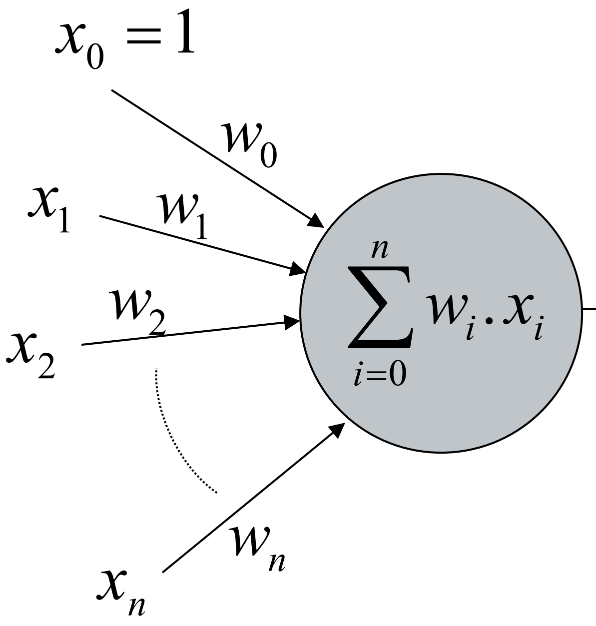
CLASIFICACIÓN BINARIA Y MULTICLASE

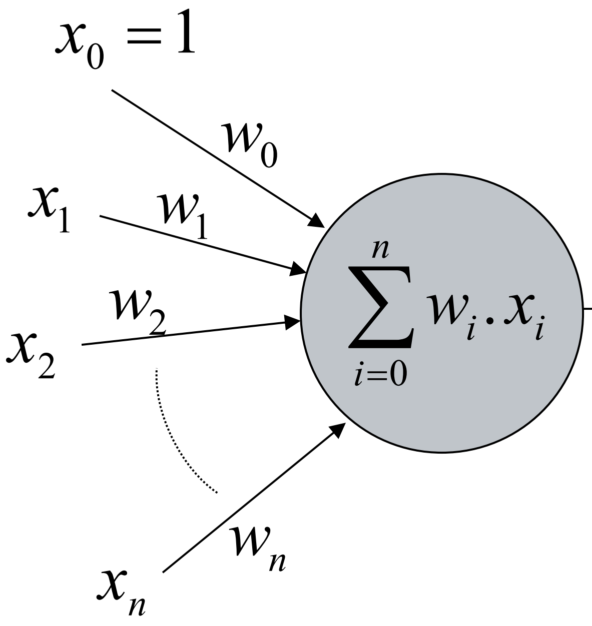
NEURONA GENERAL



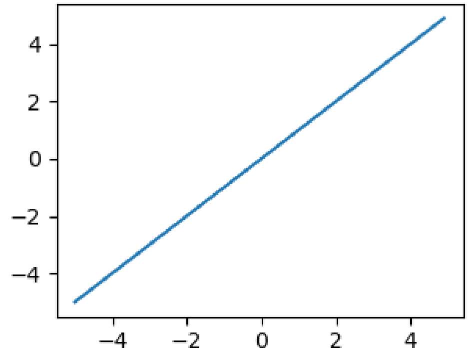
$$y_k = f\left(\sum_{i=0}^{n} w_i . x_i\right)$$

*La función **f debe ser derivable** para poder utilizar luego el descenso de gradiente*

NEURONA GENERAL



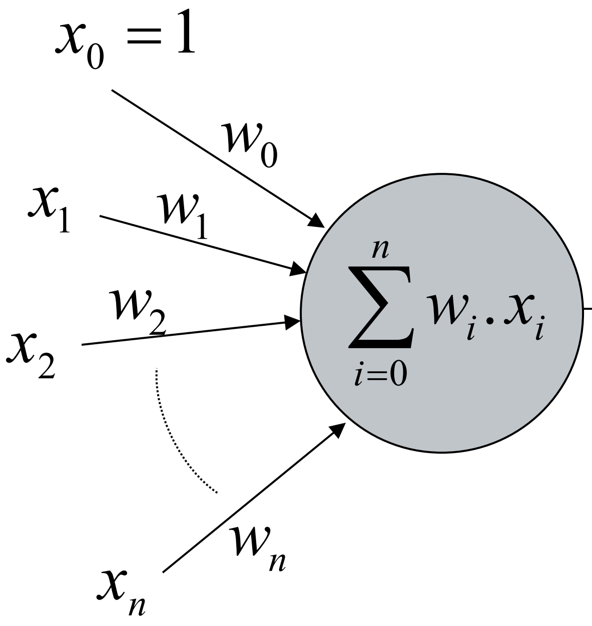
$$y_k = f\left(\sum_{i=0}^{n} w_i \, . x_i\right)$$



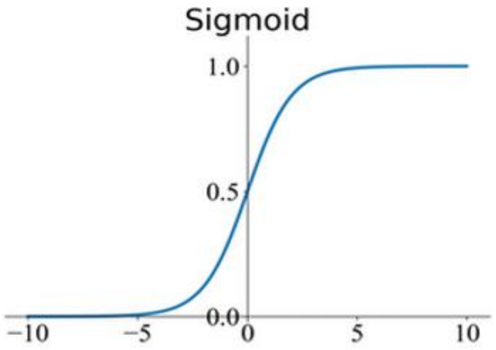
Función de activación

Lineal	$f(x) = x$

NEURONA GENERAL



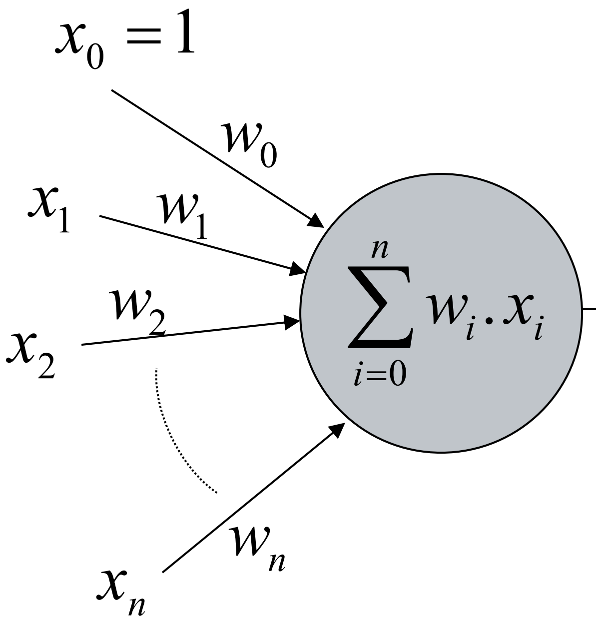
$$y_k = f\left(\sum_{i=0}^{n} w_i . x_i\right)$$



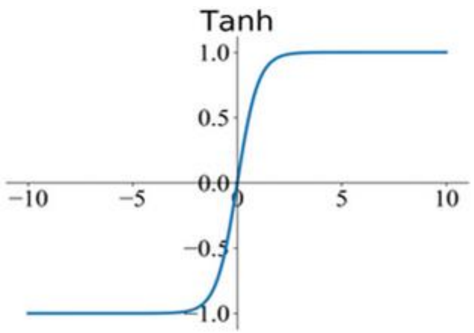
Función de activación

Lineal	$f(x) = x$
Sigmoide entre 0 y 1	$f(x) = \frac{1}{(1 + e^{-x})}$

NEURONA GENERAL



$$y_k = f\left(\sum_{i=0}^{n} w_i . x_i\right)$$



Función de activación

Lineal	$f(x) = x$
Sigmoide entre 0 y 1	$f(x) = \dfrac{1}{(1 + e^{-x})}$
Tangente hiperbólica	$f(x) = \dfrac{2}{(1 + e^{-2x})} - 1$

EJEMPLO

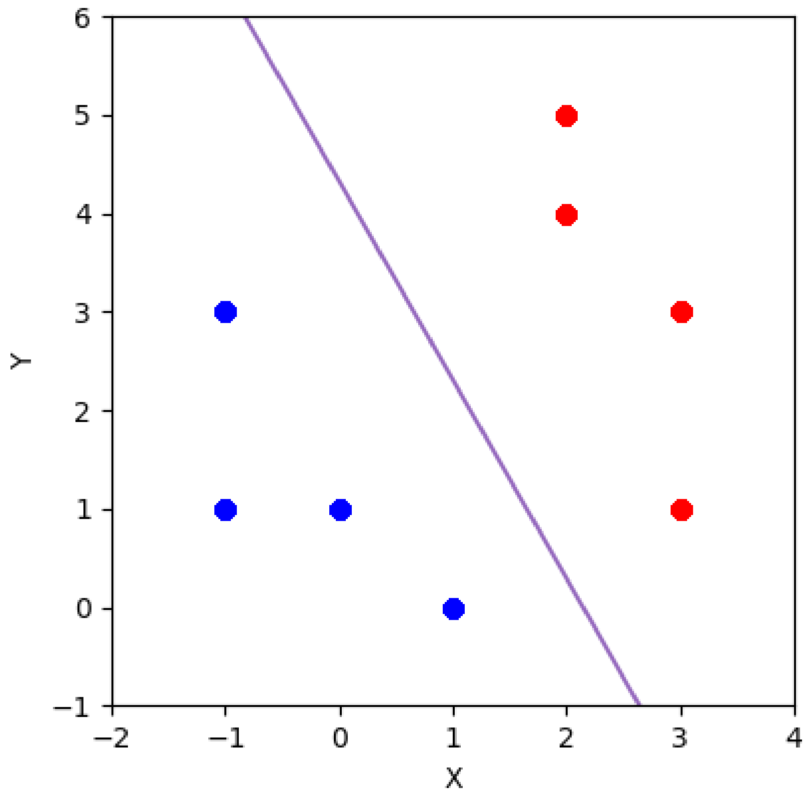
- Dados los siguientes conjuntos de puntos del plano

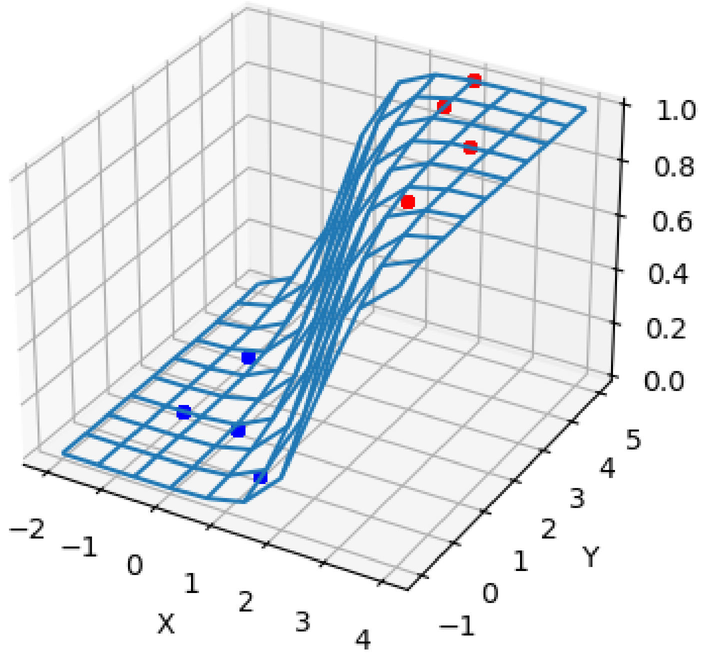
  A = {(2,2), (1,0), (0,1), (-1,1)}

  B = {(3,1), (3,3), (2,4), (2,5)}

  - Utilice una **neurona no lineal** para clasificarlos
  - Representar gráficamente la solución propuesta.

A = {(-1,3), (1,0), (0,1), (-1,1)}
B = {(3,1), (3,3), (2,4), (2,5)}





neuronaNoLineal.py

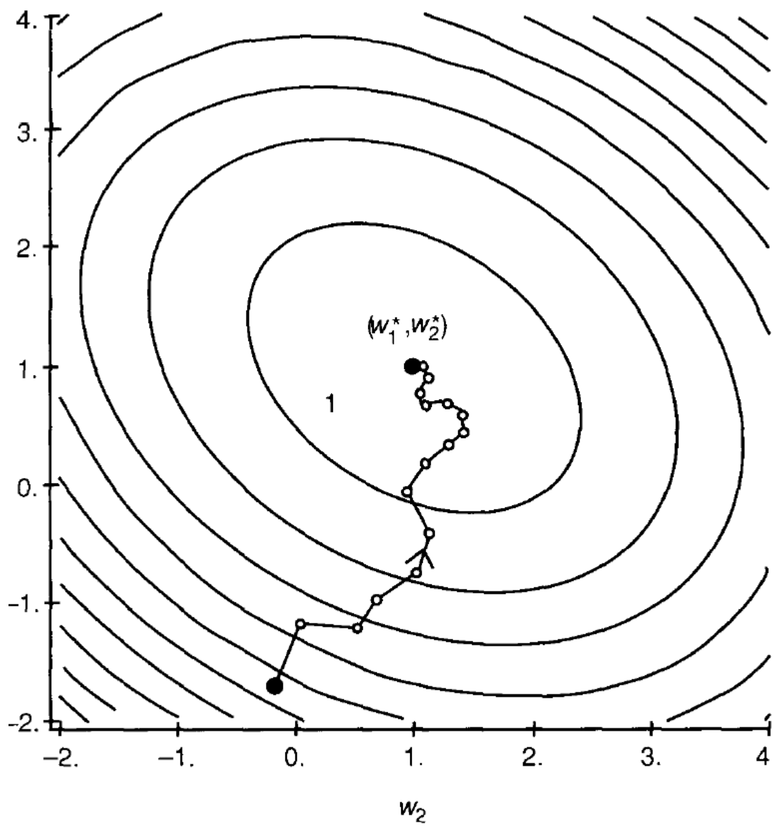
OPTIMIZANDO LOS PESOS: DESCENSO DE GRADIENTE ESTOCÁSTICO

- El descenso de gradiente es un algoritmo iterativo que ajusta los pesos para minimizar la función de costo (o función de pérdida) que mide el error cometido por la red.
- En el descenso de gradiente estocástico, se utiliza un solo ejemplo de entrenamiento en cada iteración.
- Ejemplo: Si la función de costo fuera el ECM, en lugar de usar

$$ECM = \frac{1}{3}\sum_{k=1}^{3}(t_k - y_k)^2$$

usaríamos el error cuadrático cometido en el ejemplo k

$$C = ECM_k = (t_k - y_k)^2$$



ENTRENAMIENTO DE LA NEURONA GENERAL

- Seleccionar el valor de α
- Inicializar los pesos W y b con valores random
- Mientras (la variación de la función de costo C promedio sea mayor a la cota prefijada)
  - Para cada ejemplo
    - Ingresar el ejemplo a la red y calculamos la salida.
    - Calcular el vector gradiente $\dfrac{\partial C}{\partial W}$
    - Actualizar los pesos de la red

$$w_i = w_i - \alpha \frac{\partial C}{\partial w_i}$$

¿CUÁL SERÍA EL GRADIENTE PARA LA FUNCIÓN DE COSTO ECM?

$$\frac{\partial ECM_k}{\partial w} = \left[\frac{\partial (t_k - y_k)^2}{\partial w_0}; \quad \dots \; ; \quad \frac{\partial (t_k - y_k)^2}{\partial w_n}\right]$$

$$y_k = f\left(\sum_{i=0}^{n} w_i . x_i\right)$$

$$\frac{\partial ECM_k}{\partial w_j} = -2(t_k - y_k)\frac{\partial f}{\partial (neta)}\frac{\partial (neta)}{\partial w_j}$$

$$f'$$

$$\frac{\partial (neta)}{\partial w_j} = \frac{\partial (\sum_{i=0}^{n} w_i x_i)}{\partial w_j} = x_j$$

$$\frac{\partial ECM_k}{\partial w_j} = -2(t_k - y_k)\, f'(neta)\, x_j$$

ENTRENAMIENTO DE LA NEURONA GENERAL USANDO ECM

- Seleccionar el valor de α
- Inicializar los pesos W y b con valores random
- Mientras (la variación de la función ECM_k promedio sea mayor a la cota prefijada)
  - Para cada ejemplo
    - Ingresar el k-ésimo ejemplo a la red.
    - Calcular $y_k = f(neta)$
    - Calcular el vector gradiente $\dfrac{\partial ECM_k}{\partial w_j} = -2(t_k - y_k)\, f'(neta)\, x_j$
    - Actualizar los pesos de la red

$$w_i = w_i + \alpha(t_k - y_k)\, f'(neta)\, x_j$$

AND

- Utilice una neurona con función de activación sigmoide para resolver el problema del AND.
- Entrene utilizando el descenso de gradiente estocástico con función de costo ECM.

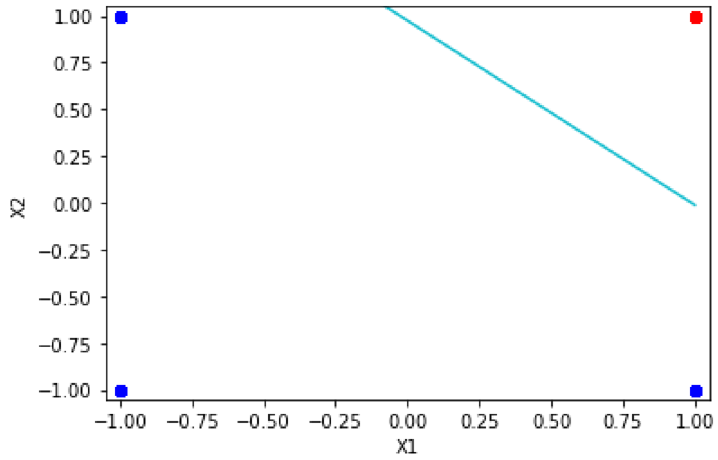
- Derivadas de cada función de activación:

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

$$f'(x) = f(x) * (1 - f(x))$$

$$f(x) = \frac{2}{1+e^{-2x}} - 1$$

$$f'(x) = 1 - f(x) * f(x)$$



Modifique el algoritmo del PERCEPTRON hecho en clase y utilice una neuronal no lineal para resolver el problema del AND

```python
import numpy as np
X = np.array([[0,0], [0,1],[1,0],[1,1]])
T = np.array([0,0,0,1])

W = np.random.uniform(-0.5, 0.5,size=2)
b = np.random.uniform(-0.5, 0.5)

alfa = 0.1
MAX_ITE = 5000
COTA = 10e-06

ite = 0
E_ant = 0
E = 1
while ((ite<MAX_ITE) and (np.abs(E_ant - E) > COTA)):
    E_ant=E
    sumaError = 0

    for p in range(len(X)):
        neta = np.dot(W,X[p,:])+b

        Y = 1/(1+np.exp(-neta))
        deriv = Y * (1-Y)

        Error = T[p]-Y

        W = W + alfa * Error * deriv * X[p,:]
        b = b + alfa * Error * deriv

        sumaError = sumaError + Error**2

    E = sumaError / len(X)
    ite = ite + 1

    print ("ite= %d   w0= %8.5f   w1=%8.5f   E=%.8f" % (ite,W[0],W[1],E))
```

Los pesos iniciales son aleatorios

```python
import numpy as np
X = np.array([[0,0], [0,1],[1,0],[1,1]])
T = np.array([0,0,0,1])

W = np.random.uniform(-0.5, 0.5,size=2)
b = np.random.uniform(-0.5, 0.5)

alfa = 0.1
MAX_ITE = 5000
COTA = 10e-06
```

Parámetros del entrenamiento

```python
ite = 0
E_ant = 0
E = 1
while ((ite<MAX_ITE) and (np.abs(E_ant - E) > COTA)):
    E_ant=E
    sumaError = 0

    for p in range(len(X)):
        neta = np.dot(W,X[p,:])+b

        Y = 1/(1+np.exp(-neta))
        deriv = Y * (1-Y)

        Error = T[p]-Y

        W = W + alfa * Error * deriv * X[p,:]
        b = b + alfa * Error * deriv

        sumaError = sumaError + Error**2

    E = sumaError / len(X)
    ite = ite + 1

    print ("ite= %d   w0= %8.5f   w1=%8.5f   E=%.8f" % (ite,W[0],W[1],E))
```

```python
import numpy as np
X = np.array([[0,0], [0,1],[1,0],[1,1]])
T = np.array([0,0,0,1])

W = np.random.uniform(-0.5, 0.5,size=2)
b = np.random.uniform(-0.5, 0.5)

alfa = 0.1
MAX_ITE = 5000
COTA = 10e-06

ite = 0
E_ant = 0
E = 1
while ((ite<MAX_ITE) and (np.abs(E_ant - E) > COTA)):
    E_ant=E
    sumaError = 0

    for p in range(len(X)):
        neta = np.dot(W,X[p,:])+b

        Y = 1/(1+np.exp(-neta))
        deriv = Y * (1-Y)

        Error = T[p]-Y

        W = W + alfa * Error * deriv * X[p,:]
        b = b + alfa * Error * deriv

        sumaError = sumaError + Error**2

    E = sumaError / len(X)
    ite = ite + 1

    print ("ite= %d   w0= %8.5f   w1=%8.5f   E=%.8f" % (ite,W[0],W[1],E))
```

Termina o bien porque realizó la máxima cantidad de intentos o porque el valor absoluto de la diferencia entre dos valores consecutivos de la función de costo es inferior a cierta cota

```python
import numpy as np
X = np.array([[0,0], [0,1],[1,0],[1,1]])
T = np.array([0,0,0,1])
W = np.random.uniform(-0.5, 0.5,size=2)
b = np.random.uniform(-0.5, 0.5)

alfa = 0.1
MAX_ITE = 5000
COTA = 10e-06
ite = 0
E_ant = 0
E = 1
while ((ite<MAX_ITE) and (np.abs(E_ant - E) > COTA)):
    E_ant=E
    sumaError = 0
    for p in range(len(X)):
        neta = np.dot(W,X[p,:])+b
        Y = 1/(1+np.exp(-neta))
        deriv = Y * (1-Y)
        Error = T[p]-Y
        W = W + alfa * Error * deriv * X[p,:]
        b = b + alfa * Error * deriv
        sumaError = sumaError + Error**2
    E = sumaError / len(X)
    ite = ite + 1
    print ("ite= %d   w0= %8.5f   w1=%8.5f   E=%.8f" % (ite,W[0],W[1],E))
```

Calculamos la salida de la neurona

Función de transferencia

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

$$f'(x) = f(x) * (1 - f(x))$$

```python
import numpy as np
X = np.array([[0,0], [0,1],[1,0],[1,1]])
T = np.array([0,0,0,1])

W = np.random.uniform(-0.5, 0.5,size=2)
b = np.random.uniform(-0.5, 0.5)

alfa = 0.1
MAX_ITE = 5000
COTA = 10e-06

ite = 0
E_ant = 0
E = 1
while ((ite<MAX_ITE) and (np.abs(E_ant - E) > COTA)):
    E_ant=E
    sumaError = 0

    for p in range(len(X)):
        neta = np.dot(W,X[p,:])+b

        Y = 1/(1+np.exp(-neta))
        deriv = Y * (1-Y)

        Error = T[p]-Y

        W = W + alfa * Error * deriv * X[p,:]
        b = b + alfa * Error * deriv

        sumaError = sumaError + Error**2

    E = sumaError / len(X)
    ite = ite + 1

    print ("ite= %d   w0= %8.5f   w1=%8.5f   E=%.8f" % (ite,W[0],W[1],E))
```

Partes del vector gradiente

Función de transferencia

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

$$f'(x) = f(x) * (1 - f(x))$$

Función de Costo

$$\frac{\partial ECM_k}{\partial w_j} = -2(t_k - y_k)\, f'(neta)\, x_j$$

```python
import numpy as np
X = np.array([[0,0], [0,1],[1,0],[1,1]])
T = np.array([0,0,0,1])

W = np.random.uniform(-0.5, 0.5,size=2)
b = np.random.uniform(-0.5, 0.5)

alfa = 0.1
MAX_ITE = 5000
COTA = 10e-06

ite = 0
E_ant = 0
E = 1
while ((ite<MAX_ITE) and (np.abs(E_ant - E) > COTA)):
    E_ant=E
    sumaError = 0

    for p in range(len(X)):
        neta = np.dot(W,X[p,:])+b

        Y = 1/(1+np.exp(-neta))
        deriv = Y * (1-Y)

        Error = T[p]-Y

        W = W + alfa * Error * deriv * X[p,:]
        b = b + alfa * Error * deriv

        sumaError = sumaError + Error**2

    E = sumaError / len(X)
    ite = ite + 1

    print ("ite= %d   w0= %8.5f   w1=%8.5f   E=%.8f" % (ite,W[0],W[1],E))
```

Función de transferencia

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

$$f'(x) = f(x) * (1 - f(x))$$

Función de Costo

$$\frac{\partial ECM_k}{\partial w_j} = -2(t_k - y_k)\, f'(neta)\, x_j$$

Actualizamos los pesos en la dirección del gradiente negativo

```python
import numpy as np
X = np.array([[0,0], [0,1],[1,0],[1,1]])
T = np.array([0,0,0,1])

W = np.random.uniform(-0.5, 0.5,size=2)
b = np.random.uniform(-0.5, 0.5)

alfa = 0.1
MAX_ITE = 5000
COTA = 10e-06

ite = 0
E_ant = 0
E = 1
while ((ite<MAX_ITE) and (np.abs(E_ant - E) > COTA)):
    E_ant=E
    sumaError = 0

    for p in range(len(X)):
        neta = np.dot(W,X[p,:])+b

        Y = 1/(1+np.exp(-neta))
        deriv = Y * (1-Y)

        Error = T[p]-Y

        W = W + alfa * Error * deriv * X[p,:]
        b = b + alfa * Error * deriv

        sumaError = sumaError + Error**2

    E = sumaError / len(X)
    ite = ite + 1

    print ("ite= %d   w0= %8.5f   w1=%8.5f   E=%.8f" % (ite,W[0],W[1],E))
```

Acumulamos el cuadrado de los errores cometidos porque estamos usando ECM

```python
import numpy as np
X = np.array([[0,0], [0,1],[1,0],[1,1]])
T = np.array([0,0,0,1])

W = np.random.uniform(-0.5, 0.5,size=2)
b = np.random.uniform(-0.5, 0.5)

alfa = 0.1
MAX_ITE = 5000
COTA = 10e-06

ite = 0
E_ant = 0
E = 1
while ((ite<MAX_ITE) and (np.abs(E_ant - E) > COTA)):
    E_ant=E
    sumaError = 0

    for p in range(len(X)):
        neta = np.dot(W,X[p,:])+b

        Y = 1/(1+np.exp(-neta))
        deriv = Y * (1-Y)

        Error = T[p]-Y

        W = W + alfa * Error * deriv * X[p,:]
        b = b + alfa * Error * deriv

        sumaError = sumaError + Error**2

    E = sumaError / len(X)
    ite = ite + 1

    print ("ite= %d   w0= %8.5f   w1=%8.5f   E=%.8f" % (ite,W[0],W[1],E))
```

Dividimos por la cantidad de ejemplos para obtener el ECM

NeuronaGral_AND.ipynb

```
import numpy as np
X = np.array([[0,0], [0,1],[1,0],[1,1]])
T = np.array([0,0,0,1])

W = np.random.uniform(-0.5, 0.5,size=2)
b = np.random.uniform(-0.5, 0.5)

alfa = 0.1
MAX_ITE = 5000
COTA = 10e-06

ite = 0
E_ant = 0
E = 1
while ((ite<MAX_ITE) and (np.abs(E_ant - E) > COTA)):
    E_ant=E
    sumaError = 0

    for p in range(len(X)):
        neta = np.dot(W,X[p,:])+b

        Y = 1/(1+np.exp(-neta))
        deriv = Y * (1-Y)

        Error = T[p]-Y

        W = W + alfa * Error * deriv * X[p,:]
        b = b + alfa * Error * deriv

        sumaError = sumaError + Error**2

    E = sumaError / len(X)
    ite = ite + 1

    print ("ite= %d   w0= %8.5f   w1=%8.5f   E=%.8f" % (ite,W[0],W[1],E))
```

```
import numpy as np
X = np.array([[0,0], [0,1],[1,0],[1,1]])
T = np.array([0,0,0,1])

T = 2 * T - 1

W = np.random.uniform(-0.5, 0.5,size=2)
b = np.random.uniform(-0.5, 0.5)

alfa = 0.1
MAX_ITE = 5000
COTA = 10e-06

ite = 0
E_ant = 0
E = 1
while ((ite<MAX_ITE) and (np.abs(E_ant - E) > COTA)):
    E_ant=E
    sumaError = 0

    for p in range(len(X)):
        neta = np.dot(W,X[p,:])+b

        Y = 2/(1+np.exp(-2*neta))-1
        deriv = 1 - Y * Y

        Error = T[p]-Y

        W = W + alfa * Error * deriv * X[p,:]
        b = b + alfa * Error * deriv

        sumaError = sumaError + Error**2

    E = sumaError / len(X)
    ite = ite + 1
```

¿Qué cambiaría si se utiliza ***tanh*** en lugar de ***sigmoid***?

Función de transferencia

$$f(x) = \frac{2}{(1 + e^{-2x})} - 1$$

$$f'(x) = 1 - f(x) * f(x)$$

Se modifica el cálculo de la salida de la neurona y su derivada

ClassNeuronaGral.py

nn = NeuronaGradiente(alpha=0.01, n_iter=50, cotaE=10E-07, FUN='sigmoid', random_state=None, draw=0, title=['X1','X2'])

- <u>Parámetros de entrada</u>
  - **alpha**: valor en el intervalo (0, 1] que representa la velocidad de aprendizaje.
  - **n_iter**: máxima cantidad de iteraciones a realizar.
  - **cotaE**: termina si la diferencia entre dos errores consecutivos es menor que este valor.
  - **FUN:** función de activación – 'sigmoid', 'tanh', 'purelin'.
  - **random_state**: None si los pesos se inicializan en forma aleatoria, un valor entero para fijar la semilla
  - **draw**: valor distinto de 0 si se desea ver el gráfico y 0 si no. Sólo si es 2D.
  - **title**: lista con los nombres de los ejes para el gráfico. Se usa sólo si **draw** no es cero.

ClassNeuronaGral.py

```
nn = NeuronaGradiente(alpha=0.01, n_iter=50, cotaE=10E-07, FUN='sigmoid',
                      random_state=None, draw=0, title=['X1','X2'])
```

nn.fit(X,T)

- Parámetros de entrada
  - **X** : arreglo de NxM donde N es la cantidad de ejemplos y M la cantidad de atributos.
  - **T** : arreglo de N elementos siendo N la cantidad de ejemplos
- Retorna
  - **w_** : arreglo de M elementos siendo M la cantidad de atributos de entrada
  - **b_** : valor numérico continuo correspondiente al bias.
  - **errors_**: errores cometidos en cada iteración.

ClassNeuronaGral.py

Y = nn.predict(X)

- Parámetros de entrada
  - **X** : arreglo de NxM donde N es la cantidad de ejemplos y M la cantidad de atributos.

- Retorna: un arreglo con el resultado de aplicar la neurona general entrenada previamente con fit() a la matriz de ejemplos X.
  - **Y** : arreglo de N elementos siendo N la cantidad de ejemplos

NeuronaGral_AND_RN.ipynb

Función de activación sigmoide con salida (0,1)
Función de Costo: ECM

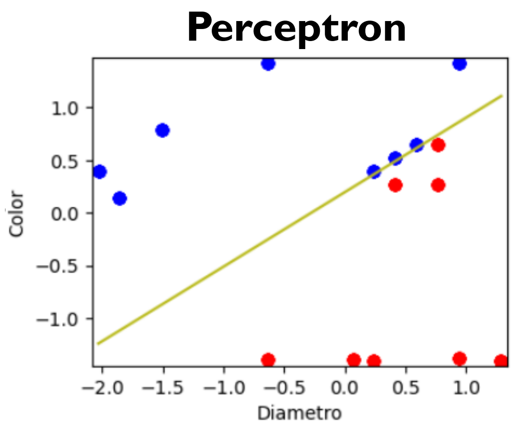
```
import numpy as np
from ClassNeuronaGral import NeuronaGradiente

# Ejemplos de entrada de la función AND
X = np.array([[0,0], [0,1],[1,0],[1,1]])
X = 2*X-1
T = np.array([0,0,0,1])

ppn = NeuronaGradiente(alpha=0.1, n_iter=50, cotaE=10e-07, FUN='sigmoid',
                       random_state=None, draw=1, title=['x1', 'x2'])
ppn.fit(X,T)

#-- % de aciertos ---
Y = (ppn.predict(X)>0.5)*1
print("Y = ", Y)
print("T = ", T)
aciertos = sum(Y == T)
print("aciertos = %d   (%.2f%%)" % (aciertos, 100*aciertos/X.shape[0]))
```

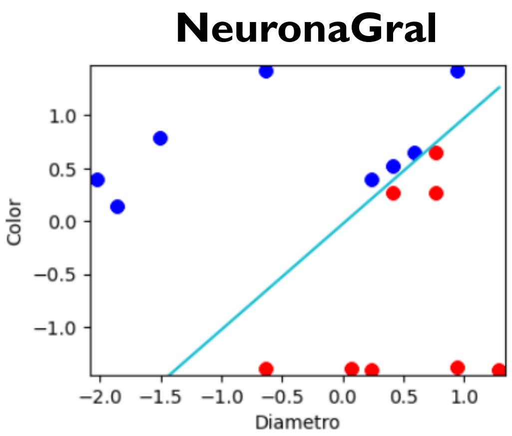
EJEMPLO

- Sobre una cinta transportadora circulan naranjas y melones. Se busca obtener un clasificador de frutas que facilite su almacenamiento. Para cada fruta se conoce su diámetro, en centímetros y su intensidad de color naranja, medida entre 0 y 255.

- Utilice la información del archivo **FrutasTrain.csv** para entrenar una **neurona no lineal** capaz de reconocer los dos tipos de fruta.

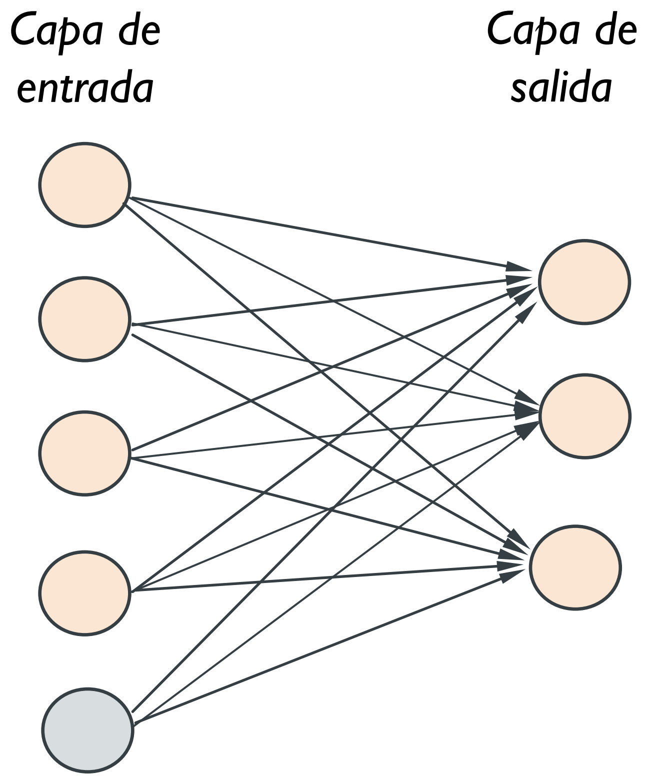
- Compare la manera de obtener la función discriminante de la neurona no lineal con respecto al perceptrón.
  - **NeuronaGral_FRUTAS_RN.ipynb**
  - **Perceptron_FRUTAS_RN.ipynb**





CLASIFICACIÓN CON MÁS DE 2 CLASES

- Pueden utilizarse varias neuronas no lineales para resolver un problema de clasificación con más de 2 clases.
- Cada neurona de la capa de salida buscará responder por un valor de clase distinto.
- El error de la capa será la suma de los errores de las neuronas que la forman.



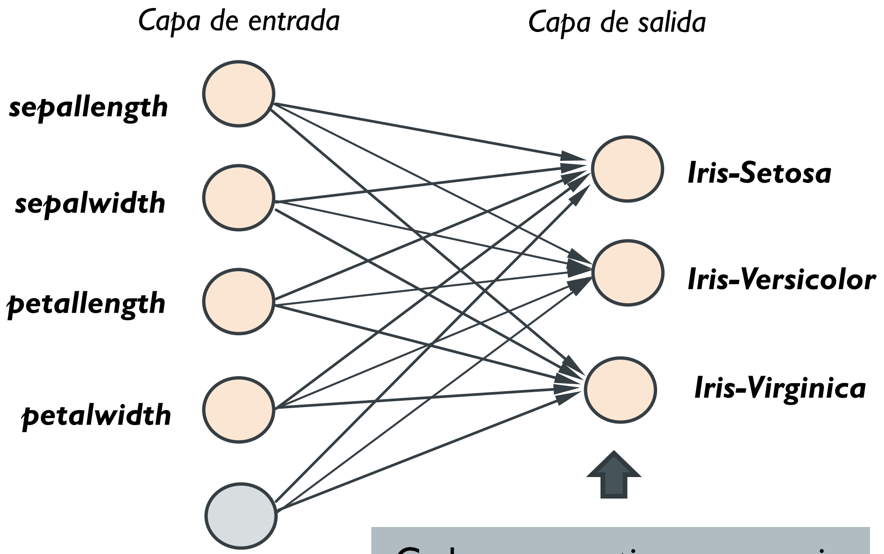
CLASIFICACIÓN CON MÁS DE 2 CLASES

X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```



Cada neurona tiene su propio vector de pesos. Luego **W es una matriz** y **b es un vector**

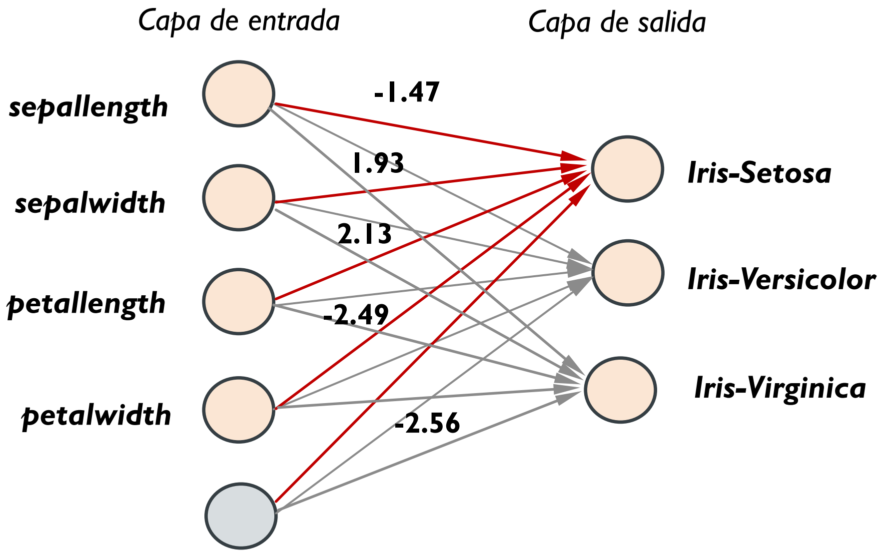
CLASIFICACIÓN CON MÁS DE 2 CLASES

X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```



Inicialmente los pesos de la red **W** y **b** son aleatorios

```
[[-1.47, 1.93, 2.13,-2.49],
 [ 0.79,-1.38, 3.36,-3.57],
 [-1.57,-0.99,-6.17, 3.92]]
```

$\underbrace{\qquad\qquad}_{\mathbf{W}}$

```
[[-2.56],
 [-0.35],
 [-7.03]]
```

$\underbrace{\qquad\qquad}_{\mathbf{b}}$

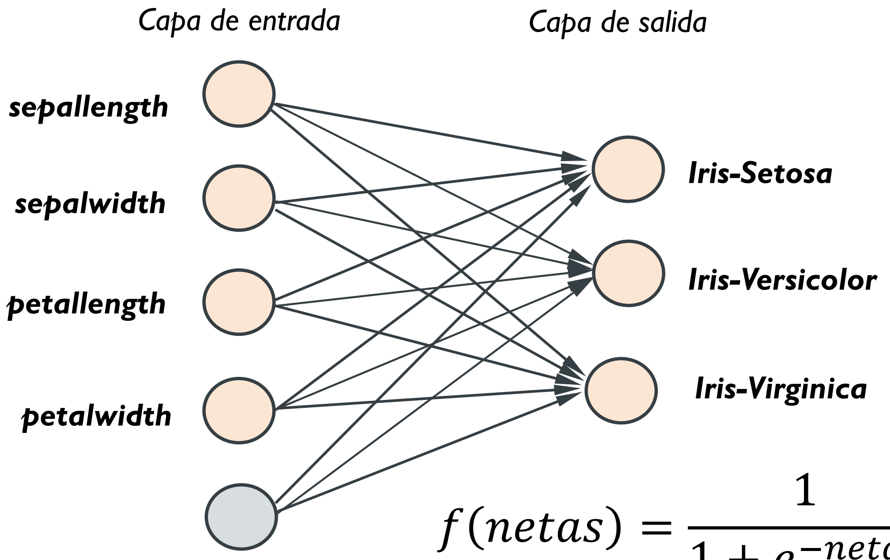
CLASIFICACIÓN CON MÁS DE 2 CLASES

X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```



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

Ingresar el primer ejemplo a la red y calcular su salida

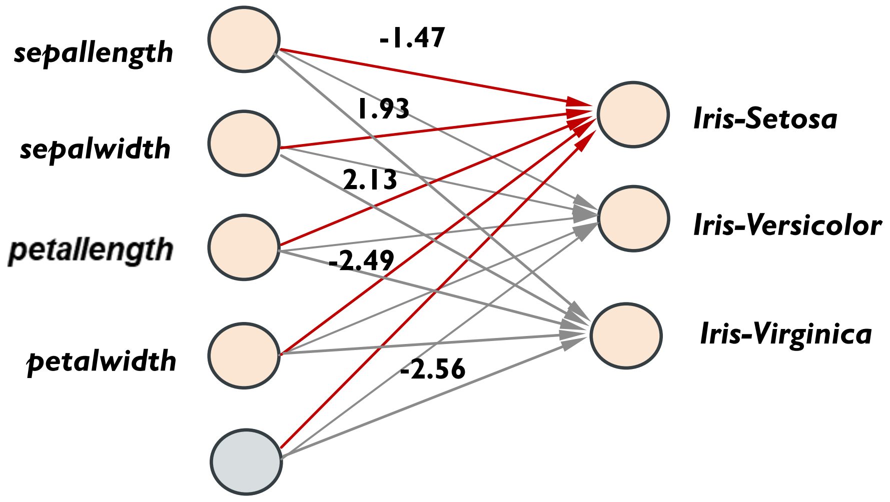
CLASIFICACIÓN CON MÁS DE 2 CLASES

X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```



W

```
[[-1.47, 1.93, 2.13,-2.49],
 [ 0.79,-1.38, 3.36,-3.57],
 [-1.57,-0.99,-6.17, 3.92]]
```

*

x^T

```
[[-1.73],
 [-0.05],
 [-1.38],
 [-1.31]]
```

+

b

```
[[-2.56],
 [-0.35],
 [-7.03]]
```

=

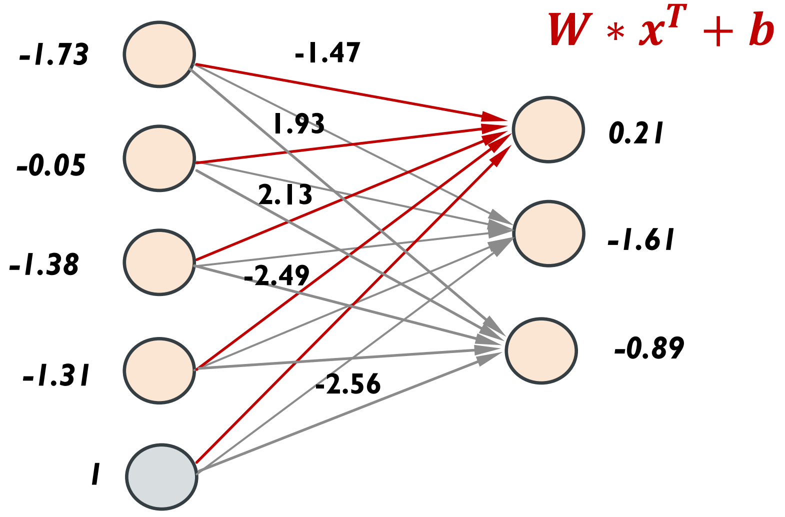
CLASIFICACIÓN CON MÁS DE 2 CLASES

X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```



W x^T **b** $netas = W * x^T + b$

```
[[-1.47, 1.93,-2.13,-2.49],      [[-1.73],      [[-2.56],      [[  0.21],
 [ 0.79,-1.38, 3.36,-3.57],  *    [-0.05],  +    [-0.35],   =   [ -1.61],
 [-1.57,-0.99, 6.17, 3.92]]       [-1.38],       [-7.03]]       [ -0.89]]
                                  [-1.31]]
```

CLASIFICACIÓN CON MÁS DE 2 CLASES

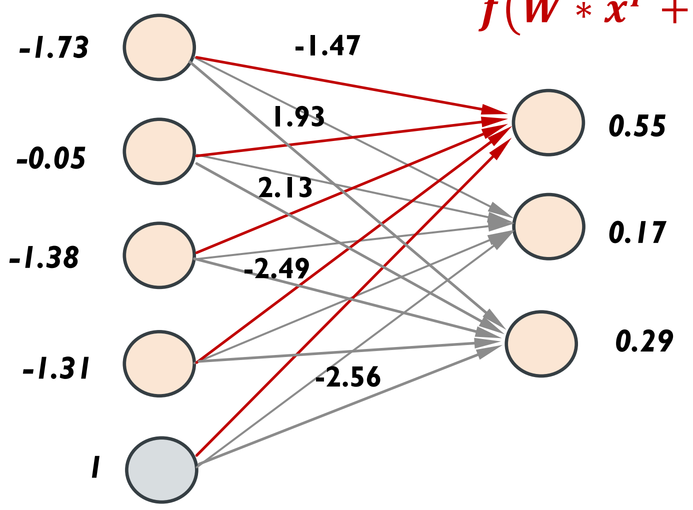
X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```

$f(W * x^T + b)$



W * x^T + **b** = $netas = W * x^T + b$

```
[[-1.47, 1.93, 2.13,-2.49],     [[-1.73],     [[-2.56],     [[  0.21],
 [ 0.79,-1.38, 3.36,-3.57],  *   [-0.05],  +   [-0.35],  =   [ -1.61],
 [-1.57,-0.99,-6.17, 3.92]]      [-1.38],      [-7.03]]      [ -0.89]]
                                 [-1.31]]
```

$y = f(netas) =$

```
[[ 1/(1+exp(-0.21))],        [[0.5521]
 [ 1/(1+exp(1.61))],     =    [0.1669]
 [ 1/(1+exp(0.89))]]          [0.2921]]
```

Calculemos el error cometido en cada neurona

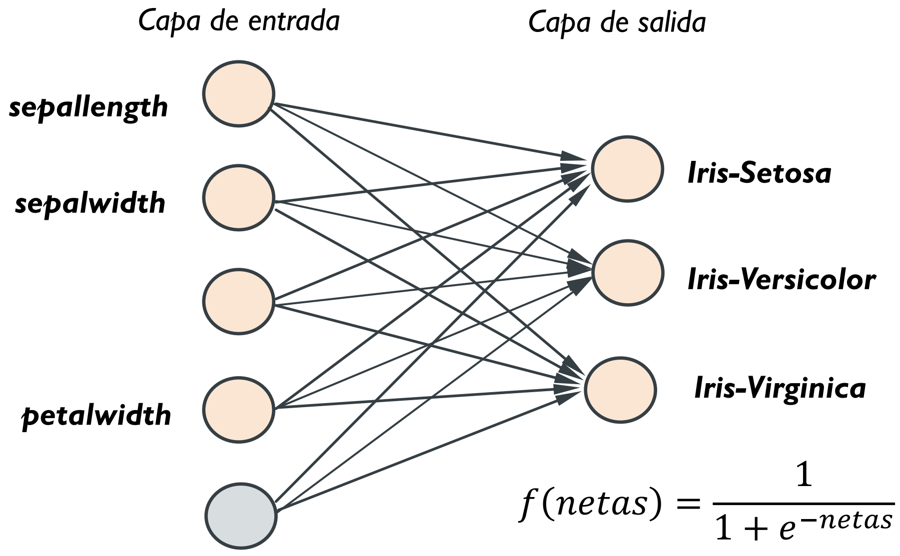
CLASIFICACIÓN CON MÁS DE 2 CLASES

X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```



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

- Error en la respuesta de la red para este ejemplo

```
ErrorSalida =  t.T - y
```

```
                [[1.0]       [[0.5521]       [[ 0.4479]
ErrorSalida =    [0.0]   -    [0.1669]   =    [-0.1669]
                 [0.0]]       [0.2921]]       [-0.2921]]
```

y

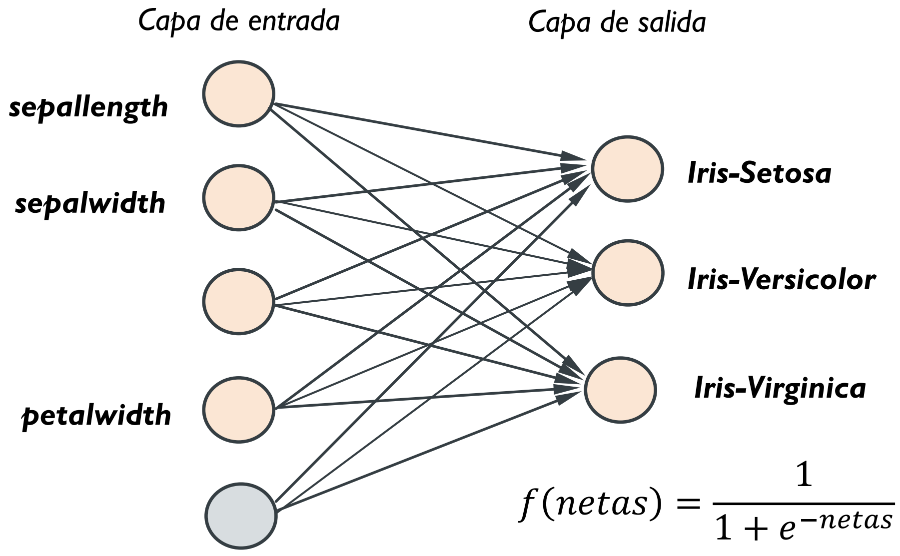
CLASIFICACIÓN CON MÁS DE 2 CLASES

X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```



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

- **Función de Costo: ECM**

  Factores para corregir W y b

```
y *(1 - y)

delta =  ErrorSalida  .*  derivada_Fun

          [[ 0.4479]          [[0.2473]         [[ 0.1108]
delta =    [-0.1669]     .*    [0.1390]     =    [-0.0232]
           [-0.2921]]          [0.2068]]         [-0.0604]]
```

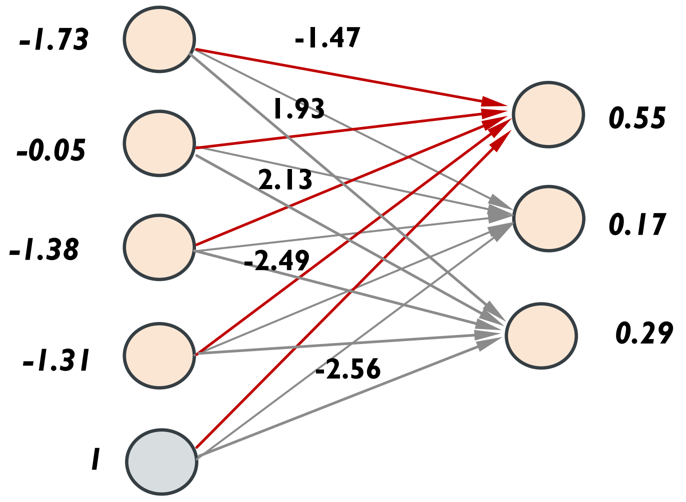
CLASIFICACIÓN CON MÁS DE 2 CLASES

X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```

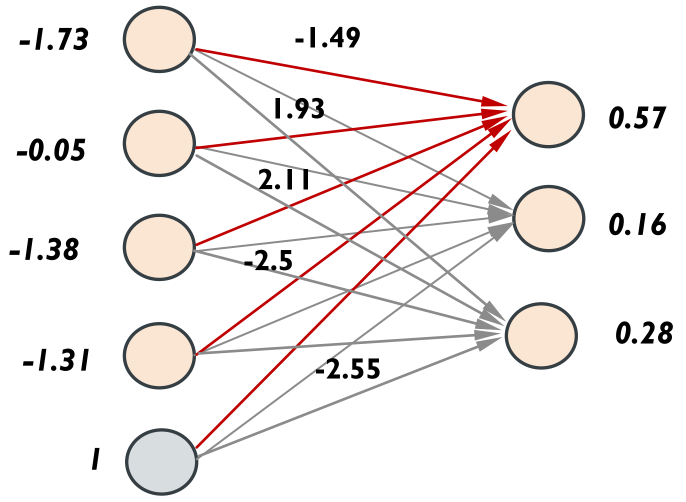


- **Modificación de W y b**

```
W = W + alfa * delta * X

     [[-1.47, 1.93, 2.13,-2.49],         [[ 0.1108]                                 [[-1.49  1.93  2.11 -2.5 ]
W  = [ 0.79,-1.38, 3.36,-3.57], + alfa * [-0.0232]  * [[-1.73,-0.05,-1.38,-1.31]]= [ 0.79 -1.38  3.36 -3.57]
     [-1.57,-0.99,-6.17, 3.92]]           [-0.0604]]                                 [-1.56 -0.99 -6.16  3.93]]

                          [[-2.56],           [[ 0.1108]      [[-2.55]
b = b + alfa * delta =     [-0.35],  + alfa *  [-0.0232]   =   [-0.35]
                           [-7.03]]            [-0.0604]]      [-7.04]]
```

CLASIFICACIÓN CON MÁS DE 2 CLASES

X	T
[[-1.73,-0.05,-1.38,-1.31],	[[1,0,0],
[-0.37,-1.62, 0.22, 0.18],	[0,1,0],
[1.11,-0.05, 0.93, 1.54],	[0,0,1],
[-0.99, 0.39,-1.44,-1.31],	[1,0,0],
[1.73, 1.29, 1.46, 1.81]]	[0,0,1]]



- **Modificación de W y b**

```
W = W + alfa * delta * X
```

```
   [[-1.47, 1.93, 2.13,-2.49],          [[ 0.1108]                                  [[-1.49  1.93  2.11 -2.5 ]
W = [ 0.79,-1.38, 3.36,-3.57], + alfa * [-0.0232]  * [[-1.73,-0.05,-1.38,-1.31]]= [ 0.79 -1.38  3.36 -3.57]
    [-1.57,-0.99,-6.17, 3.92]]           [-0.0604]]                                 [-1.56 -0.99 -6.16  3.93]]
```

```
                            [[-2.56],            [[ 0.1108]      [[-2.55]
b = b + alfa * delta =       [-0.35],  + alfa *   [-0.0232]   =   [-0.35]
                            [-7.03]]             [-0.0604]]      [-7.04]]
```

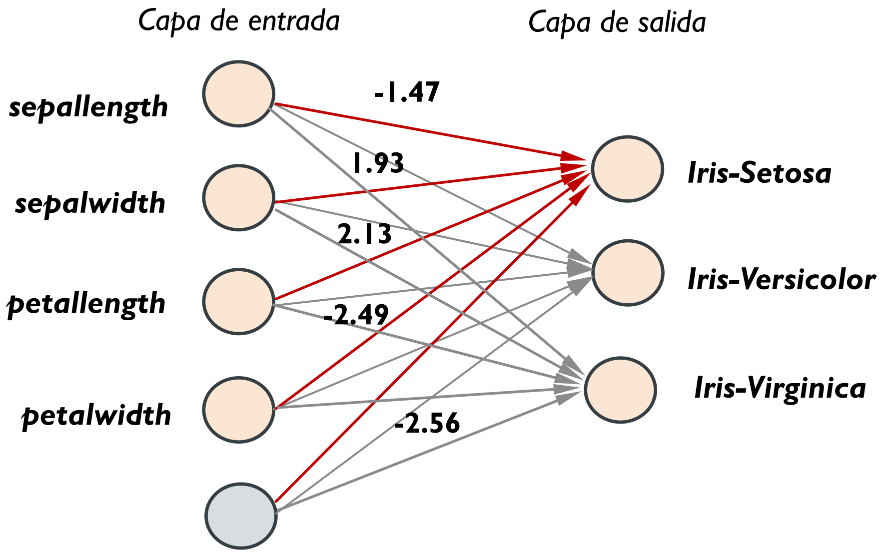
CLASIFICACIÓN CON MÁS DE 2 CLASES

X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```



Para obtener el resultado de la red debe calcularse

$$f\left(W * x^T + b\right)$$

siendo f la función de activación

```
[[-1.47, 1.93,-2.13,-2.49],
 [ 0.79,-1.38, 3.36,-3.57],
 [-1.57,-0.99, 6.17, 3.92]]
```

W

```
[[-2.56],
 [-0.35],
 [-7.03]]
```

b

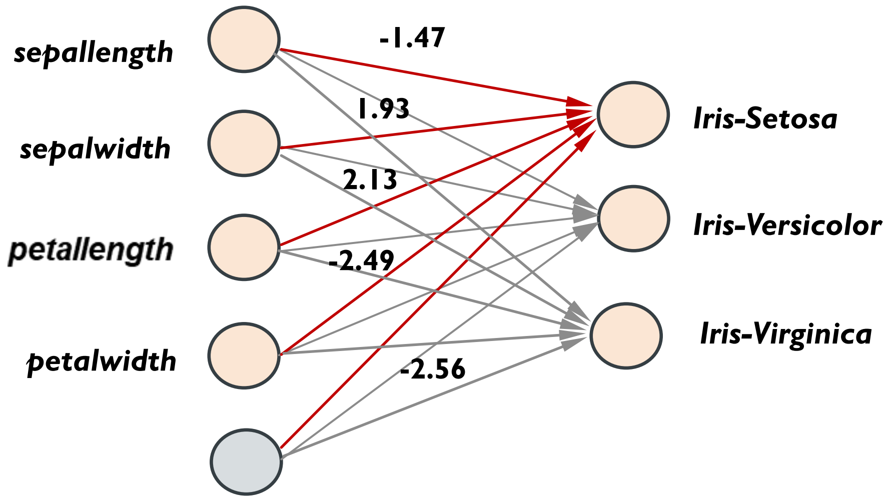
CLASIFICACIÓN DE FLORES DE IRIS

X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```



W

```
[[-1.47, 1.93,-2.13,-2.49],
 [ 0.79,-1.38, 3.36,-3.57],
 [-1.57,-0.99, 6.17, 3.92]]
```

*

x^T

```
[[-1.73],
 [-0.05],
 [-1.38],
 [-1.31]]
```

+

b

```
[[-2.56],
 [-0.35],
 [-7.03]]
```

=

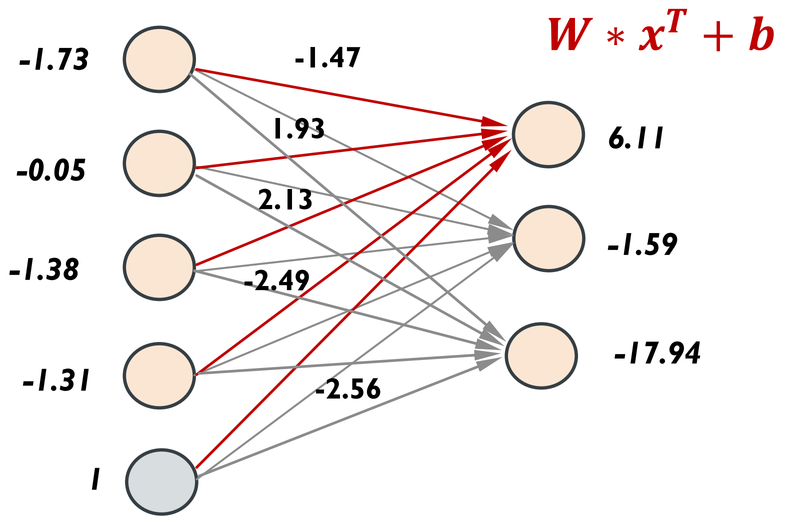
CLASIFICACIÓN DE FLORES DE IRIS

X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```



W

```
[[-1.47, 1.93,-2.13,-2.49],
 [ 0.79,-1.38, 3.36,-3.57],
 [-1.57,-0.99, 6.17, 3.92]]
```

*

x^T

```
[[-1.73],
 [-0.05],
 [-1.38],
 [-1.31]]
```

+

b

```
[[-2.56],
 [-0.35],
 [-7.03]]
```

=

$W * x^T + b$

```
[[  6.11],
 [ -1.59],
 [-17.94]]
```

CLASIFICACIÓN DE FLORES DE IRIS

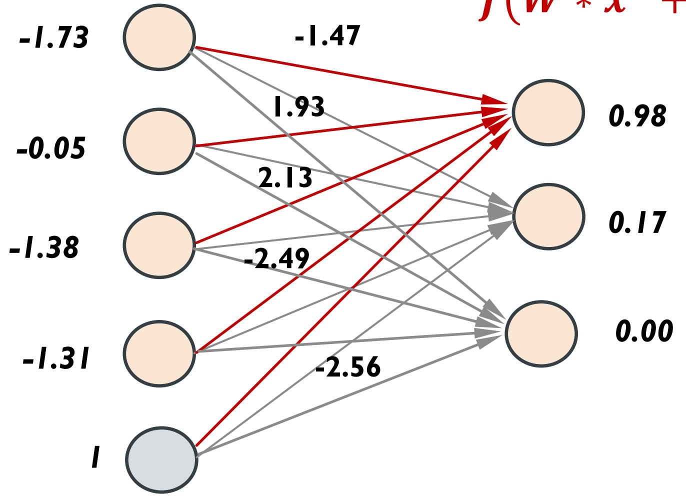
X

```
[[-1.73,-0.05,-1.38,-1.31],
 [-0.37,-1.62, 0.22, 0.18],
 [ 1.11,-0.05, 0.93, 1.54],
 [-0.99, 0.39,-1.44,-1.31],
 [ 1.73, 1.29, 1.46, 1.81]]
```

T

```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```

$f(W * x^T + b)$



W * x^T + **b** = $W * x^T + b$

```
[[-1.47, 1.93,-2.13,-2.49],     [[-1.73],     [[-2.56],     [[  6.11],
 [ 0.79,-1.38, 3.36,-3.57],  *   [-0.05],  +   [-0.35],  =   [ -1.59],
 [-1.57,-0.99, 6.17, 3.92]]      [-1.38],      [-7.03]]      [-17.94]]
                                 [-1.31]]
```

$f(W * x^T + b)$ =

```
[[ 1/(1+exp( -6.11))],       [[ 0.98],
 [ 1/(1+exp(  1.59))],   =    [ 0.17],
 [ 1/(1+exp( 17.94))]]        [ 0.00]]
```

Se interpreta como

```
[[ 1 ],
 [ 0 ],
 [ 0 ]]
```

ENTROPÍA CRUZADA BINARIA

- Es una función de costo que puede usarse con neuronas con función de activación sigmoide entre 0 y 1

$$C = -\frac{1}{n}\sum_{n}[\ t\ln y + (1-t)\ \ln(1-y)\ ]$$

donde

- t es el valor binario esperado
- $y = 1/(1 + e^{-\sum x_i w_i})$ es la salida de la neurona

- Ver que es una función de costo
  - C > 0
  - C tiende a 0 (cero) a medida que la neurona aprende la salida deseada.

DERIVADA DE LA ENTROPÍA CRUZADA BINARIA

$$C = -\frac{1}{n}\sum_{n}[t \ln \mathrm{y} + (1-t) \ln(1-y)]$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_{n}\left(\frac{t}{y} - \frac{1-t}{1-y}\right)\frac{\partial y}{\partial w_j}$$

$f(neta)$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_{n}\left(\frac{t}{f(neta)} - \frac{1-t}{1-f(neta)}\right)\frac{\partial f(neta)}{\partial w_j}$$

DERIVADA DE LA ENTROPÍA CRUZADA BINARIA

$$C = -\frac{1}{n}\sum_{n}[t \ln y + (1-t)\ \ln(1-y)]$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_{n}\left(\frac{t}{f(neta)} - \frac{1-t}{1-f(neta)}\right)\frac{\partial f(neta)}{\partial w_j}$$

$$\frac{t-f(neta)}{f(neta)(1-f(neta))}$$

$$f'(neta)x_j$$

DERIVADA DE LA ENTROPÍA CRUZADA BINARIA

$$C = -\frac{1}{n}\sum_{n}[t \ln y + (1-t)\ \ln(1-y)]$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_{n}\left(\frac{t - f(neta)}{f(neta)(1-f(neta))}\right) f'(neta)\, x_j$$

Si $f(neta) = \frac{1}{1+e^{-neta}}$, $f'(neta) = f(neta)\big(1 - f(neta)\big)$

DERIVADA DE LA ENTROPÍA CRUZADA BINARIA

$$C = -\frac{1}{n}\sum_{n}[t \ln \mathrm{y} + (1-t)\ \ln(1-y)]$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_{n}\left(\frac{t - f(neta)}{f(neta)(1-f(neta))}\right) f'(neta)\, x_j$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_{n}(t - f(neta))\, x_j$$

DERIVADA DE LA ENTROPÍA CRUZADA BINARIA

$$C = -\frac{1}{n}\sum_{n}[t \ln \mathrm{y} + (1-t)\ \ln(1-y)]$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_{n}\left(\frac{t-f(neta)}{f(neta)(1-f(neta))}\right) f'(neta)\, x_j$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_{n}(t-y)\, x_j$$

DERIVADA DE LA ENTROPÍA CRUZADA BINARIA

$$C = -\frac{1}{n}\sum_{n}[t \ln \mathrm{y} + (1-t)\ \ln(1-y)]$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_{n}(t-y)\, x_j$$

- Si utilizamos descenso de gradiente estocástico sólo se mide el error cometido en el ejemplo k

$$\frac{\partial C_k}{\partial w_j} = -(t_k - y_k)\, x_j$$

Gradiente de la función de costo ECM calculado sobre el k-ésimo ejemplo

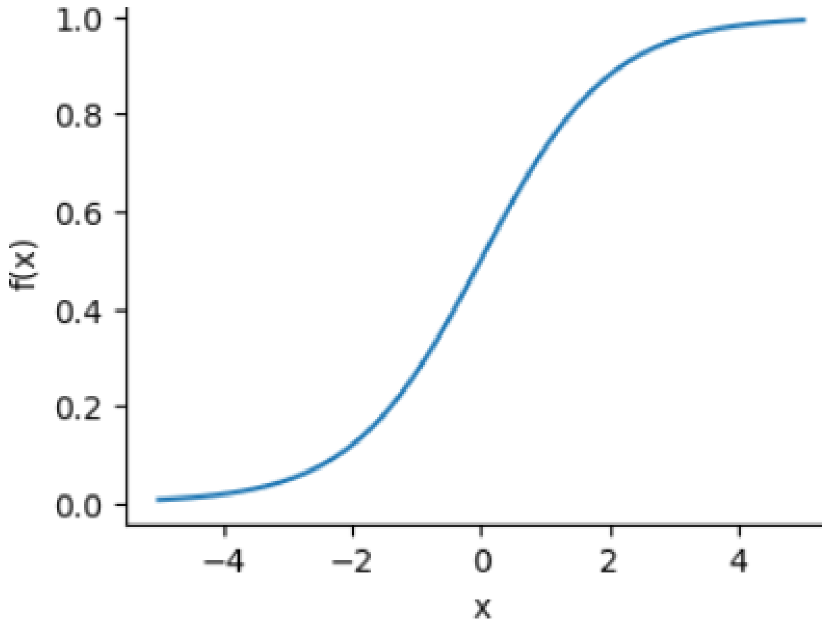
$$\frac{\partial ECM_k}{\partial w_j} = -(t_k - y_k)\boxed{f'(neta)}\,x_j$$

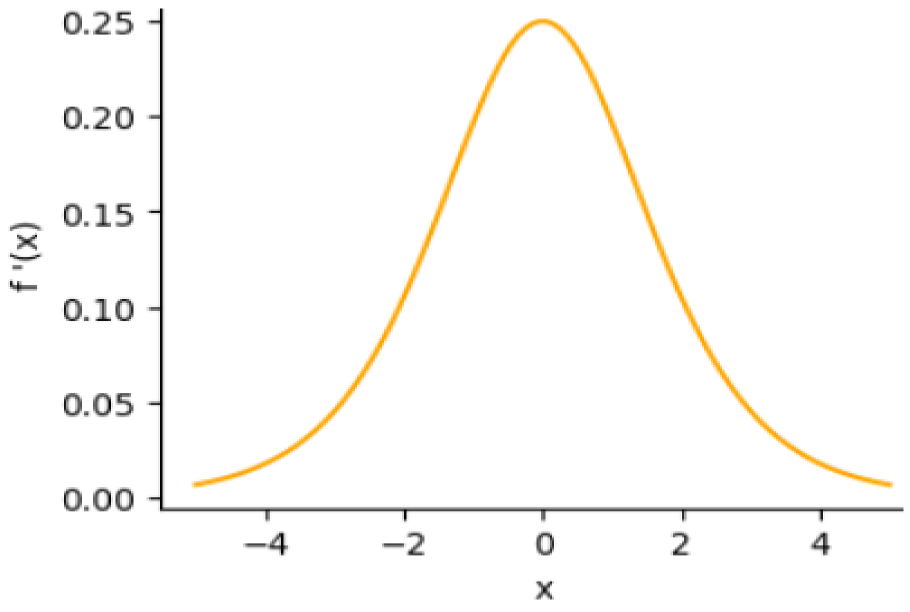
¿Qué valores toma f'(neta) cuando se trata de la función sigmoide entre 0 y 1?

FUNCIÓN SIGMOIDE

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

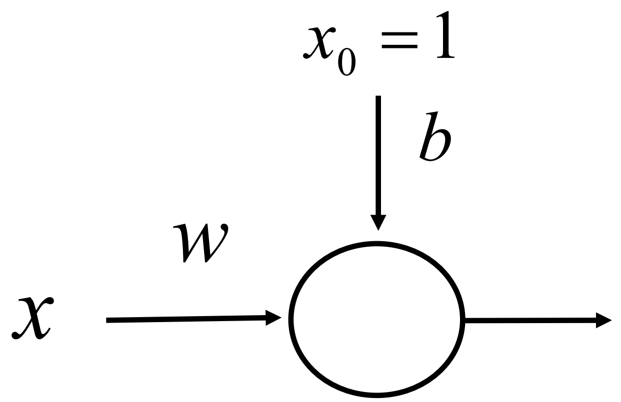
$$f'(x) = f(x) * (1 - f(x))$$





EJEMPLO: Entrene una neurona con función de activación sigmoide entre 0 y 1 para que reciba un 1 y responda 0

- Usando como Función de Costo el **Error Cuadrático Medio** (ECM)



$$y = \frac{1}{1 + e^{-(wx+b)}}$$

$x \; = \; 1$ (entrada)

t $\; = \; 0$ (salida esperada)

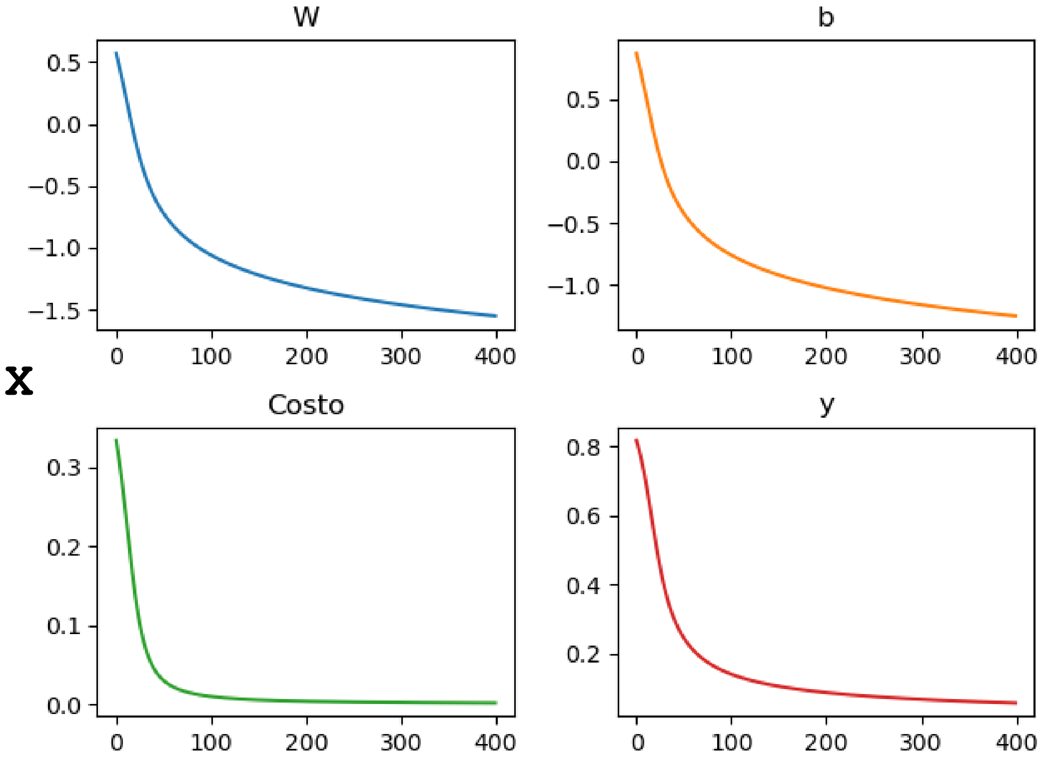
Función de costo (para 1 ejemplo)

$$C = \frac{(t-y)^2}{2}$$

$$\frac{\partial C}{\partial w} = -(t-y)\, \underbrace{[y\,(1-y)]}_{f'(neta)}\; x$$

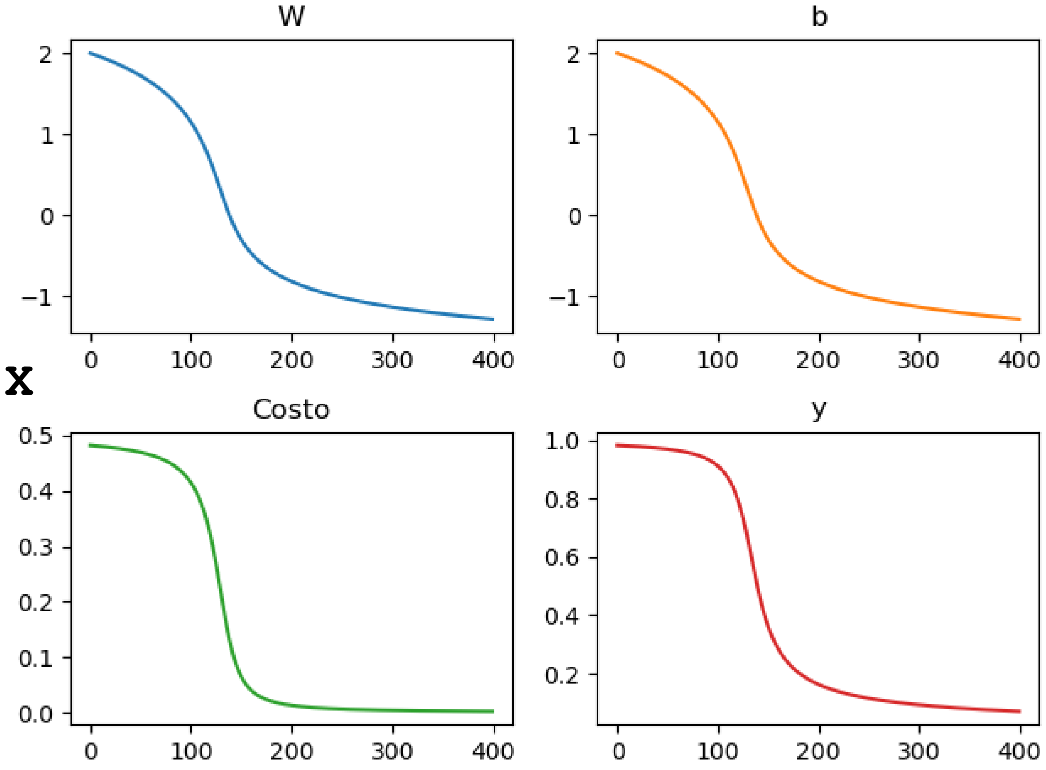
NeuronaNoLineal_1Ej.ipynb

```
X = 1
T = 0
W = 0.6
b = 0.9
MAX_ITE, alfa = 2000, 0.25
ite = 0
C_ant, Costo = 0, 1
while (ite<MAX_ITE) and (np.abs(C_ant-Costo)>10e-05):
    C_ant = Costo

    neta = W * X + b
    y = 1.0/(1+np.exp(-neta))
    Error = T - y
    Costo = (Error**2)/2
    gradiente_W = - Error * ( y * (1-y)) * X
    gradiente_b = - Error * ( y * (1-y))
    W = W - alfa * gradiente_W
    b = b - alfa * gradiente_b
    ite = ite + 1
```



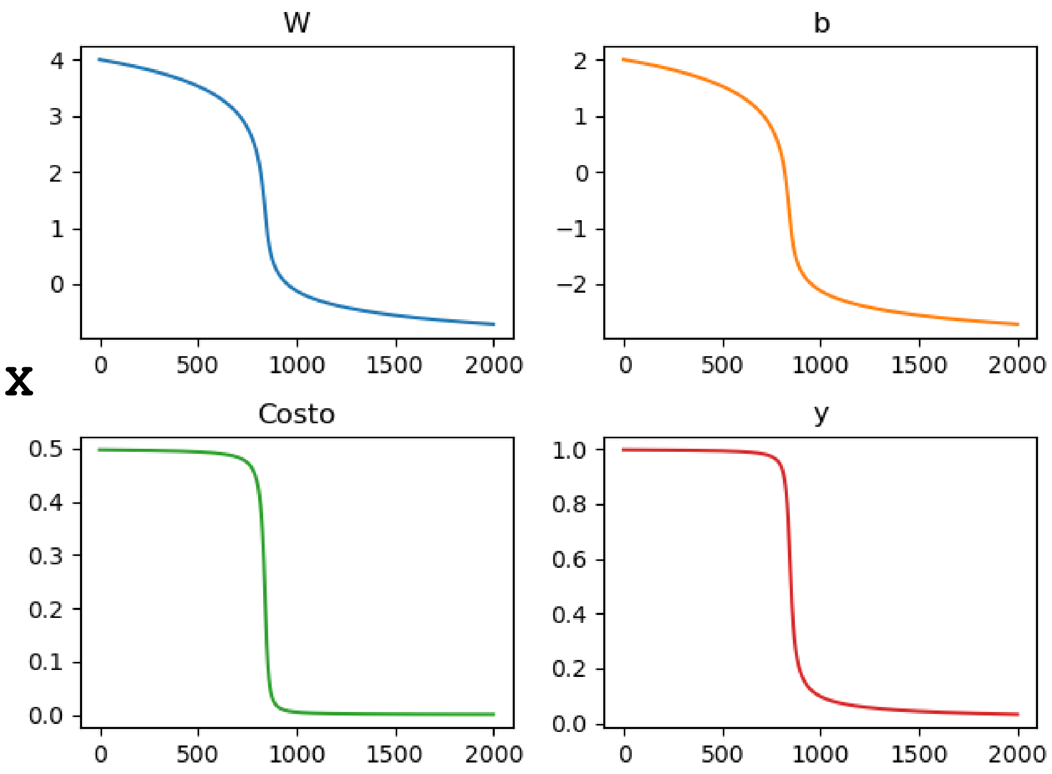
NeuronaNoLineal_1Ej.ipynb

```
X = 1
T = 0
W = 2
b = 2
MAX_ITE, alfa = 2000, 0.25
ite = 0
C_ant, Costo = 0, 1
while (ite<MAX_ITE) and (np.abs(C_ant-Costo)>10e-05):
    C_ant = Costo
    neta = W * X + b
    y = 1.0/(1+np.exp(-neta))
    Error = T - y
    Costo = (Error**2)/2
    gradiente_W = - Error * ( y * (1-y)) * X
    gradiente_b = - Error * ( y * (1-y))
    W = W - alfa * gradiente_W
    b = b - alfa * gradiente_b
    ite = ite + 1
```



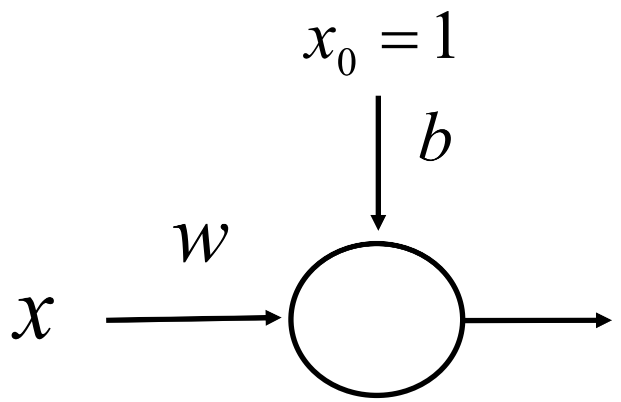
NeuronaNoLineal_1Ej.ipynb

```python
X = 1
T = 0
W = 4
b = 2
MAX_ITE, alfa = 2000, 0.25
ite = 0
C_ant, Costo = 0, 1
while (ite<MAX_ITE) and (np.abs(C_ant-Costo)>10e-05):
    C_ant = Costo

    neta = W * X + b
    y = 1.0/(1+np.exp(-neta))

    Error = T - y

    Costo = (Error**2)/2

    gradiente_W = - Error * ( y * (1-y)) * X
    gradiente_b = - Error * ( y * (1-y))

    W = W - alfa * gradiente_W
    b = b - alfa * gradiente_b

    ite = ite + 1
```



EJEMPLO: Entrene una neurona con función de activación sigmoide entre 0 y 1 para que reciba un 1 y responda 0

- Función de Costo: **Entropía Cruzada Binaria** (EC_binaria)



$$y = \frac{1}{1+e^{-(wx+b)}}$$

$x = 1$ (entrada)

t $= 0$ (salida esperada)

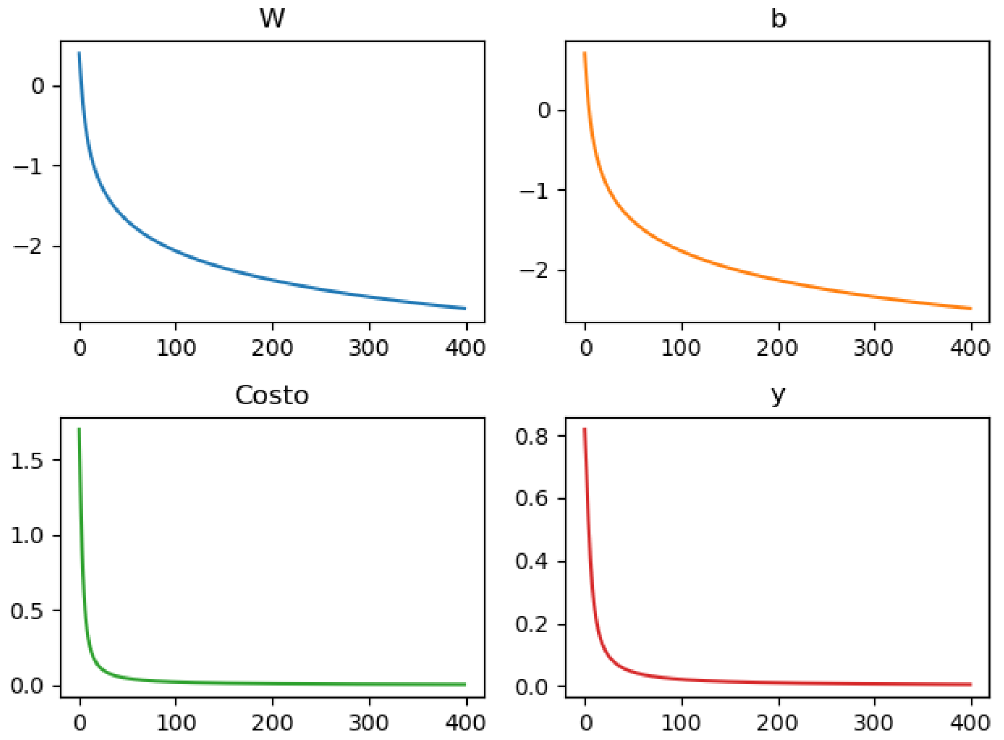
Función de costo (para 1 ejemplo)

$$C = -(t \ln y + (1-t)\ln(1-y))$$

$$\frac{\partial C}{\partial w} = -(t-y)\ x$$

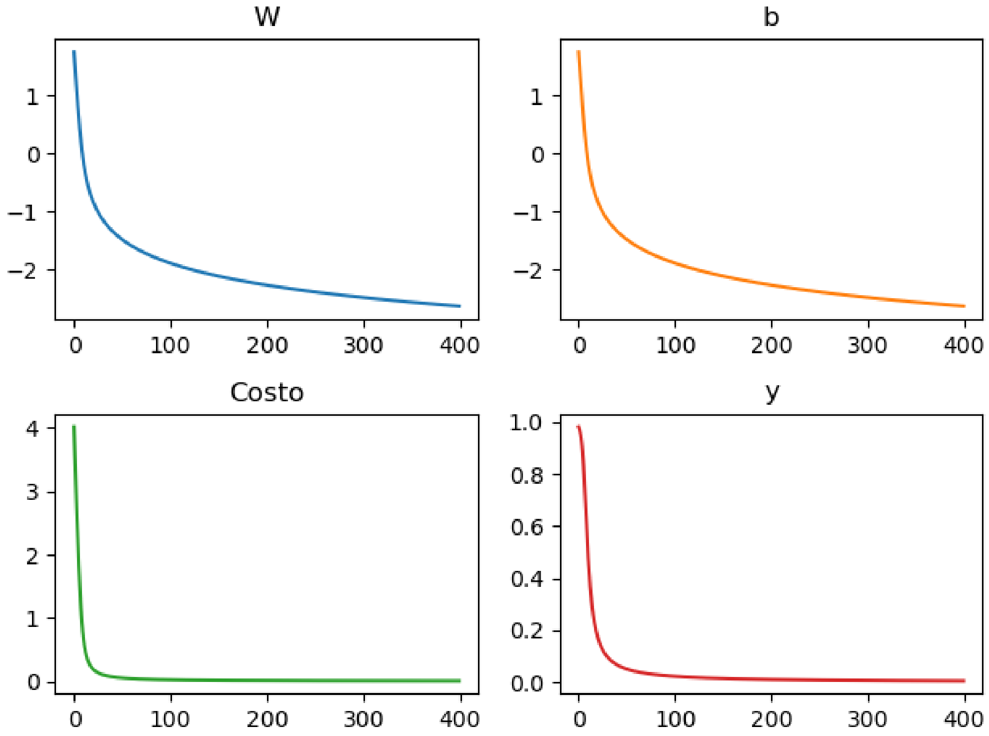
NeuronaNoLineal_1Ej_EC.ipynb

```
X = 1
T = 0
W = 0.6
b = 0.9
MAX_ITE, alfa = 2000, 0.25
ite = 0
C_ant, Costo = 0, 1
while (ite<MAX_ITE) and (np.abs(C_ant-Costo)>10e-05):
    C_ant = Costo

    neta = W * X + b
    y = 1.0/(1+np.exp(-neta))
    Error = T - y

    Costo = -(T*np.log(y)+(1-T)*np.log(1-y))

    gradiente_W = - Error * X
    gradiente_b = - Error

    W = W - alfa * gradiente_W
    b = b - alfa * gradiente_b

    ite = ite + 1
```



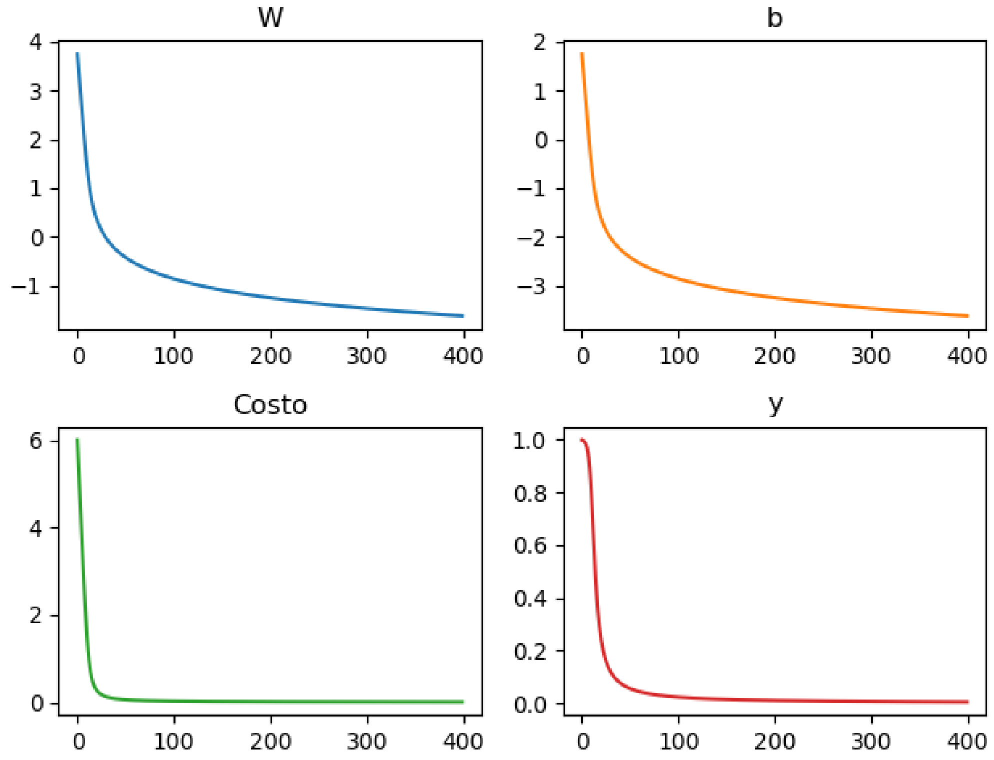
NeuronaNoLineal_1Ej_EC.ipynb

```
X = 1
T = 0
W = 2
b = 2
MAX_ITE, alfa = 2000, 0.25
ite = 0
C_ant, Costo = 0, 1
while (ite<MAX_ITE) and (np.abs(C_ant-Costo)>10e-05):
    C_ant = Costo

    neta = W * X + b
    y = 1.0/(1+np.exp(-neta))
    Error = T - y

    Costo = -(T*np.log(y)+(1-T)*np.log(1-y))

    gradiente_W = - Error * X
    gradiente_b = - Error

    W = W - alfa * gradiente_W
    b = b - alfa * gradiente_b

    ite = ite + 1
```



NeuronaNoLineal_1Ej_EC.ipynb

```
X = 1
T = 0
W = 4
b = 2
MAX_ITE, alfa = 2000, 0.25
ite = 0
C_ant, Costo = 0, 1
while (ite<MAX_ITE) and (np.abs(C_ant-Costo)>10e-05):
    C_ant = Costo

    neta = W * X + b
    y = 1.0/(1+np.exp(-neta))
    Error = T - y

    Costo = -(T*np.log(y)+(1-T)*np.log(1-y))

    gradiente_W = - Error * X
    gradiente_b = - Error

    W = W - alfa * gradiente_W
    b = b - alfa * gradiente_b

    ite = ite + 1
```



FUNCIONES DE COSTO

Entropía Cruzada Binaria

- **Mejor ajuste** para problemas de clasificación binaria.
- Produce **gradientes más grandes** cuando las predicciones están muy alejadas de las etiquetas verdaderas, acelerando el entrenamiento.

*La **entropía cruzada** es más adecuada para clasificación binaria, ya que aprovecha la naturaleza probabilística de la sigmoide.*

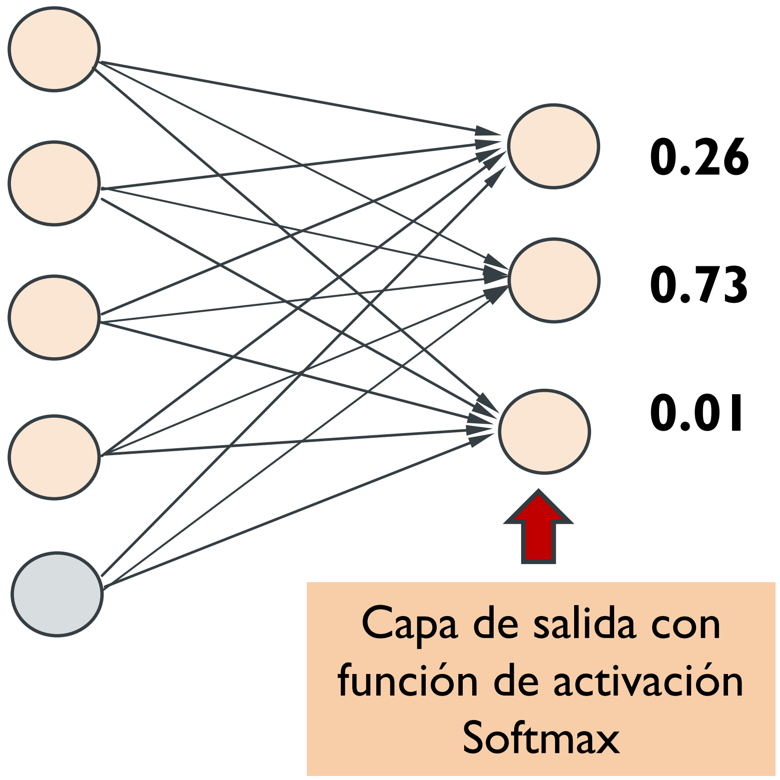
Error Cuadrático Medio

- Usualmente empleado en problemas de **regresión**.
- Gradientes más pequeños especialmente cuando el valor de la función sigmoide se acerca a 0 o a 1 lo que puede provocar **aprendizaje más lento**.

*El **ECM**, aunque puede usarse en clasificación, no genera gradientes tan eficientes, especialmente cuando las predicciones son extremas (cercanas a 0 o 1).*

FUNCIÓN SOFTMAX

- Se utiliza como función de activación en la última capa para normalizar la salida de la red a una distribución de probabilidad.



CAPA SOFTMAX

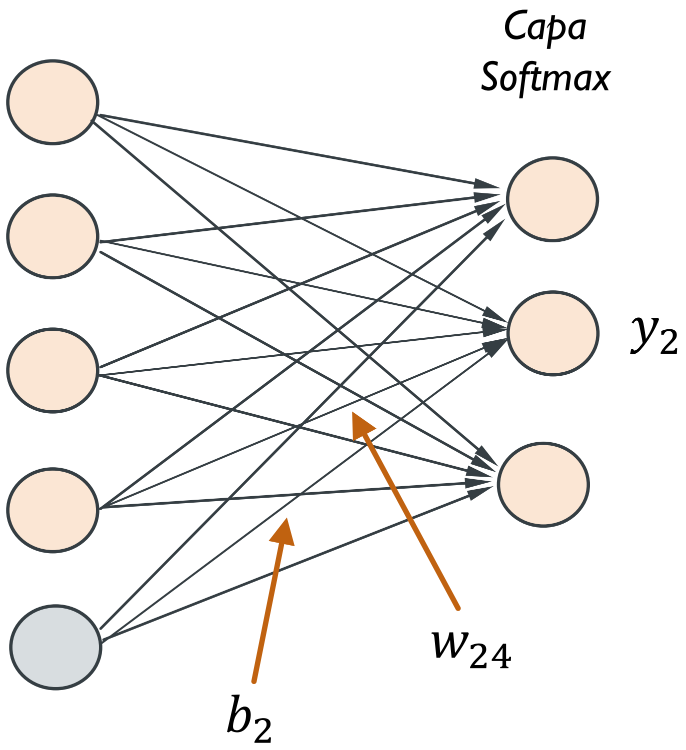
$$neta_j = \sum_i w_{ji}\, x_i + b_j$$

$$y_j = \frac{e^{neta_j}}{\sum_k e^{neta_k}}$$

- La salida de la capa es una distribución de probabilidad

$$y_j > 0 \qquad j = 1..k$$

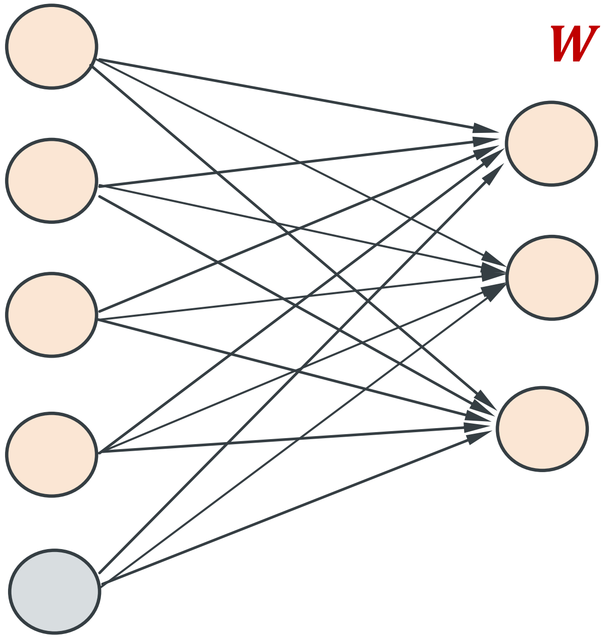
$$\sum_j y_j = 1$$



Note que el incremento en algún y_j producirá disminuciones en el resto

FUNCIÓN SOFTMAX

- Ejemplo



$\boldsymbol{W * x^T + b}$

5

4

2

Función sigmoide

$\boldsymbol{f(W * x^T + b)}$

0.99

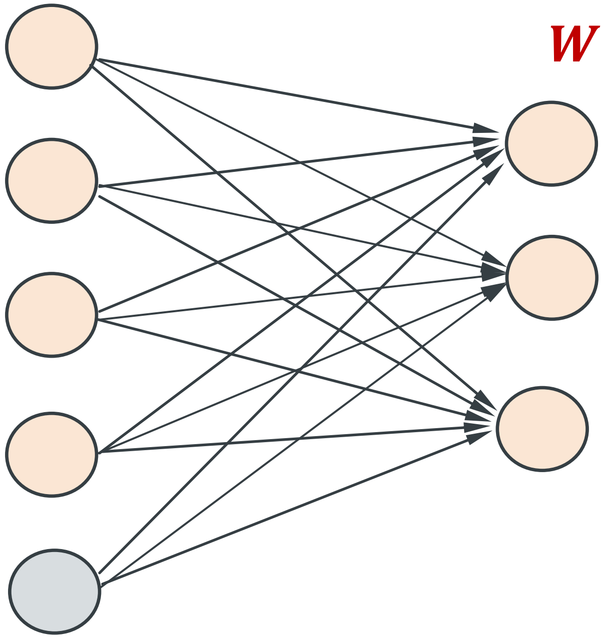
0.98

0.88

FUNCIÓN SOFTMAX

- Ejemplo

Neta	sigmoid	exp(neta)	softmax
5	0,99	148,41	0,71
4	0,98	54,60	0,26
2	0,88	7,39	0,04



$W * x^T + b$

5

4

2

Función softmax →

$f(W * x^T + b)$

0.71

0.26

0.04

FUNCIÓN SOFTMAX

Entrada

Propagación hacia adelante →

Salidas de la propagación hacia adelante

cat	dog	horse
5	4	2
4	2	8
4	4	1

Función sigmoide →

Salidas de la capa con función sigmoide

cat	dog	horse
0.99	0.98	0.88
0.98	0.88	0.99
0.98	0.98	0.73

FUNCIÓN SOFTMAX

Neta	sigmoid	exp(neta)	softmax
5	0,99	148,41	0,71
4	0,98	54,60	0,26
2	0,88	7,39	0,04

Entrada

Propagación hacia adelante

Salidas de la propagación hacia adelante

cat	dog	horse
5	4	2
4	2	8
4	4	1

Función Softmax

Salidas de la capa Softmax

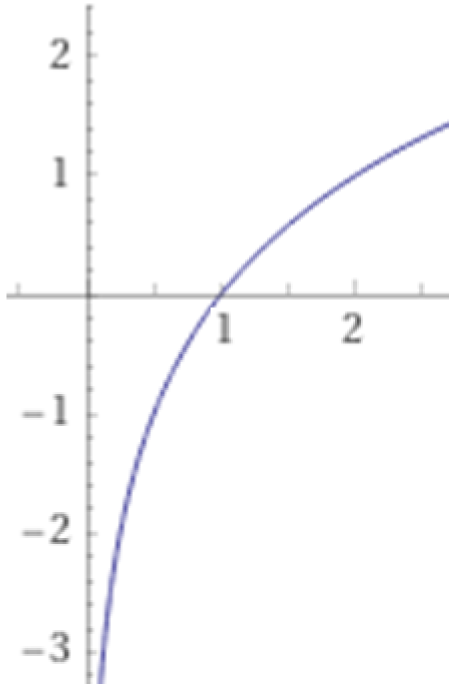
cat	dog	horse
0.71	0.26	0.04
0.02	0.00	0.98
0.49	0.49	0.02

ENTROPIA CRUZADA CATEGORICA

$$y_j = \frac{e^{neta_j}}{\sum_k e^{neta_k}}$$

- **Función de costo**

$$C = -\sum_k t_k \ln y_k$$



donde $\boldsymbol{t}$ es un vector binario que vale 1 sólo en la posición correspondiente al valor de clase esperado.

Luego

$$C = -\ln y_s$$

s es la neurona correspondiente al valor de clase esperado

ENTROPIA CRUZADA CATEGORICA

$$y_j = \frac{e^{neta_j}}{\sum_k e^{neta_k}}$$

- **Función de costo**

$$C = -\ln y_s$$

S es la neurona correspondiente al valor de clase esperado

- Derivada de la función de costo

$$\frac{\partial C}{\partial w_{jk}} = -(t_j - y_j)\, x_k \qquad \frac{\partial C}{\partial b_j} = -(t_j - y_j)$$

Coincide con la derivada de la entropía cruzada binaria

CAPA SOFTMAX - ENTROPIA CRUZADA CATEGORICA

Entrada	**Salida Softmax** cat	dog	horse	**Loss, L(a)** NLL
	0.71	0.26	0.04	0.34
	0.02	0.00	0.98	0.02
	0.49	0.49	0.02	0.71

$-\log(y_s)$ *en la clase correcta*

La clase correcta está pintada de rojo

Total = 1.07

- Sólo se evalúa en la neurona correspondiente a la salida esperada.
- Cuando más cerca está de 1 menor será el error.
- A menor valor de la neurona softmax correspondiente a la clase correcta, mayor error.

ClassRNMulticlase.py

nn = RNMulticlase (alpha=0.01, n_iter=50, cotaE=10E-07, FUN='sigmoid', COSTO='ECM', random_state=None)

- Parámetros de entrada
  - **alpha**: valor en el intervalo (0, 1] que representa la velocidad de aprendizaje.
  - **n_iter**: máxima cantidad de iteraciones a realizar.
  - **cotaE**: termina si la diferencia entre dos errores consecutivos es menor que este valor.
  - **FUN:** función de activación – 'sigmoid', 'tanh', 'softmax'.
  - **COSTO**: función de costo – 'ECM', 'EC_binaria', 'EC'
  - **random_state**: None si los pesos se inicializan en forma aleatoria, un valor entero para fijar la semilla

ClassRNMulticlase.py

nn = RNMulticlase(alpha=0.01, n_iter=50, cotaE=10E-07, FUN='sigmoid',
COSTO='ECM', random_state=None)

nn.fit(X,T)

- Parámetros de entrada
  - **X** : arreglo de NxM donde N es la cantidad de ejemplos y M la cantidad de atributos.
  - **T** : arreglo de NxK donde N es la cantidad de ejemplos y K es la cantidad de neuronas de salida
- Retorna
  - **w_** : arreglo de KxM siendo K el tamaño de la capa de salida y M la cantidad de atributos de entrada.
  - **b_** : arreglo de K elementos formado por los bias de cada una de las K neuronas de la capa de salida.
  - **errors_**: errores cometidos en cada iteración.

ClassRNMulticlase.py

```
nn = RNMulticlase(alpha=0.01, n_iter=50, cotaE=10E-07, FUN='sigmoid',
                  COSTO='ECM', random_state=None)
nn.fit(X,T)
nn.save( nomArch )
```

- Parámetro de entrada
  - **nomArch** : nombre del archivo donde se guardará la matriz de pesos w_ y el vector de bias b_
- Retorna
  - Quedará generado el archivo con extensión npz si no se indicó ninguna.

ClassRNMulticlase.py

 # antes de cargar los pesos debe definirse el objeto indicando al menos la función de activación

nn = RNMulticlase(FUN=‘softmax’)

nn.load(nomArch)

- Parámetro de entrada
  - **nomArch** : nombre del archivo del cual se cargarán la matriz de pesos w_ y el vector de bias b_

ClassRNMulticlase.py

Y = nn.predict_nOut(X)

- Parámetros de entrada
  - **X** : arreglo de NxM donde N es la cantidad de ejemplos y M la cantidad de atributos.

- Retorna: un arreglo con el resultado de aplicar la neurona general entrenada previamente con fit() a la matriz de ejemplos X.
  - **Y** : arreglo de NxK donde N es la cantidad de ejemplos y K es la cantidad de neuronas de salida con valores continuos.

ClassRNMulticlase.py

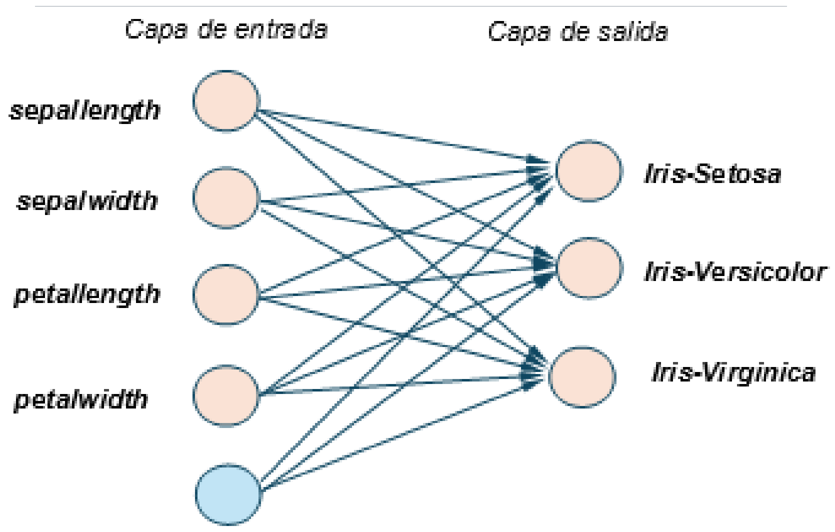
Y = nn.predict(X)

- Parámetros de entrada
  - **X** : arreglo de NxM donde N es la cantidad de ejemplos y M la cantidad de atributos.

- Retorna: un arreglo con el resultado de aplicar la neurona general entrenada previamente con fit() a la matriz de ejemplos X.
  - **Y** : arreglo de N elementos indicando para cada ejemplo el número de la clase predicha.

RNMulticlase_IRIS_RN_COMPLETO.ipynb

SIGMOIDE vs SOFTMAX



- Si la función de activación es **sigmoide**
  - Las neuronas de salida aprenden en forma independiente.
  - La salida no tiene que sumar 1.
  - Permite interpretar cada neurona como una probabilidad independiente.
  - Puede ser útil si las clases no son mutuamente excluyentes.
  - Es conveniente utilizar como función de costo la Entropía cruzada binaria (gradientes más grandes).

- Si la función de activación es **Softmax**
  - Las neuronas de salida aprenden en forma conjunta.
  - La suma de las salidas siempre es 1, lo que permite interpretar las salidas como una distribución de probabilidad.
  - Muy útil cuando las clases son mutuamente excluyentes.
  - Utiliza como función de costo la Entropía Cruzada Categórica.