UNIVERSIDAD EAFIT

Lecture 02
Deep Learning

Ph.D Juan David Martínez Vargas
Ph. D Raul Andrés Castañeda

Escuela de Ciencias Aplicadas e Ingeniería

Agenda

- Modelos neuronales
- Perceptron: Single layer NNs
- Álgebra lineal y cálculo para Deep Learning

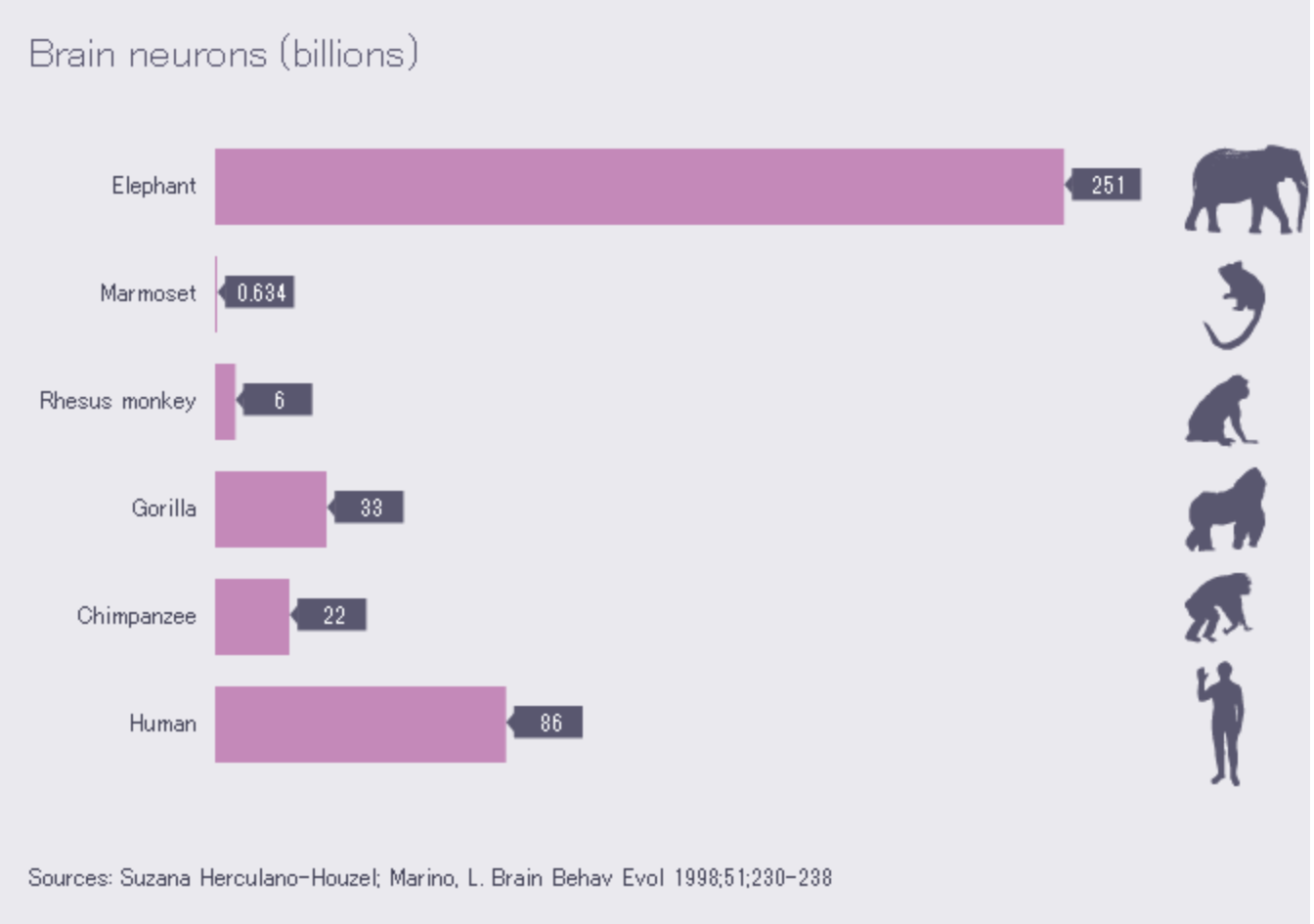
Redes Neuronales: Inspiradas por el Cerebro

https://www.verywellmind.com/how-brain-cells-communicate-with-each-other-2584397

Redes Neuronales: Inspiradas por el Cerebro

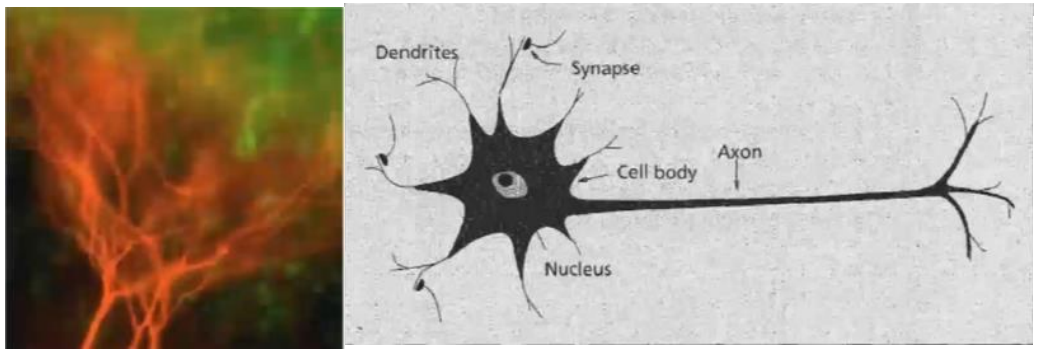
https://medium.com/@adsactly/is-it-a-bird-is-it-a-plane-biomimicry-in-airplanes-9862d331df2e

Número de neuronas en el cerebro



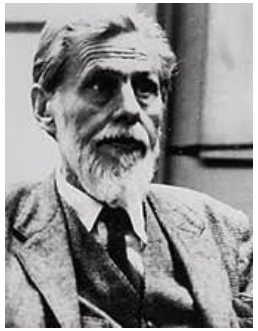
https://en.wikipedia.org/wiki/List_of_animals_by_number_of_neurons#/media/File:Brain_size_comparison_-_Brain_neurons_(billions).png

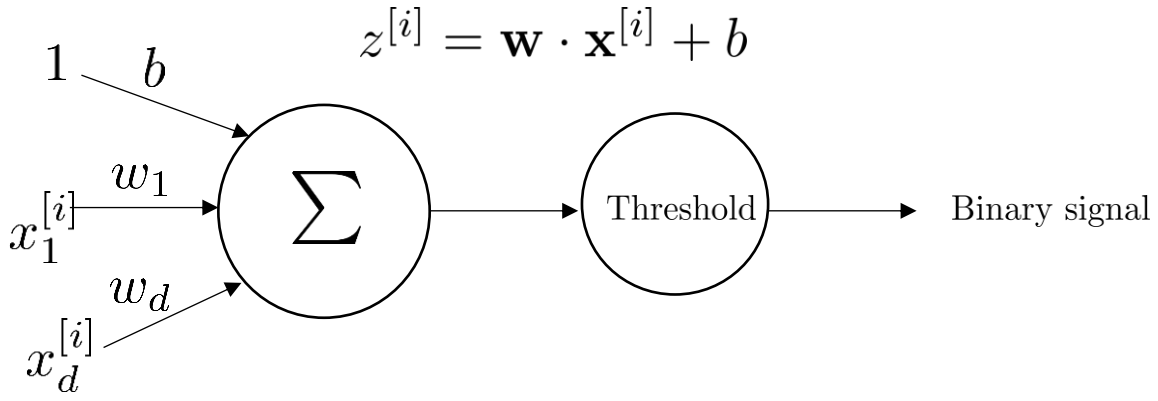
Modelo de Neurona de McCulloch Pitts



A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY

WARREN S. MCCULLOCH and WALTER H. PITTS 1943

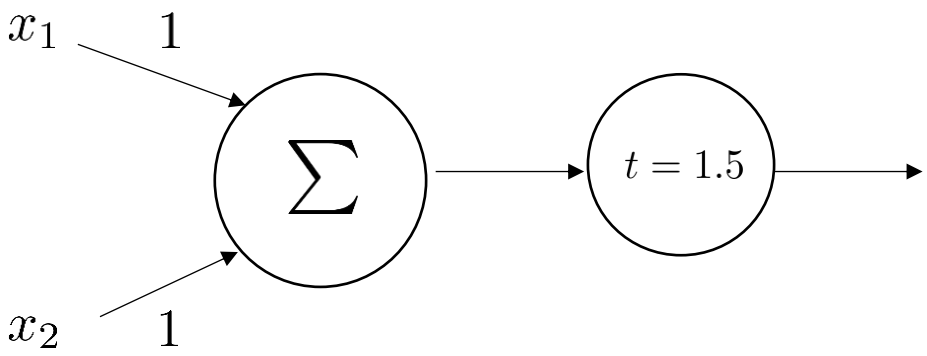




Compuertas Lógicas

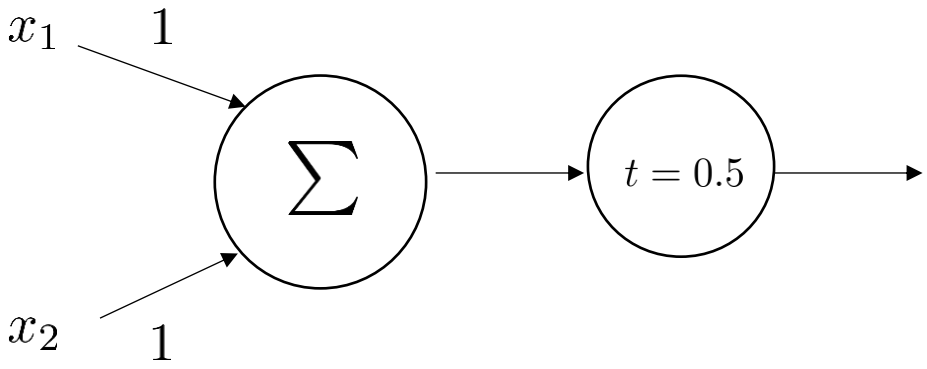
And

x_1	x_2	Out
0	0	0
0	1	0
1	0	0
1	1	1



Or

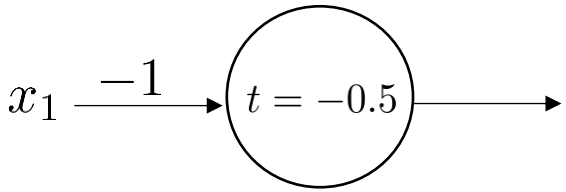
x_1	x_2	Out
0	0	0
0	1	1
1	0	1
1	1	1



Compuertas Lógicas

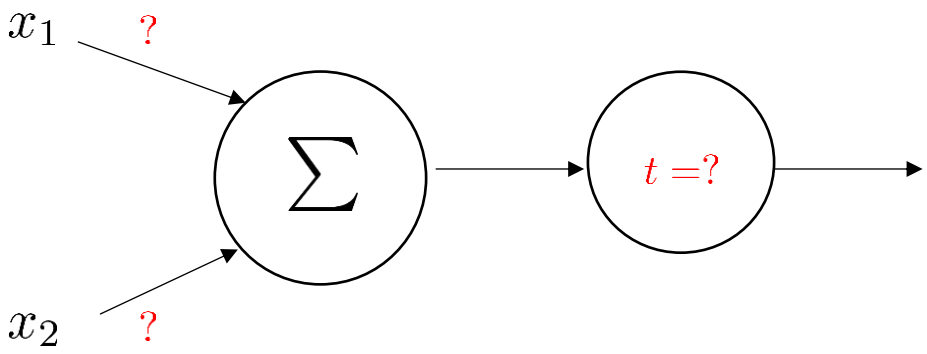
Not

x_1	Out
0	1
1	0

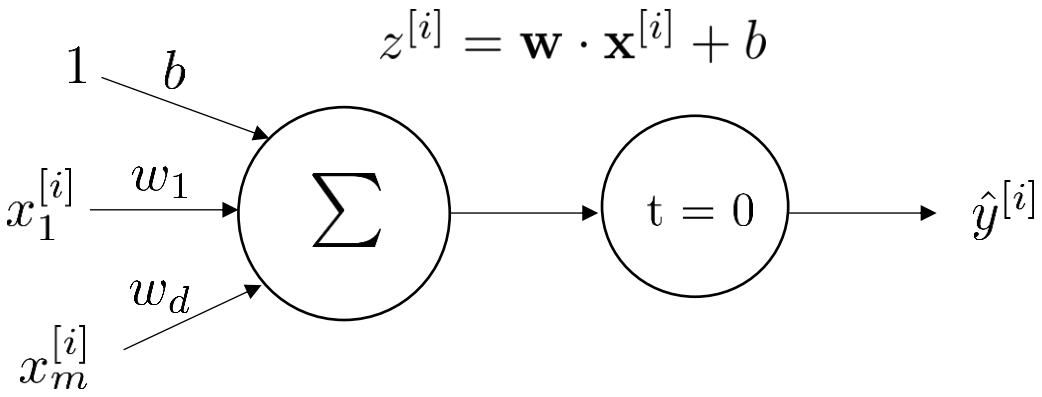


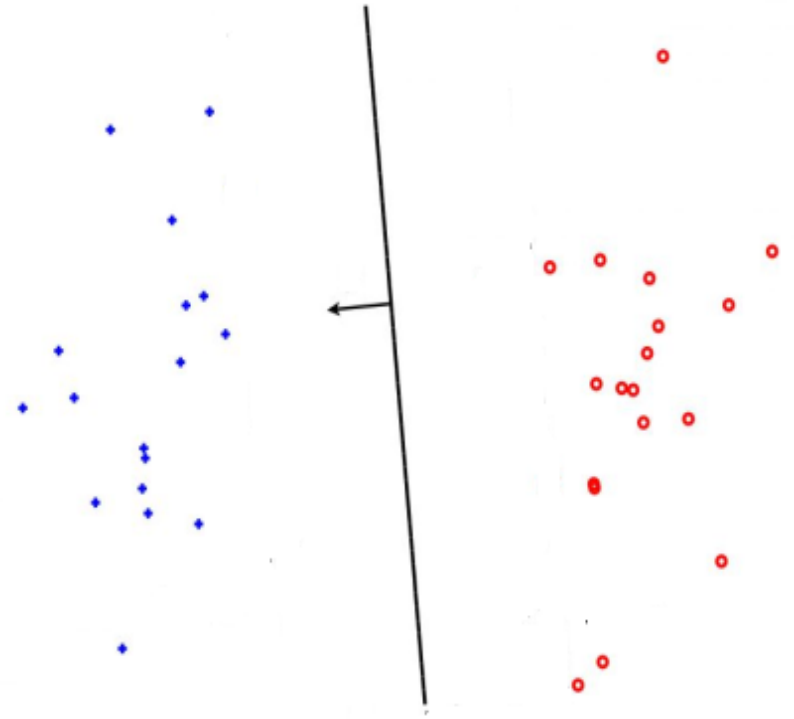
Xor

x_1	x_2	Out
0	0	0
0	1	1
1	0	1
1	1	0



Perceptrón





$$\mathbf{x}^{[i]} = [1, x_1^{[i]}, \ldots, , x_m^{[i]}] \qquad w_0 = b$$

$$z^{(i)} = \mathbf{w} \cdot \mathbf{x}^{[i]}$$

Perceptrón: Algoritmo

$$\mathcal{D} = \{(\mathbf{x}^{[i]}, y^{[i]}) | i = 1, \ldots, n\} \qquad \mathbf{x}^{[i]} \in \Re^{m+1} \quad y^{[i]} \in \{0, 1\}$$

Result: $\mathbf{w}$
$\mathbf{w} = \mathbf{0}_{m+1}$;
for $t = 1, \ldots, T$ **do**
  for $i = 1, \ldots, n$ **do**
    instructions;
    if $y^{[i]}(\mathbf{w} \cdot \mathbf{x}^{[i]}) \leq 0$ **then**
      $\mathbf{w} = \mathbf{w} + y^{[i]}\mathbf{x}^{[i]}$
    else
      do nothing
    end
  end
end

Vectorización

In [2]:

```
z = 0.
for i in range(len(x)):
    z += x[i] * w[i]

print(z)
```

```
2.2
```

In [3]:

```
z = sum(x_i*w_i for x_i, w_i in zip(x, w))
print(z)
```

```
2.2
```

In [4]:

```
import numpy as np

x_vec, w_vec = np.array(x), np.array(w)

z = (x_vec.transpose()).dot(w_vec)
print(z)

z = x_vec.dot(w_vec)
print(z)
```

```
2.2
2.2
```

Vectorización

```
In [6]: %timeit -r 100 -n 10 forloop(x, w)
```

```
38.9 ms ± 1.32 ms per loop (mean ± std. dev. of 100 r
uns, 10 loops each)
```

```
In [7]: %timeit -r 100 -n 10 listcomprehension(x, w)
```

```
29.7 ms ± 842 µs per loop (mean ± std. dev. of 100 ru
ns, 10 loops each)
```

```
In [8]: %timeit -r 100 -n 10 vectorized(x_vec, w_vec)
```

```
46.8 µs ± 8.07 µs per loop (mean ± std. dev. of 100 r
uns, 10 loops each)
```

Implementación

```python
class Perceptron():
    def __init__(self, num_features):
        self.num_features = num_features
        self.weights = np.zeros((num_features, 1), dtype=np.float)
        self.bias = np.zeros(1, dtype=np.float)

    def forward(self, x):
        linear = np.dot(x, self.weights) + self.bias
        predictions = np.where(linear > 0., 1, 0)
        return predictions

    def backward(self, x, y):
        predictions = self.forward(x)
        errors = y - predictions
        return errors

    def train(self, x, y, epochs):
        for e in range(epochs):

            for i in range(y.shape[0]):

                errors = self.backward(x[i].reshape(1, self.num_features), y[i]).reshape(-1)
                self.weights += (errors * x[i]).reshape(self.num_features, 1)
                self.bias += errors

    def evaluate(self, x, y):
        predictions = self.forward(x).reshape(-1)
        accuracy = np.sum(predictions == y) / y.shape[0]
        return accuracy
```

```python
class Perceptron():
    def __init__(self, num_features):
        self.num_features = num_features


        self.weights = torch.zeros(num_features, 1,
                                   dtype=torch.float32, device=device)
        self.bias = torch.zeros(1, dtype=torch.float32, device=device)

        # placeholder vectors so they don't
        # need to be recreated each time
        self.ones = torch.ones(1)
        self.zeros = torch.zeros(1)

    def forward(self, x):
        linear = torch.add(torch.mm(x, self.weights), self.bias)
        predictions = torch.where(linear > 0., self.ones, self.zeros)
        return predictions

    def backward(self, x, y):
        predictions = self.forward(x)
        errors = y - predictions
        return errors

    def train(self, x, y, epochs):
        for e in range(epochs):

            for i in range(y.shape[0]):

# use view because backward expects a matrix (i.e., 2D tensor)
                errors = self.backward(x[i].reshape(1, self.num_features), y[i]).reshape(-1)
                self.weights += (errors * x[i]).reshape(self.num_features, 1)
                self.bias += errors

    def evaluate(self, x, y):
        predictions = self.forward(x).reshape(-1)
        accuracy = torch.sum(predictions == y).float() / y.shape[0]
        return accuracy
```

Perceptrón: Fun Fact

[...] Where a perceptron had been trained to distinguish between - this was for military purposes - it was looking at a scene of a forest in which there were camouflaged tanks in one picture and no camouflaged tanks in the other. And the perceptron - after a little training - made a 100% correct distinction between these two different sets of photographs. Then they were embarrassed a few hours later to discover that the two rolls of film had been developed differently. And so these pictures were just a little darker than all of these pictures and the perceptron was just measuring the total amount of light in the scene. But it was very clever of the perceptron to find some way of making the distinction.

-- Marvin Minsky, AI researcher & author of the "Perceptrons" book

Source: https://www.webofstories.com/play/marvin.minsky/122

Perceptrón: Fun Fact

https://qph.fs.quoracdn.net/main-qimg-305eb8136c4a20f348bb7ab465bc2e10

http://theconversation.com/want-to-beat-climate-change-protect-our-natural-forests-121491

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

Sea

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

Modo *On-line*

Result: $\mathbf{w}, b$
$\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$
for $t = 1, \ldots, T$ **do**
  for $i = 1, \ldots, n$ **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 $t = 1, \ldots, T$ **do**
  for $i = 1, \ldots, n$ **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 *On-line*

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

Modo *Batch*

Result: $\mathbf{w}, b$
$\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$
for $t = 1, \ldots, T$ **do**
  $\Delta\mathbf{w} := 0, \Delta b := 0;$
  for $i = 1, \ldots, n$ **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

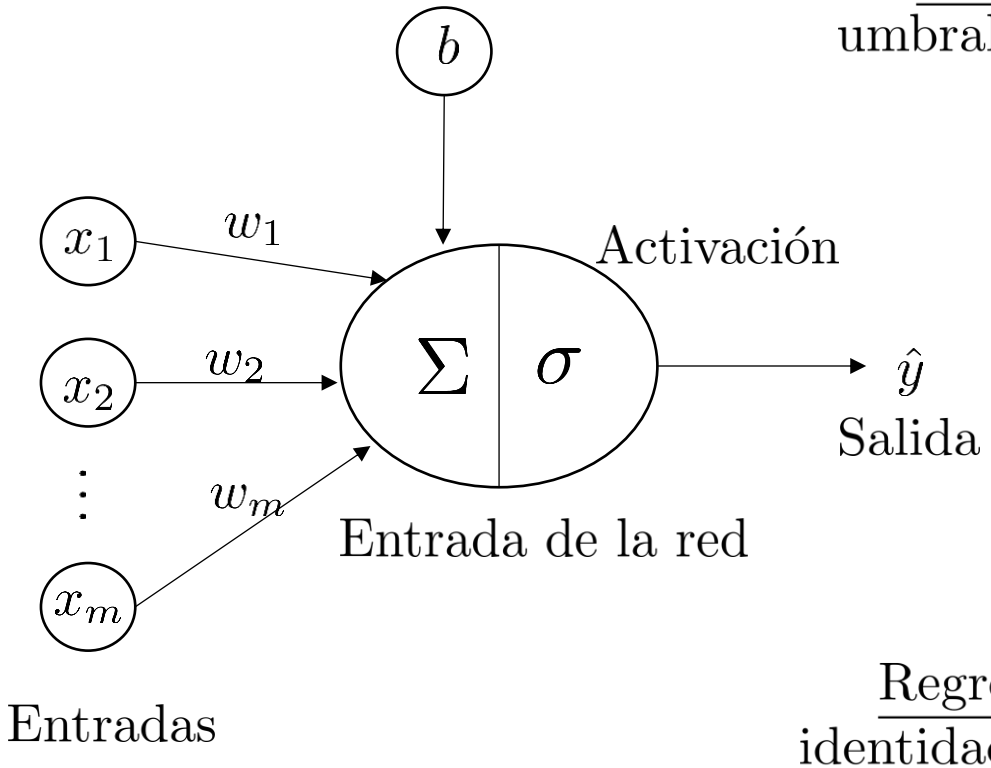
Modo *minibatch*

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

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

Result: $\mathbf{w}, b$
$\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$
for $t = 1, \ldots, T$ **do**
  for $j = 1, \ldots, n/k$ **do**
    $\Delta\mathbf{w} := 0, \Delta b := 0;$
    for $\{(\mathbf{x}^{[i]}, y^{[i]}), \ldots (\mathbf{x}^{[i+k]}, y^{[i+k]})\} \subset D$ **do**
      cálculo de la salida;
      cálculo del error ;
      actualización de $\Delta\mathbf{w}, \Delta b$;
    end
    actualización $\mathbf{w}, b$;
    $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}$;
    $b := b + \Delta w$;
  end
end

Regresión lineal



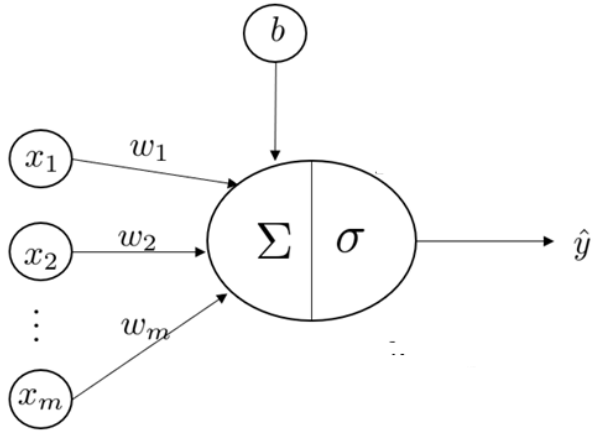
Perceptrón: la función de activación es la función de umbral. La salida es la etiqueta binaria $\hat{y} \in \{0, 1\}$

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

Regresión Lineal

$$\hat{y} = w_1 x_1 + b$$

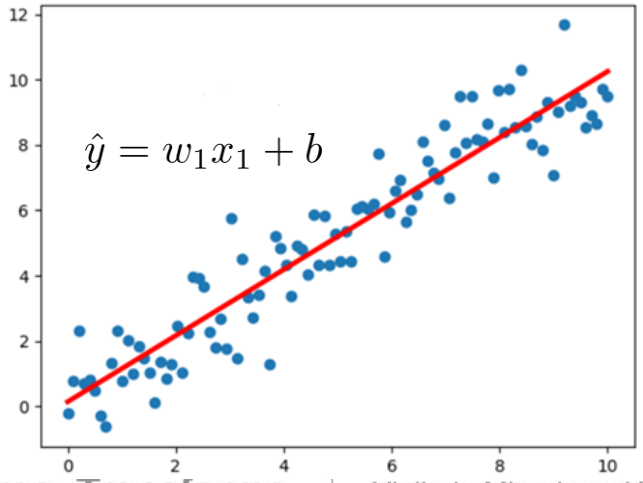
Describe la relación entre una variable dependiente $\hat{y}$ y una única variable independiente x.

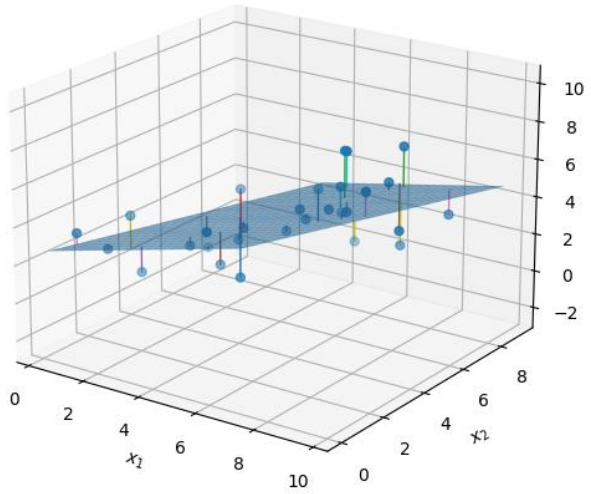


Regresión Lineal Múltiple

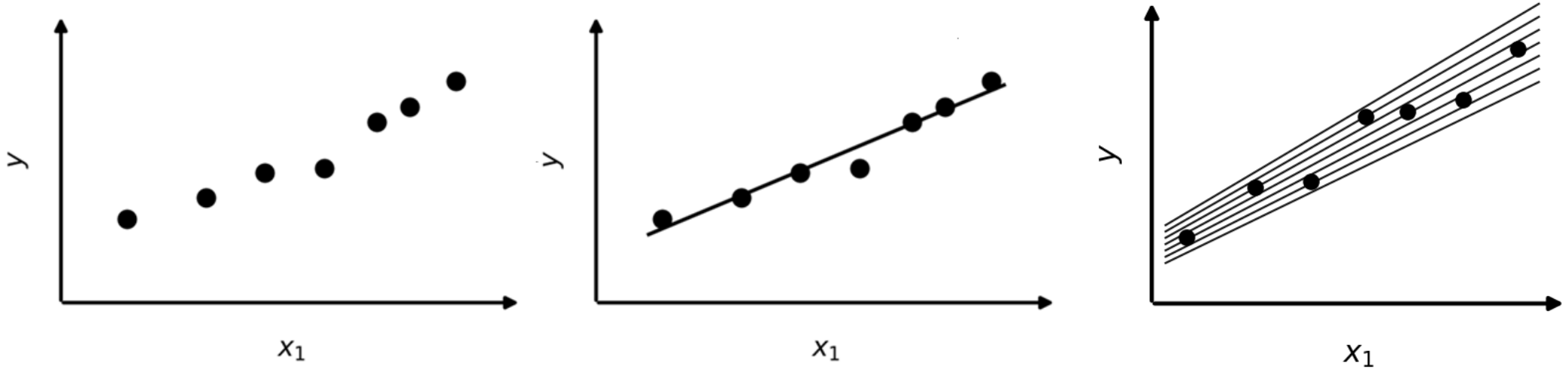
$$\hat{y} = b + w_1 x_1 + w_2 x_2 + \cdots + w_m x_m$$

Relaciona una variable dependiente $\hat{y}$ con dos o más variables independientes.





Regresión lineal

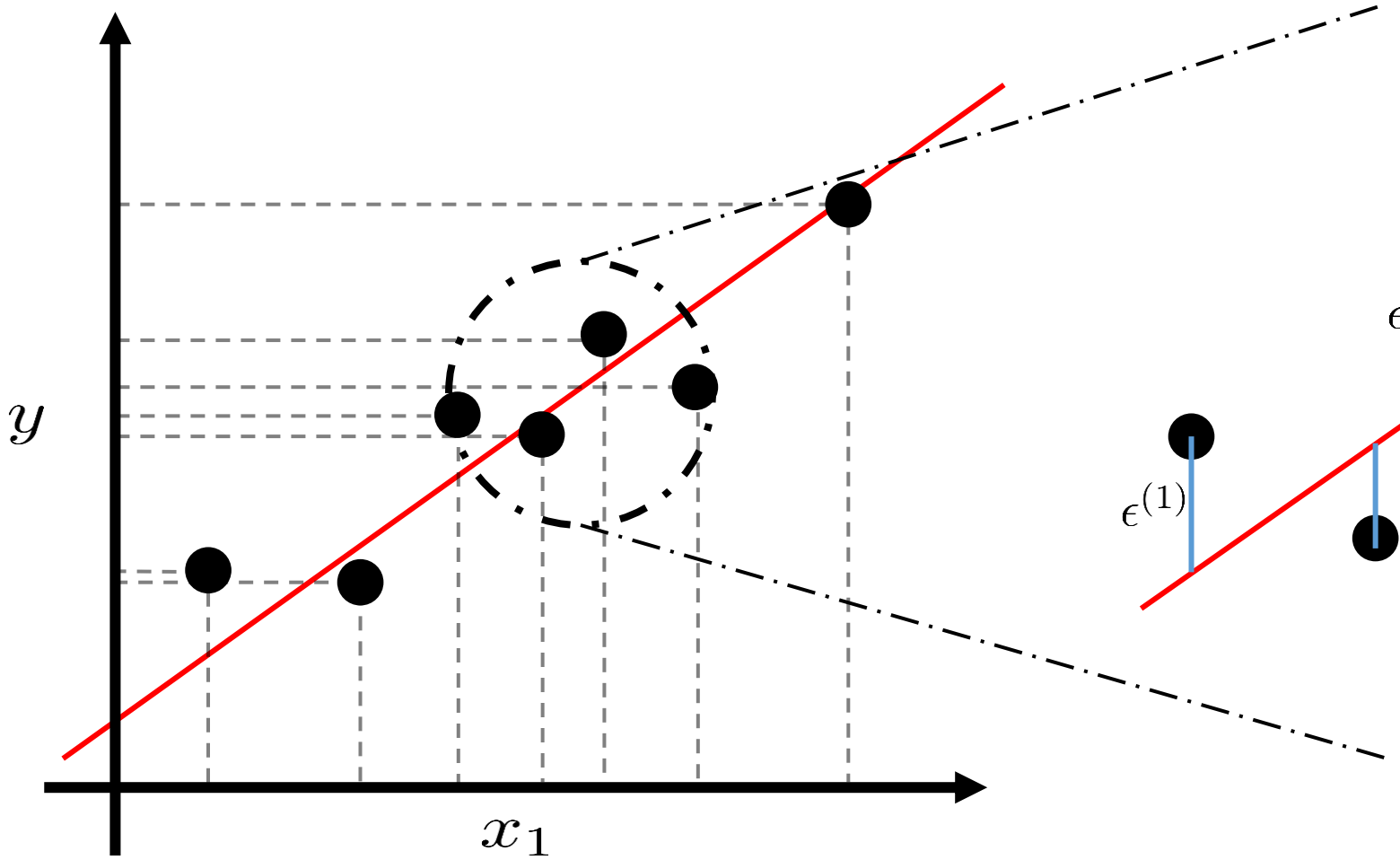


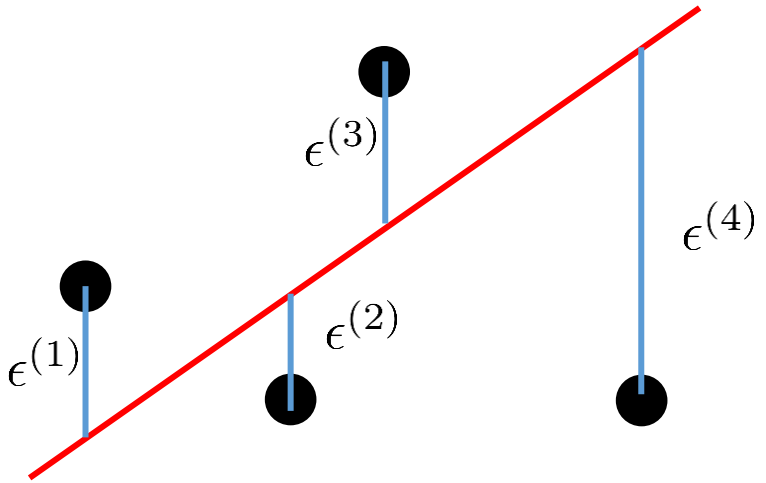
¿Cuál es la línea que mejor se ajusta a los datos?

Minimizar la distancia entre la línea y cada punto

Diferencia entre el valor predicho y el valor real

Regresión lineal



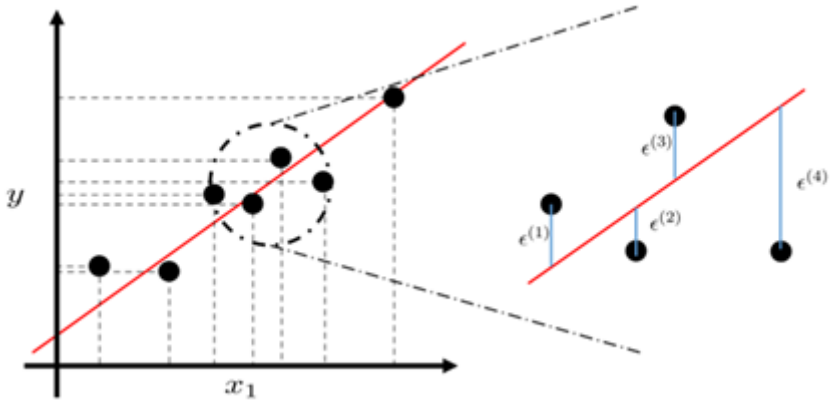


$$\hat{y} = w_1 x_1 + b$$

$$\epsilon^{(i)} = y^{(i)} - \hat{y}^{(i)}$$

$$\epsilon^{(i)} = y^{(i)} - (w_1 x_1^{(i)} + b)$$

Regresión lineal



Minimizar la distancia entre la línea y cada punto

Función de costo:

$$L(w_1, b) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - (w_1 x^{(i)} + b))^2 \qquad L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - (\mathbf{w}^\top \mathbf{x}^{(i)} + b))^2$$

Regresión lineal (mínimos cuadrados)

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

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

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

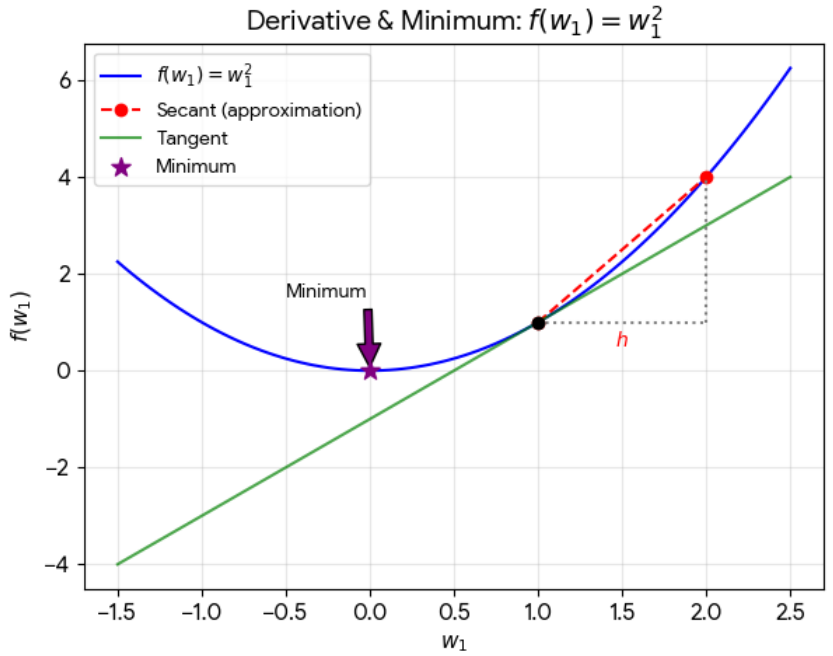
Regresión lineal (mínimos cuadrados)

Una mejor manera

Una mejor forma de hacerlo, es analizando qué efecto tiene un cambio en un parámetro en el desempeño preditivo (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.

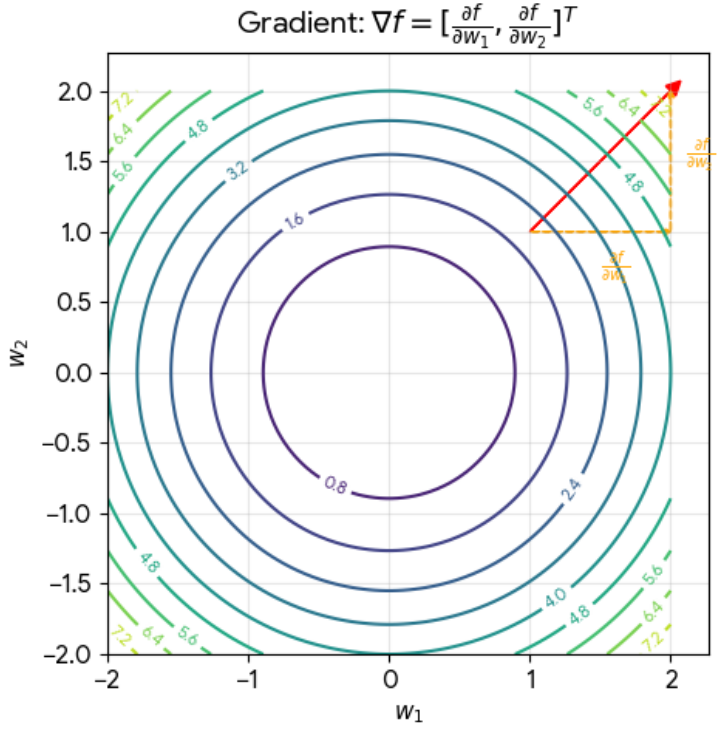
Derivadas de funciones univariadas

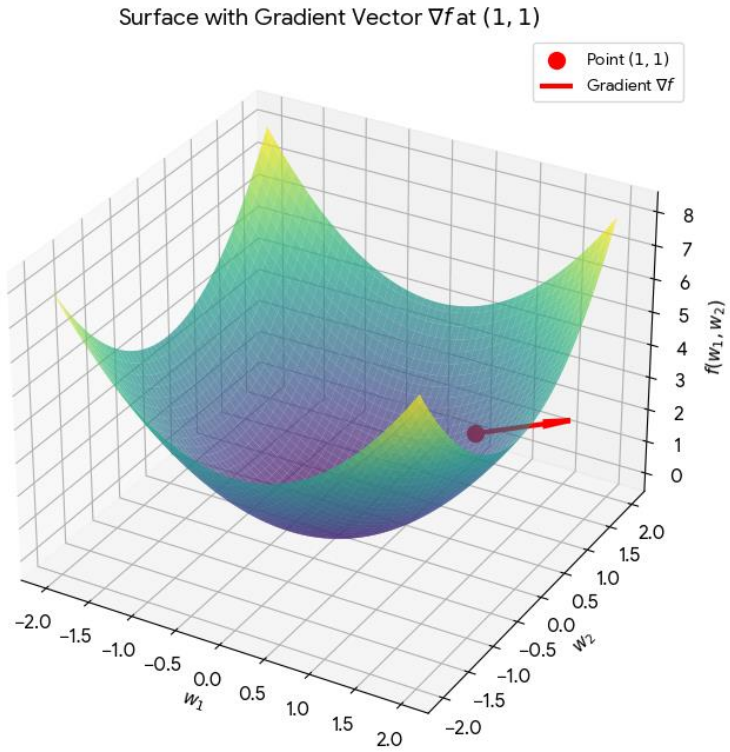
$$\frac{d}{dw_1} f(w_1) = \lim_{h \to 0} \frac{f(w_1+h) - f(w_1)}{h}$$



Gradiente: Derivadas de funciones multivariadas

$$\nabla f(w_1, w_2) = \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \frac{\partial f}{\partial w_2} \end{bmatrix} = \begin{bmatrix} \lim_{h \to 0} \frac{f(w_1+h, w_2) - f(w_1, w_2)}{h} \\ \lim_{h \to 0} \frac{f(w_1, w_2+h) - f(w_1, w_2)}{h} \end{bmatrix}$$





Gradiente: Derivadas de funciones multivariables

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

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

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

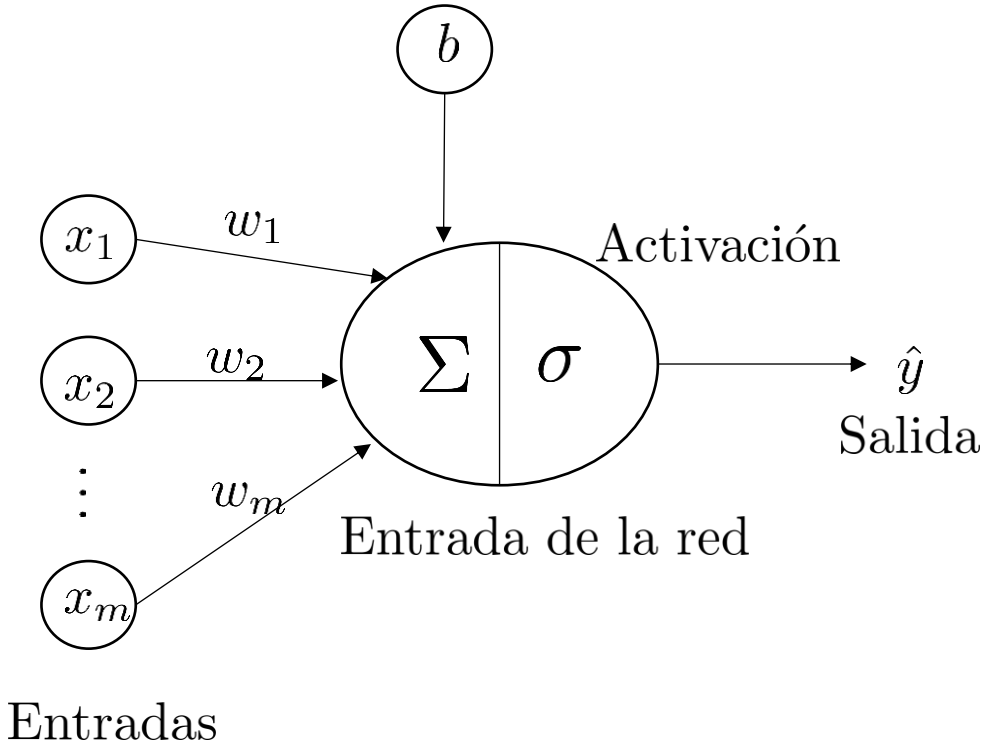
y

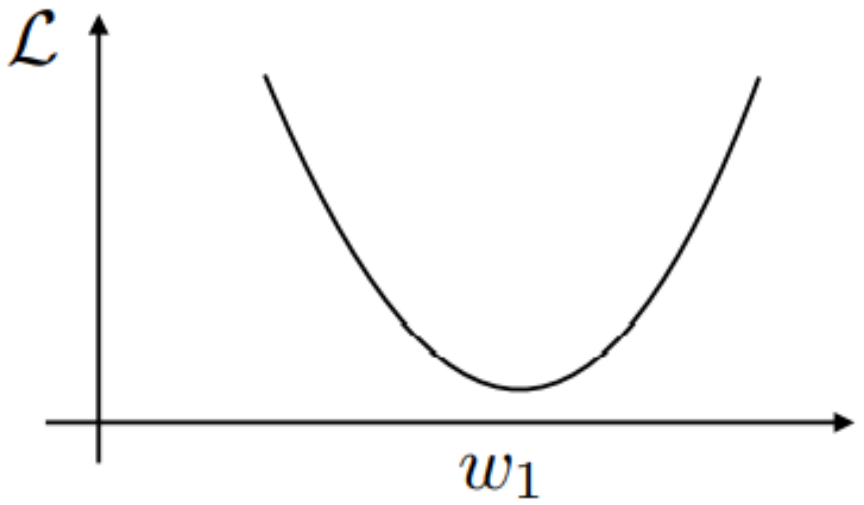
$$\frac{\partial f}{\partial y} = \frac{\partial}{\partial y}x^2y + y = x^2 + 1.$$

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

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

Entrenamiento del regresor lineal con gradiente descendente

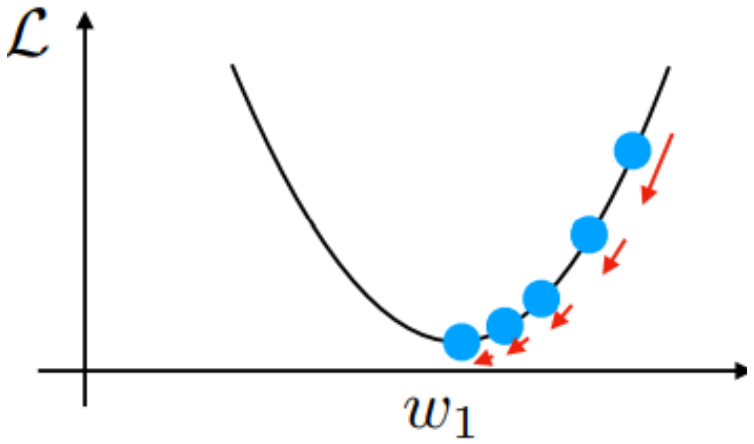


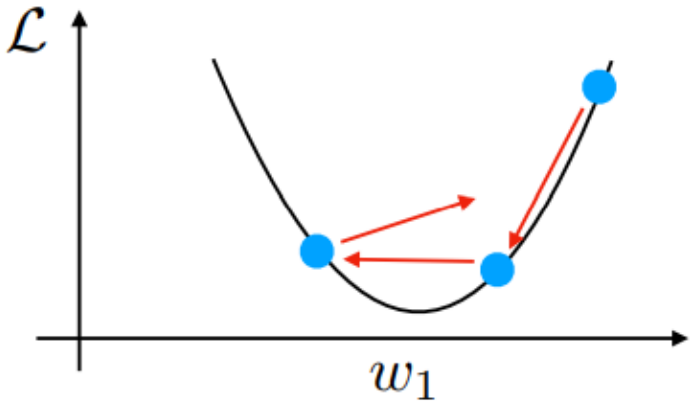


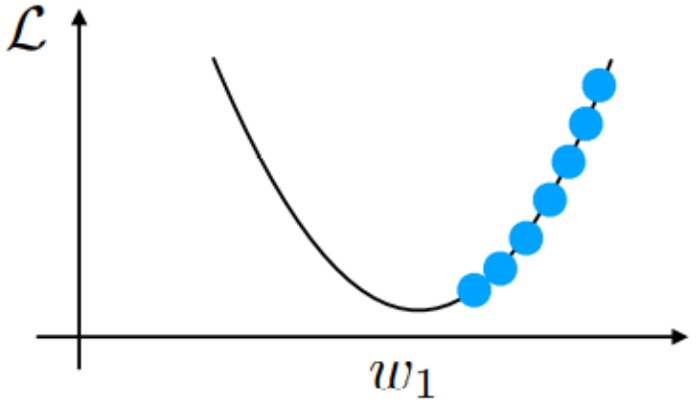
Función de pérdida convexa

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

Entrenamiento del regresor lineal con gradiente descendente







Entrenamiento del regresor lineal con gradiente descendente

Regla de aprendizaje del perceptrón

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

tasa de aprendizaje

Vectorizado

Result: $\mathbf{w}, b$
$\mathbf{w} := \mathbf{0} \in \Re^m, \mathbf{b} := 0;$
for $t = 1, \ldots, T$ **do**
  for $i = 1, \ldots, n$ **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

Ciclo *for*

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

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

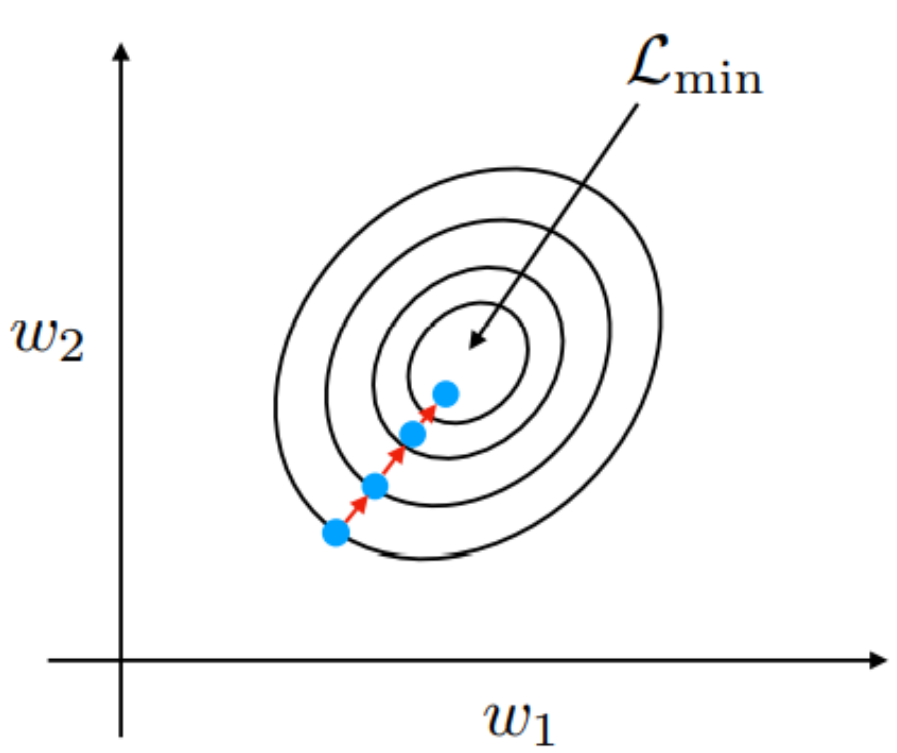
A menudo, la función de pérdida se escala por un factor de $1/2$ por conveniencia:

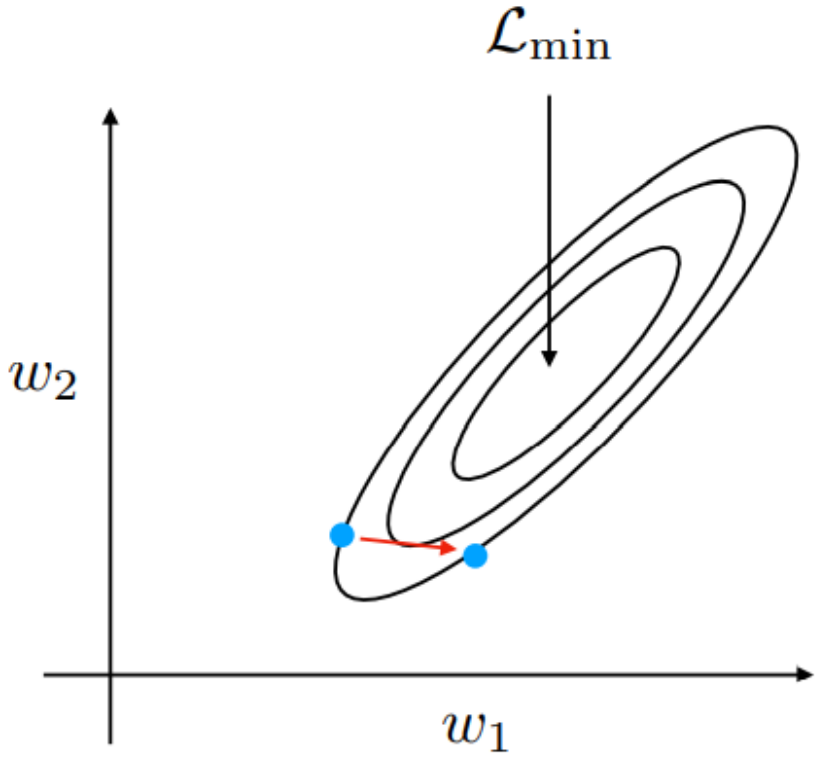
$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2n} \sum_i (\hat{y}^{[i]} - y^{[i]})^2.$$

La derivada estaría dada por

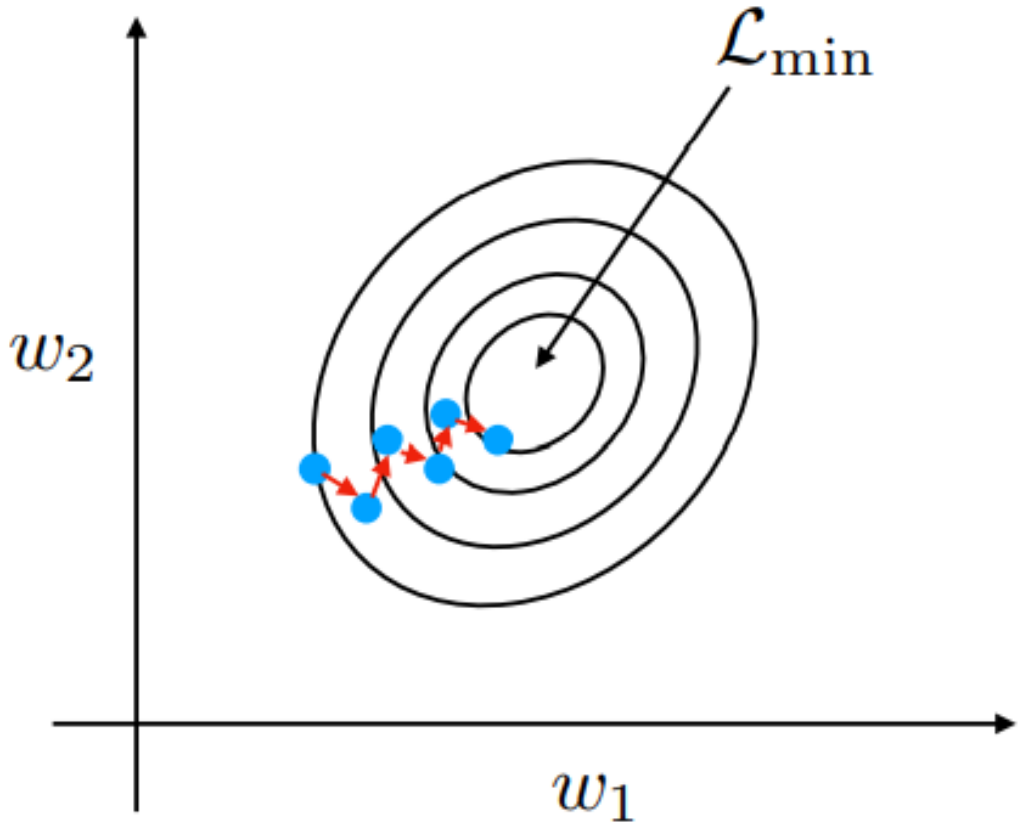
$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2n} \sum_i (\hat{y}^{[i]} - y^{[i]})^2 \\
&= \frac{\partial}{\partial w_j} \sum_i \frac{1}{2n} (\sigma(\mathbf{w}^\top \mathbf{x}^{[i]}) - y^{[i]})^2 \\
&= \sum_i \frac{1}{n} (\sigma(\mathbf{w}^\top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

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.".*

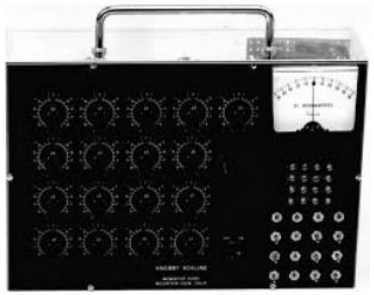
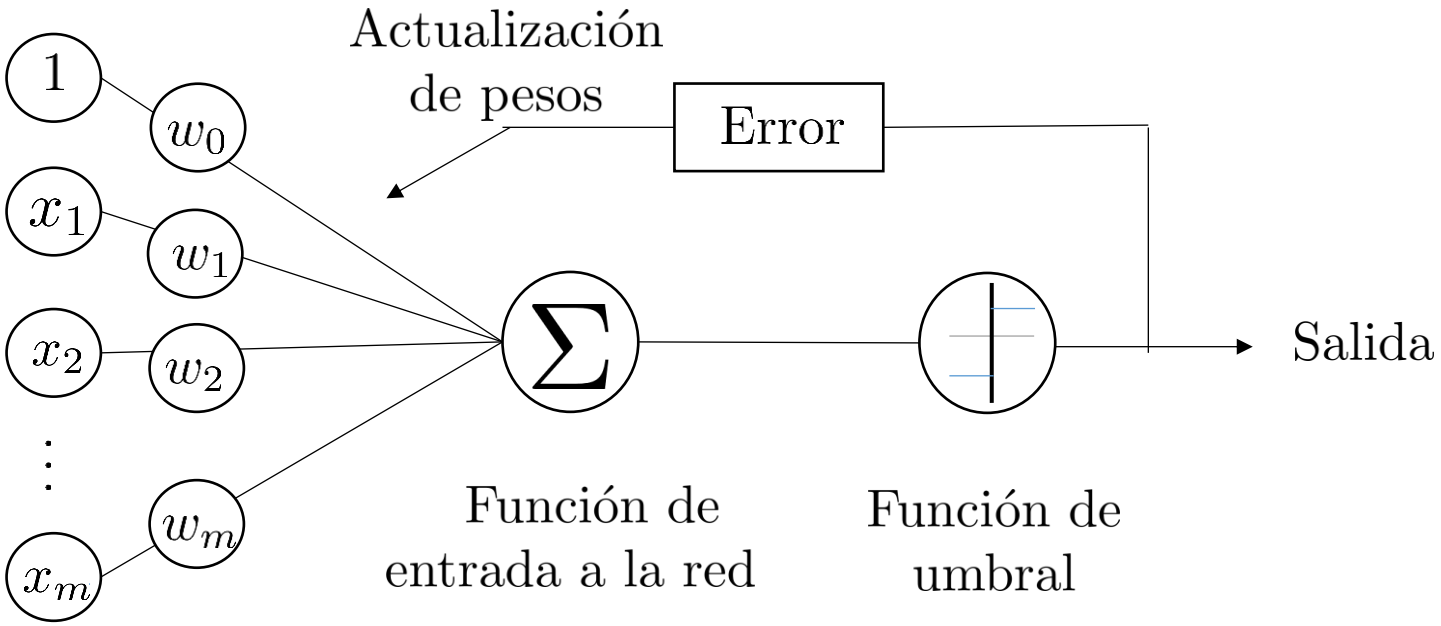


Image source: https://www.researchgate.net/profile/Alexander_Magoun2/publication/265789430/figure/fig2/AS:392335251787780@1470551421849/ADALINE-An-adaptive-linear-neuron-Manually-adapted-synapses-Designed-and-built-by-Ted.png

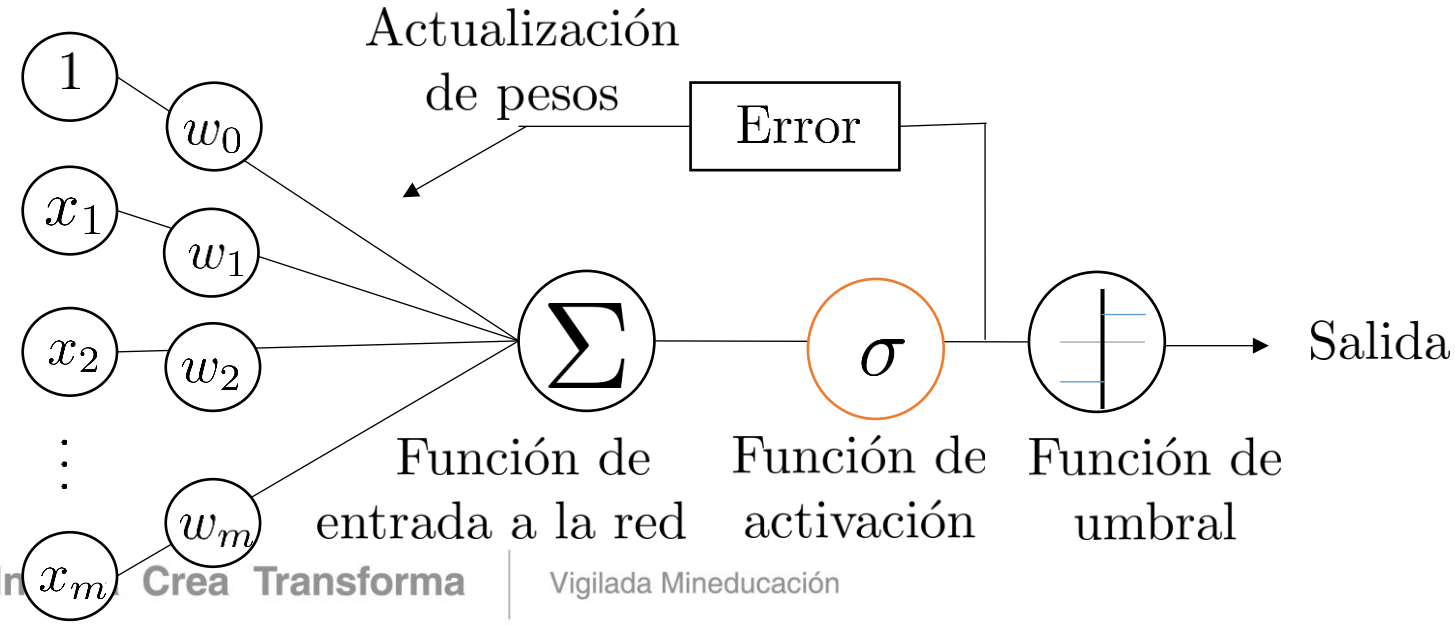
THIS REPORT HAS BEEN DELIMITED AND CLEARED FOR PUBLIC RELEASE UNDER DOD DIRECTIVE 5200.20 AND NO RESTRICTIONS ARE IMPOSED UPON ITS USE AND DISCLOSURE.

DISTRIBUTION STATEMENT A

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.



Perceptrón



ADAptive LInear NEuron

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/`