

Lecture 03

Artificial Intelligence

Agenda

- Modo *On-line*, *Batch* y *minibatch*
- Relación entre el perceptrón y la regresión lineal
- Algoritmo de entrenamiento iterativo para regresión lineal
- Gradientes
- Entrenamiento para Adaline

¿Qué se desea después del perceptrón?

Recapitulando, para el perceptrón se tiene:

$$\sigma\left(\sum_{i=1}^m x_i w_i b\right) = \sigma(\mathbf{x}^\top \mathbf{w} + b) = \hat{y} \quad \sigma(z) \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases} \quad b = -\theta$$

Sea

$$\mathcal{D} = ((\mathbf{x}^{[1]}, y^{[1]}), (\mathbf{x}^{[2]}, y^{[2]}), \dots, (\mathbf{x}^{[n]}, y^{[n]})) \in (\Re^m \times \{0, 1\})^n$$

Result: \mathbf{w}, b

$\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$

for $t = 1, \dots, T$ **do**

for $i = 1, \dots, n$ **do**

 cálculo de la salida $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]}\top \mathbf{w} + b);$

 cálculo del error $err := (y^{[i]} - \hat{y}^{[i]});$

 actualización de parámetros $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]}, b := b + err;$

end

end

Principio general de aprendizaje

Como principio general de aprendizaje, aplicable a todos los modelos comunes de neuronas y arquitecturas de redes neuronales (profundas y no profundas):

Sea

$$\mathcal{D} = ((\mathbf{x}^{[1]}, y^{[1]}), (\mathbf{x}^{[2]}, y^{[2]}), \dots, (\mathbf{x}^{[n]}, y^{[n]})) \in (\mathfrak{R}^m \times \{0, 1\})^n$$

Modo *On-line*

Result: \mathbf{w}, b

$\mathbf{w} := \mathbf{0} \in \mathfrak{R}^m, \mathbf{b} := 0;$

for $t = 1, \dots, T$ **do**

for $i = 1, \dots, n$ **do**

 cálculo de la salida;

 cálculo del error ;

 actualización de parámetros \mathbf{w}, b ;

end

end

Modo *On-line*

Result: \mathbf{w}, b
 $\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$
for $t = 1, \dots, T$ **do**
 for $i = 1, \dots, n$ **do**
 cálculo de la salida;
 cálculo del error ;
 actualización de
 parámetros \mathbf{w}, b ;
 end
end

Modo *On-line II*

Result: \mathbf{w}, b
 $\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$
for j iteraciones **do**
 Elija un $(\mathbf{x}^{[i]}, y^{[i]}) \in \mathcal{D}$
 aleatorio;
 cálculo de la salida;
 cálculo del error ;
 actualización \mathbf{w}, b ;
end

Modo *Batch*

Modo *On-line*

Result: \mathbf{w}, b
 $\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$
for $t = 1, \dots, T$ **do**
 for $i = 1, \dots, n$ **do**
 cálculo de la salida;
 cálculo del error ;
 actualización de
 parámetros \mathbf{w}, b ;
 end
end

Result: \mathbf{w}, b
 $\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$
for $t = 1, \dots, T$ **do**
 $\Delta\mathbf{w} := 0, \Delta b := 0;$
 for $i = 1, \dots, n$ **do**
 cálculo de la salida;
 cálculo del error ;
 actualización de
 parámetros $\Delta\mathbf{w}, \Delta b$;
 end
 actualización de parámetros
 \mathbf{w}, b ;
 $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}$ $b := b + \Delta b$;
end

Modo *minibatch*

Modo más común en Deep Learning. Combina *On-line* y *Batch*.

$$\mathcal{D} = ((\mathbf{x}^{[1]}, y^{[1]}), (\mathbf{x}^{[2]}, y^{[2]}), \dots, (\mathbf{x}^{[n]}, y^{[n]})) \in (\Re^m \times \{0, 1\})^n$$

Result: \mathbf{w}, b

$\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$

for $t = 1, \dots, T$ **do**

for $j = 1, \dots, n/k$ **do**

$\Delta\mathbf{w} := 0, \Delta b := 0;$

for $\{(\mathbf{x}^{[i]}, y^{[i]}), \dots, (\mathbf{x}^{[i+k]}, y^{[i+k]})\} \subset D$ **do**

 cálculo de la salida;

 cálculo del error ;

 actualización de $\Delta\mathbf{w}, \Delta b$;

end

 actualización \mathbf{w}, b ;

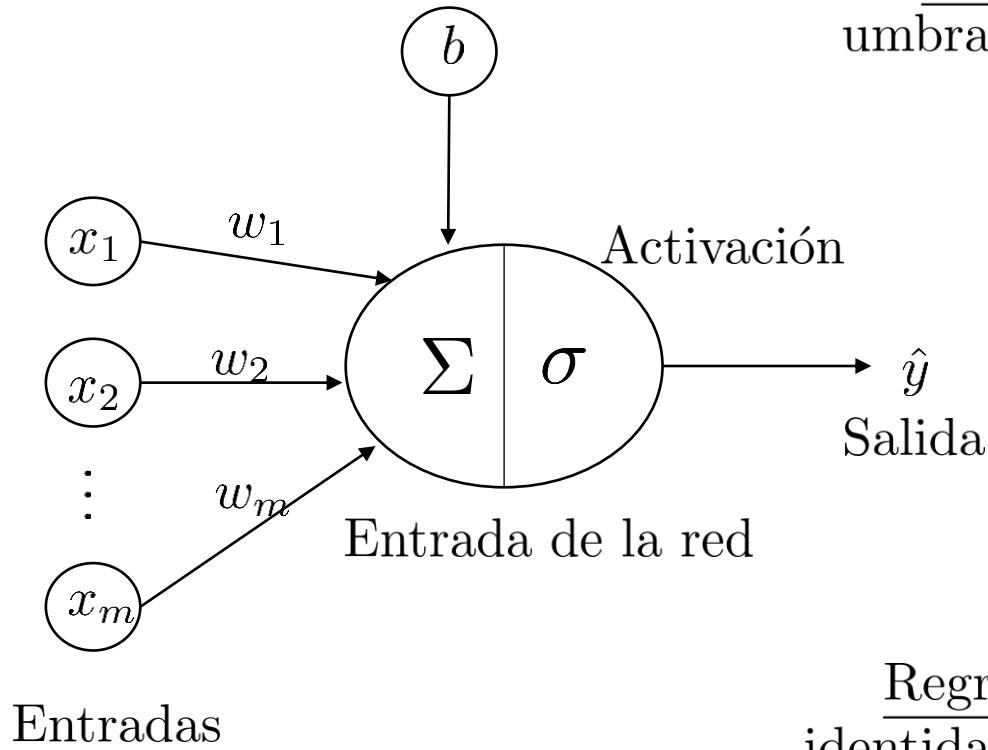
$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w};$

$b := b + \Delta b;$

end

end

Regresión lineal



Perceptrón: la función de activación es la función de umbral. La salida es la etiqueta binaria $\hat{y} \in \{0, 1\}$

Regresión lineal: la función de activación es la función identidad $\sigma(x) = x$. La salida es el número $\hat{y} \in \mathbb{R}$

Regresión lineal (mínimos cuadrados)

El modelo de regresión lineal que se puede usar es el siguiente:

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top,$$

asumiendo que el bias está incluido en \mathbf{w} , y la matriz de diseño tiene un vector adicional. Es posible resolver este modelo usando *ecuaciones normales*, i.e., solución analítica.

Regresión lineal (mínimos cuadrados)

Fuerza bruta

Una forma de ajustar un modelo lineal (y cualquier red neuronal) es inicializando los parámetros en 0's o en valores aleatorios muy pequeños. De esta forma, en k iteraciones:

- Elegir otro conjunto aleatorio de pesos.
- Si el desempeño del modelo es mejor, se conservan los nuevos pesos.
- Si el desempeño del modelo es peor, se descartan.

Este método asegura encontrar la solución óptima para un k bastante grande, pero sería demasiado lento.

Regresión lineal (mínimos cuadrados)

Una mejor manera!

Una mejor forma de hacerlo, es analizando qué efecto tiene un cambio en un parámetro en el desempeño predictivo (pérdida) del modelo; posteriormente, se cambiará solo un poco el peso en la dirección que mejora el desempeño (minimiza la pérdida). Esto se hace con bastantes pasos pequeños, hasta que la pérdida no se decremente más.

Regla de aprendizaje del perceptrón

Result: \mathbf{w}, b
 $\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$
for $t = 1, \dots, T$ **do**
 for $i = 1, \dots, n$ **do**
 $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]\top} \mathbf{w} + b);$
 $err := (y^{[i]} - \hat{y}^{[i]});$
 $\mathbf{w} := \mathbf{w} + err \times \mathbf{x}^{[i]},$
 $b := b + err;$
 end
end

Gradiente descendente estocástico

Result: \mathbf{w}, b
 $\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$
for $t = 1, \dots, T$ **do**
 for $i = 1, \dots, n$ **do**
 $\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]\top} \mathbf{w} + b);$
 $\Delta_{\mathbf{w}} \mathcal{L} = (y^{[i]} - \hat{y}^{[i]}) \mathbf{x}^{[i]};$
 $\Delta_b \mathcal{L} = (y^{[i]} - \hat{y}^{[i]});$
 $\mathbf{w} := \mathbf{w} + \eta \times (-\Delta_{\mathbf{w}} \mathcal{L});$
 $b := b + \eta \times (-\Delta_b \mathcal{L});$
 end
end

gradiente negativo

tasa de aprendizaje

Ciclo *for*

Vectorizado

Result: \mathbf{w}, b

$\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$

for $t = 1, \dots, T$ **do**

for $i = 1, \dots, n$ **do**

$\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]\top} \mathbf{w} + b);$

$\Delta_{\mathbf{w}} \mathcal{L} = (y^{[i]} - \hat{y}^{[i]}) \mathbf{x}^{[i]};$

$\Delta_b \mathcal{L} = (y^{[i]} - \hat{y}^{[i]});$

$\mathbf{w} := \mathbf{w} + \eta \times (-\Delta_{\mathbf{w}} \mathcal{L});$

$b := b + \eta \times (-\Delta_b \mathcal{L});$

end

end

Result: \mathbf{w}, b

$\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$

for $t = 1, \dots, T$ **do**

$\Delta_{\mathbf{w}} \mathcal{L} := 0, \Delta_b \mathcal{L} := 0;$

for $i = 1, \dots, n$ **do**

$\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]\top} \mathbf{w} + b);$

end

for $j = 1, \dots, m$ **do**

$\frac{\partial \mathcal{L}}{\partial w_j} = (y^{[i]} - \hat{y}^{[i]}) x_j^{[i]};$

$w_j := w_j + \eta \times \left(-\frac{\partial \mathcal{L}}{\partial w_j} \right);$

end

$\frac{\partial \mathcal{L}}{\partial b} = (y^{[i]} - \hat{y}^{[i]});$

$b := b + \eta \times \left(-\frac{\partial \mathcal{L}}{\partial b} \right);$

end

Modo *On-line*

Result: \mathbf{w}, b

$\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$

for $t = 1, \dots, T$ **do**

$\Delta\mathbf{w} := 0, \Delta b := 0;$

for $i = 1, \dots, n$ **do**

$\hat{y}^{[i]} := \sigma(\mathbf{x}^{[i]\top} \mathbf{w} + b);$

end

for $j = 1, \dots, m$ **do**

$\frac{\partial \mathcal{L}}{\partial w_j} = (y^{[i]} - \hat{y}^{[i]})x_j^{[i]};$

$w_j := w_j + \eta \times \left(-\frac{\partial \mathcal{L}}{\partial w_j} \right);$

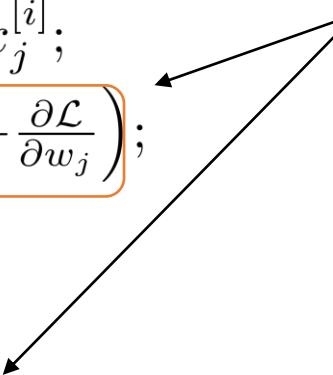
end

$\frac{\partial \mathcal{L}}{\partial b} = (y^{[i]} - \hat{y}^{[i]});$

$b := b + \eta \times \left(-\frac{\partial \mathcal{L}}{\partial b} \right);$

end

Coincidencialmente, parece casi el mismo de la regla del perceptrón, exceptuando que la predicción es un número real y se tiene una tasa de aprendizaje. Esta regla de aprendizaje se llama
gradiente descendente estocástico



Gradiente: derivadas de funciones multivariadas

Si se tiene la función $f(x, y, z, \dots)$,

el gradiente tendrá la forma $\Delta f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \\ \vdots \end{bmatrix}$.

Ejemplo

Para la función $f(x, y) = x^2y + y$ tendremos el gradiente $\Delta f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix}$,
donde

$$\frac{\partial f}{\partial x} = \frac{\partial}{\partial x}x^2y + y = 2xy$$

y

$$\frac{\partial f}{\partial y} = \frac{\partial}{\partial y}x^2y + y = x^2 + 1.$$

Por tanto, el gradiente de la función f estará definido como

$$\Delta f(x, y) = \begin{bmatrix} 2xy \\ x^2 + 1 \end{bmatrix}$$

Gradientes y regla de la cadena

Suponga que se tiene la siguiente función compuesta:

$$f(g(x), h(x))$$

La regla de la cadena regular para una sola entrada es de la forma:

$$\frac{d}{dx}[f(g(x))] = \frac{df}{dg} \frac{dg}{dx}$$

Para dos entradas como es el caso de la función inicial, se tiene

$$\frac{d}{dx}[f(g(x), h(x))] = \frac{\partial f}{\partial g} \frac{dg}{dx} + \frac{\partial f}{\partial h} \frac{dh}{dx}$$

Ejemplo

Calcule la derivada de la función

$$f(g, h) = g^2h + h,$$

donde $g(x) = 3x$ y $h(x) = x^2$.

Solución

$$\frac{\partial f}{\partial g} = 2gh \quad \frac{\partial f}{\partial h} = g^2 + 1$$

$$\frac{dg}{dx} = \frac{d}{dx} 3x = 3 \quad \frac{dh}{dx} = \frac{d}{dx} x^2 = 2x$$

$$\begin{aligned}\frac{d}{dx}[f(g(x), h(x))] &= [(2gh)(3)] + [(g^2 + 1)2x] \\ &= 2xg^2 + 6gh + 2x\end{aligned}$$

Ahora, en forma vectorial, se tendría

$$\frac{d}{dx}[f(g(x), h(x))] = \frac{\partial f}{\partial g} \frac{dg}{dx} + \frac{\partial f}{\partial h} \frac{dh}{dx} = \nabla f \cdot \mathbf{v}'(x),$$

donde ∇f es la matriz Jacobiana de f y

$$\mathbf{v} = \begin{bmatrix} g(x) \\ h(x) \end{bmatrix} \quad \mathbf{v}'(x) = \frac{d}{dx} \begin{bmatrix} g(x) \\ h(x) \end{bmatrix} = \begin{bmatrix} dg/dx \\ dh/dx \end{bmatrix}$$

finalmente, se tiene

$$\nabla f \cdot \mathbf{v}'(x) = \begin{bmatrix} \partial f / \partial g \\ \partial f / \partial h \end{bmatrix} \cdot \begin{bmatrix} dg/dx \\ dh/dx \end{bmatrix} = \frac{\partial f}{\partial g} \frac{dg}{dx} + \frac{\partial f}{\partial h} \frac{dh}{dx}$$

Matriz Jacobiana

De forma general, la matriz Jacobiana de la función

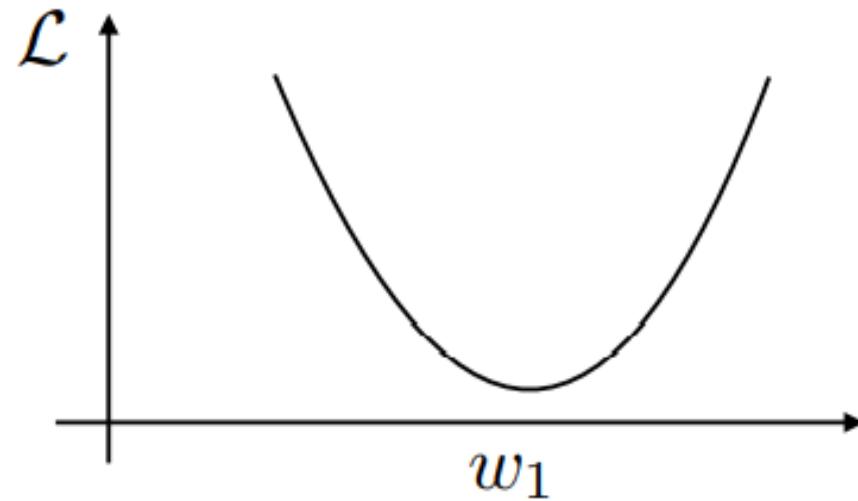
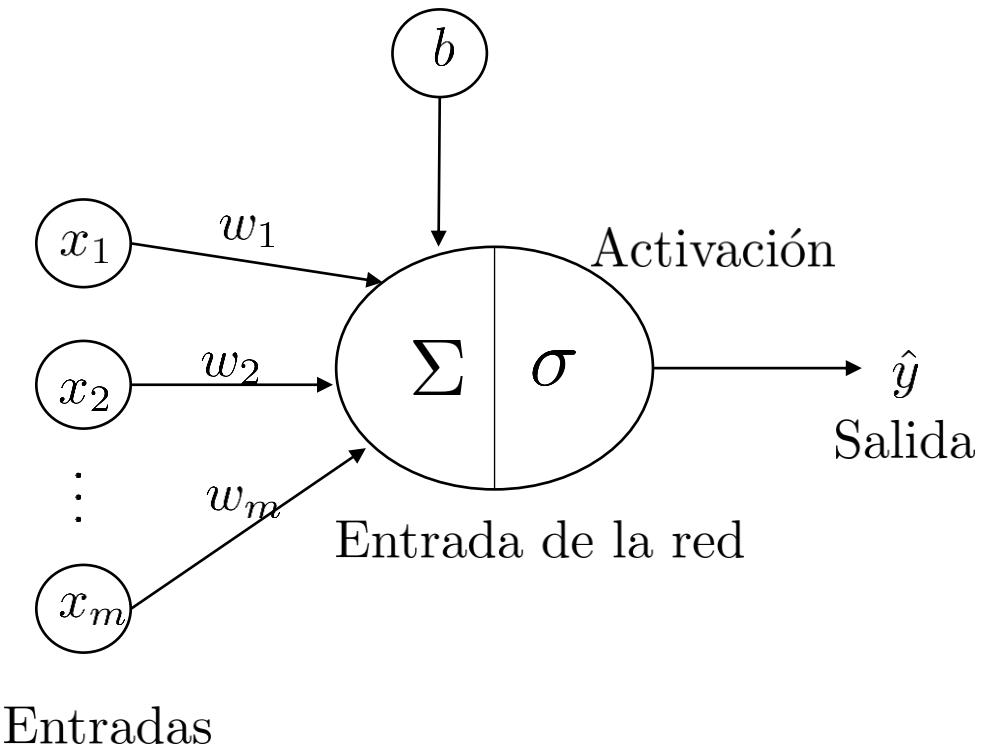
$$\mathbf{f}(x_1, x_2, \dots, x_m) = \begin{bmatrix} f_1(x_1, x_2, \dots, x_m) \\ f_2(x_1, x_2, \dots, x_m) \\ f_3(x_1, x_2, \dots, x_m) \\ \vdots \\ f_m(x_1, x_2, \dots, x_m) \end{bmatrix},$$

estará dado por

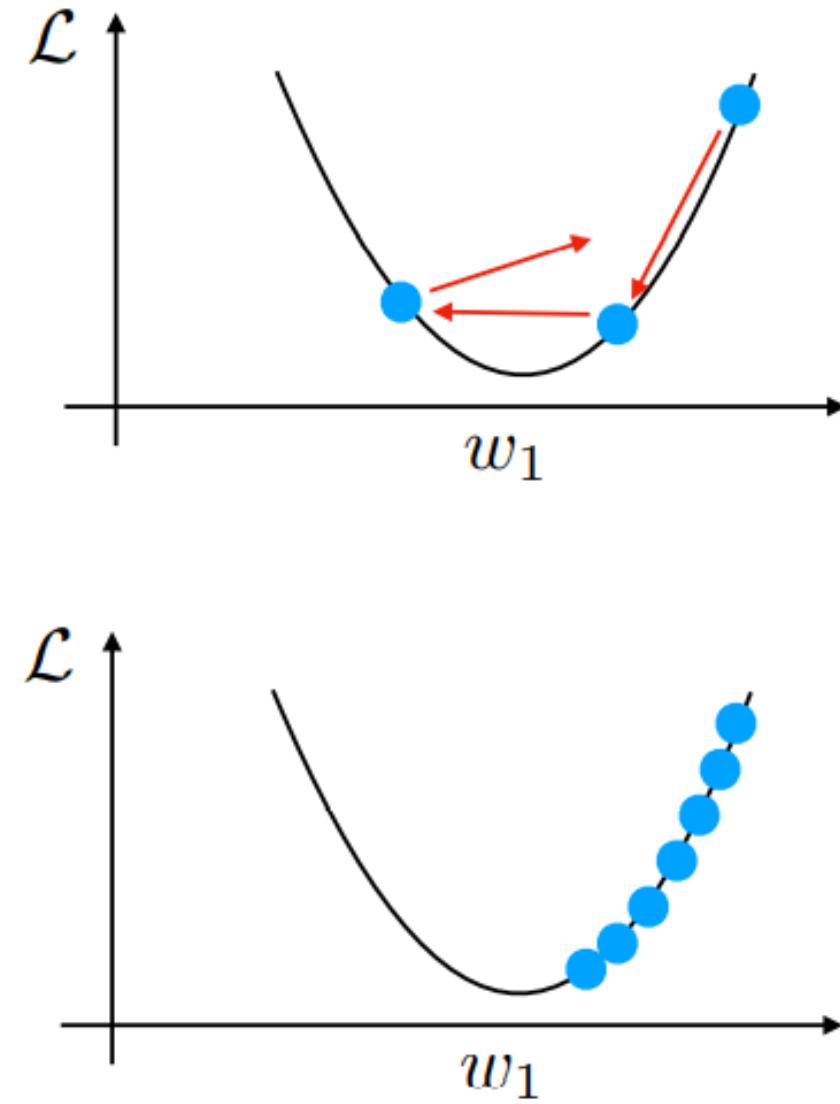
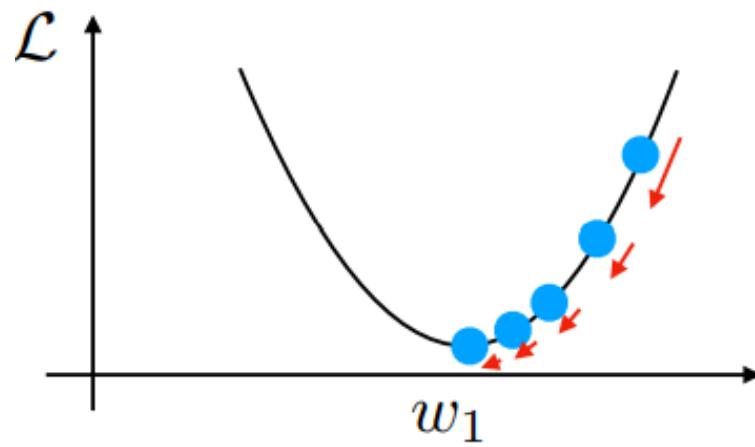
$$J(x_1, x_2, \dots, x_m) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} & \cdots & \frac{\partial f_2}{\partial x_m} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} & \cdots & \frac{\partial f_3}{\partial x_m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \frac{\partial f_m}{\partial x_3} & \cdots & \frac{\partial f_m}{\partial x_m} \end{bmatrix},$$

(↑ $(\nabla f_1)^\top$)

Entrenamiento del regresor lineal con gradiente descendente



Función de pérdida convexa
$$\mathcal{L}(\mathbf{w}, b) = \sum_i (\hat{y}^{[i]} - y^{[i]})^2$$



Derivada de la función de pérdida para regresión lineal

Dada la función de pérdida

$$\mathcal{L}(\mathbf{w}, b) = \sum_i (\hat{y}^{[i]} - y^{[i]})^2,$$

la derivada estaría dada por

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\partial}{\partial w_j} \sum_i (\hat{y}^{[i]} - y^{[i]})^2 \\&= \frac{\partial}{\partial w_j} \sum_i (\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]})^2 \\&= \sum_i 2(\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]}) \frac{\partial}{\partial w_j} (\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]}) \\&= \sum_i 2(\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^\top \mathbf{x}^{[i]})} \frac{\partial}{\partial w_j} \mathbf{w}^\top \mathbf{x}^{[i]} \\&= \sum_i 2(\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^\top \mathbf{x}^{[i]})} x_j^{[i]} \\&= \sum_i 2(\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]}) x_j^{[i]}\end{aligned}$$

Note que la función de activación para el caso de la regresión lineal es la función identidad.

Derivada de la función de pérdida para regresión lineal: caso alternativo

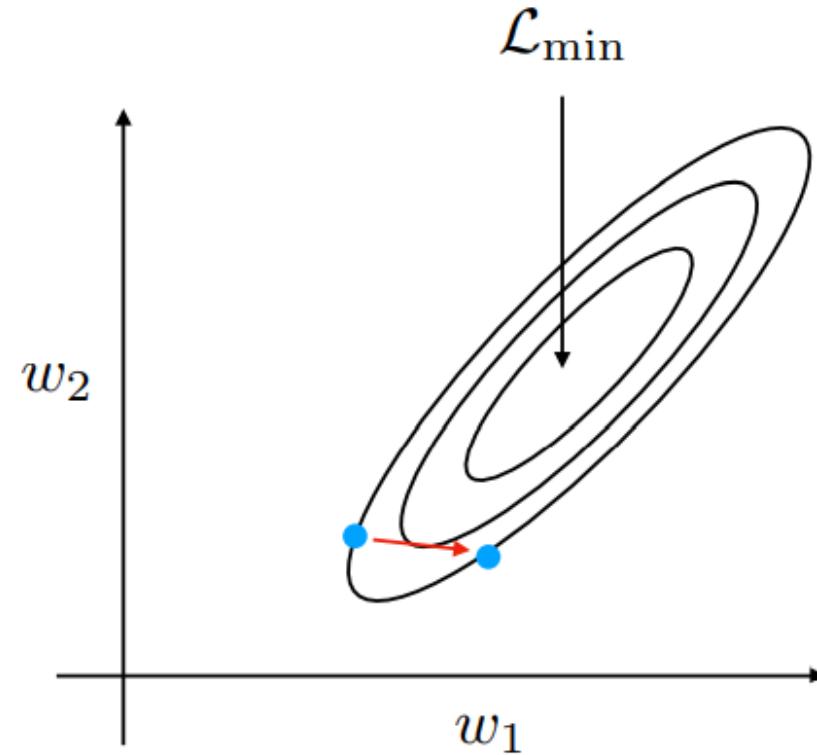
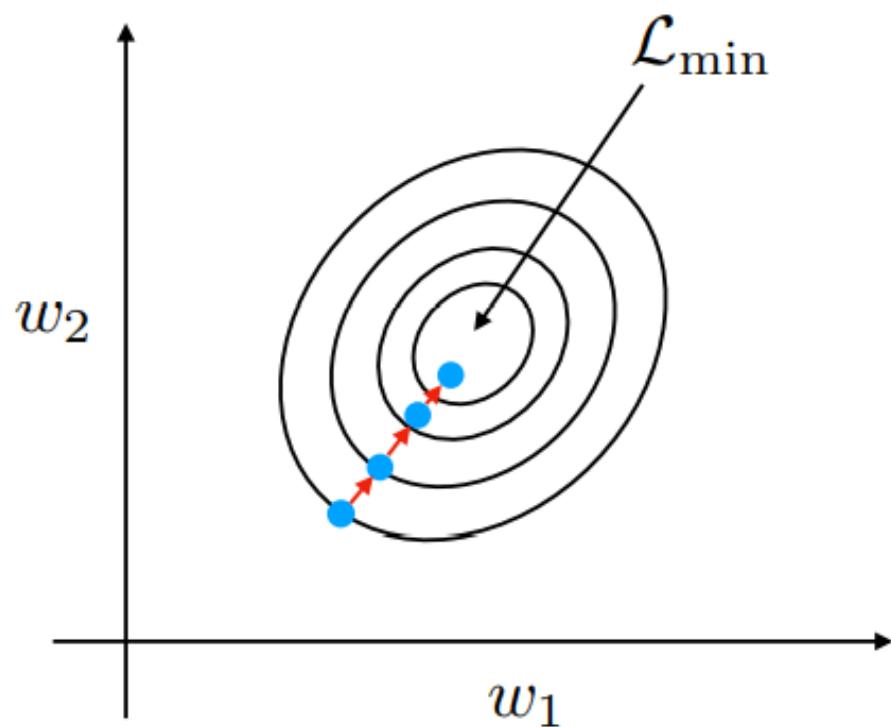
A menudo, la función de pérdida se escala por un factor de $1/2$ por conveniencia:

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2n} \sum_i (\hat{y}^{[i]} - y^{[i]})^2.$$

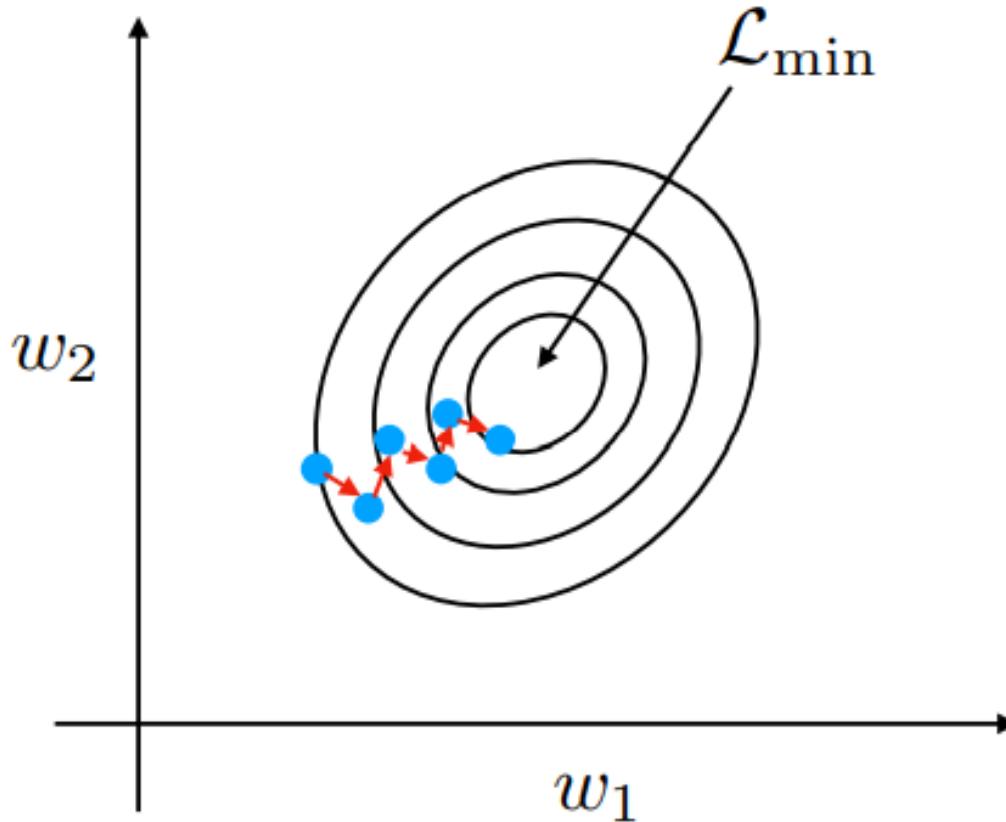
La derivada estaría dada por

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2n} \sum_i (\hat{y}^{[i]} - y^{[i]})^2 \\&= \frac{\partial}{\partial w_j} \sum_i \frac{1}{2n} (\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]})^2 \\&= \sum_i \frac{1}{n} (\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]}) \frac{\partial}{\partial w_j} (\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]}) \\&= \sum_i \frac{1}{n} (\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^\top \mathbf{x}^{[i]})} \frac{\partial}{\partial w_j} \mathbf{w}^\top \mathbf{x}^{[i]} \\&= \sum_i \frac{1}{n} (\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]}) \frac{d\sigma}{d(\mathbf{w}^\top \mathbf{x}^{[i]})} x_j^{[i]} \\&= \sum_i \frac{1}{n} (\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]}) x_j^{[i]}\end{aligned}$$

Gradiente descendente por lotes



Gradiente descendente estocástico



Entrenamiento de una red neuronal de una sola capa con gradiente descendente

Widrow and Hoff's ADALINE (1960)

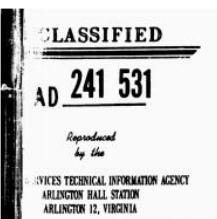
A nicely differentiable neuron model

Widrow, B., & Hoff, M. E. (1960). *Adaptive switching circuits* (No. TR-1553-1). Stanford Univ Ca Stanford Electronics Labs.

Widrow, B. (1960). *Adaptive "adaline" Neuron Using Chemical" memistors.*".



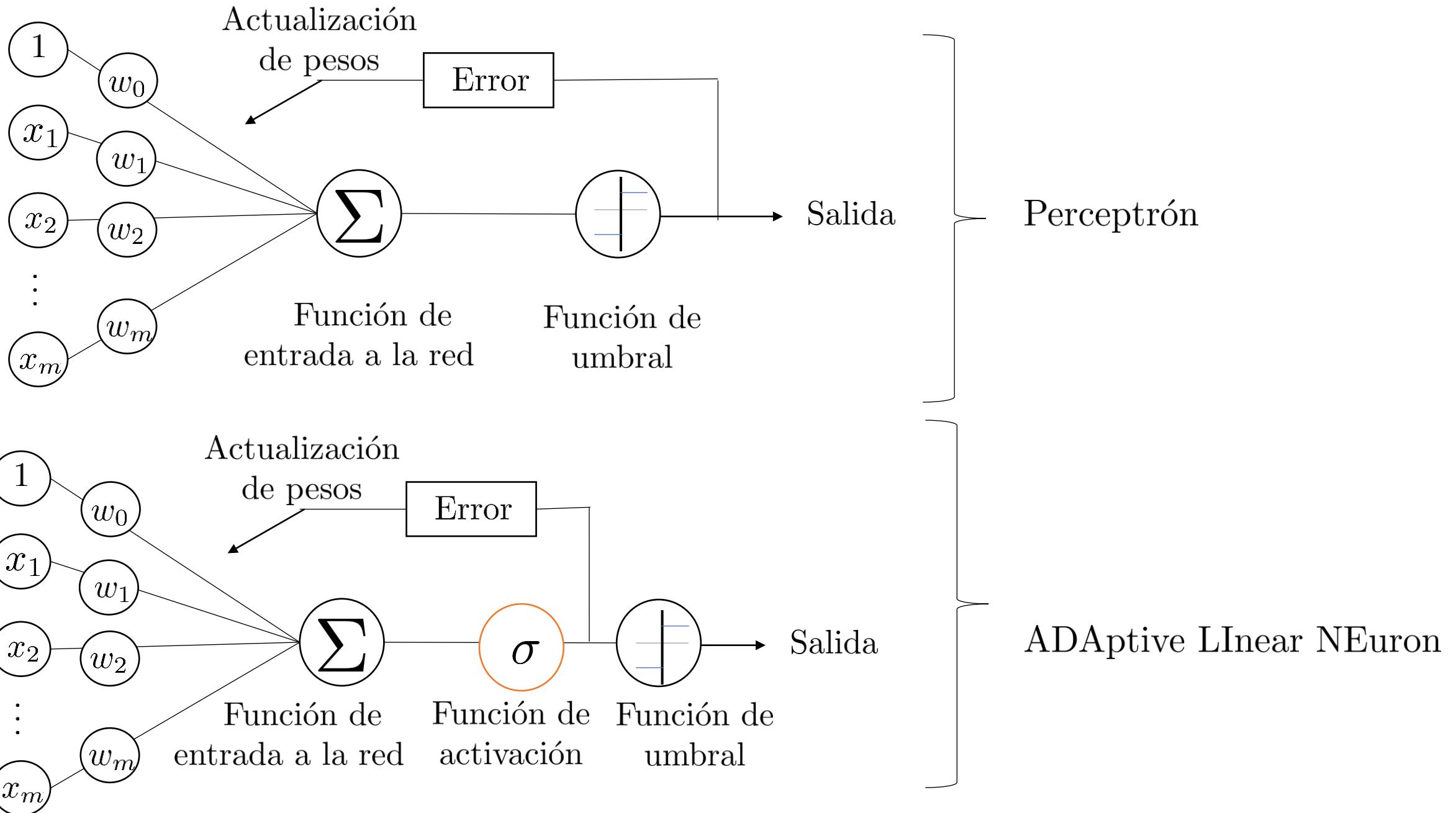
Image source: https://www.researchgate.net/profile/Alexander_Magoun/publication/265789430/figure/fig2/AS:392335251787780b147051421849/ADALINE-An-adaptive-linear-neuron-Manually-adapted-synapses-Designed-and-built-by-Ted.png

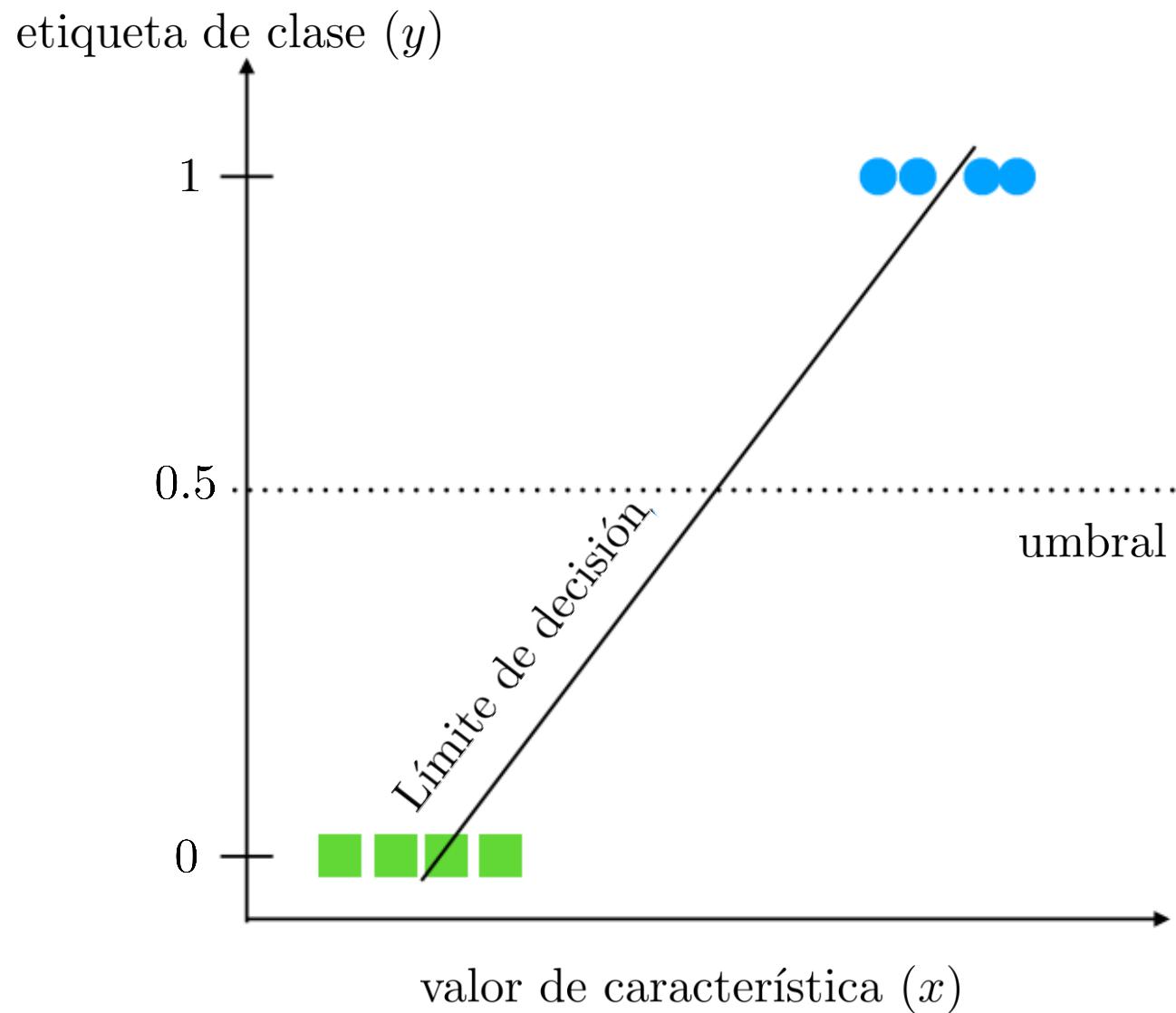


THIS REPORT HAS BEEN DELIMITED
AND CLEARED FOR PUBLIC RELEASE
UNDER DOD DIRECTIVE 5200.20 AND
NO RESTRICTIONS ARE IMPOSED UPON
ITS USE AND DISCLOSURE.

DISTRIBUTION STATEMENT A

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED.

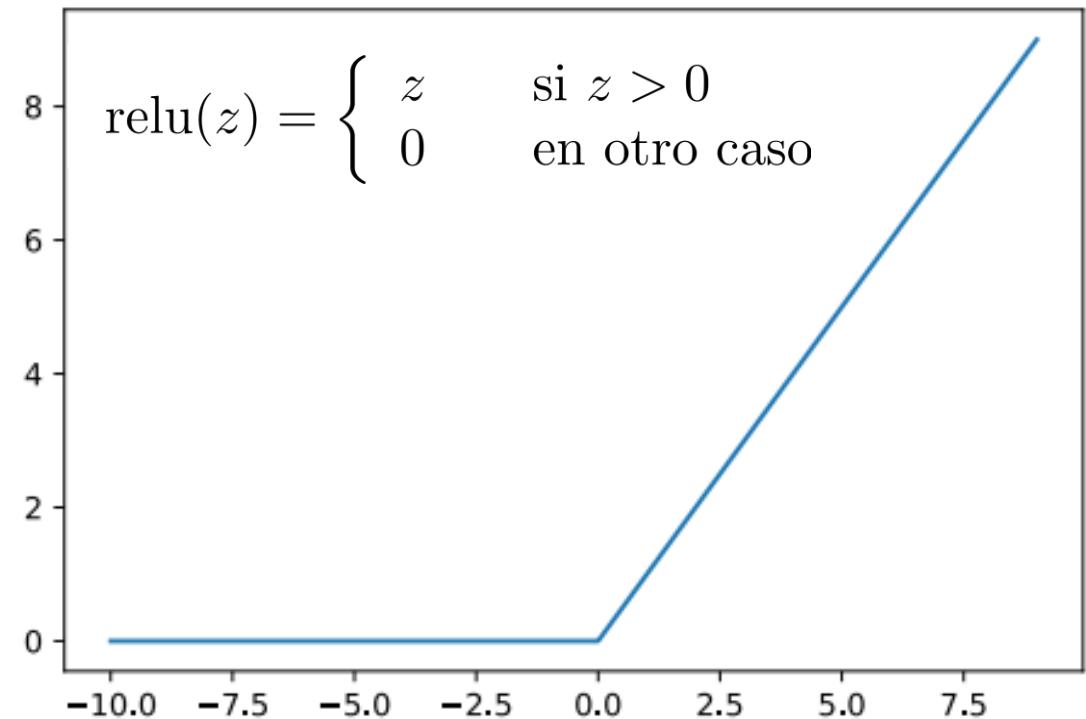




Entendiendo la derivación automática usando grafos computacionales

Se supone la siguiente función de activación:

$$a(x, w, b) = \text{relu}(wx + b)$$



En el sentido estricto

$$\sigma'(z) = \begin{cases} 0 & \text{si } z < 0 \\ 1 & \text{si } z > 0 \\ \text{DNE} & \text{si } z = 0 \end{cases}$$

¿Por qué no es diferenciable?

Pero en el contexto del aprendizaje de máquina y las ciencias de la computación, por conveniencia, es posible decir que

$$\sigma'(z) = \begin{cases} 0 & \text{si } z \leq 0 \\ 1 & \text{si } z > 0 \end{cases}$$

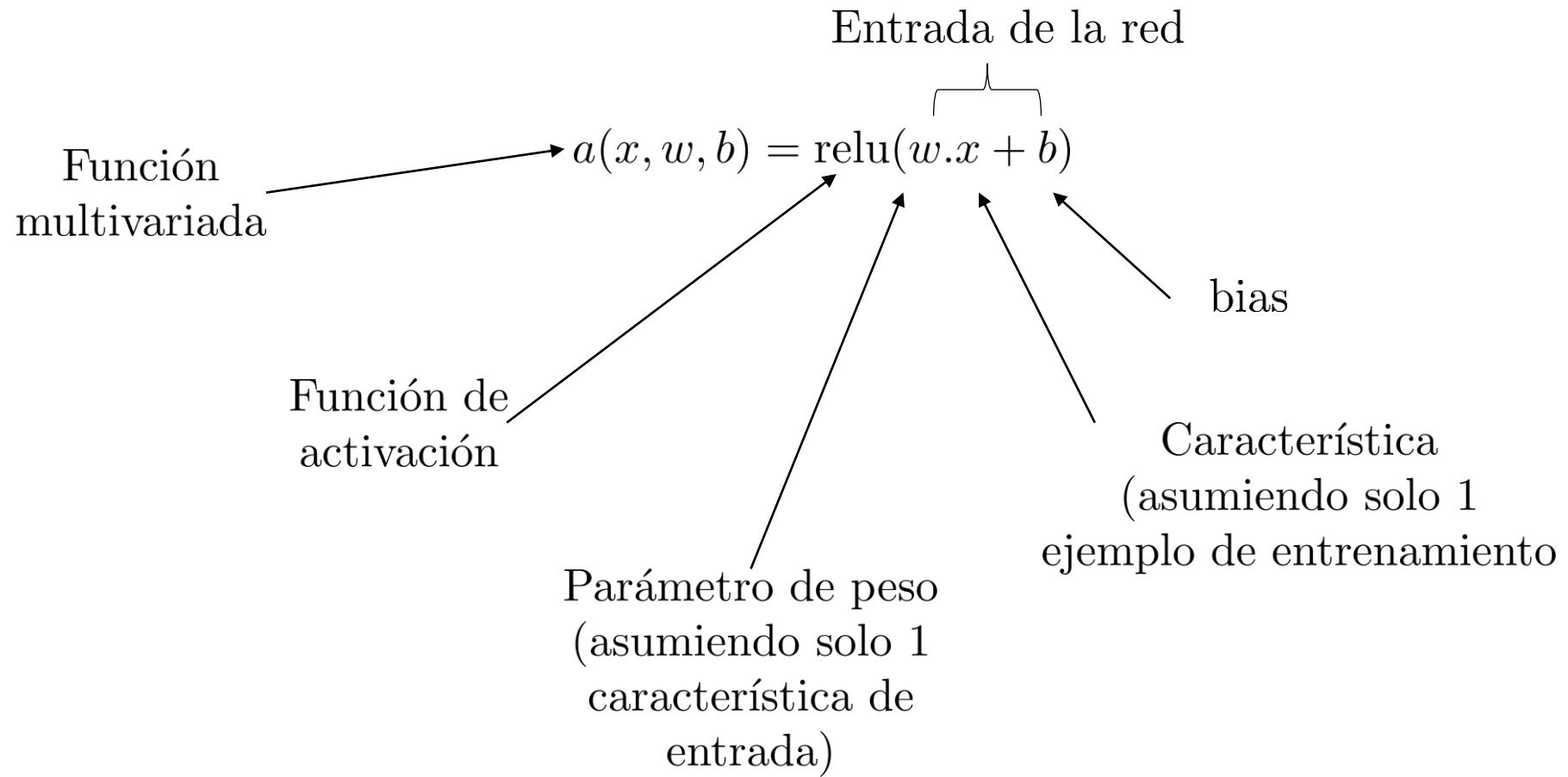
La derivada no existe (DNE) en 0, porque la derivada es diferente si nos acercamos al límite por la izquierda y por la derecha, i.e.,

$$\sigma'(z) = \lim_{z \rightarrow 0} \frac{\max(0, z + \Delta z) - \max(0, z)}{\Delta z}$$

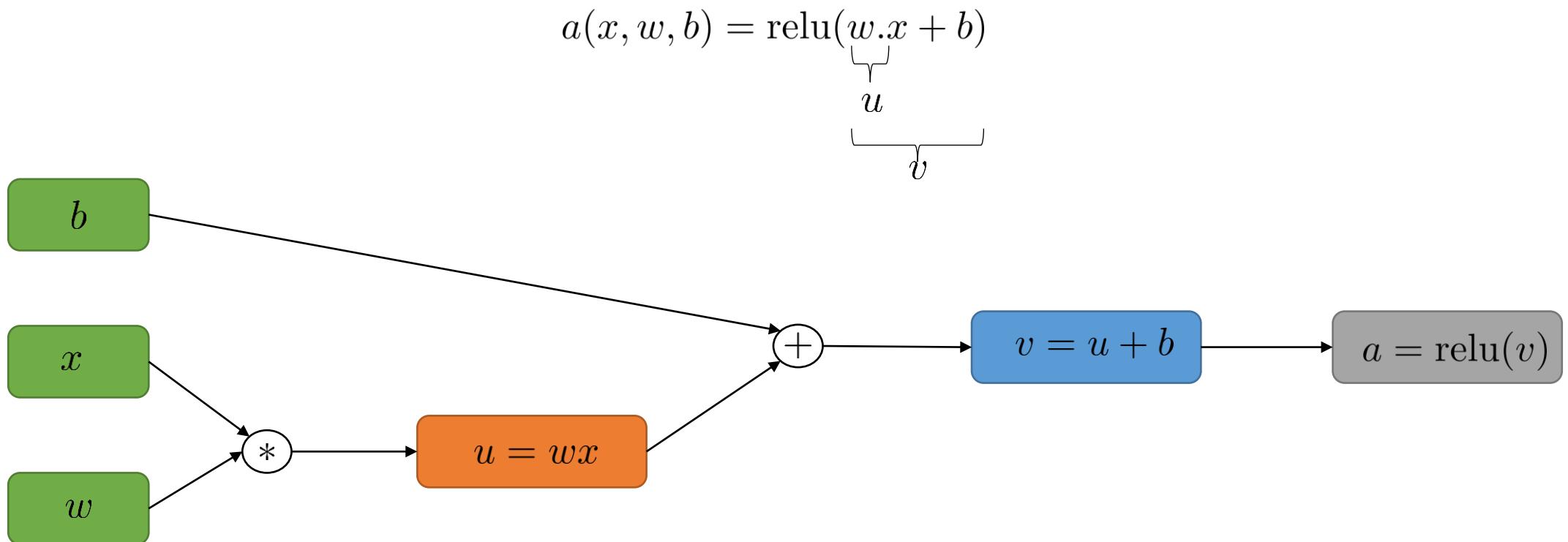
$$\sigma'(0) = \lim_{z \rightarrow 0^+} \frac{0 + \Delta z - 0}{\Delta z} = 1$$

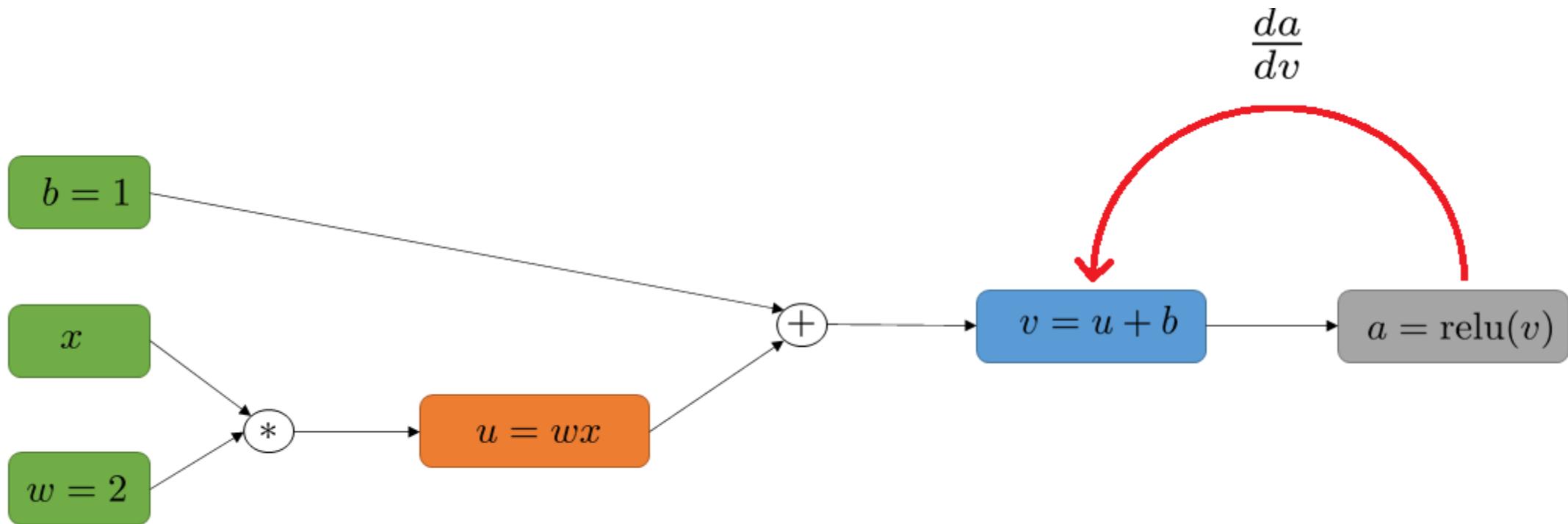
$$\sigma'(0) = \lim_{z \rightarrow 0^-} \frac{0 - 0}{\Delta z} = 0$$

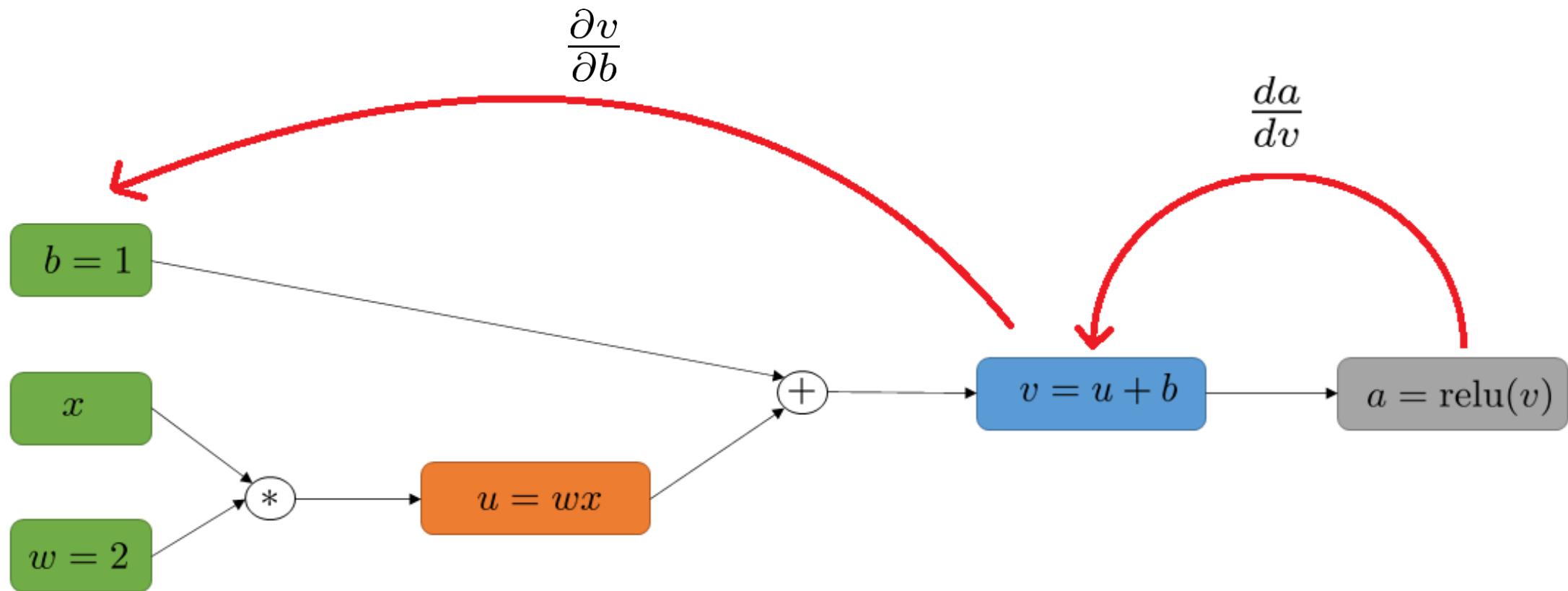
Regresando a la función de activación

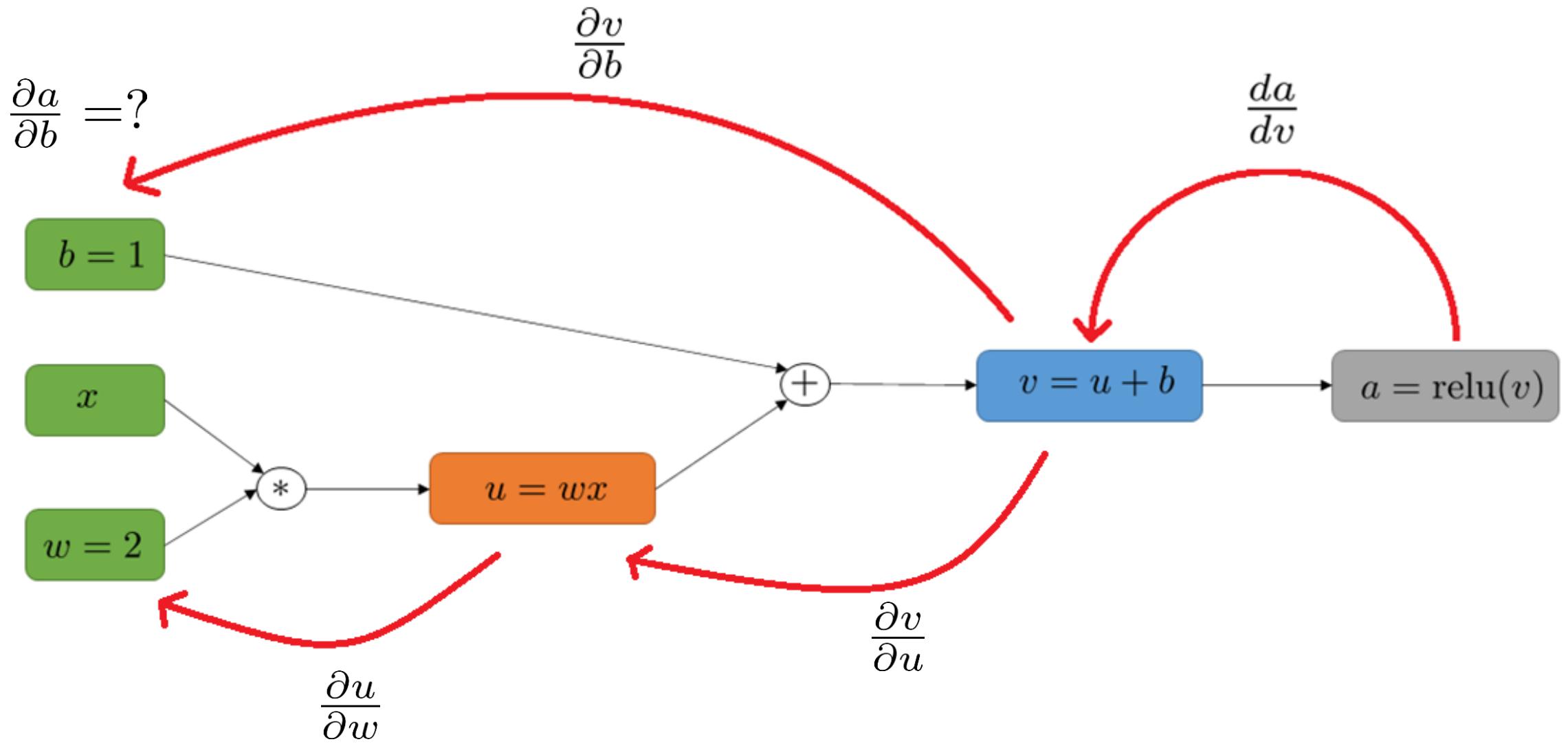


Vista como un grafo sería

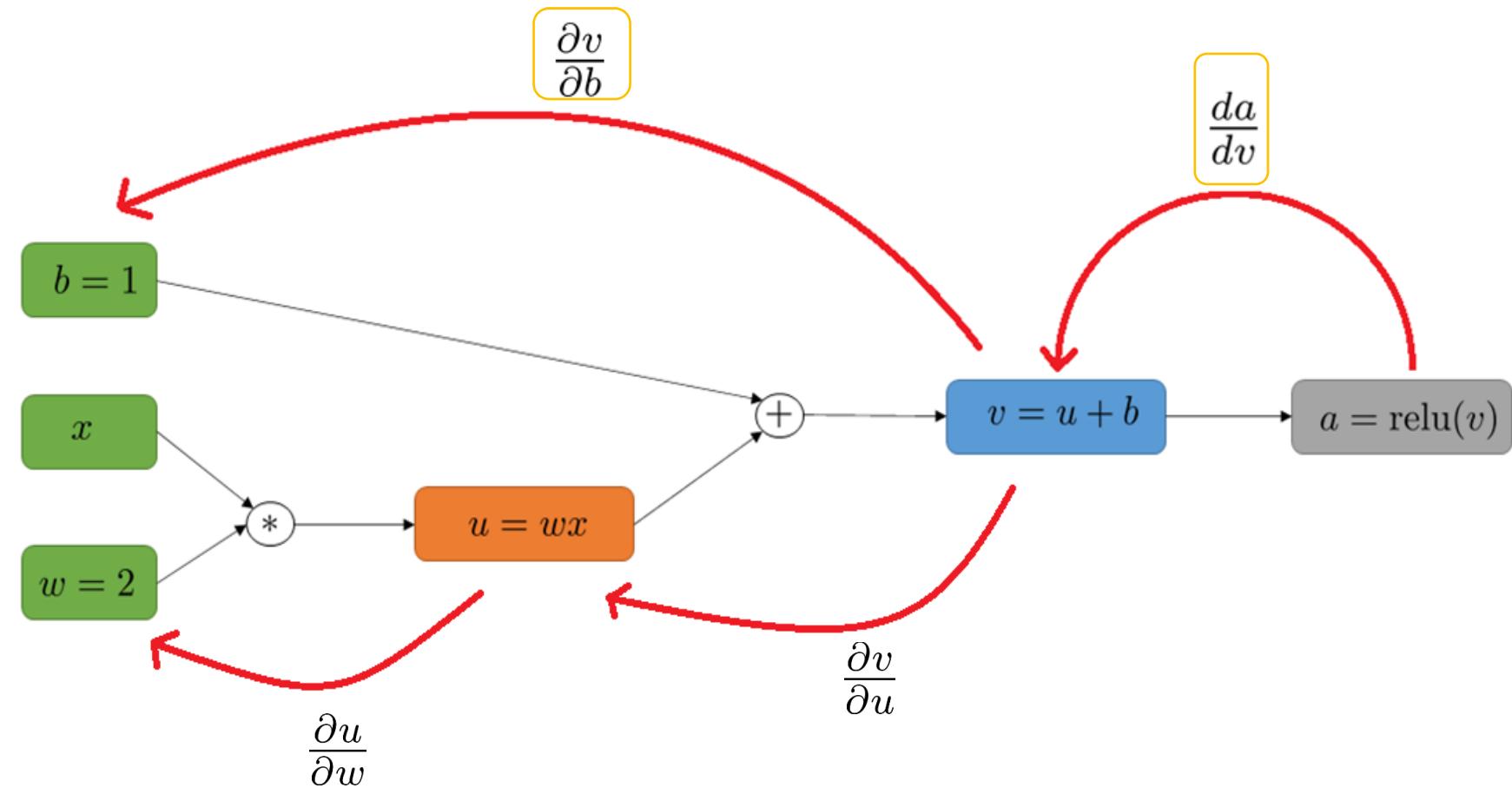




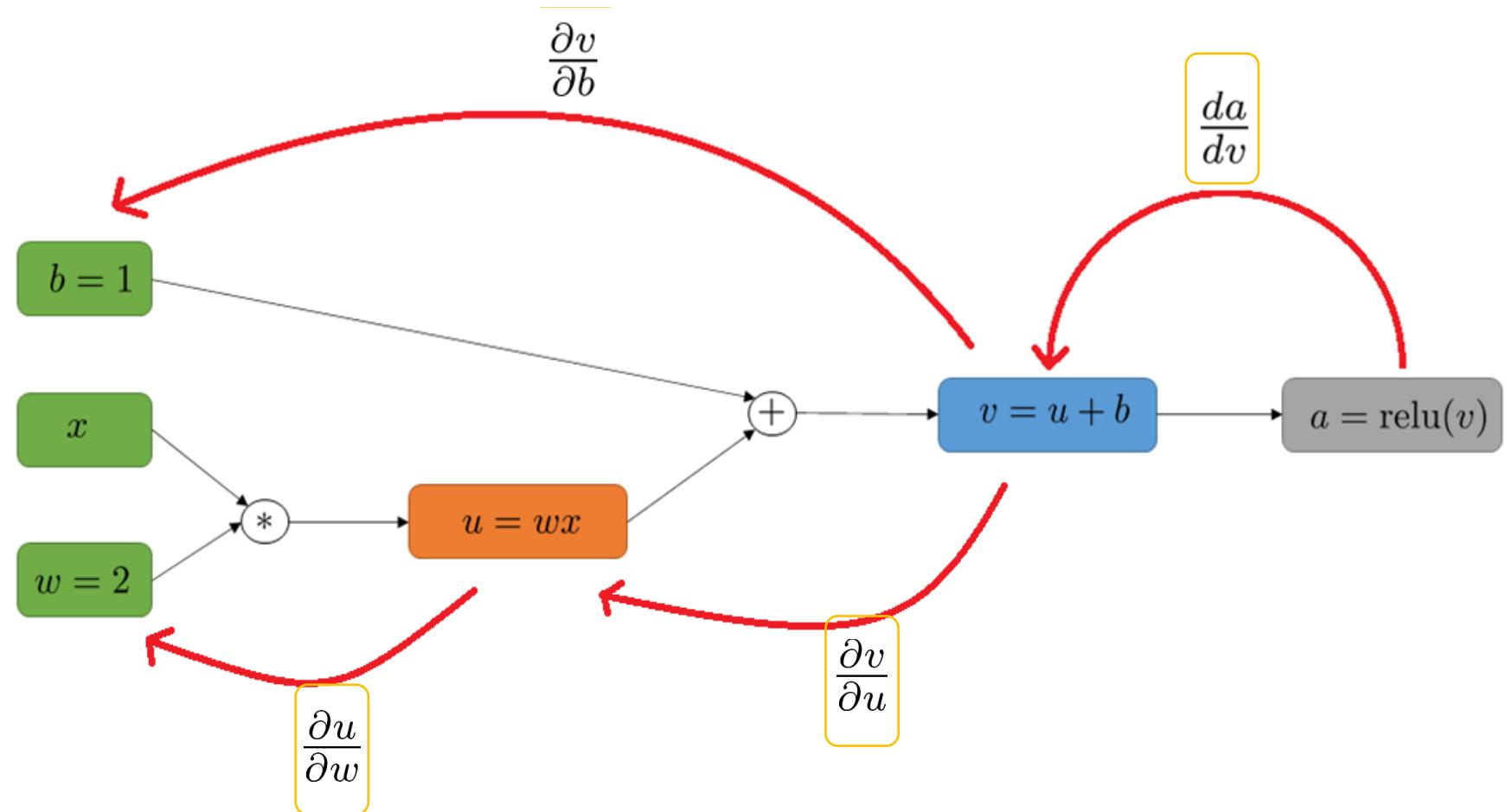




$$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{da}{dv}$$



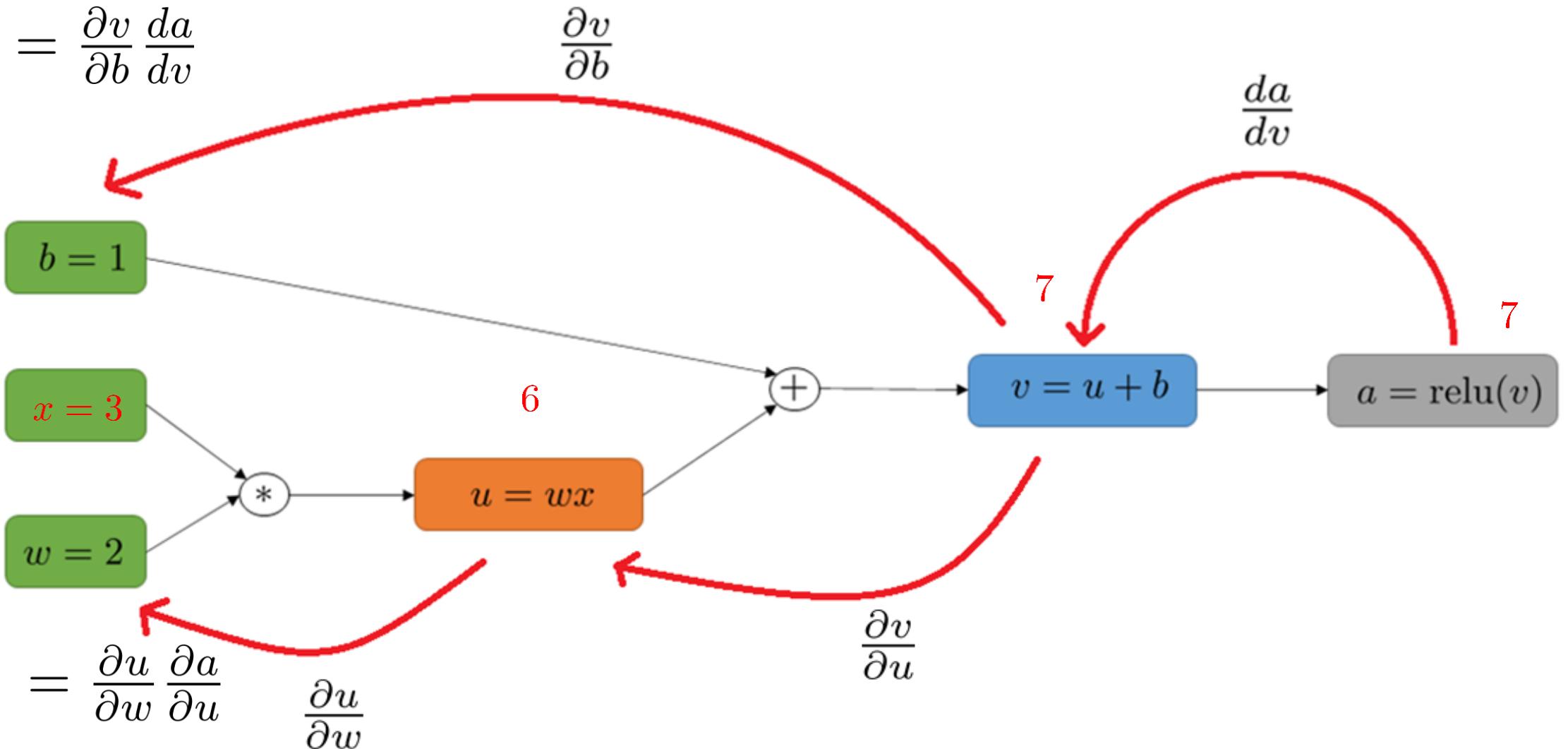
$$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{da}{dv}$$



$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u}$$

$$= \frac{\partial u}{\partial w} \frac{\partial u}{\partial v} \frac{da}{dv}$$

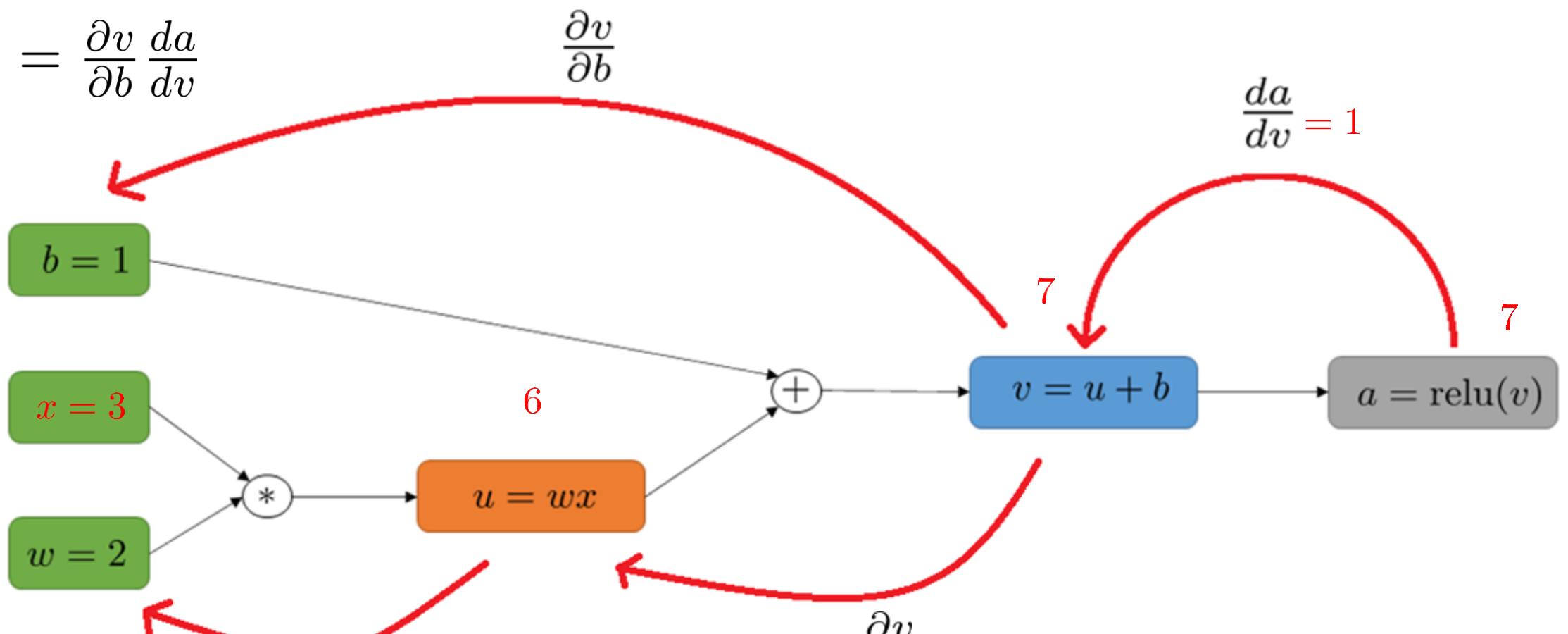
$$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{da}{dv}$$



$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u} \quad \frac{\partial u}{\partial w}$$

$$= \frac{\partial u}{\partial w} \frac{\partial u}{\partial v} \frac{da}{dv}$$

$$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{da}{dv}$$



$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u} \quad \frac{\partial u}{\partial w}$$

$$= \frac{\partial u}{\partial w} \frac{\partial u}{\partial v} \frac{da}{dv}$$

$\text{relu}(z) = \begin{cases} z & \text{si } z > 0 \\ 0 & \text{en otro caso} \end{cases}$
--

$$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{da}{dv}$$

$$\frac{\partial v}{\partial b} = ?$$

$$\frac{da}{dv} = 1$$

$$b = 1$$

$$x = 3$$

$$w = 2$$

$$\begin{aligned}\frac{\partial a}{\partial w} &= \frac{\partial u}{\partial w} \frac{\partial a}{\partial u} \quad \frac{\partial u}{\partial w} \\ &= \frac{\partial u}{\partial w} \frac{\partial u}{\partial v} \frac{da}{dv}\end{aligned}$$

6

$$u = wx$$



$$v = u + b$$

7

$$a = \text{relu}(v)$$

$$\frac{\partial v}{\partial u} = ?$$

Función	Derivada
$f(x) + g(x)$	$f'(x) + g'(x)$

$$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{da}{dv}$$

$$\frac{\partial v}{\partial b} = ?$$

$$\frac{da}{dv} = 1$$

$$b = 1$$

$$x = 3$$

$$w = 2$$

6

$$u = wx$$



$$v = u + b$$

7

$$a = \text{relu}(v)$$

$$\frac{\partial a}{\partial w} = \frac{\partial u}{\partial w} \frac{\partial a}{\partial u}$$

$$\frac{\partial u}{\partial w} = ?$$

$$\frac{\partial v}{\partial u} = 1$$

$$= \frac{\partial u}{\partial w} \frac{\partial u}{\partial v} \frac{da}{dv}$$

$$\frac{\partial a}{\partial b} = \frac{\partial v}{\partial b} \frac{da}{dv} = 1$$

$$\frac{\partial v}{\partial b} = ?$$

$$\frac{da}{dv} = 1$$

$$b = 1$$

$$x = 3$$

$$w = 2$$

6

$$u = wx$$

$$+$$

$$v = u + b$$

$$a = \text{relu}(v)$$

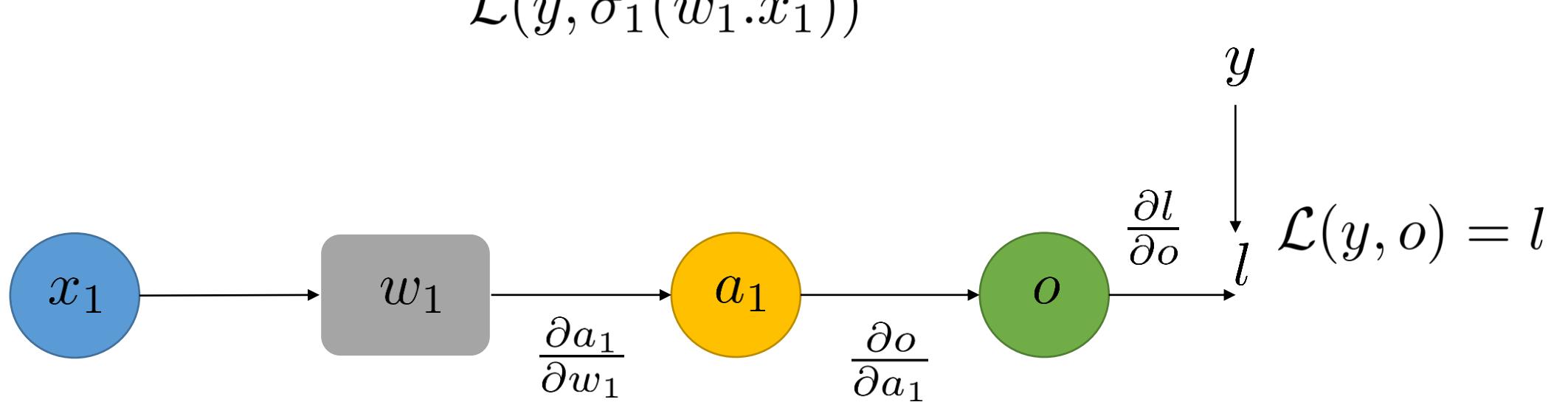
$$\begin{aligned}\frac{\partial a}{\partial w} &= \frac{\partial u}{\partial w} \frac{\partial a}{\partial u} \quad \frac{\partial u}{\partial w} = 3 \\ &= \frac{\partial u}{\partial w} \frac{\partial u}{\partial v} \frac{da}{dv} = 3 * 1 * 1 = 3\end{aligned}$$

$$\frac{\partial v}{\partial u} = 1$$

7

7

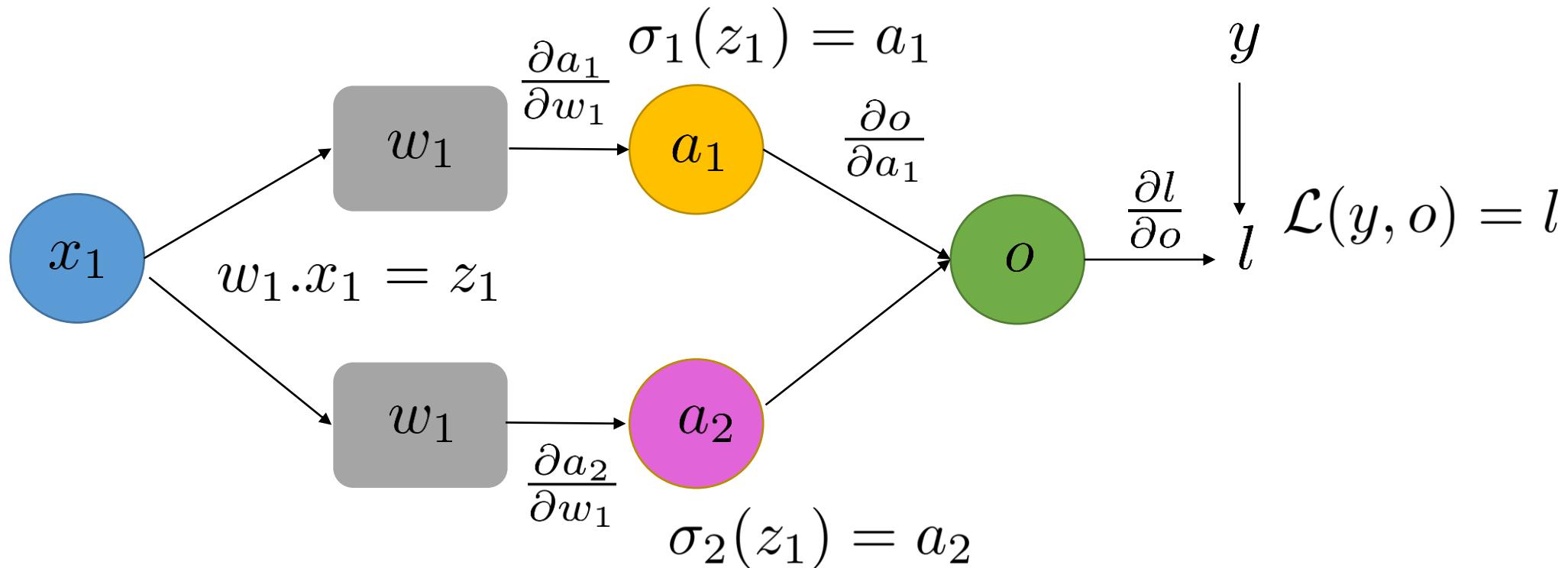
Grafo con un solo camino



$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \frac{\partial a_1}{\partial w_1} \text{ Regla de la cadena univariada}$$

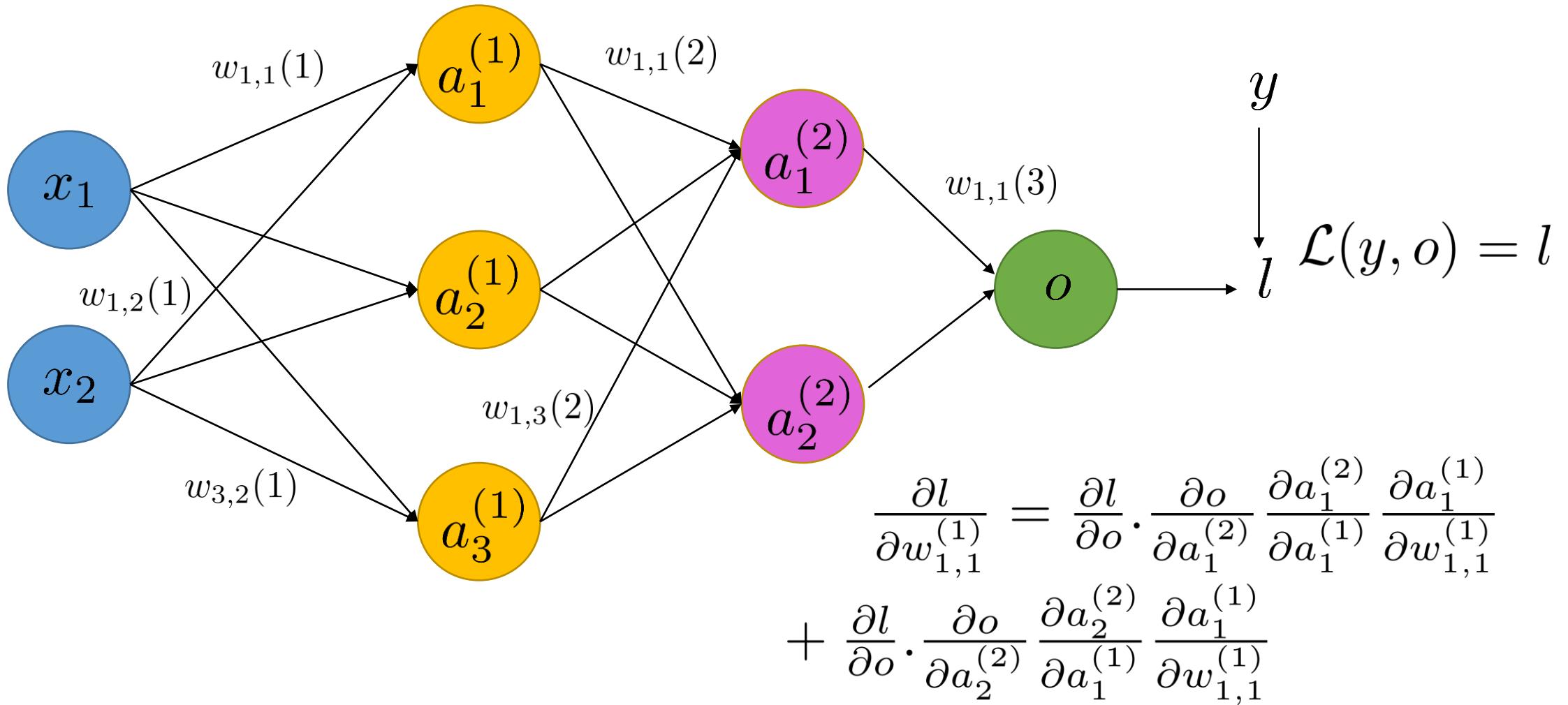
Grafo con peso compartido

$$\mathcal{L}(y, \sigma_3[\sigma_1(w_1 \cdot x_1), \sigma_2(w_1 \cdot x_1)])$$



$$\frac{\partial l}{\partial w_1} = \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_1} \frac{\partial a_1}{\partial w_1} + \frac{\partial l}{\partial o} \cdot \frac{\partial o}{\partial a_2} \frac{\partial a_2}{\partial w_1} \text{ Regla de la cadena multivariada}$$

Capas completamente conectadas



Una mirada más cercana a la API de PyTorch

Algunas características de PyTorch son:

- Se basa en Torch 7, que estuvo basado en Lua e inspirado por Lush
- PyTorch inició en 2016
- Se centra en la flexibilidad y en minimizar la sobrecarga cognitiva
- Naturaleza dinámica de la API autograduada inspirada en Chainer
- Diferenciación automática
- Gráficos de cálculo dinámicos
- Integración de NumPy
- Escrito en C++ y CUDA (CUDA es como C++ para la GPU). En este sentido, se tiene una ganancia del 10% en velocidad al sacar a python de los cálculos.
- Python es el *framework* para la usabilidad

Algunos tutoriales sobre PyTorch

- <https://pytorch.org/tutorials/>
- https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
- <https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf>

y el foro de discusión

<https://discuss.pytorch.org/>



Uso de PyTorch: Paso 1 (Definición)

```
class MultilayerPerceptron(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_feat, num_h1)

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_h1, num_h2)

        ### Output layer
        self.linear_out = torch.nn.Linear(num_h2, num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = self.linear_2(out)
        out = F.relu(out)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas
```

El *backward* se deducirá automáticamente si usamos la nn.

La definición de los parámetros del modelo será instanciada cuando se cree un objeto de su clase

Definir cómo y en qué orden se deben usar los parámetros del modelo en el paso hacia adelante *forward*

Uso de PyTorch: Paso 2 (Creación)

```
torch.manual_seed(random_seed)
model = MultilayerPerceptron(num_features=num_features,
                             num_classes=num_classes)

model = model.to(device)

optimizer = torch.optim.SGD(model.parameters(),
                           lr=learning_rate)
```

Instanciar el modelo
(crea los parámetros del modelo)

Define un método de optimización

```
torch.manual_seed(random_seed)
model = MultilayerPerceptron(num_features=num_features
                            num_classes=num_classes)

model = model.to(device)

optimizer = torch.optim.SGD(model.parameters(),
                           lr=learning_rate)
```

Opcionalmente mueve *modelo* a la GPU,
donde *device* e.g. `torch.device('cuda:0')`

Uso de PyTorch: Paso 3 (Entrenamiento)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        cost = F.cross_entropy(probas, targets)
        optimizer.zero_grad()

        cost.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

model.eval()
with torch.no_grad():
    # compute accuracy
```

Corre para un número específico
de épocas

Itera sobre *minibatches*
en cada época

Si su modelo está sobre la GPU,
los datos también deben estarlo

y = model(x) llama `__call__` y luego `.forward()`,
donde algunas cosas extra se hacen en `__call__`; no llame
y = model.`forward`(x) directamente.

Los gradientes en cada nodo hoja se acumulan bajo
el atributo `.grad`, no solamente se almacenan.
Por ello se deben llevar a cero antes de cada
paso en *backward*

Uso de PyTorch: Paso 3 (Entrenamiento)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad() _____→ Fija el gradiente en cero
                                    (puede no ser cero por un paso previo de forward)
        loss.backward() _____→ Calculo de los gradientes, el backward es construido
                                automáticamente por autograd basado en el método
                                forward() y la función de pérdida
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # compute accuracy
```

Esto corre el método **forward()**

Define la función de pérdida a optimizar

Usa los gradientes para actualizar los pesos de acuerdo con el método de optimización

Uso de PyTorch: Paso 3 (Entrenamiento)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # compute accuracy
```

Para la evaluación, se configura el modelo en modo eval (no será relevante más adelante cuando se use *DropOut* o *BatchNorm*)

Esto previene que se construya el cálculo automático del gradiente en el *backporpagation* para salvar memoria

Uso de PyTorch: Paso 3 (Entrenamiento)

```
for epoch in range(num_epochs):
    model.train()
    for batch_idx, (features, targets) in enumerate(train_loader):

        features = features.view(-1, 28*28).to(device)
        targets = targets.to(device)

        ### FORWARD AND BACK PROP
        logits, probas = model(features)
        loss = F.cross_entropy(logits, targets)
        optimizer.zero_grad()

        loss.backward()

        ### UPDATE MODEL PARAMETERS
        optimizer.step()

    model.eval()
    with torch.no_grad():
        # compute accuracy
```

Se usa `logits` por eficiencia computacional.
Básicamente, internamente usa una función
`log_softmax(logits)`, que
es más estable que `log(softmax(logits))`.

API Orientada a Objetos vs Funcional

```
import torch.nn.functional as F

class MultilayerPerceptron(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()
        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features,
                                        num_hidden_1)
        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_hidden_1,
                                        num_hidden_2)
        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden_2,
                                         num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = self.linear_2(out)
        out = F.relu(out)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas

class MultilayerPerceptron(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()
        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features,
                                        num_hidden_1)
        self.relu1 = torch.nn.ReLU()
        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_hidden_1,
                                        num_hidden_2)
        self.relu2 = torch.nn.ReLU()
        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden_2,
                                         num_classes)
        self.softmax = torch.nn.Softmax()

    def forward(self, x):
        out = self.linear_1(x)
        out = self.relu1(out)
        out = self.linear_2(out)
        out = self.relu2(out)
        logits = self.linear_out(out)
        probas = self.softmax(logits, dim=1)
        return logits, probas
```

Es innecesario dado que estas funciones no necesitan almacenar un estado, pero quizás útil para mantener el seguimiento del orden de las operaciones (cuando se implementa `forward`).

API Orientada a Objetos vs Funcional

Usando *secuencial*

```
import torch.nn.functional as F

class MultilayerPerceptron(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        ### 1st hidden layer
        self.linear_1 = torch.nn.Linear(num_features,
                                       num_hidden_1)

        ### 2nd hidden layer
        self.linear_2 = torch.nn.Linear(num_hidden_1,
                                       num_hidden_2)

        ### Output layer
        self.linear_out = torch.nn.Linear(num_hidden_2,
                                         num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.relu(out)
        out = self.linear_2(out)
        out = F.relu(out)
        logits = self.linear_out(out)
        probas = F.log_softmax(logits, dim=1)
        return logits, probas

class MultilayerPerceptron(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super(MultilayerPerceptron, self).__init__()

        self.my_network = torch.nn.Sequential(
            torch.nn.Linear(num_features, num_hidden_1),
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_1, num_hidden_2),
            torch.nn.ReLU(),
            torch.nn.Linear(num_hidden_2, num_classes)
        )

    def forward(self, x):
        logits = self.my_network(x)
        probas = F.softmax(logits, dim=1)
        return logits, probas
```

Mucho más compacto y claro,
pero **forward** puede ser difícil
de depurar si hay errores.
Sin embargo, si se usa *secuencial*
se pueden tener salidas intermedias.

¿Preguntas?