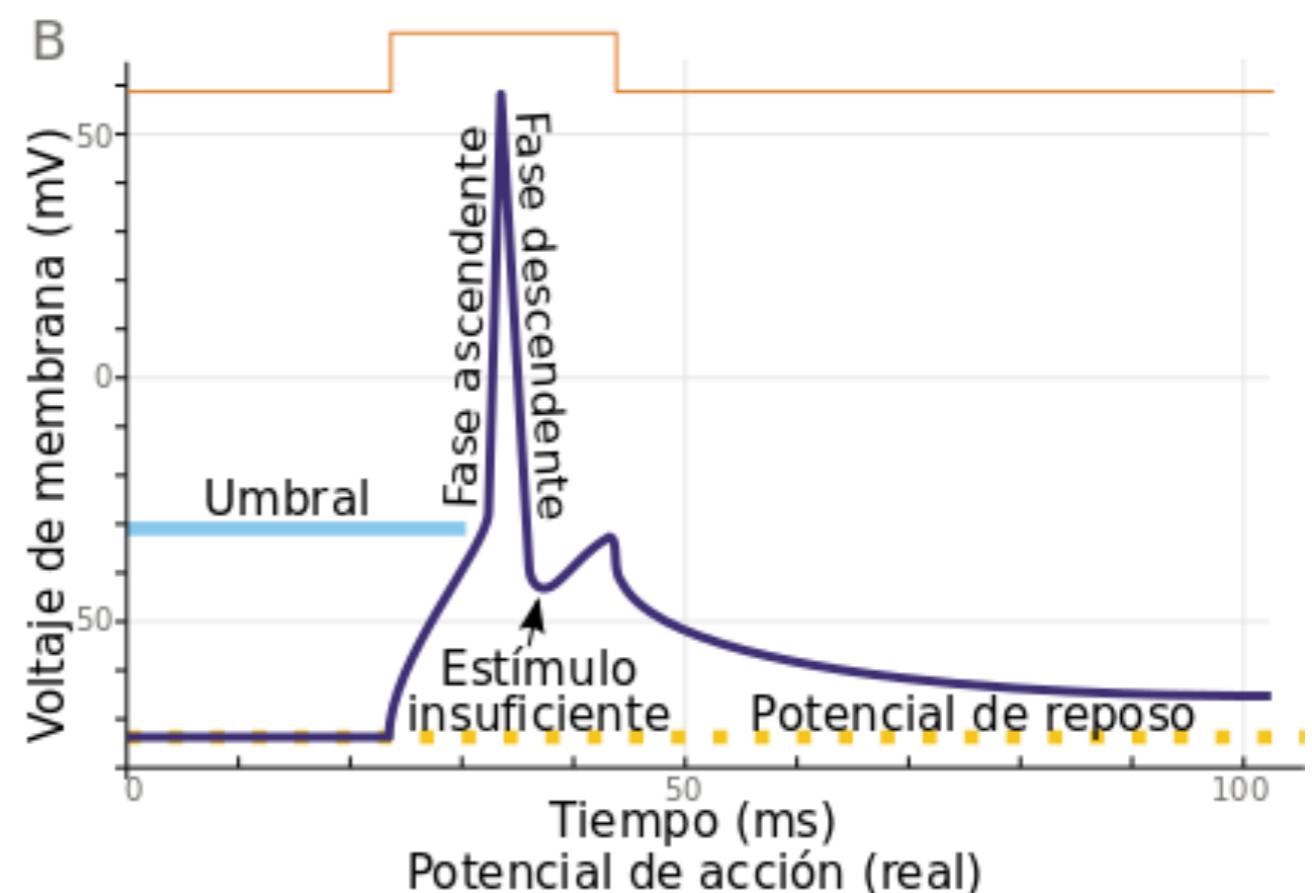
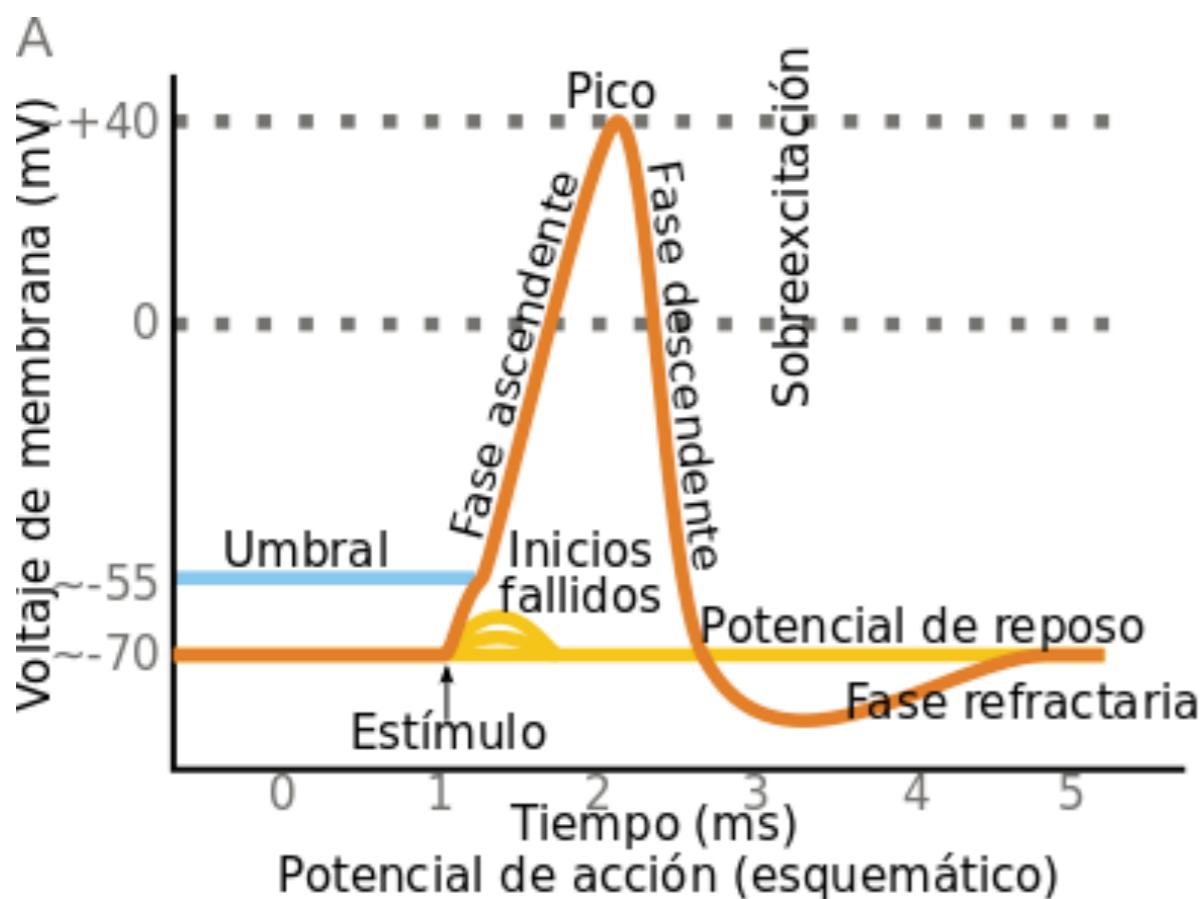


# Redes Neuronales



# Redes Neuronales

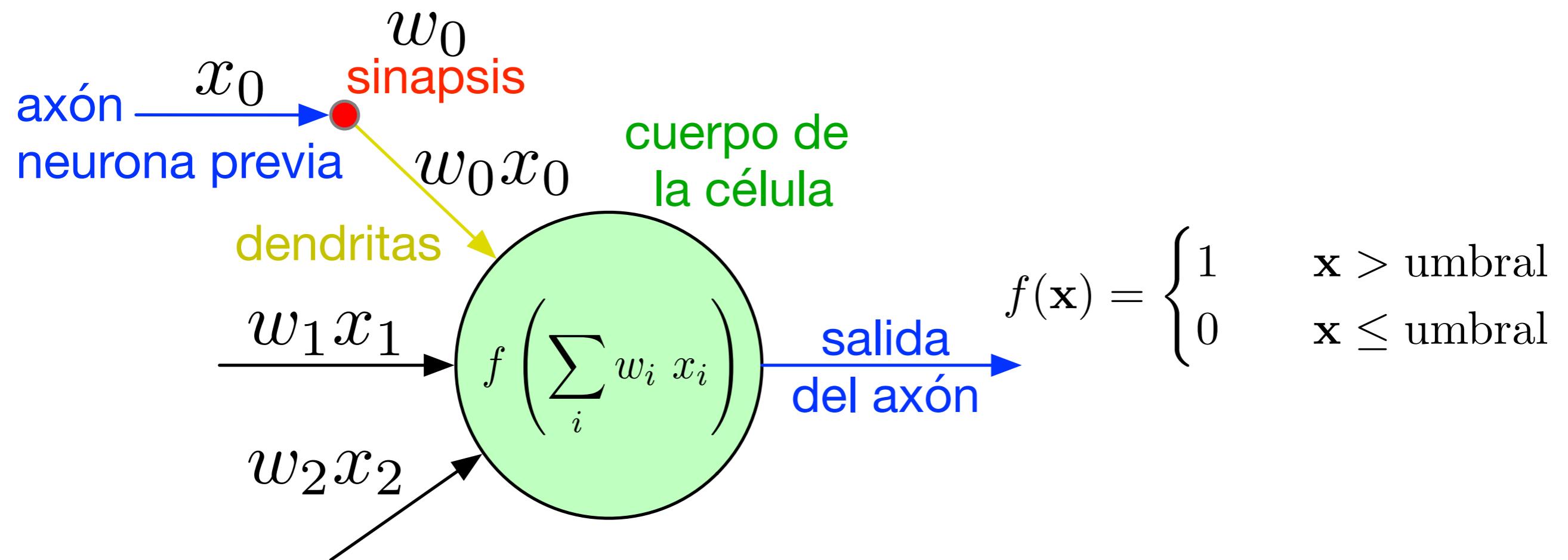
Si el estímulo es mayor que cierto umbral se produce un fenómeno altamente no lineal que se propaga por el axón



Una vez que la perturbación llega a los terminales del axón produce una liberación de neurotransmisores y se produce la sinapsis

# Redes Neuronales

- ❖ McCulloch-Pitts Neuron (1943)

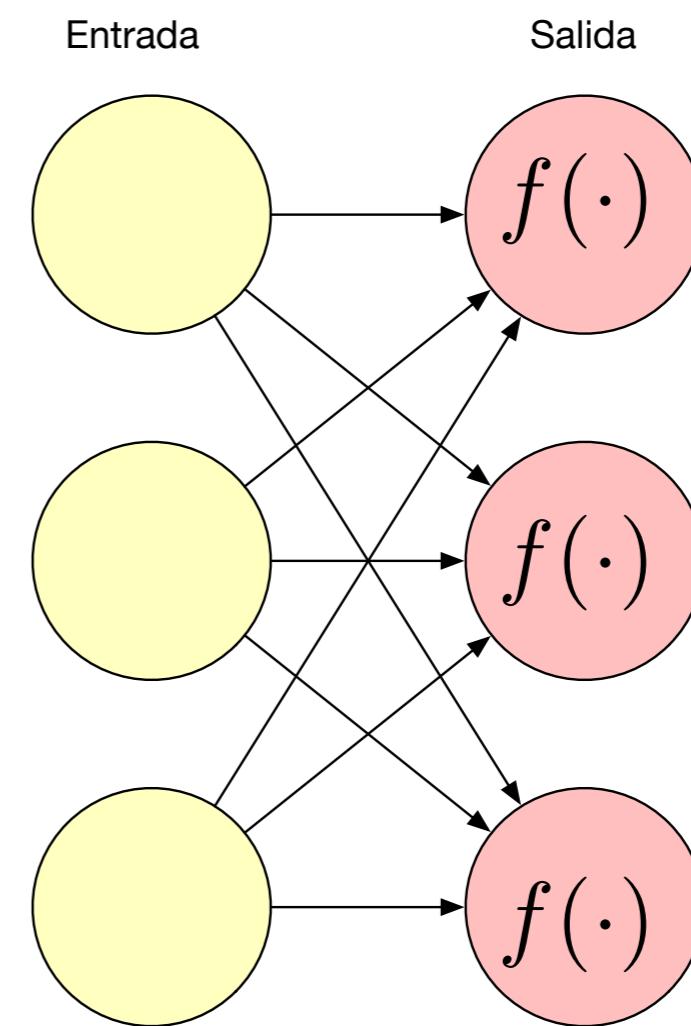
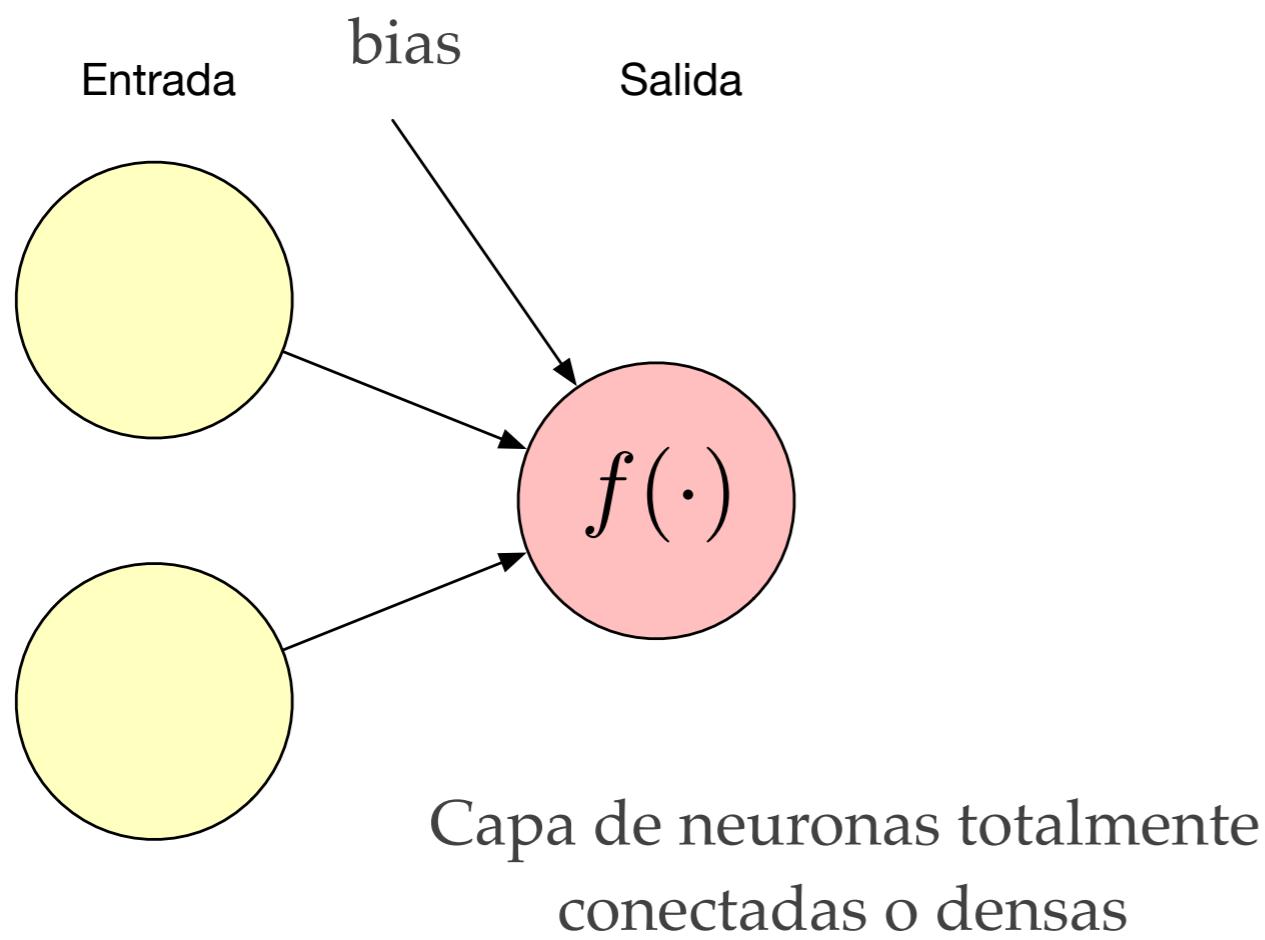


$$f\left(\sum_i w_i x_i - b\right) = \text{sign}\left(\sum_i w_i x_i - b\right)$$

# Redes Neuronales

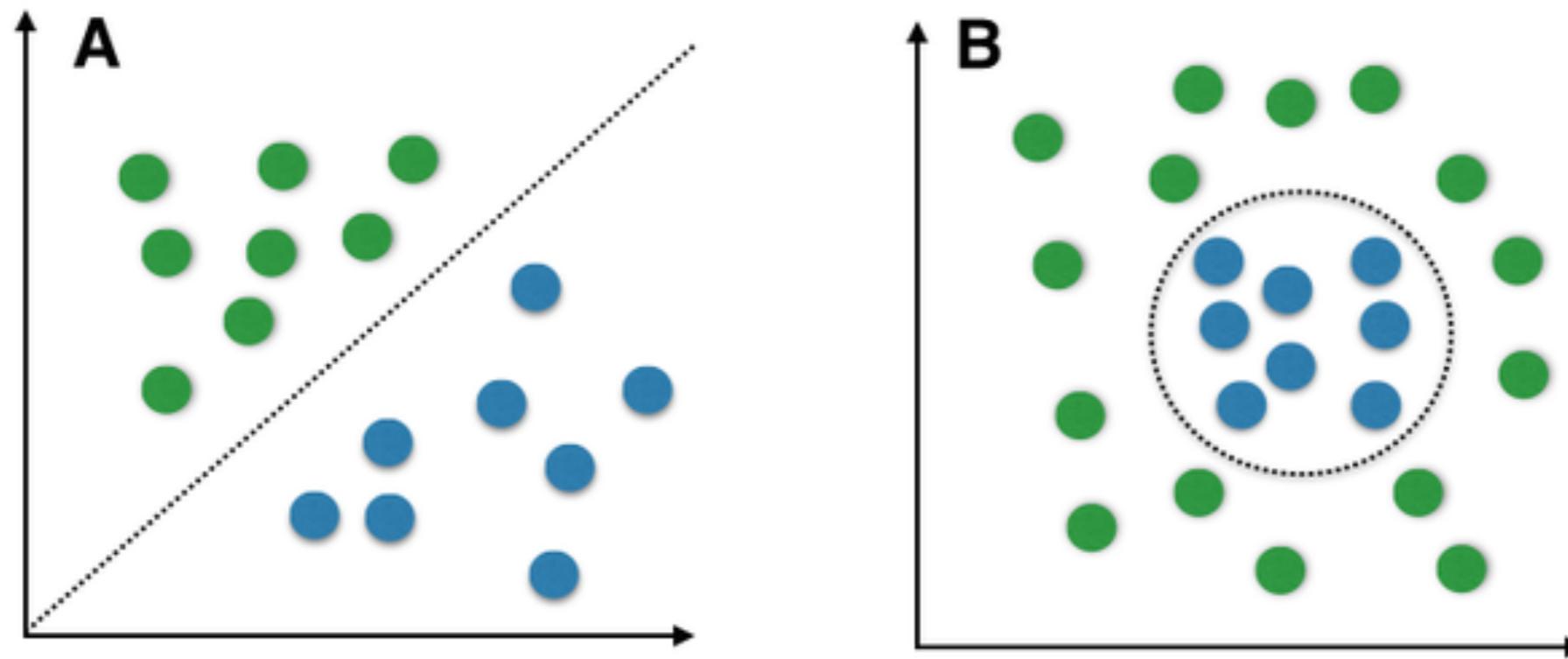
- ❖ Perceptrón (Rosenblatt, 1958, 1962)

Surgen las primeras Redes Neuronales



# Redes Neuronales

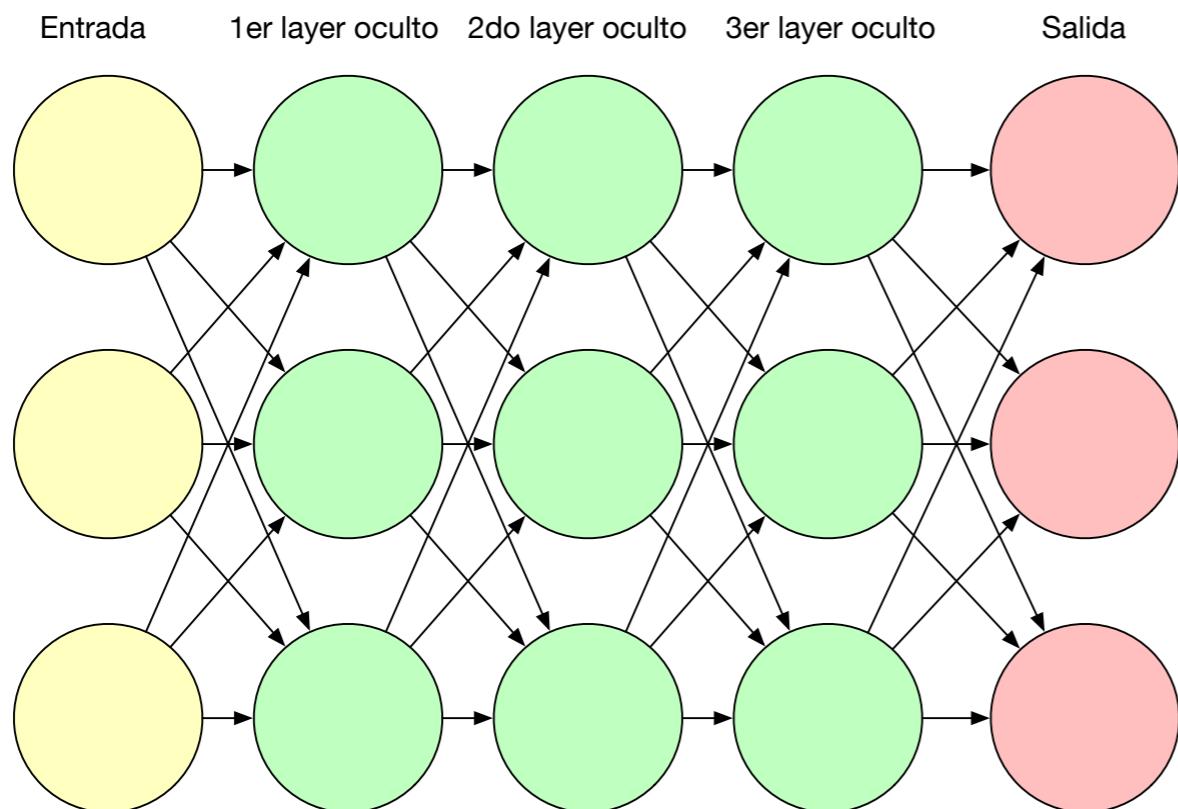
- ❖ Perceptrón (Rosenblatt, 1958, 1962): solo puede resolver problemas linealmente separables



Si el problema es linealmente separable, el algoritmo del perceptrón para encontrar los pesos encuentra la solución en tiempo finito

# Redes Neuronales

- ❖ Perceptron multi-capas (Tesis de P. Werbos, 1974 )



- ❖ Feed-forward: Nos vamos a centrar en este tipo de redes, pero también están las recurrentes.
- ❖ Una red con más de una capa puede resolver un problema arbitrario si tiene suficientes neuronas.
- ❖ No hay algoritmo con un teorema de convergencia

# Redes Neuronales

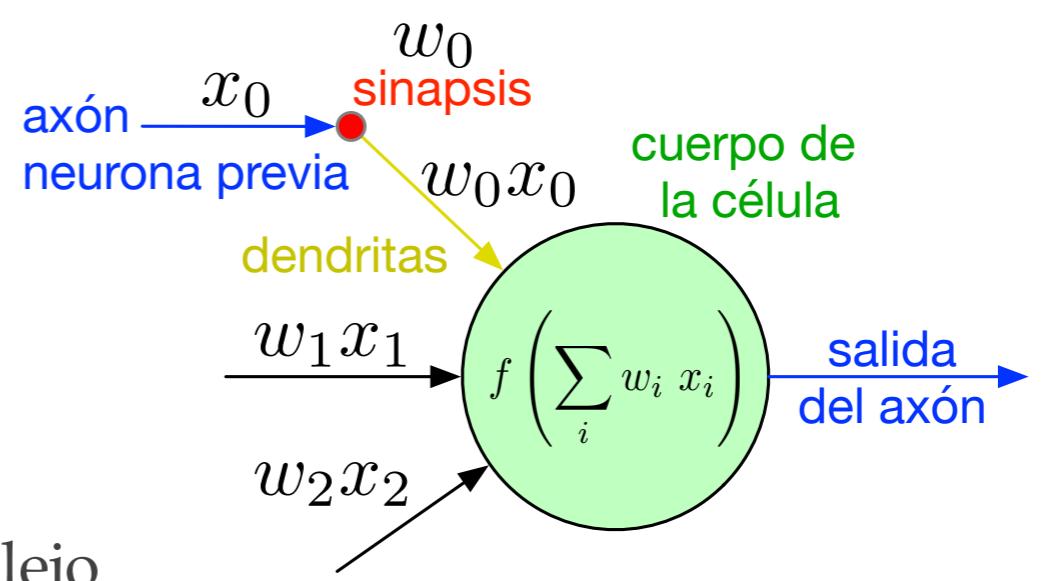
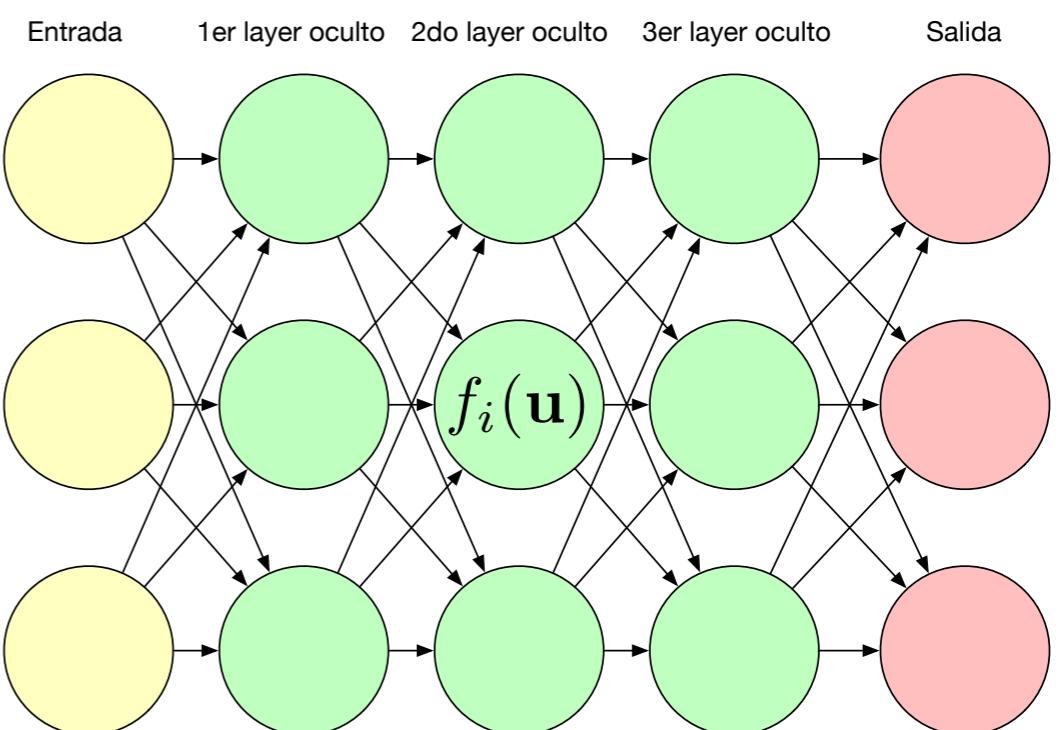
## ❖ Neural Networks (NN).

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

neuronas/unidades  
 $z_i = f_i(\mathbf{u})$

Las redes neuronales modernas **no buscan** modelar el cerebro humano.

- ❖ Neuronas biológicas:
  - ❖ Diferentes tipos
  - ❖ Las dendritas computan funciones no lineales muy complejas.
  - ❖ La sinapsis no es un solo peso sino un sistema dinámico no lineal muy complejo



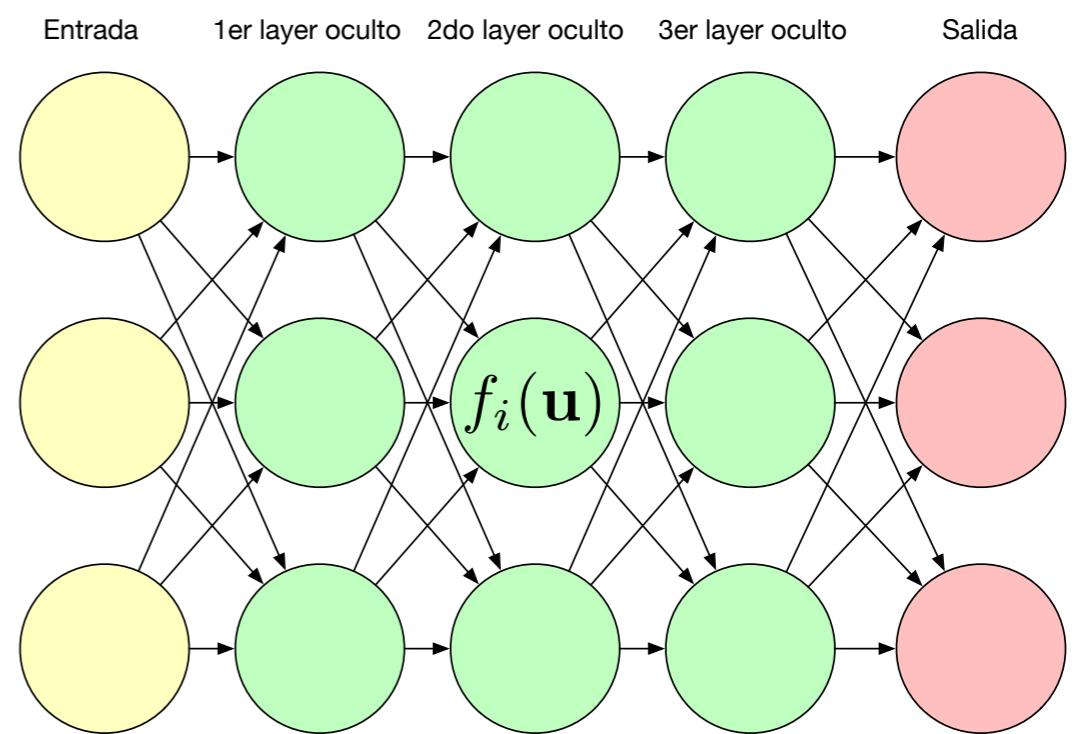
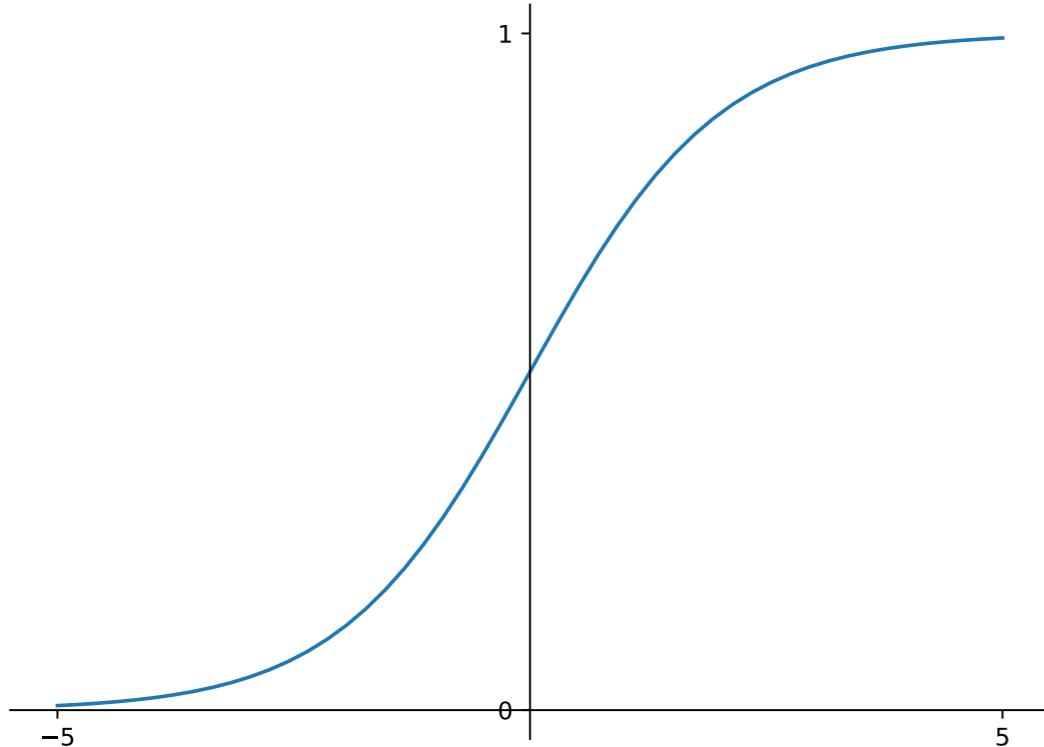
# Redes Neuronales

- ❖ Neural Networks (NN).

- ❖ neuronas

$$f_i \left( \sum_j w_{ij} x_j \right)$$

- ❖ función de activación: Sigmoid



- ❖ Arquitectura:  
¿Cuántas neuronas?  
¿Cómo se conectan las neuronas?  
¿Qué funciones de activación utilizar?

---

# Redes Neuronales

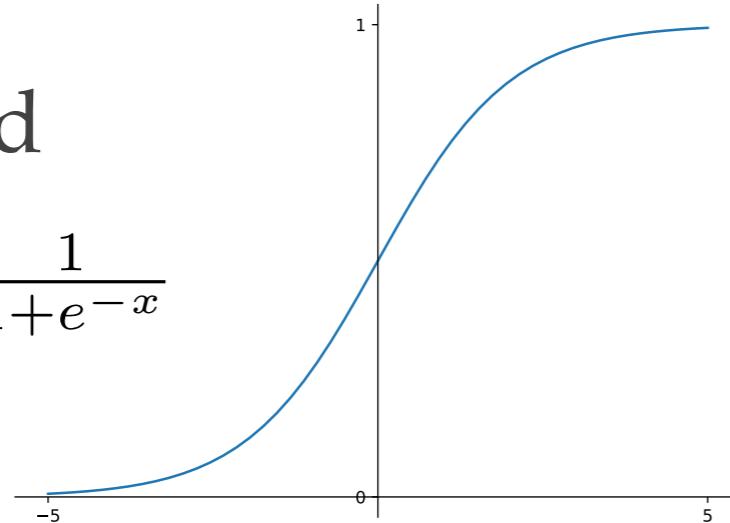
---

- ❖ ¿Por qué necesitamos de la no linealidad de las funciones de activación para aproximar funciones complejas arbitrarias ?
- ❖ ¿Qué pasa si tenemos múltiples capas cuya función de activación es lineal ?

# Funciones de activación

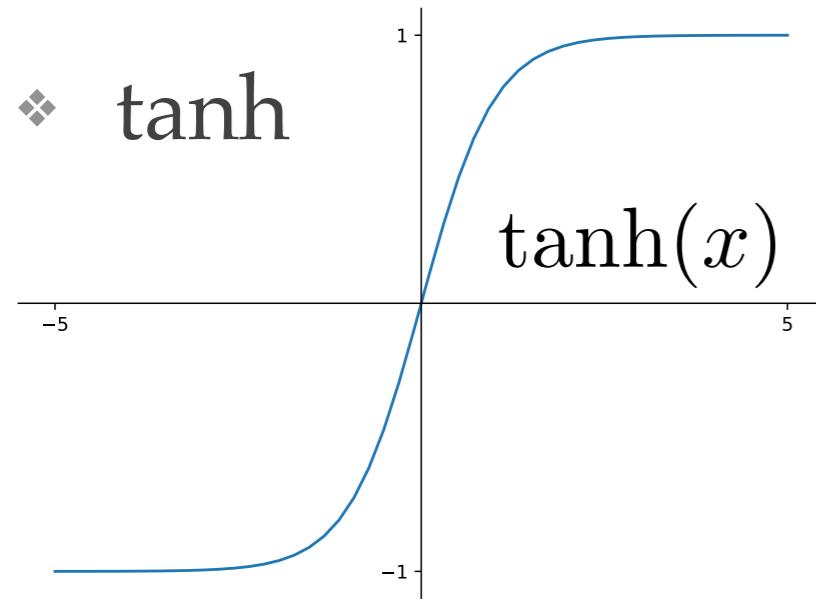
- ❖ Sigmoid

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



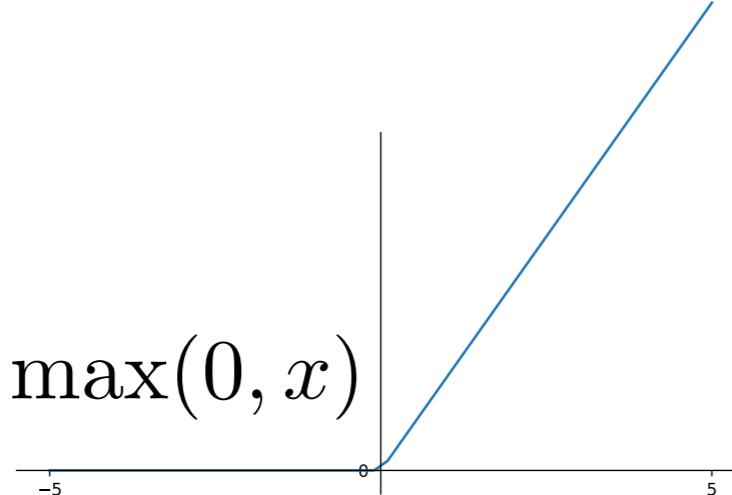
- ❖ tanh

$$\tanh(x)$$



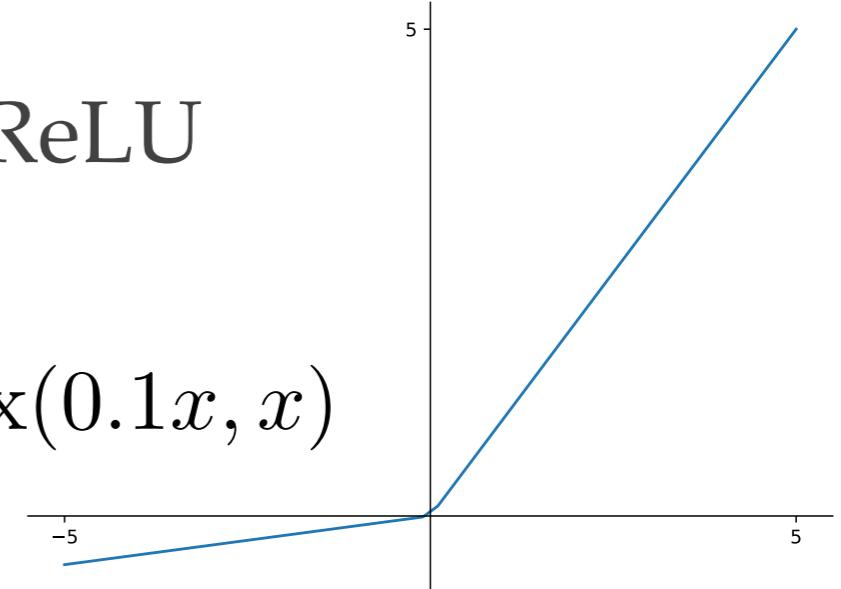
- ❖ ReLU

$$\max(0, x)$$



- ❖ Leaky ReLU

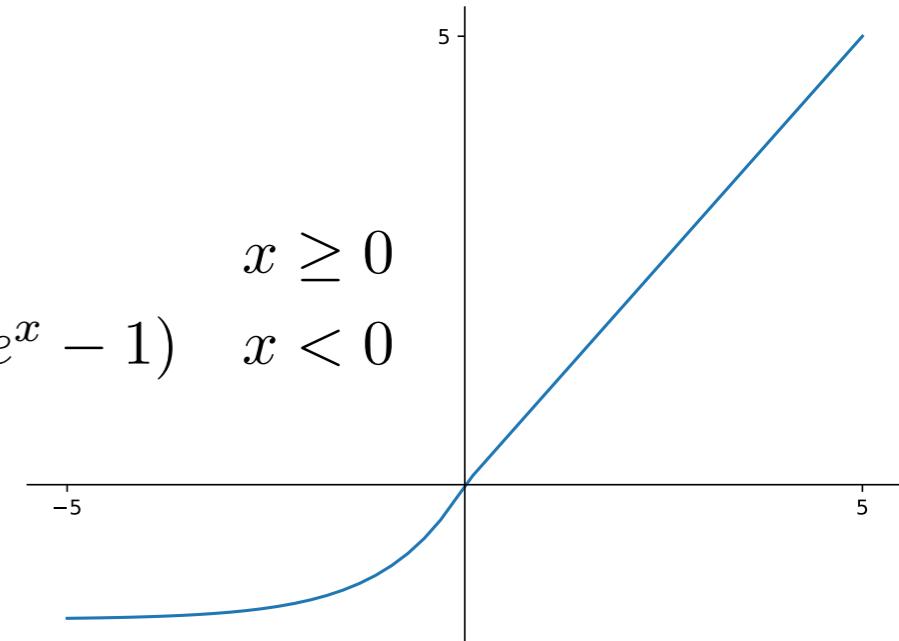
$$\max(0.1x, x)$$



- ❖ Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

- ❖ ELU
- $$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



# Redes Neuronales

- ❖ Las arquitecturas feed-forward implementan mapeos entrada-salida. Pensar la NN como una función compuesta por una gran cantidad de composición de funciones no lineales.
  - ❖ Clasificación (salidas es discreta).
  - ❖ Aproximación de funciones (salidas continuas).

¿ Cómo elegimos los pesos ?

- ❖ Rosenblatt (Perceptrón, 1958, 1962) propone una regla ad-hoc
- ❖ Back-propagation (D.E. Rumelhart, G.E. Hinton et al., 1986; LeCun, 1985)

¿Backpropagation?

# Aprendizaje

1. Se presenta una entrada en particular
2. Se predice la salida
3. Se calcula el error según la función de costo, por ejemplo, el error cuadrático medio (MSE)
4. Se minimiza el error respecto a los parámetros de la red (optimización), por ejemplo utilizando el método mini-batch GD (BGD) o SGD
  - ❖ Necesitamos calcular el gradiente de la función de costo respecto a los parámetros de la NN
$$\nabla_W \text{loss}$$

# Redes Neuronales

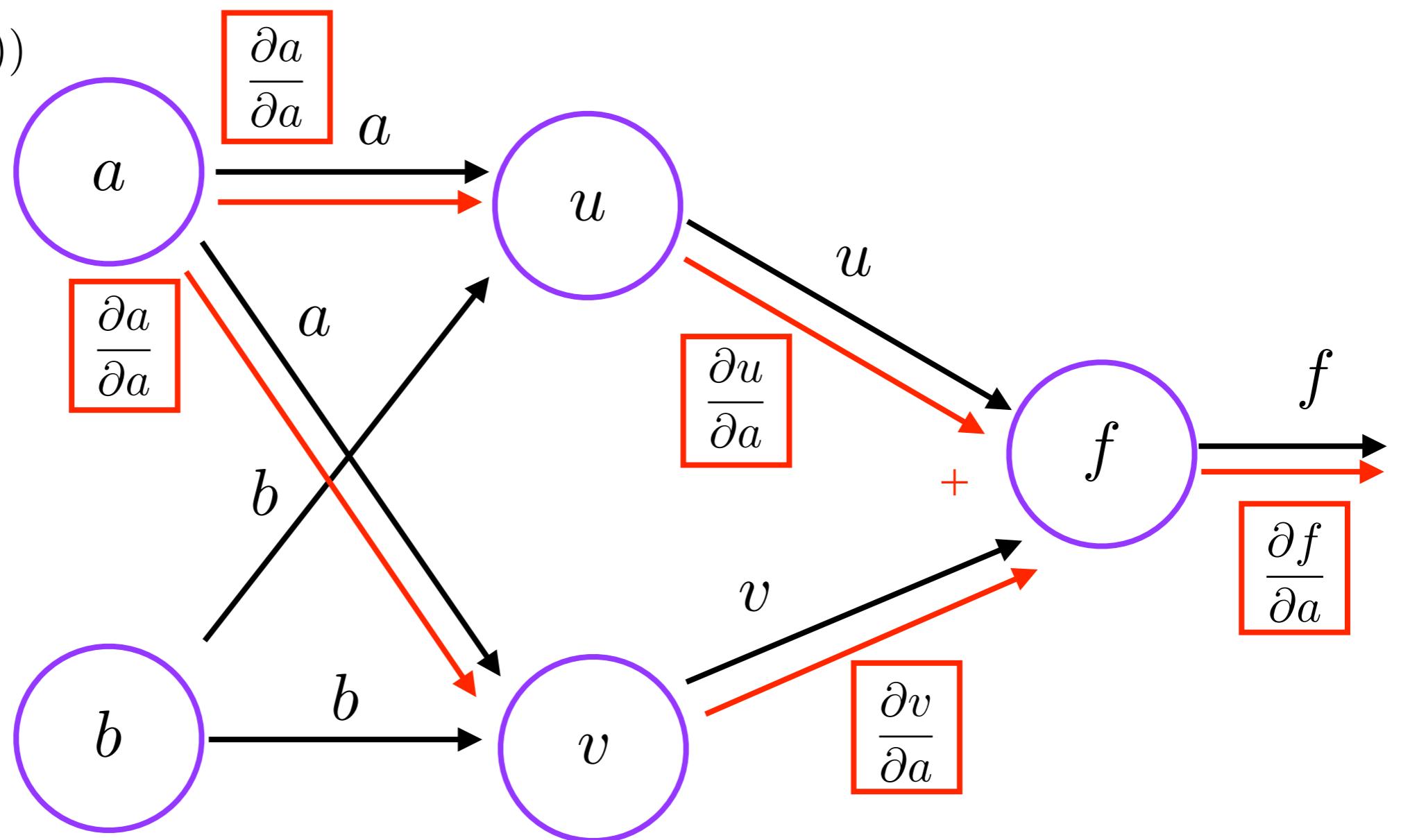
---

- ❖ Back-propagation (D.E. Rumelhart, G.E. Hinton et al., 1986; LeCun, 1985)
  - ❖ En sí es una forma **inteligente** de aplicar la **regla de la cadena** para calcular derivadas.
  - ❖ Aunque se asocie siempre a redes neuronales, el método de back-propagation **NO** es solo para redes neuronales.

# Backpropagation

- ❖ Veamos como se aplica la regla de la cadena en este contexto:

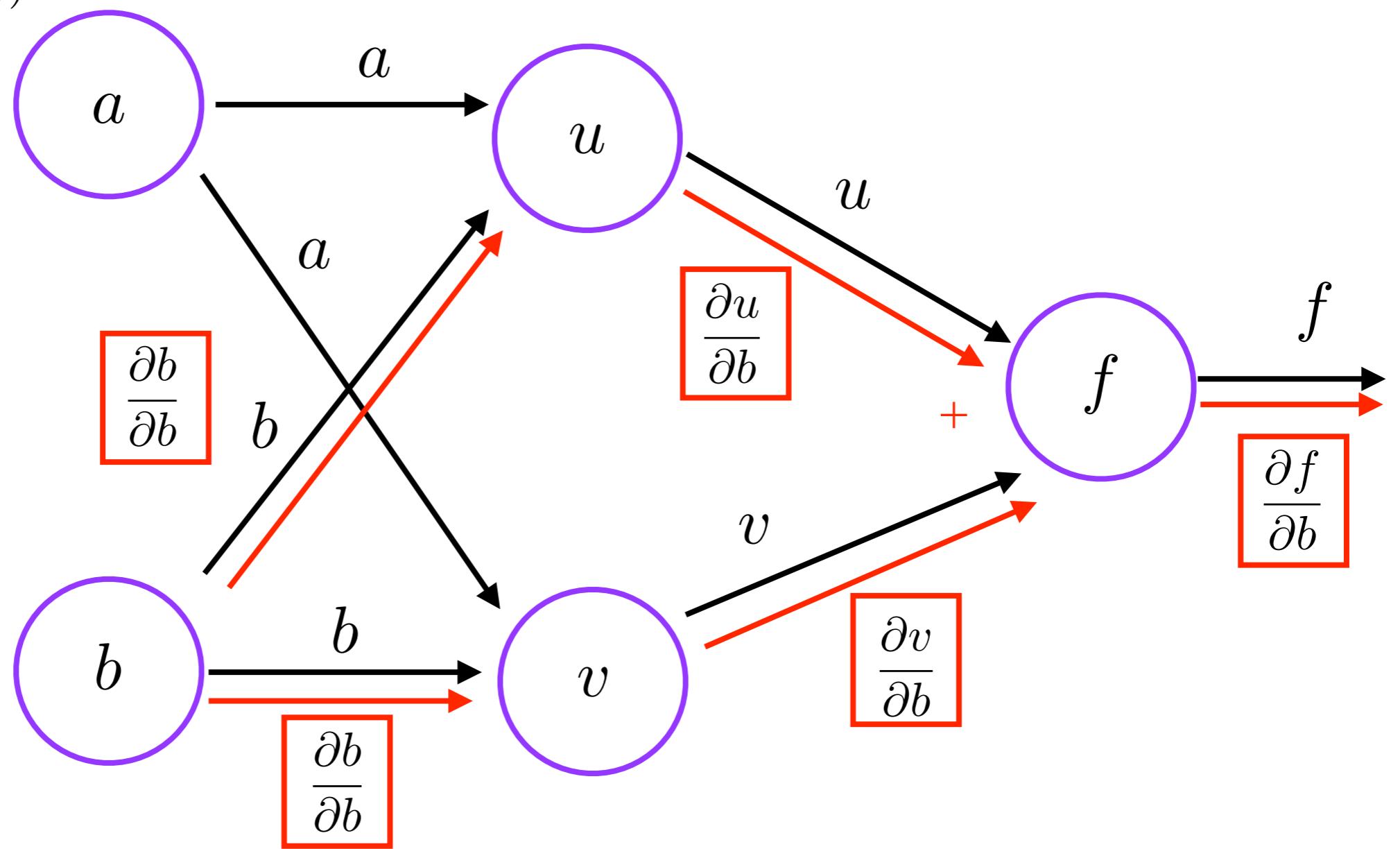
$$f(u(a, b), v(a, b))$$



# Backpropagation

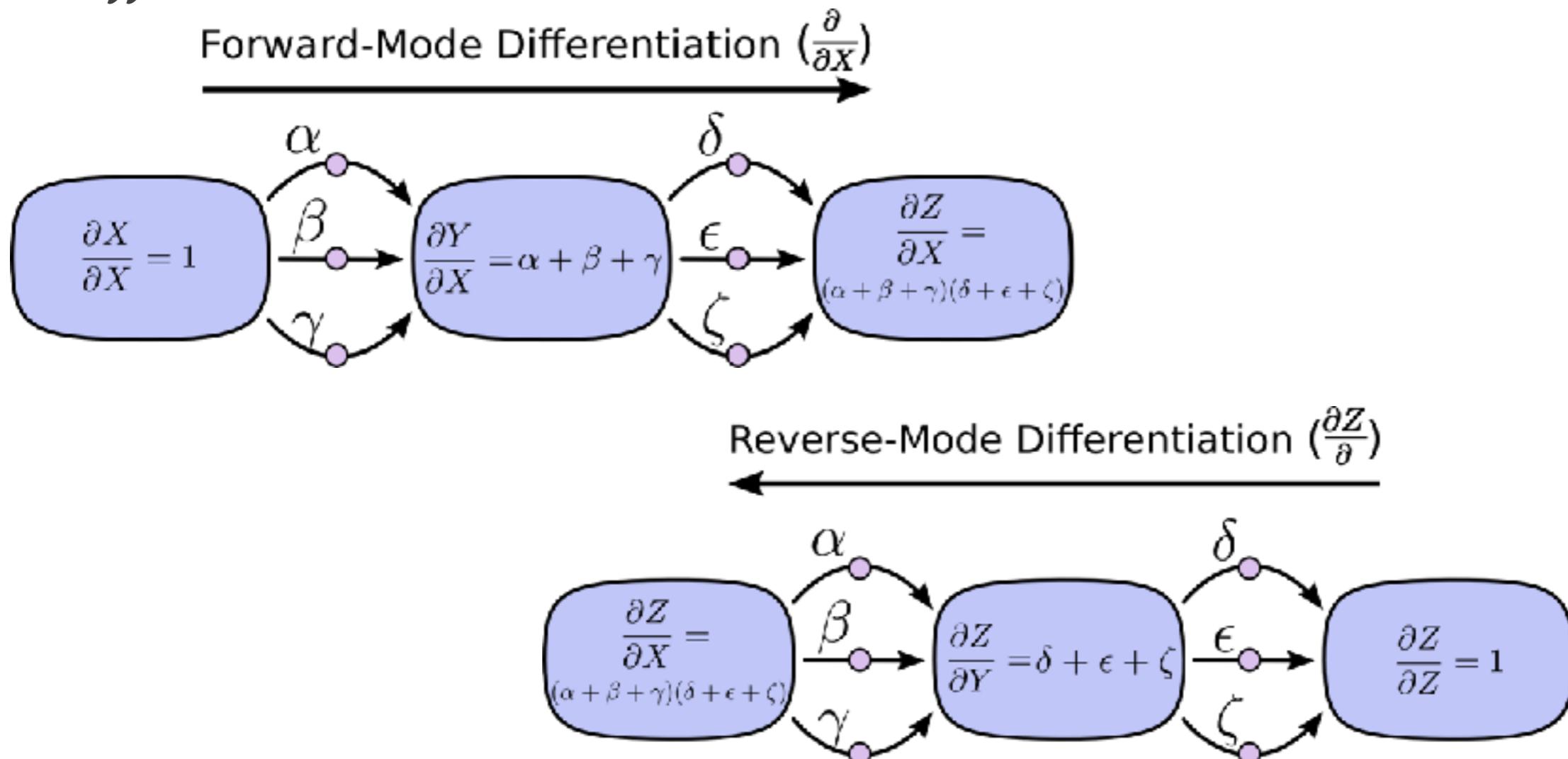
- ❖ Veamos como se aplica la regla de la cadena en este contexto:

$$f(u(a, b), v(a, b))$$



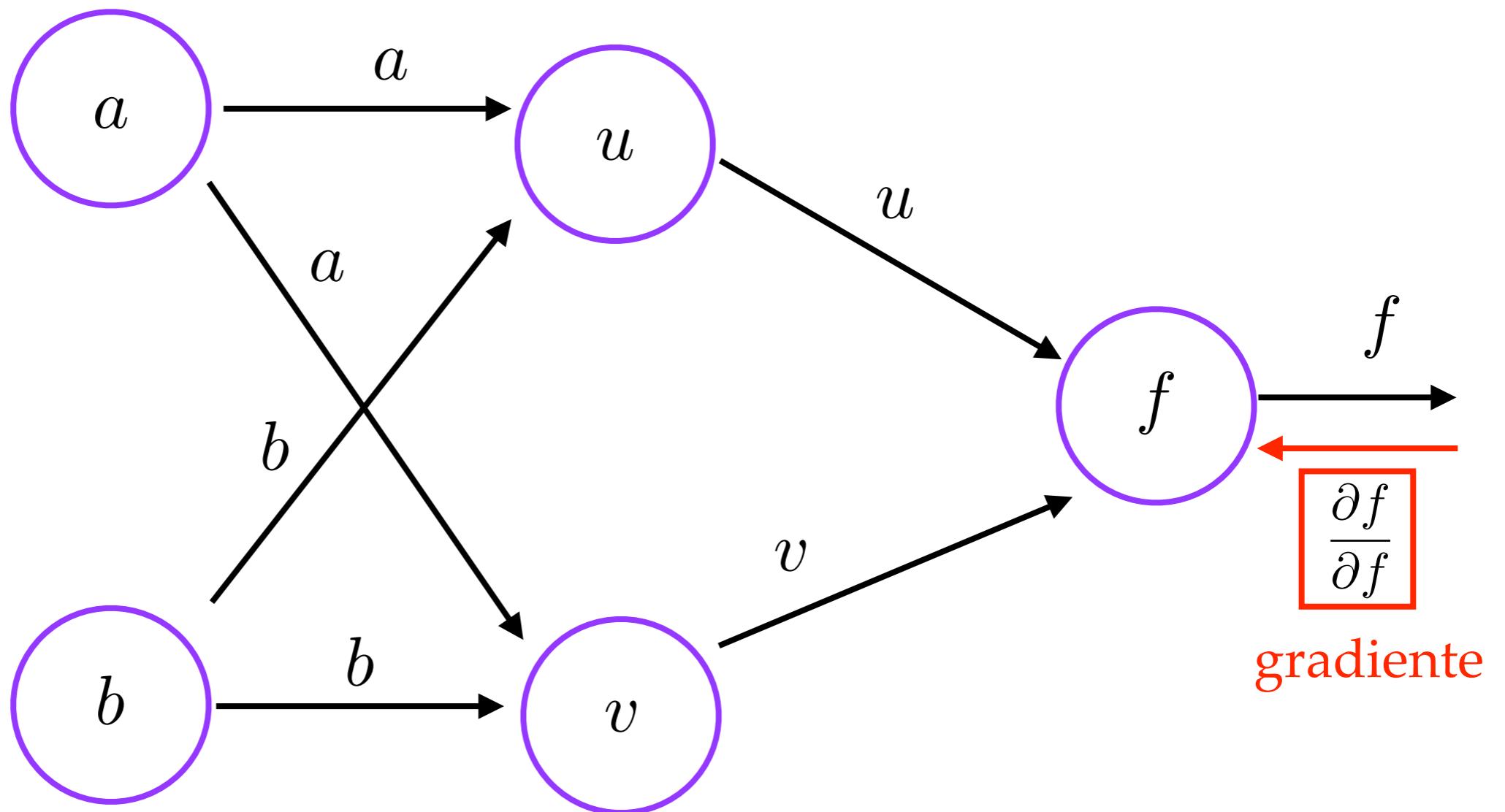
# Backpropagation

- ❖ Back-propagation también conocido como *reverse mode differentiation*



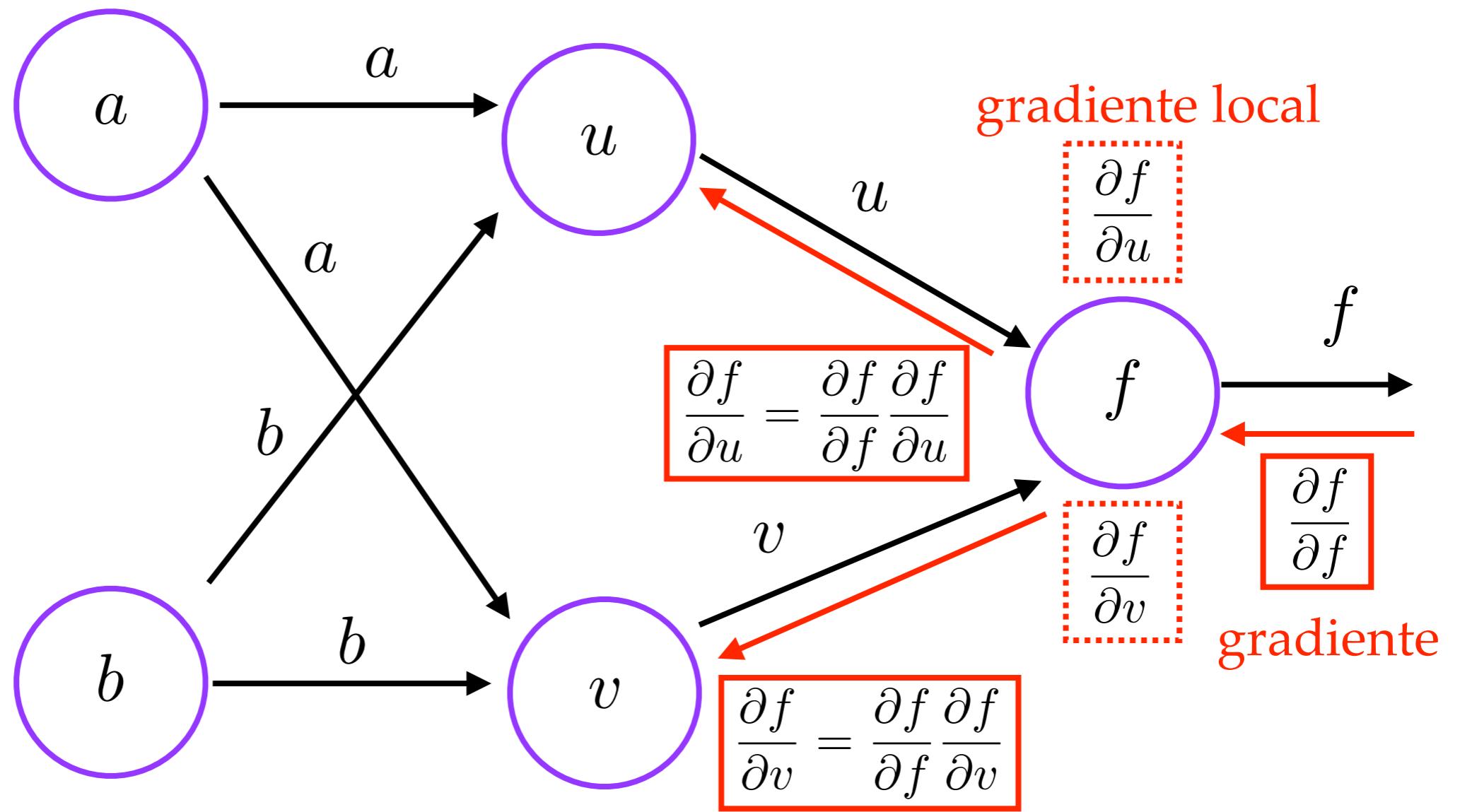
# Backpropagation

$$f(u(a, b), v(a, b))$$



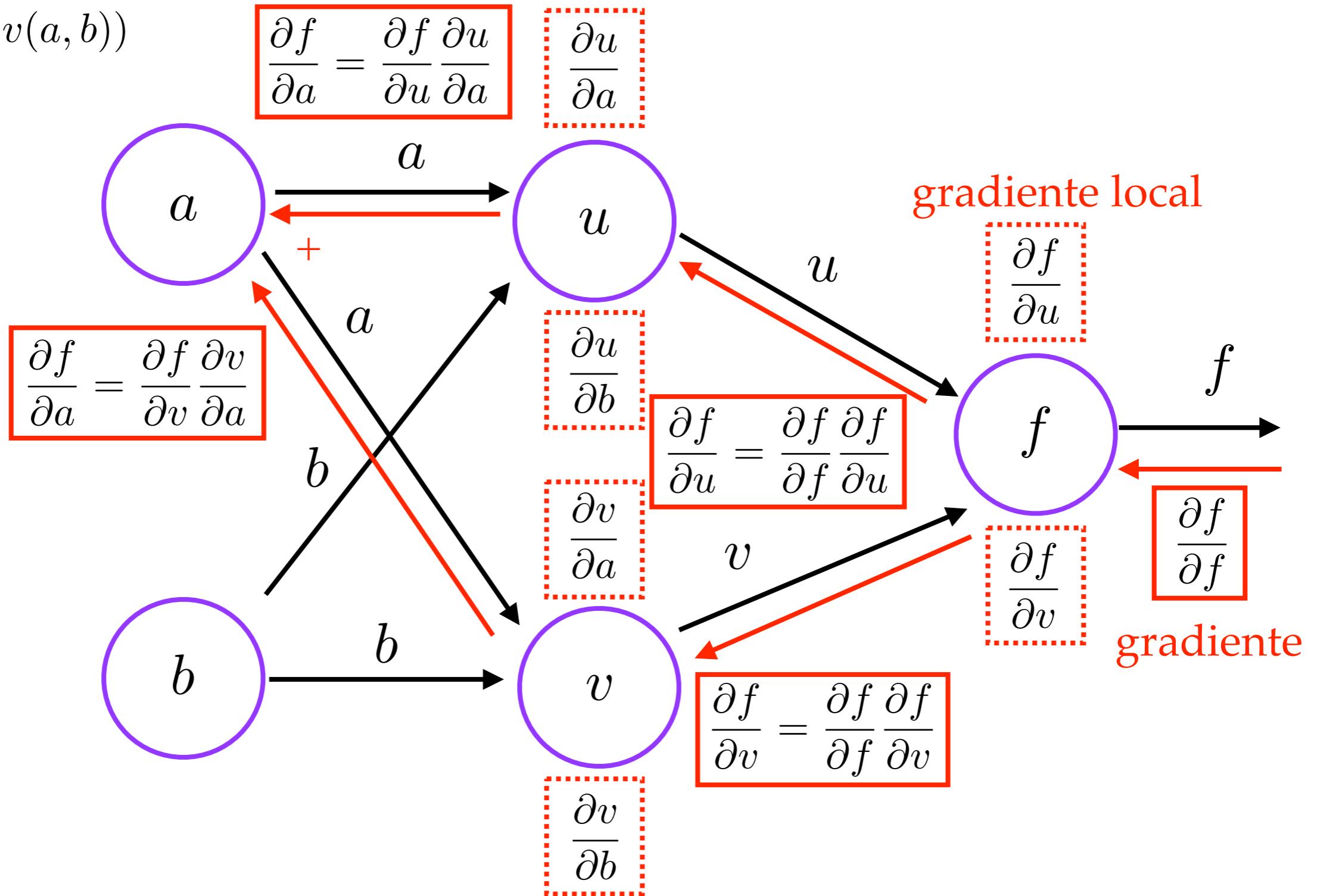
# Backpropagation

$$f(u(a, b), v(a, b))$$



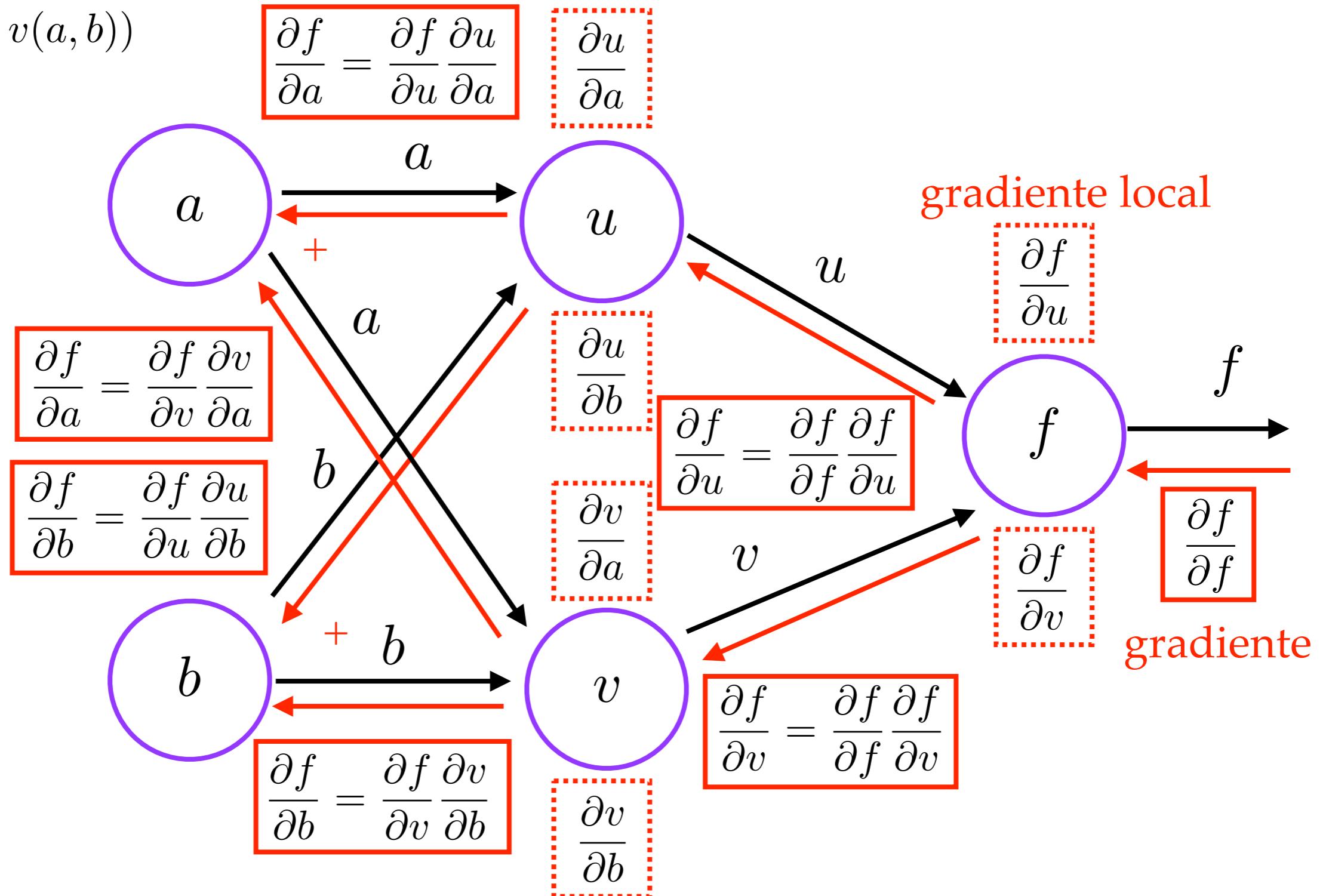
# Backpropagation

$$f(u(a, b), v(a, b))$$



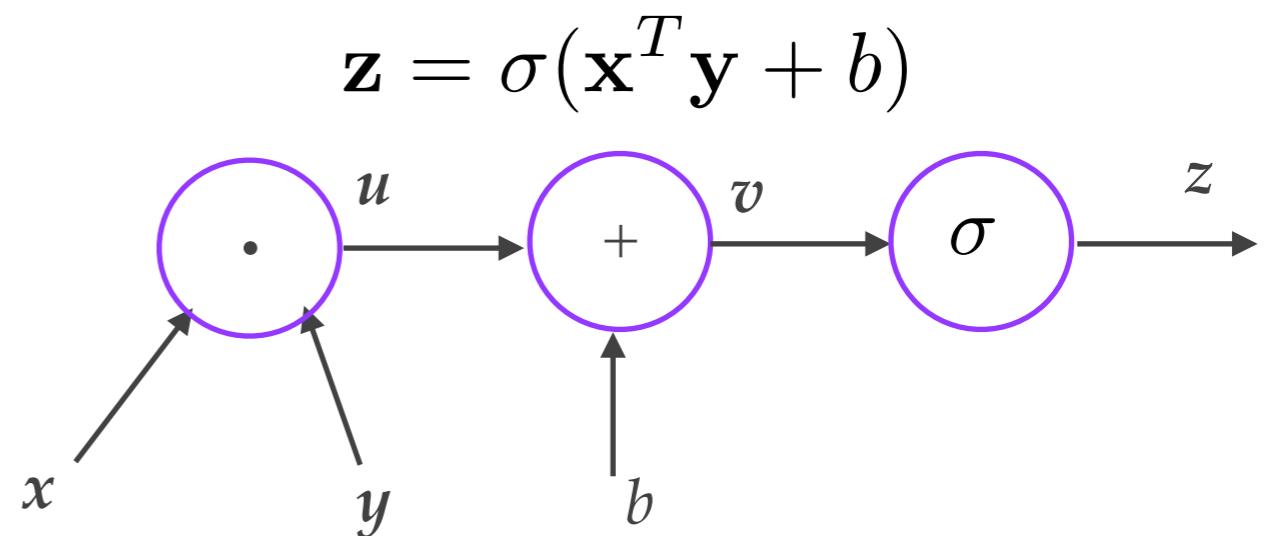
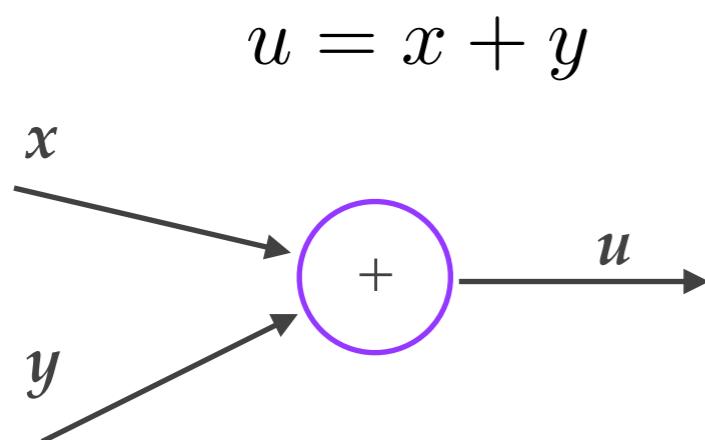
# Backpropagation

$$f(u(a, b), v(a, b))$$



# Computational Graphs

- ❖ Algunas ideas sobre cálculo de grafos o *computational graphs*: son una buena forma de pensar en expresiones matemáticas.

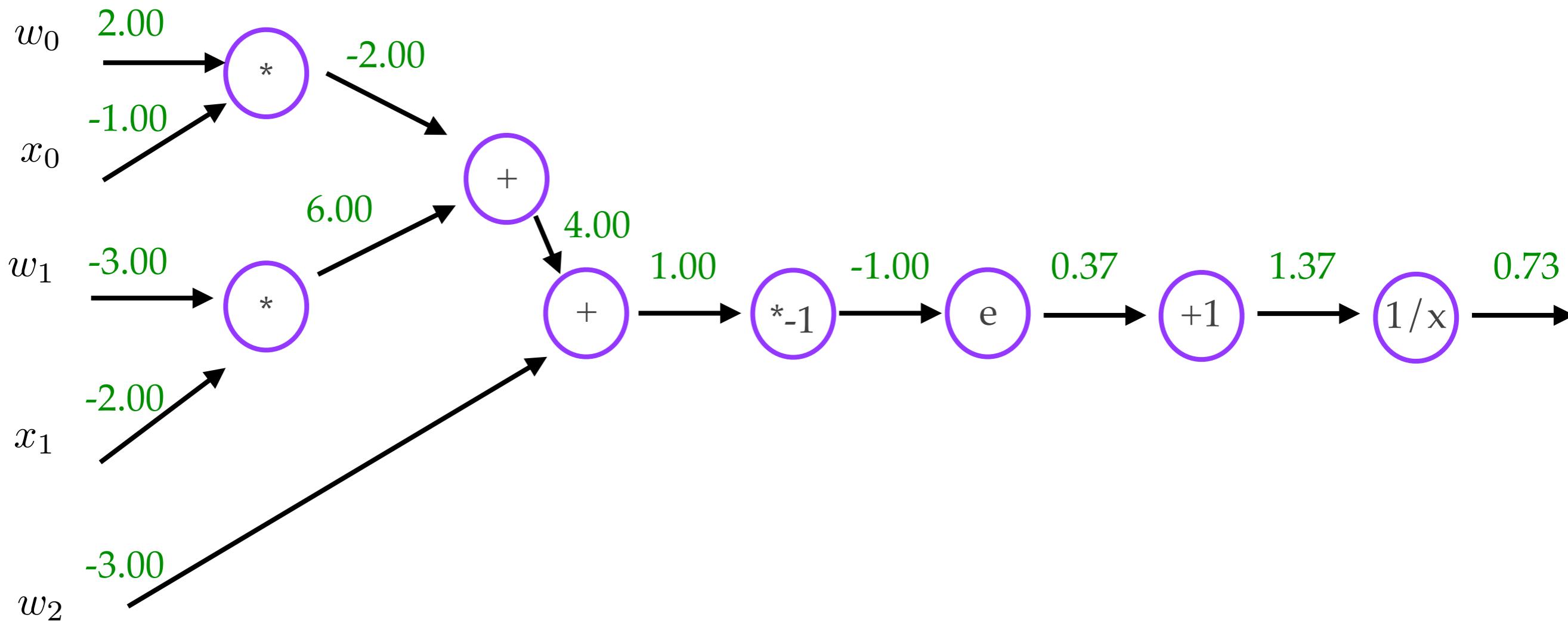


Variables: escalares, vectores, matrices, **tensores**, ...  
Operaciones: funciones

- ❖ En este sentido la **red neuronal** es una masiva composición de funciones.

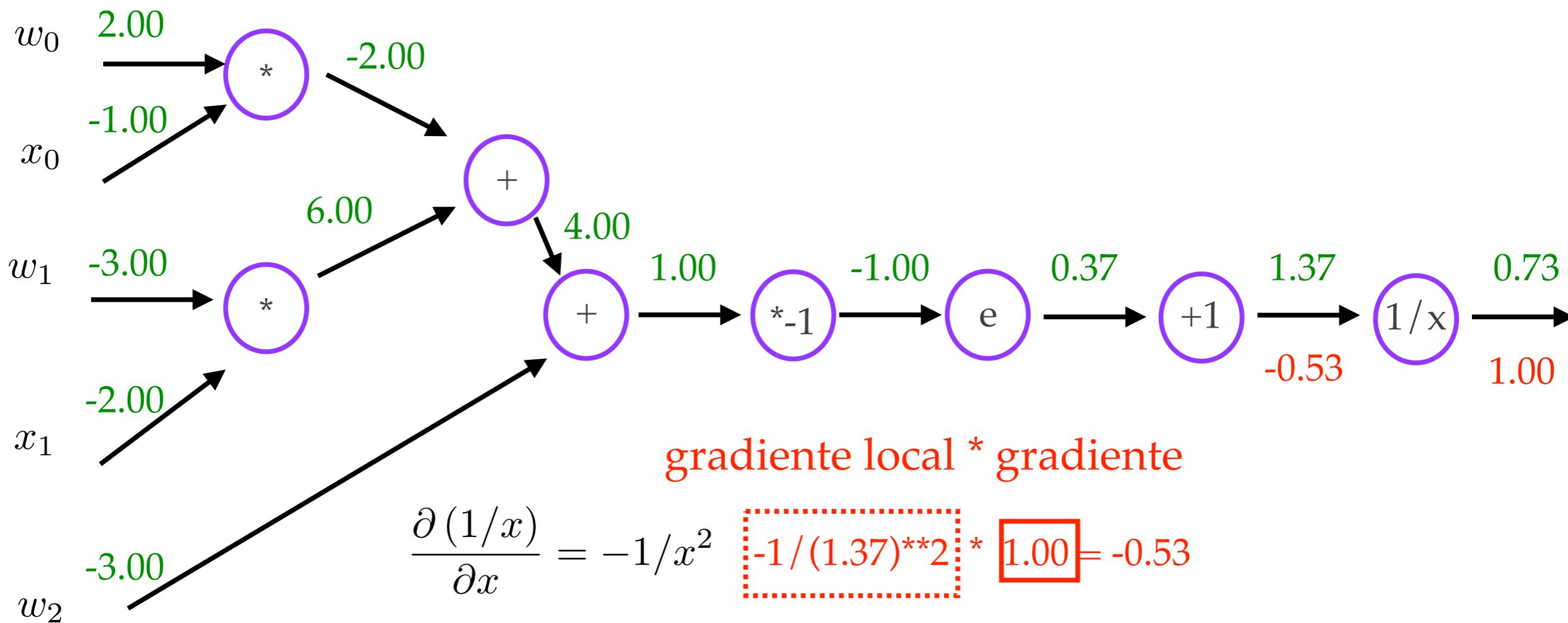
# Backpropagation

- ❖ Ejemplo: Perceptron + sigmoid  $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



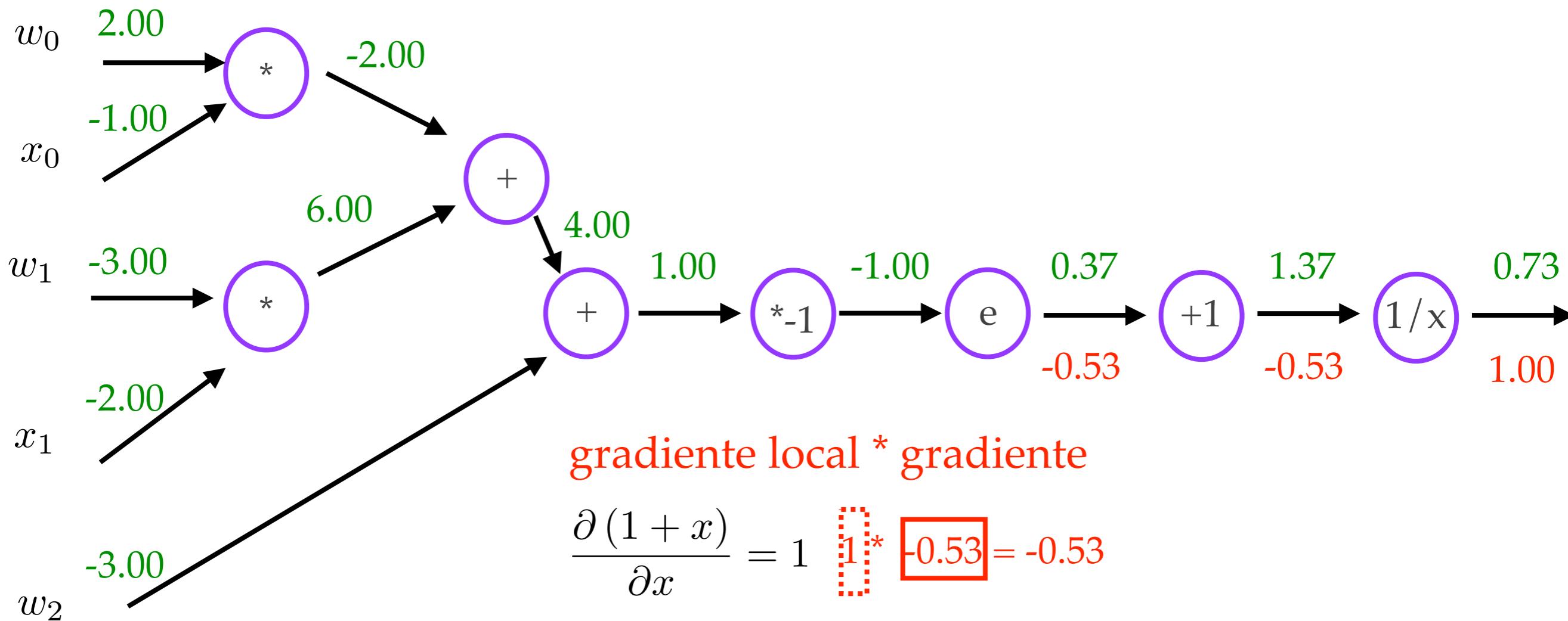
# Backpropagation

- ❖ Ejemplo: Perceptron + sigmoid  $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



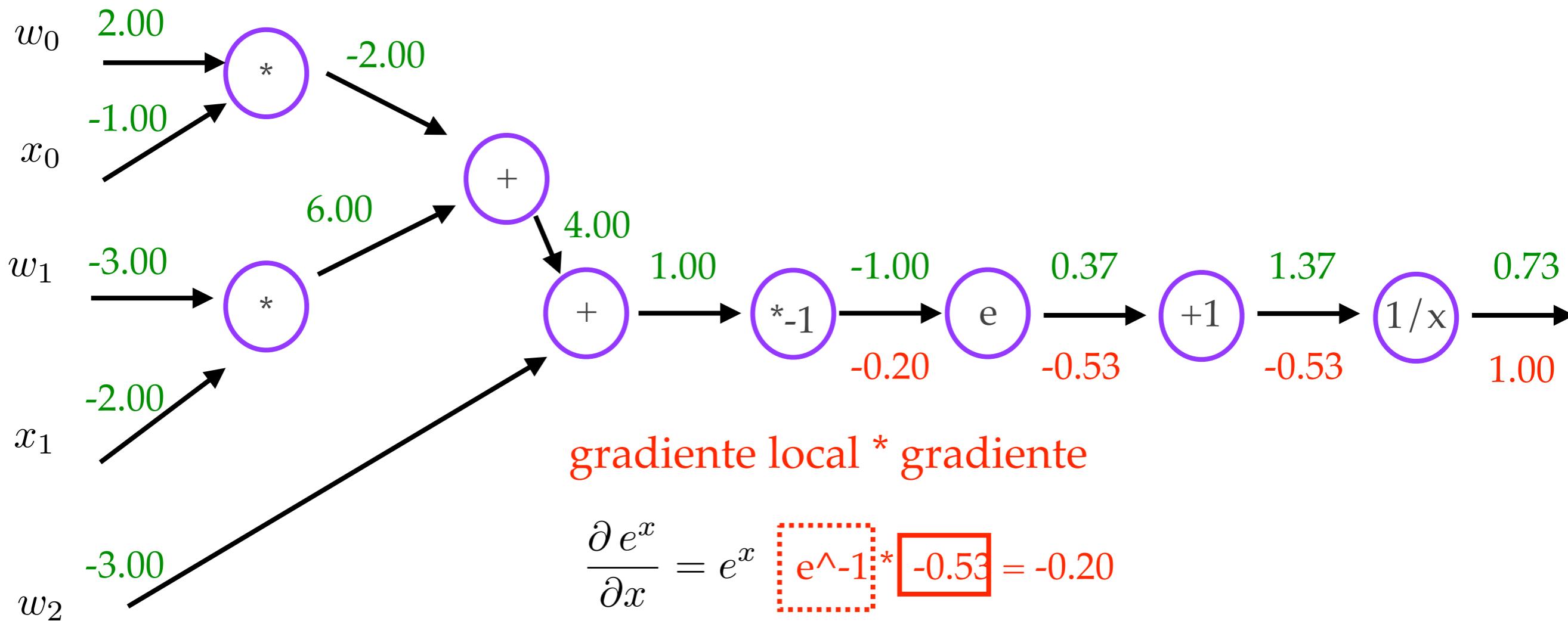
# Backpropagation

- ❖ Ejemplo: Perceptron + sigmoid  $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



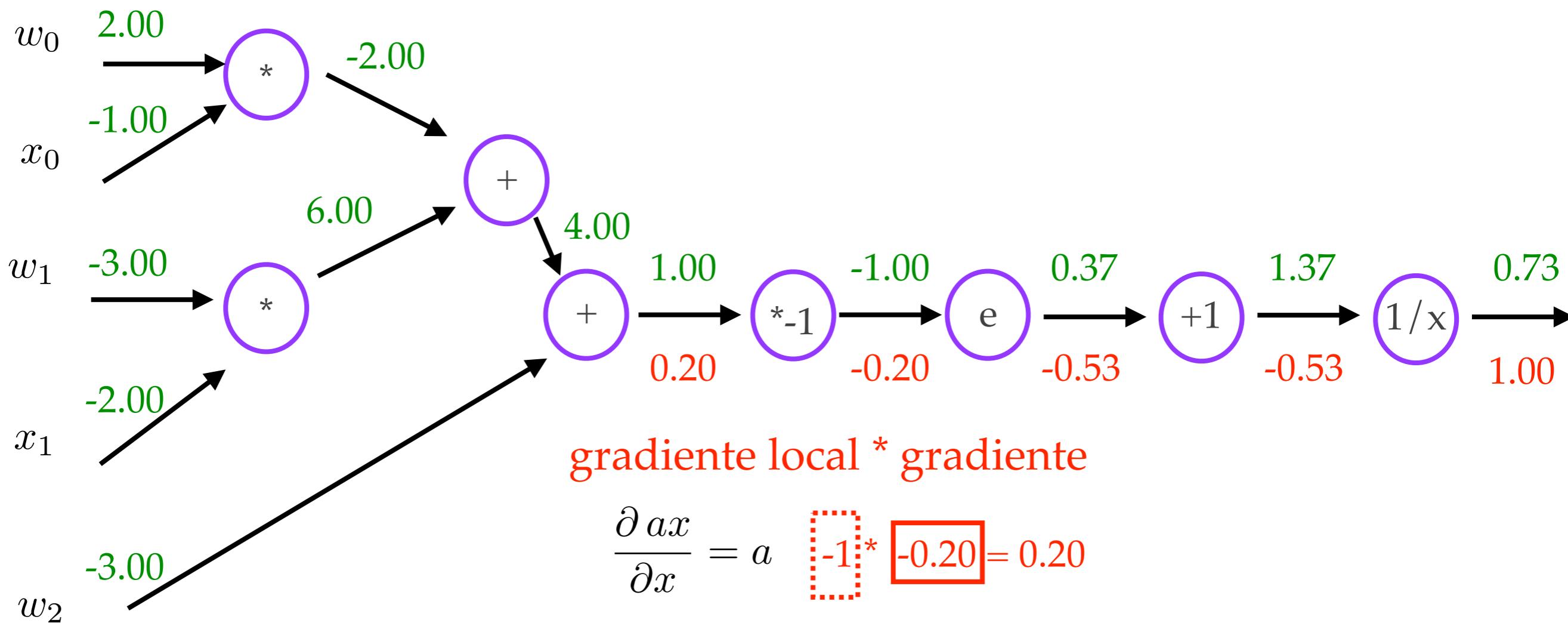
# Backpropagation

- ❖ Ejemplo: Perceptron + sigmoid  $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



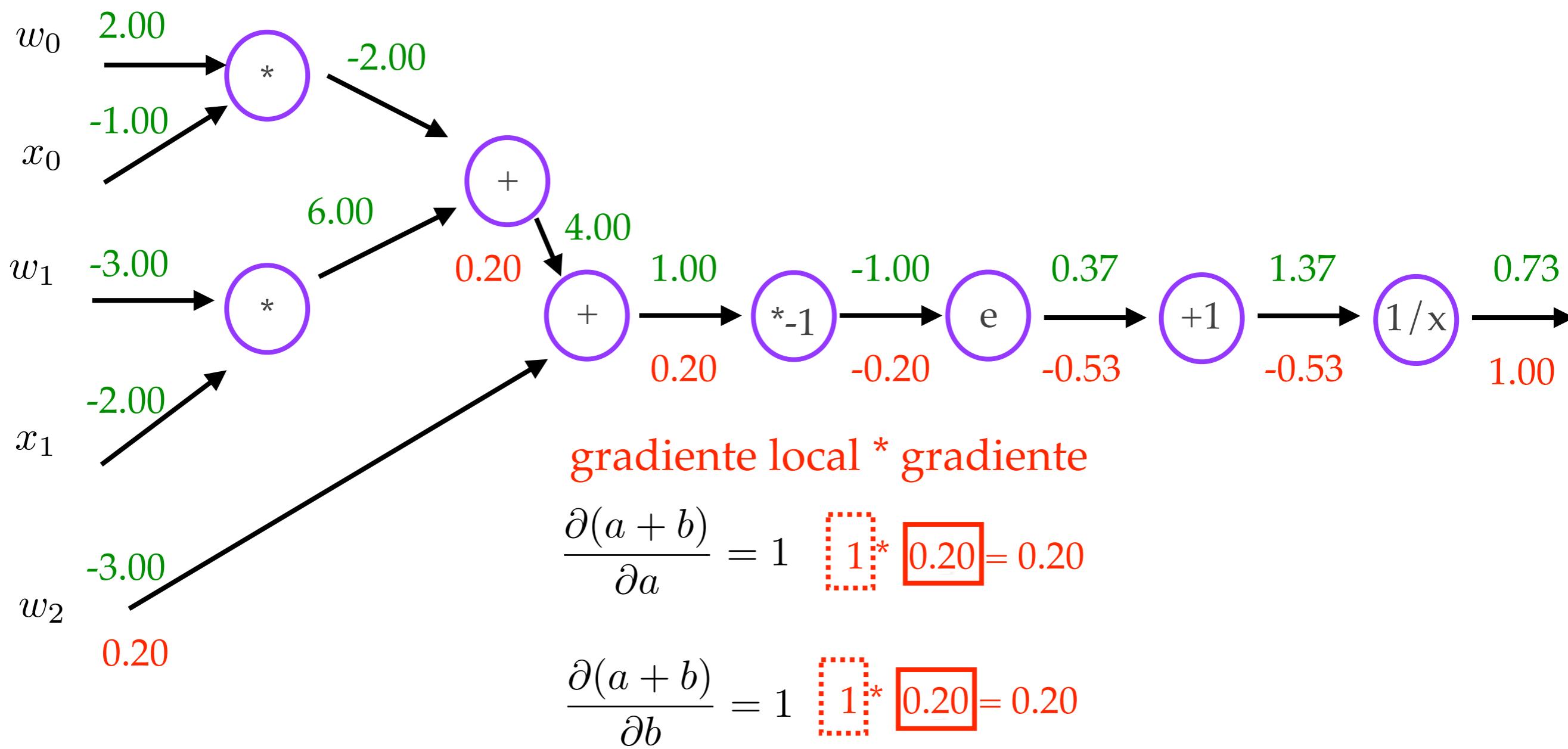
# Backpropagation

- ❖ Ejemplo: Perceptron + sigmoid  $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



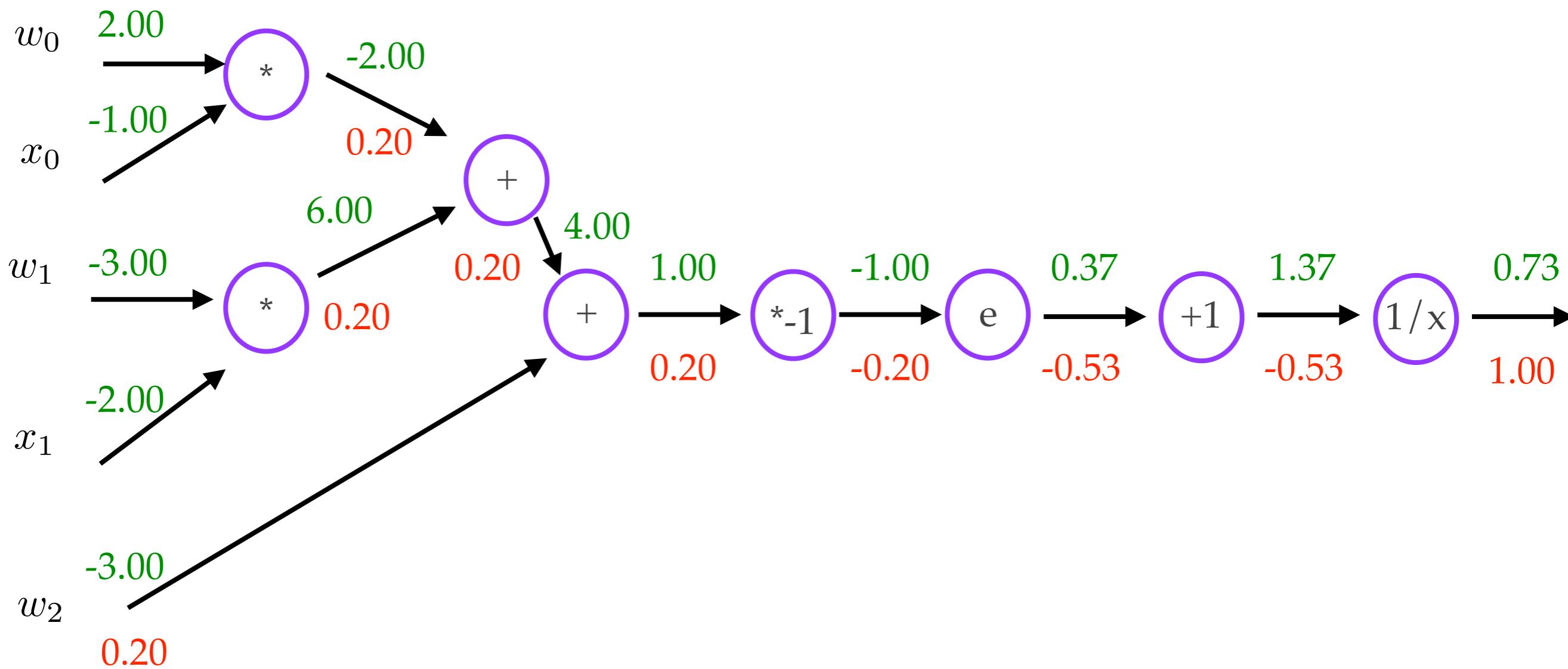
# Backpropagation

- ❖ Ejemplo: Perceptron + sigmoid  $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



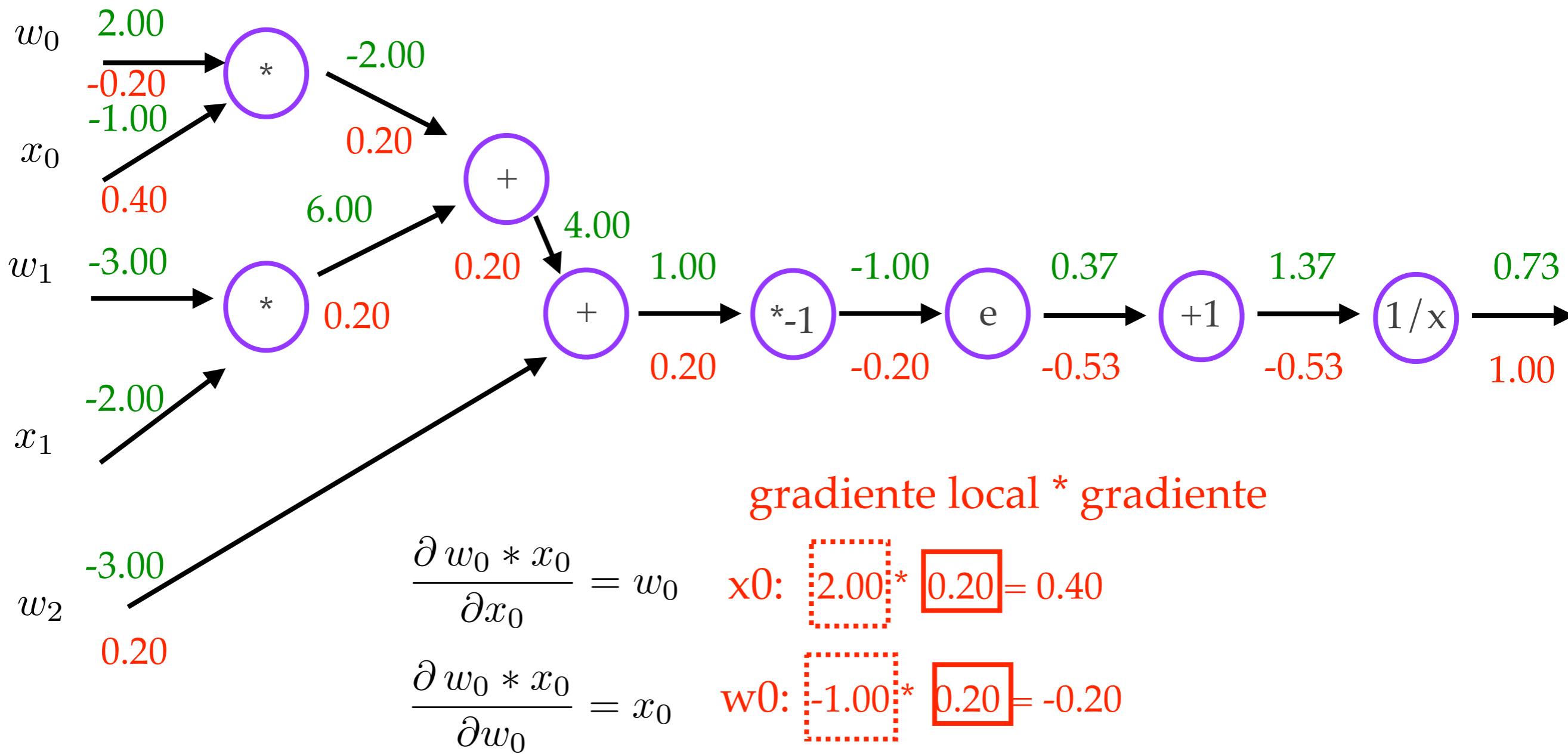
# Backpropagation

- ❖ Ejemplo: Perceptron + sigmoid  $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



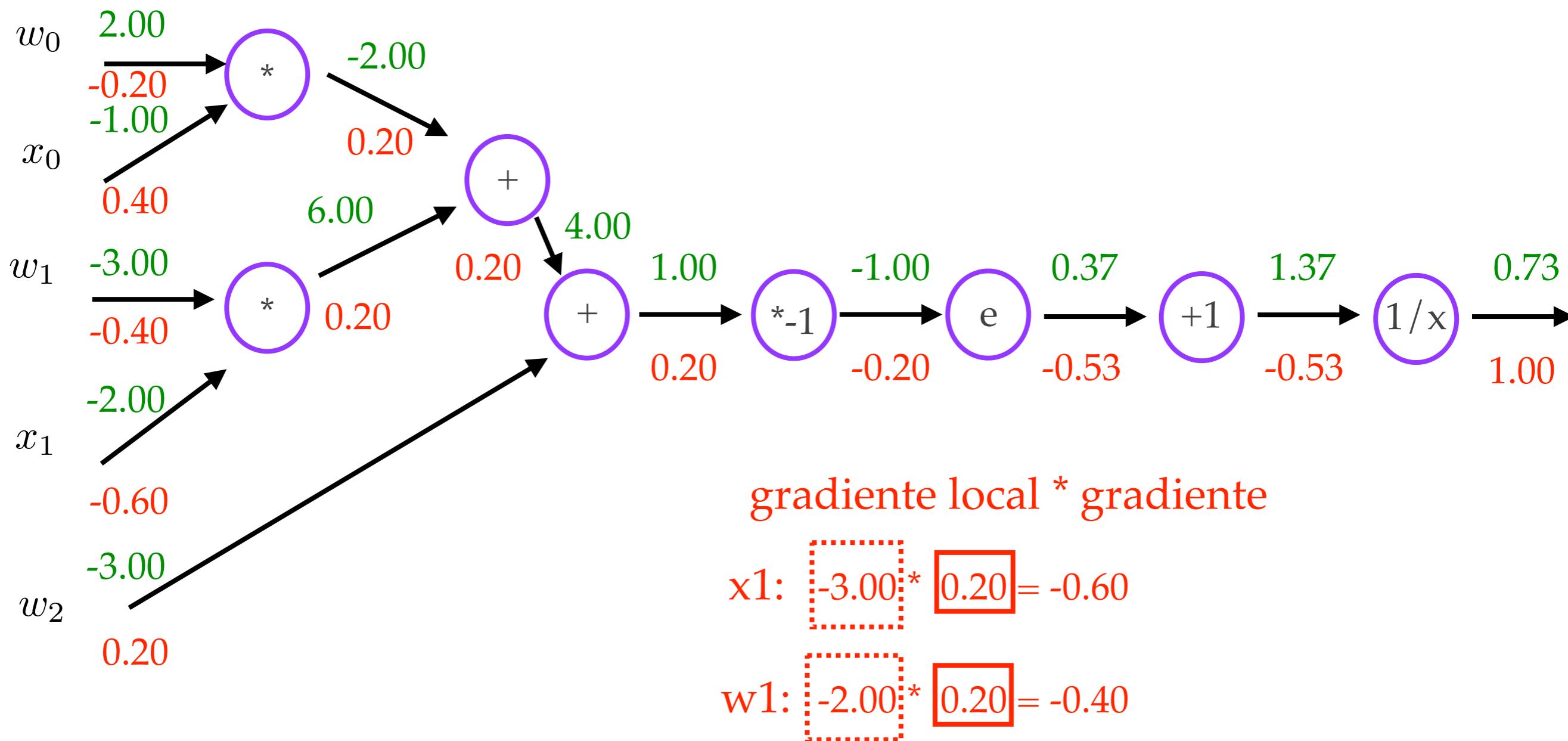
# Backpropagation

- ❖ Ejemplo: Perceptron + sigmoid  $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



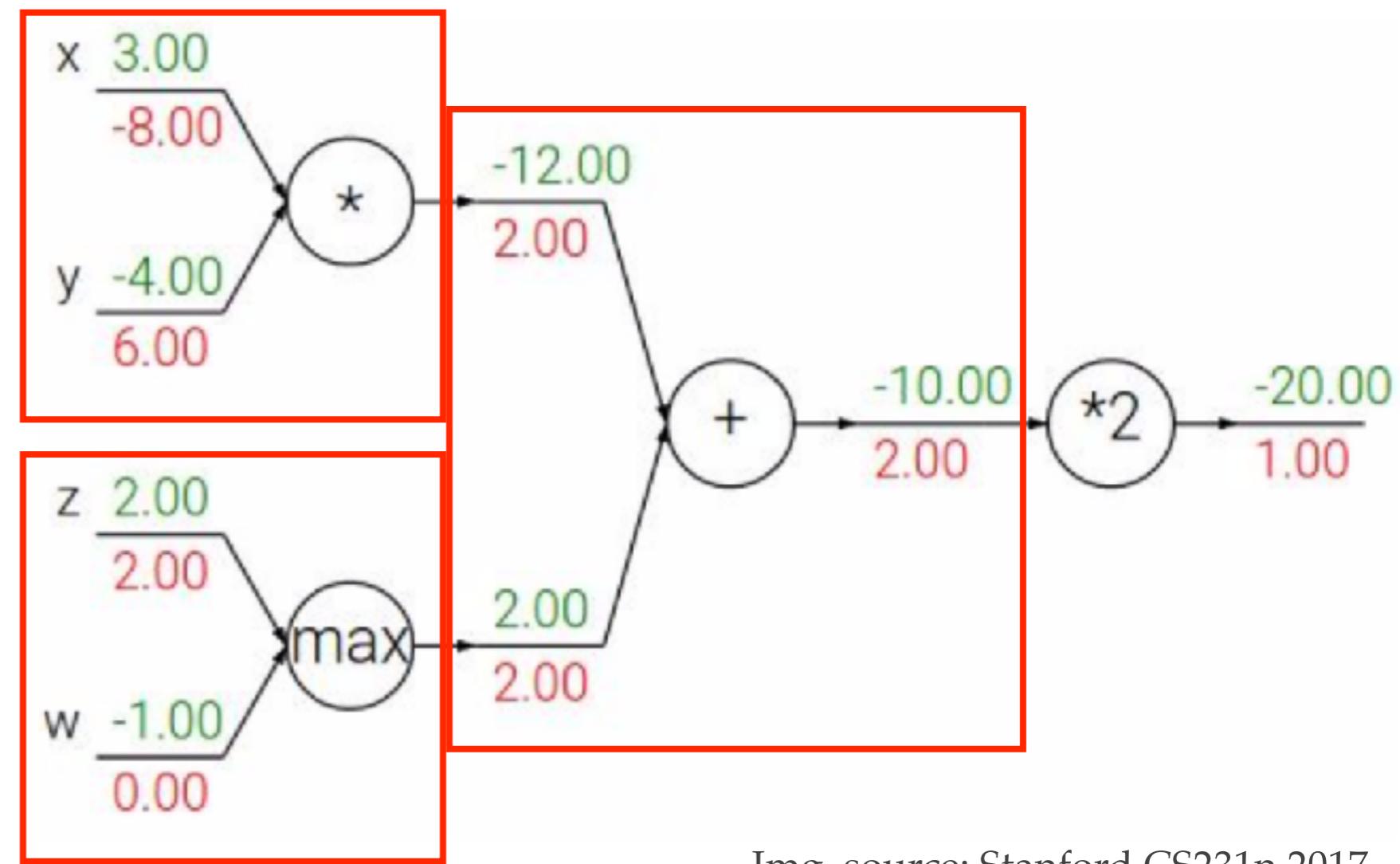
# Backpropagation

- ❖ Ejemplo: Perceptron + sigmoid  $f(x, w) = \frac{1}{1 + e^{-(w_0x_0 + w_1x_1 + w_2)}}$



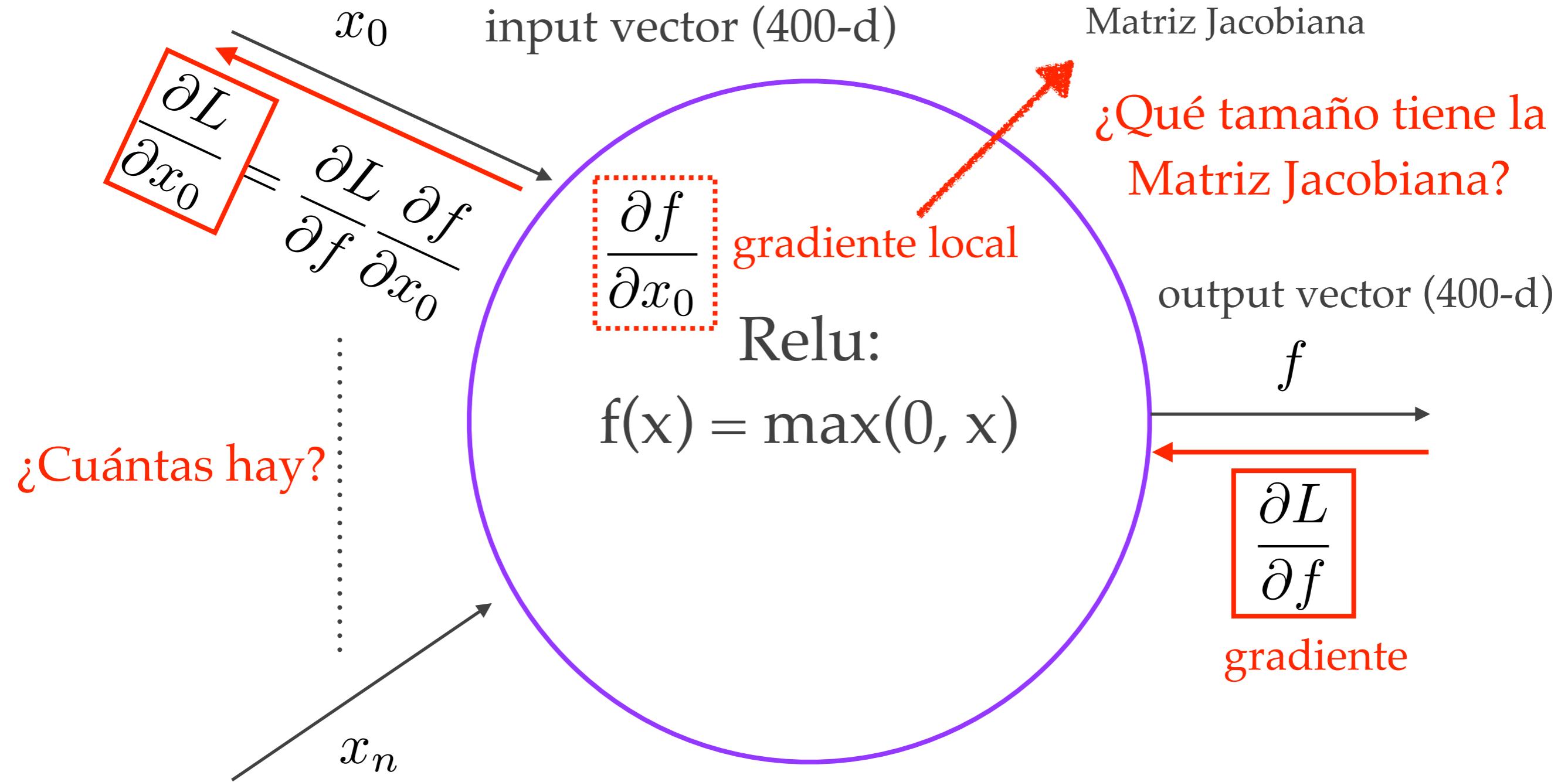
# Backpropagation

- ❖  $+$ : Distribuye el gradiente
- ❖  $*$ : escala el gradiente
- ❖ max: es un router  
(sólo uno pasa)

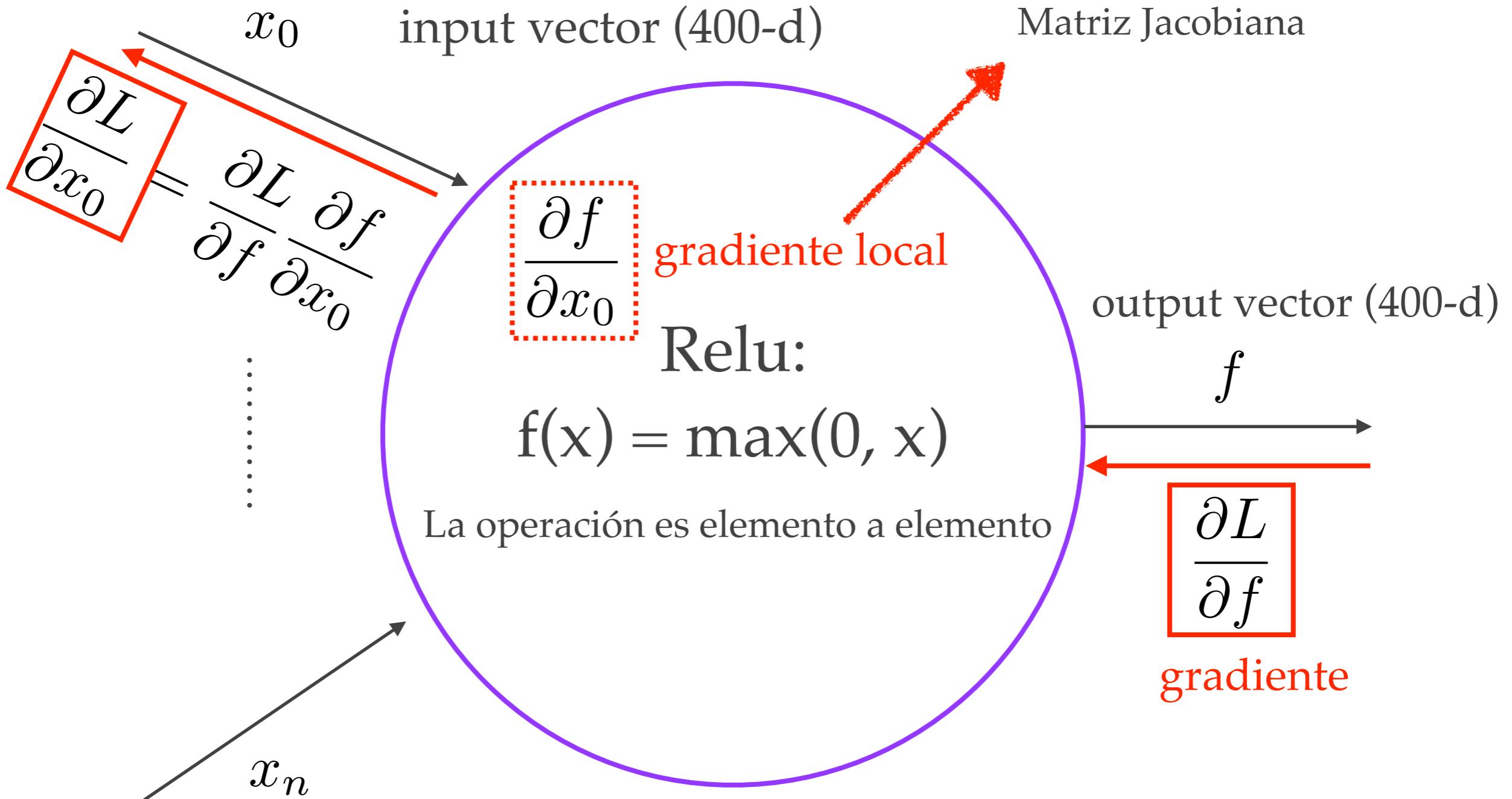


Img. source: Stanford CS231n 2017

# Backpropagation



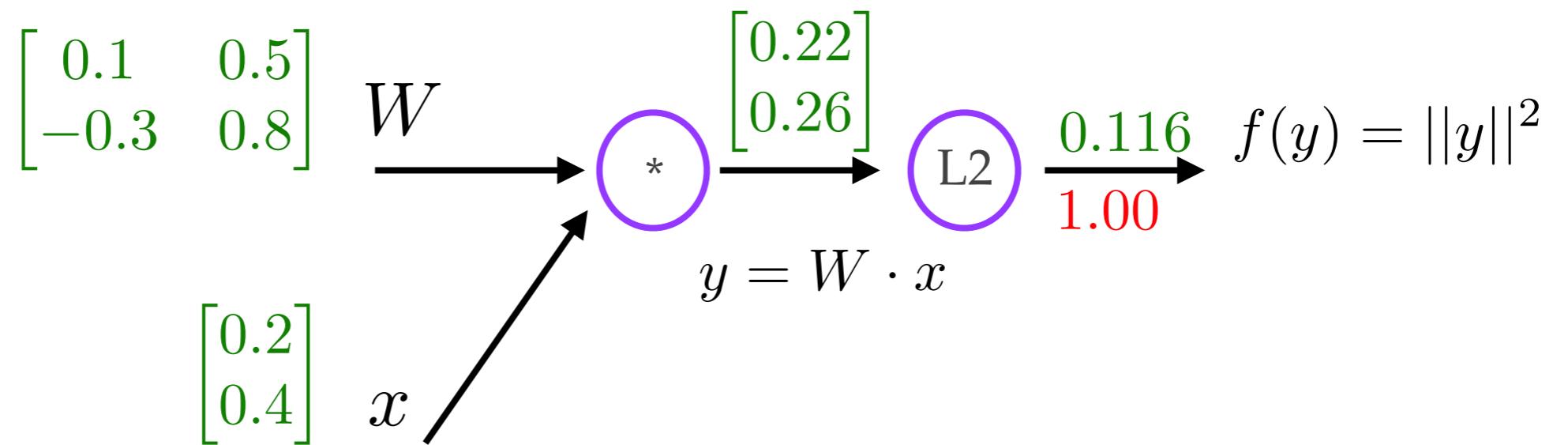
# Backpropagation



¿Qué representan los 400-d?

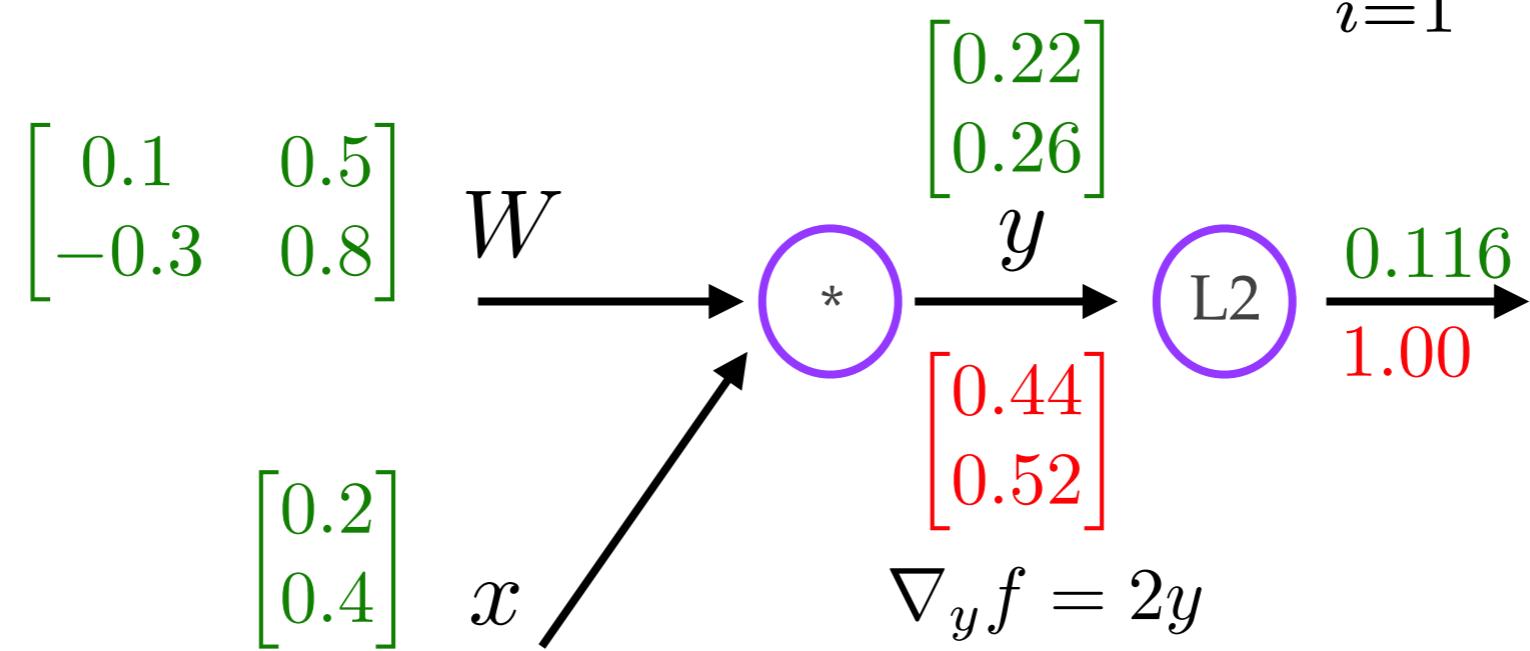
# Backpropagation

- ❖ Ejemplo:  $f(x, W) = \text{L2} = \|W \cdot x\|^2 = \sum_{i=1}^n (W \cdot x)_i^2$



# Backpropagation

- ❖ Ejemplo:  $f(x, W) = \text{L2} = \|W \cdot x\|^2 = \sum_{i=1}^n (W \cdot x)_i^2$



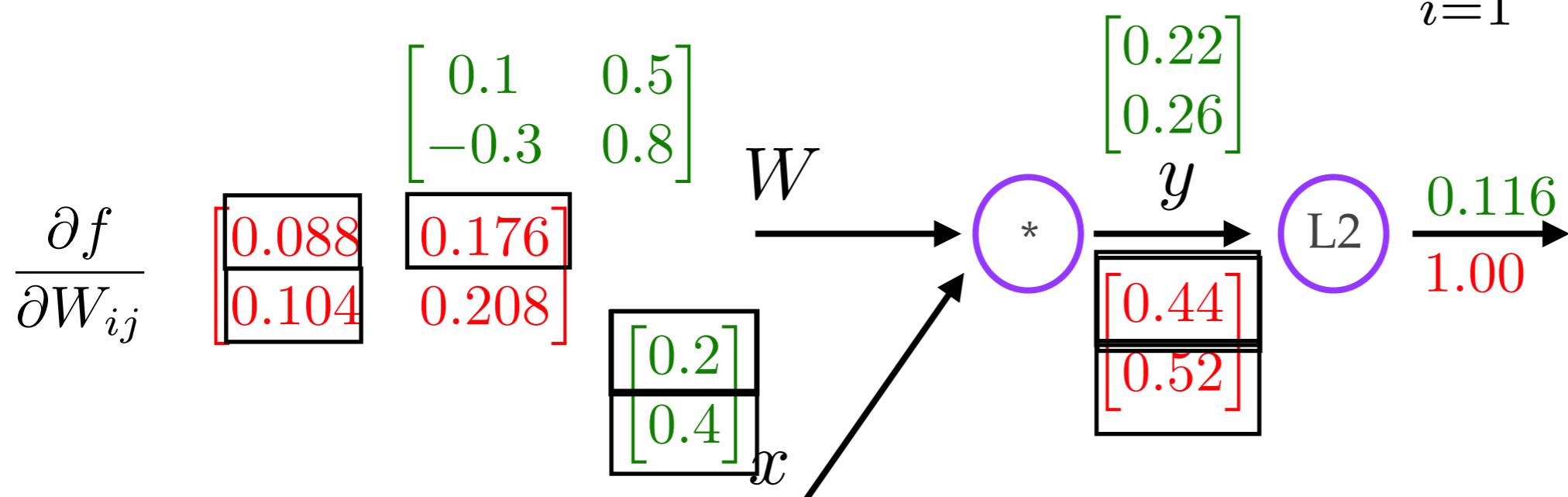
$$y = W \cdot x$$

$$\frac{\partial f}{\partial y_i} = 2y_i$$

$$f(y) = \|y\|^2$$

# Backpropagation

- ❖ Ejemplo:  $f(x, W) = \text{L2} = \|W \cdot x\|^2 = \sum_{i=1}^n (W \cdot x)_i^2$



$$y = W \cdot x = \begin{bmatrix} \sum_{k=1}^n W_{1k} x_k \\ \vdots \\ \sum_{k=1}^n W_{nk} x_k \end{bmatrix}$$

$$\frac{\partial y_k}{\partial W_{ij}} = 1_{k=i} x_j \quad \text{gradiente local}$$

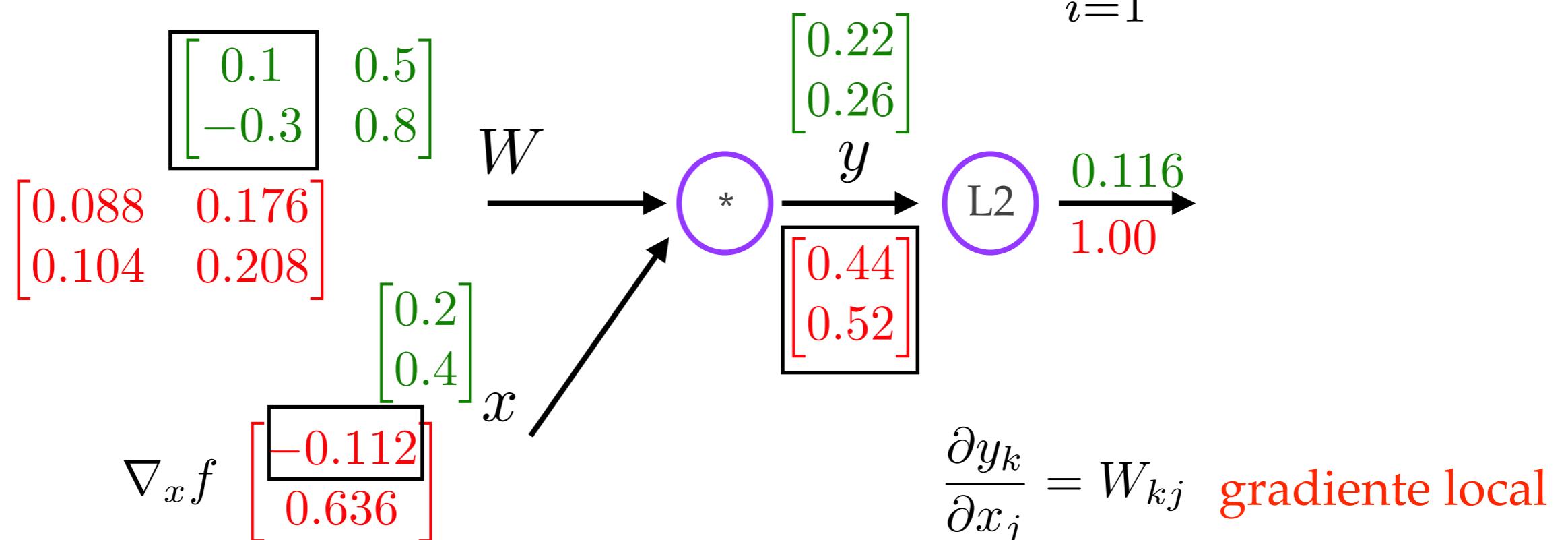
$$\text{gradiente local} * \text{gradiente}$$

$$\frac{\partial f}{\partial W_{ij}} = 2y_i x_j$$

En este caso es una matriz cuadrada pero no necesariamente tiene que ser cuadrada

# Backpropagation

- ❖ Ejemplo:  $f(x, W) = \text{L2} = \|W \cdot x\|^2 = \sum_{i=1}^n (W \cdot x)_i^2$



$$y = W \cdot x = \begin{bmatrix} \sum_{k=1}^n W_{1k} x_k \\ \vdots \\ \sum_{k=1}^n W_{nk} x_k \end{bmatrix}$$

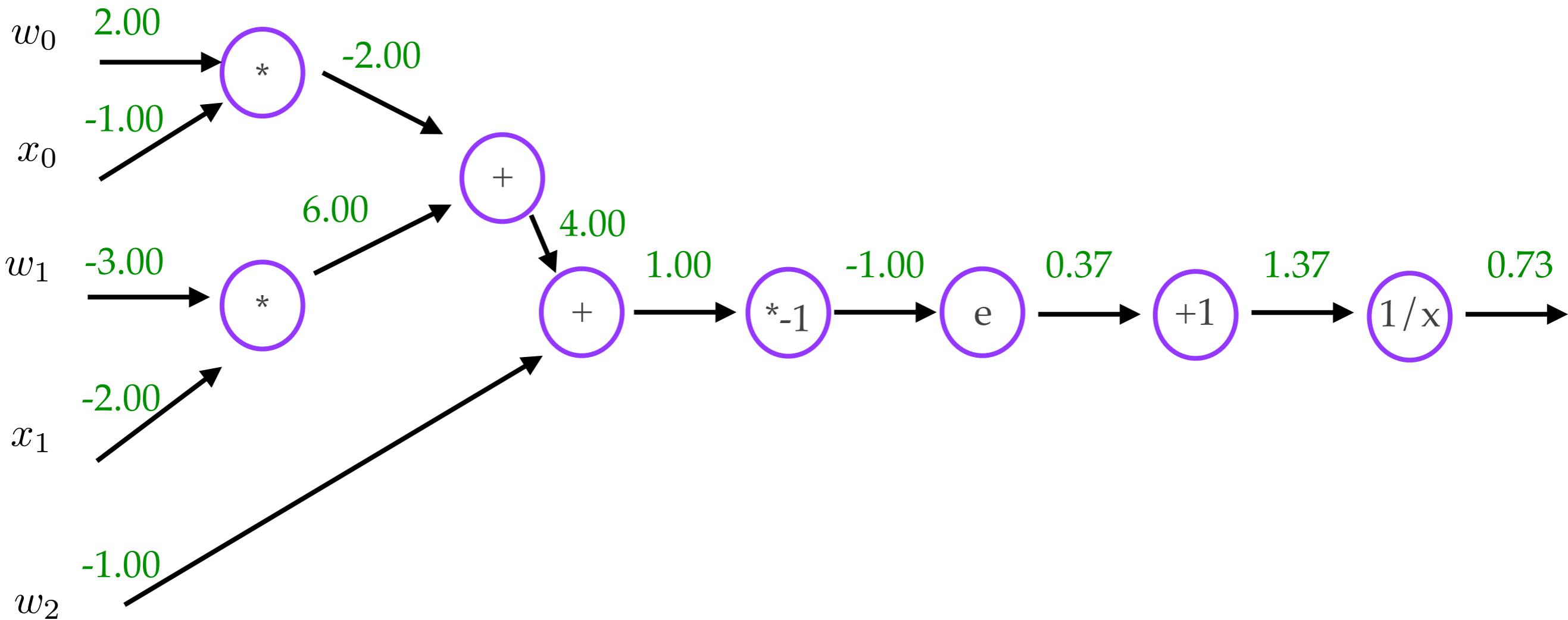
gradiente local \* gradiente

$$\frac{\partial f}{\partial x_j} = \nabla_y f \cdot \nabla_{x_j} y = \sum_k \frac{\partial f}{\partial y_k} \frac{\partial y_k}{\partial x_j} = \sum_k 2y_k W_{kj}$$

$$\nabla_x f = 2W^T \cdot y$$

# Backpropagation

- ❖ Esta forma de realizar cálculos mediante la utilización de grafos o *data flow graphs* es la que implementan la mayoría de las librerías especializadas en NN como Theano o Tensorflow.

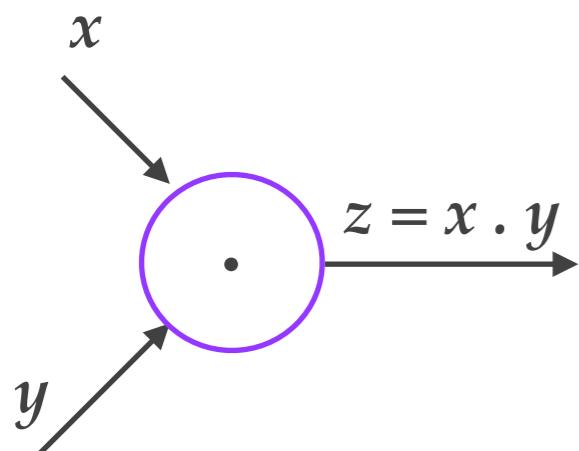


# Implementación

```
class ComputationalGraph(object):  
    #...  
    def forward(inputs):  
        # 1. [pass inputs to input gates...]  
        # 2. forward the computational graph:  
        for gate in self.graph.nodes_topologically_sorted():  
            gate.forward()  
        return loss # the final gate in the graph outputs the loss  
    def backward():  
        for gate in reversed(self.graph.nodes_topologically_sorted()):  
            gate.backward() # little piece of backprop (chain rule applied)  
        return inputs_gradients
```

# Implementación

- ❖ Luego se implementan las operaciones



```
class MultiplyGate(object):  
    def forward(x,y):  
        z = x*y  
        return z  
    def backward(dz):  
        # dx = ... #todo  
        # dy = ... #todo  
        return [dx, dy]
```

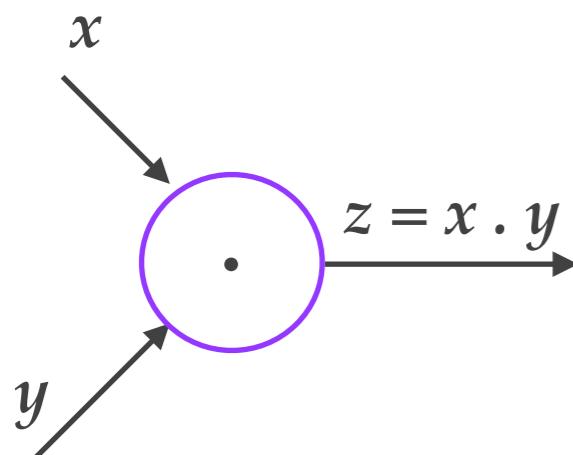
$$\frac{\partial L}{\partial z}$$

$$\frac{\partial L}{\partial x}$$

Img. source: Stanford CS231n 2017

# Implementación

- ❖ Luego se implementan las operaciones

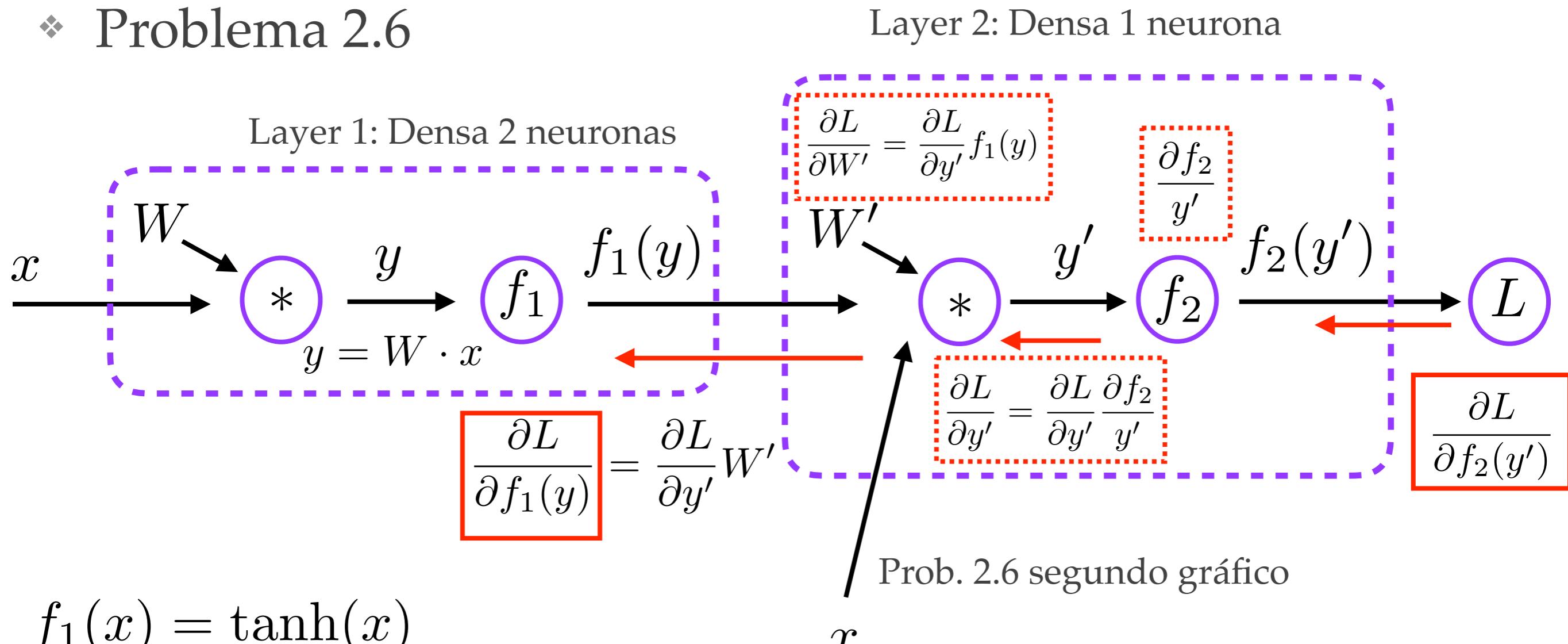


```
class MultiplyGate(object):  
    def forward(x,y):  
        z = x*y  
        self.x = x # must keep these around!  
        self.y = y  
        return z  
  
    def backward(dz):  
        dx = self.y * dz # [dz/dx * dL/dz]  
        dy = self.x * dz # [dz/dy * dL/dz]  
        return [dx, dy]
```

Img. source: Stanford CS231n 2017

# Implementación

## ❖ Problema 2.6



$$f_1(x) = \tanh(x)$$

$$f_2(x) = \tanh(x)$$

$$\text{MSE} = L(y') = \frac{1}{N} \sum_i \|y_t^{(i)} - y'^{(i)}\|_2^2$$