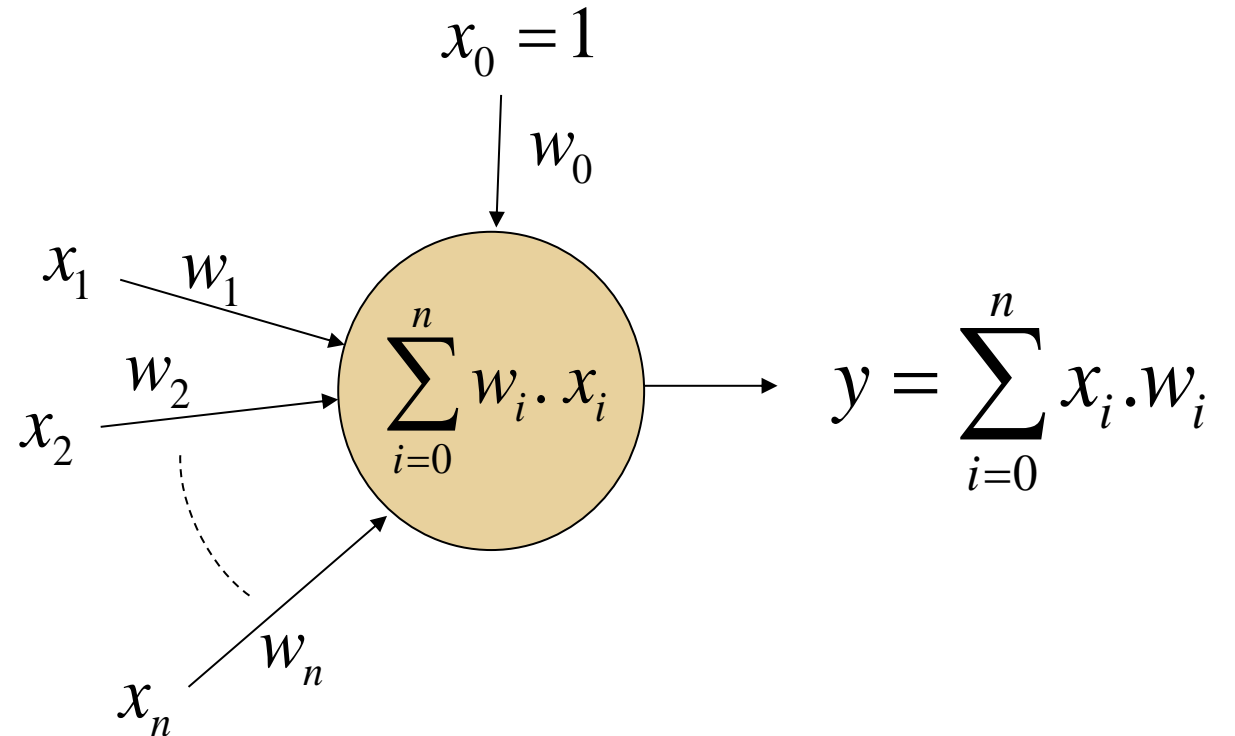


# Combinador Lineal

- Resuelve un problema de **Regresión Lineal**
- Función de Error
  - ▣ Error cuadrático medio
- Técnica de optimización
  - ▣ Descenso de gradiente estocástico



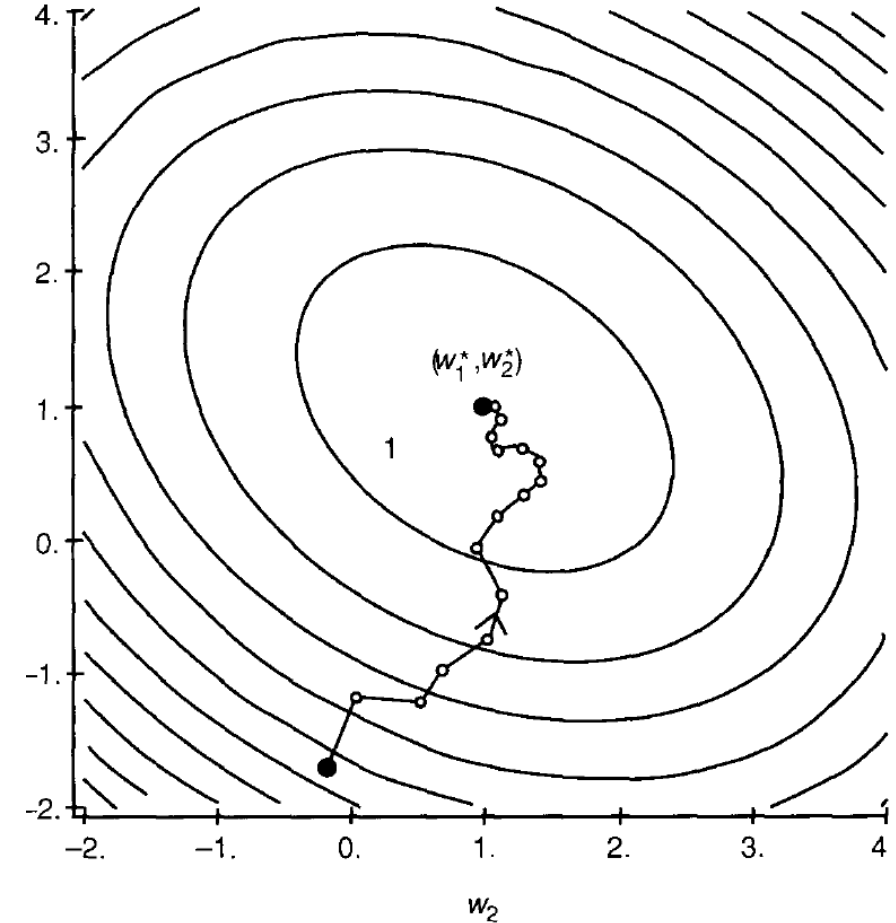
# Técnica del descenso del gradiente estocástico

$$w(t+1) = w(t) + \Delta w(t)$$

$$w(t+1) = w(t) - \mu \nabla \xi(w(t))$$

□ se utiliza

$$\xi = \langle \varepsilon_k^2 \rangle \approx \varepsilon_k^2 = (d_k - \sum_{i=0}^N x_{ik} w_i)^2$$



# Técnica del descenso del gradiente estocástico

$$w(t+1) = w(t) + \Delta w(t)$$

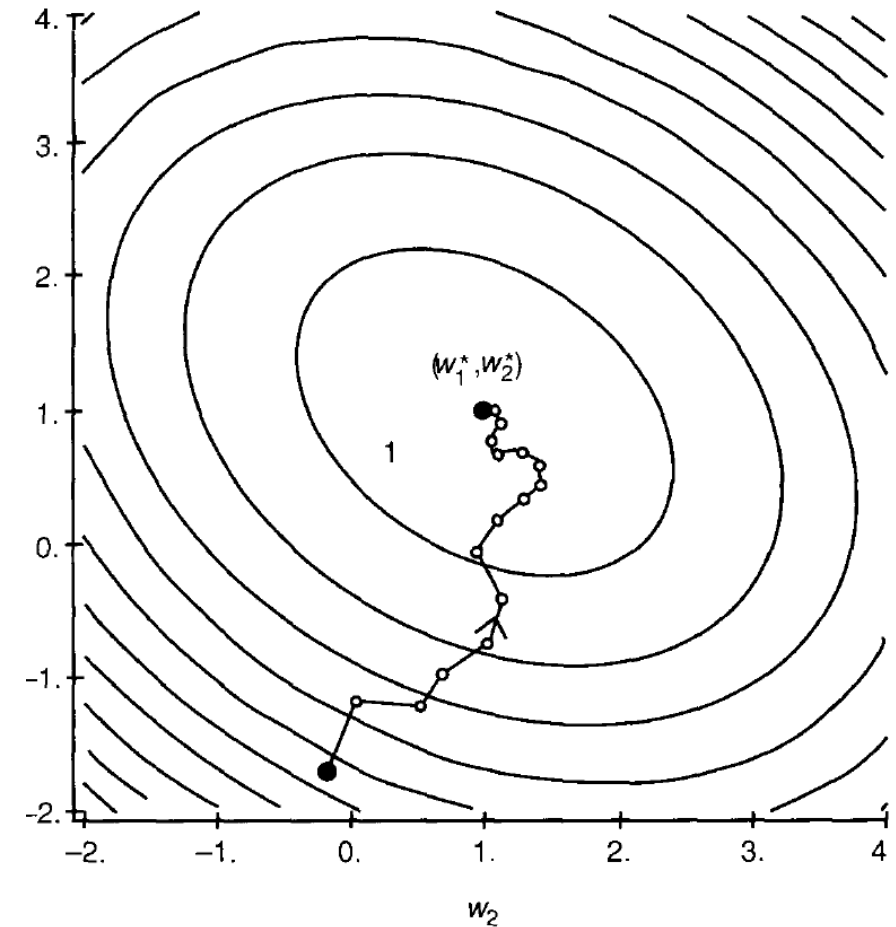
$$w(t+1) = w(t) - \mu \nabla \xi(w(t))$$

□ se utiliza

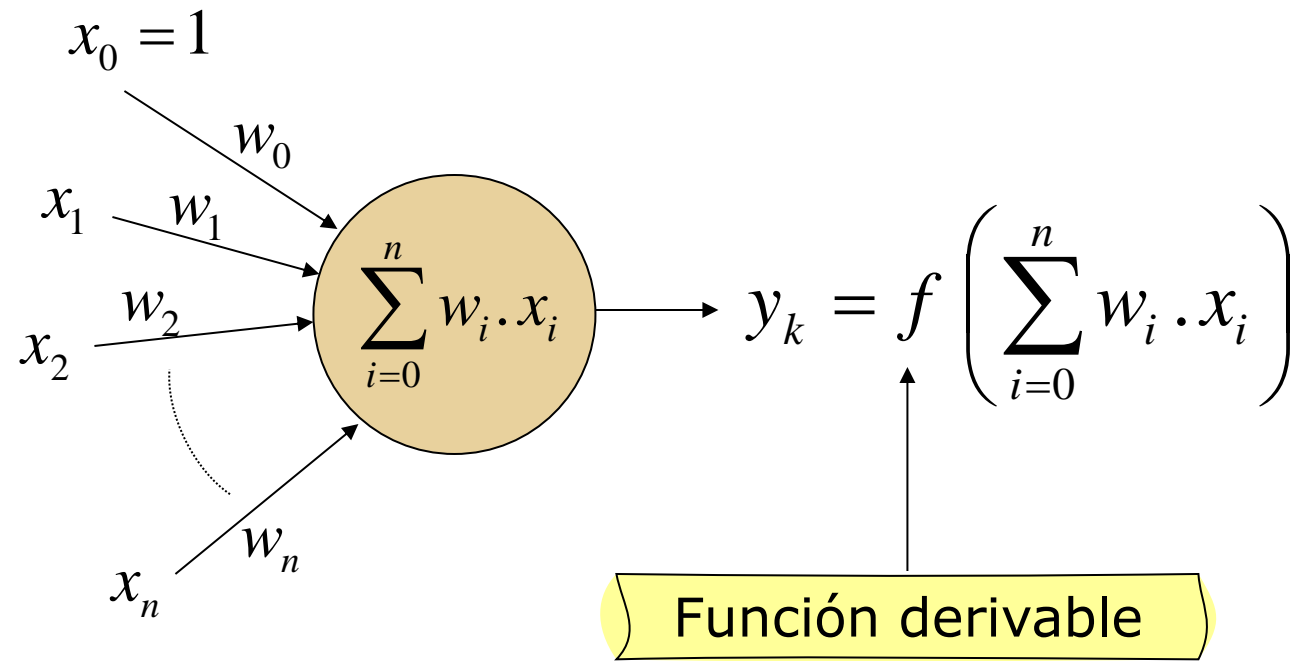
$$\xi = \langle \varepsilon_k^2 \rangle \approx \varepsilon_k^2 = (d_k - \sum_{i=0}^N x_{ik} w_i)^2$$

□ veamos que

$$\nabla \varepsilon_k^2(t) = -2\varepsilon_k(t)x_k$$



# Neurona General



# Función de Salida LINEAL

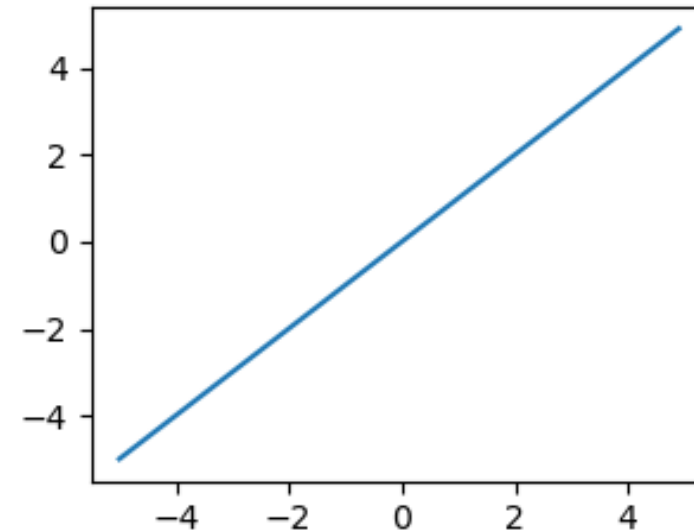
$$f(x) = x$$

$$f'(x) = 1$$

desde Python

```
import numpy as np
import grafica as gr
from matplotlib import pyplot as plt

x = np.array(range(-50,50,1))/10.0
y = gr.evaluar('purelin', x)
plt.plot(x,y,'-')
```



# Función SIGMOIDE $\in (0,1)$

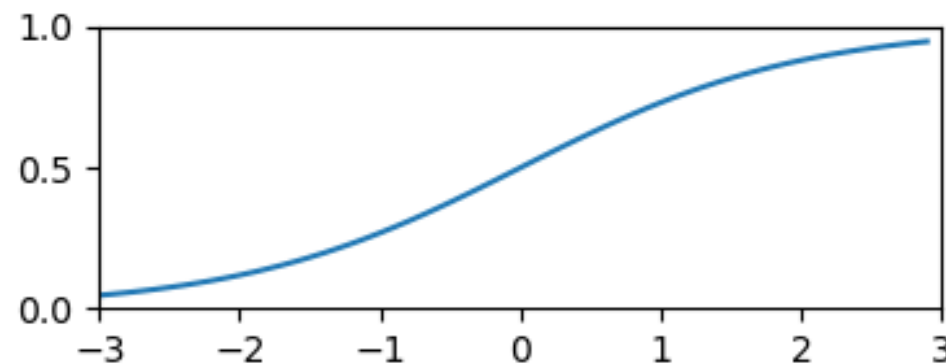
$$f(x) = \frac{1}{1 + e^{-x}}$$

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

desde Python

```
import numpy as np
import grafica as gr
from matplotlib import pyplot as plt

x = np.array(range(-30,30,1))/10.0
y = gr.evaluar('logsig', x)
plt.plot(x,y, '-')
plt.axis([-3, 3, 0, 1])
plt.show()
```



# Función SIGMOIDE $\in (-1,1)$

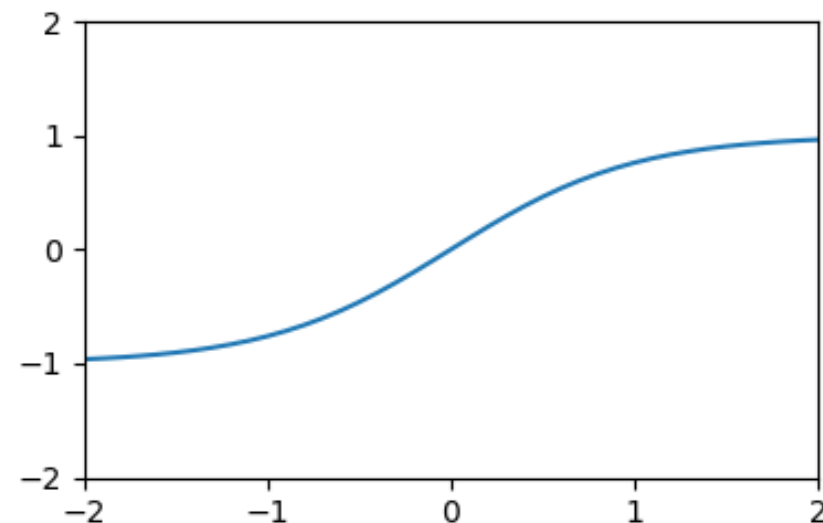
$$f(x) = \frac{2}{1 + e^{-2x}} - 1$$

$$f'(x) = 1 - f(x)^2$$

desde Python

```
import numpy as np
import grafica as gr
from matplotlib import pyplot as plt

x = np.array(range(-30,30,1))/10.0
y = gr.evaluar('tansig', x)
plt.plot(x,y, '-')
plt.axis([-2, 2, -2, 2])
plt.show()
```



# Ejemplo

- Dados los siguientes conjuntos de puntos del plano

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

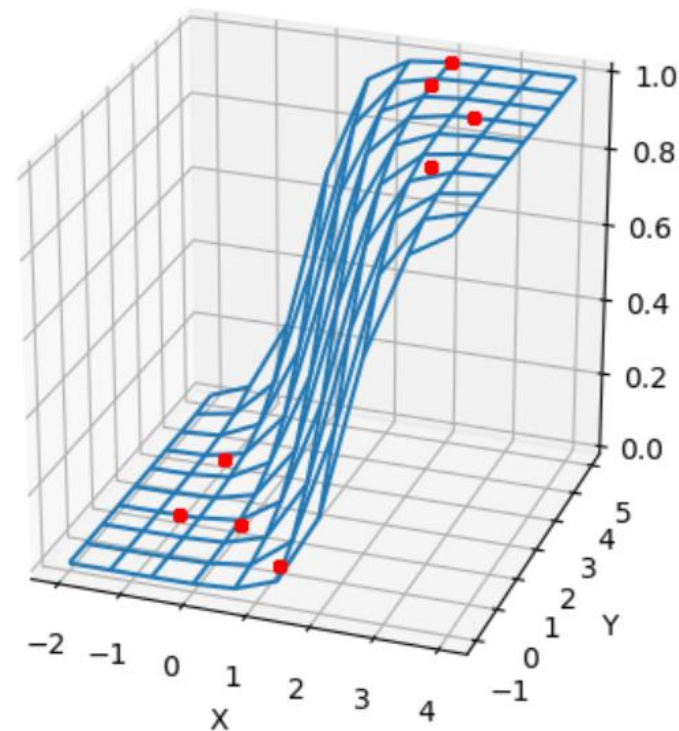
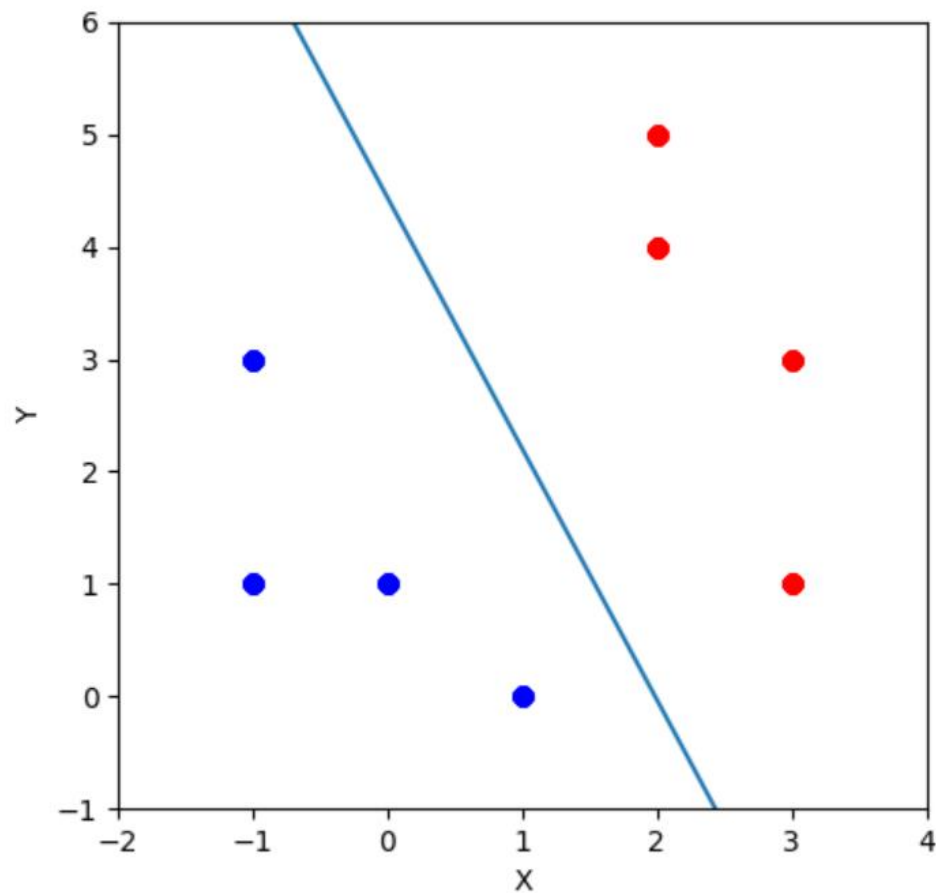
$$B = \{(3,1), (3,3), (2,4), (2,5)\}$$

- ▣ 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

# Entrenamiento de una neurona no lineal

- Seleccionar el valor de  $\alpha$
- Inicializar los pesos  $W$  y  $b$  con valores random
- Mientras (la variación del ECM sea mayor a la cota prefijada)
  - ▣ Para cada ejemplo
    - Ingresar el ejemplo a la red.
    - Calcular el error  $\varepsilon = (y - \hat{y})$  y  $\frac{\partial \varepsilon^2}{\partial w} = \frac{\partial (y - \hat{y})^2}{\partial w}$
    - Actualizar los pesos de la red

$$w_i = w_i - \alpha \frac{\partial \varepsilon^2}{\partial w_i}$$

## ¿Cómo sería la derivada del error si la neurona no es lineal?

$$\frac{\partial \varepsilon^2}{\partial w} = \left[ \frac{\partial (y - \hat{y})^2}{\partial w_0}; \quad \dots; \quad \frac{\partial (y - \hat{y})^2}{\partial w_n} \right]$$

$$\frac{\partial \varepsilon^2}{\partial w_j} = -2(y - \hat{y}) \frac{\partial f}{\partial (neta)} \frac{\partial (neta)}{\partial w_j}$$

$f'$  →  $\frac{\partial f}{\partial (neta)}$

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

$$\frac{\partial \varepsilon^2}{\partial w_j} = -2(y - \hat{y}) f'(neta) x_j$$

# Entrenamiento de una neurona no lineal

- Seleccionar el valor de  $\alpha$
- Inicializar los pesos  $W$  y  $b$  con valores random
- Mientras (la variación del ECM sea mayor a la cota prefijada)
  - ▣ Para cada ejemplo
    - Ingresar el ejemplo a la red.
    - Calcular  $\frac{\partial \varepsilon^2}{\partial w_i} = -2 * \varepsilon * f'(neta) * x_i$
    - Actualizar los pesos de la red

$$w_i = w_i - \alpha \frac{\partial \varepsilon}{\partial w_i} = w_i + 2\alpha * \varepsilon * f'(neta) * x_i$$

# AND

- Utilice una neurona no lineal con salida sigmoide para resolver el problema del AND

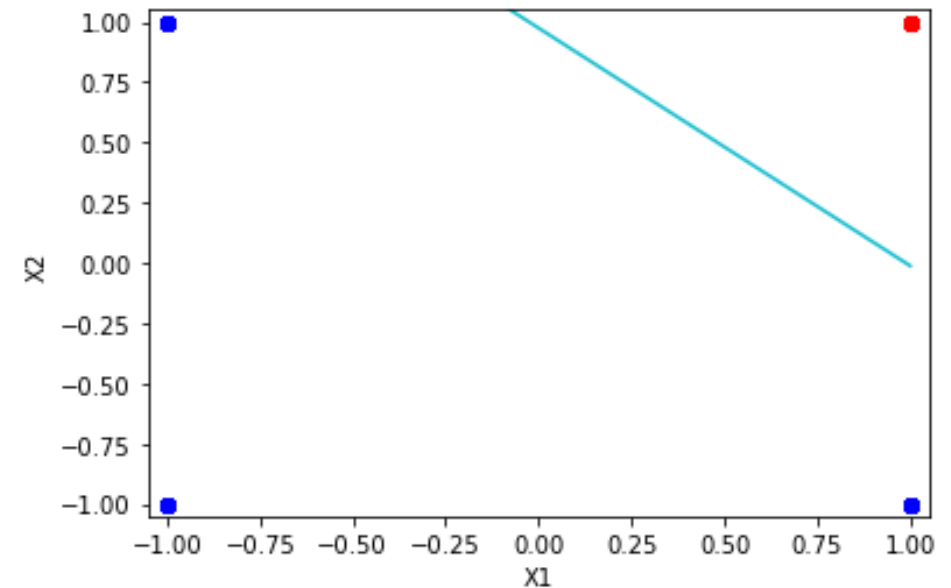
$$f(x) = \frac{1}{1 + e^{-x}}$$

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

---

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

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



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

```

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=%8.5f" % (ite,W[0],W[1],E))

```

*Los pesos iniciales son aleatorios*

```

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=%8.5f" % (ite,W[0],W[1],E))

```

*Parámetros del entrenamiento*

```

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=%8.5f" % (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 es inferior a cierta cota*



```

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 del combinador lineal y evaluamos el error cometido*

```

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=%8.5f" % (ite,W[0],W[1],E))

```

*Son parte del vector gradiente*

```

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=%8.5f" % (ite,W[0],W[1],E))

```

*Actualizamos los pesos en la  
dirección del gradiente negativo*

```

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=%8.5f" % (ite,W[0],W[1],E))

```

*Acumulamos el cuadrado de los errores cometidos*

```

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=%8.5f" % (ite,W[0],W[1],E))

```

Dividimos por la cantidad de ejemplos para obtener el ECM

```

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=%8.5f" % (ite,W[0],W[1],E))

```

# ClassNeuronaGral.py

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

## □ 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', '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

```
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]))
```

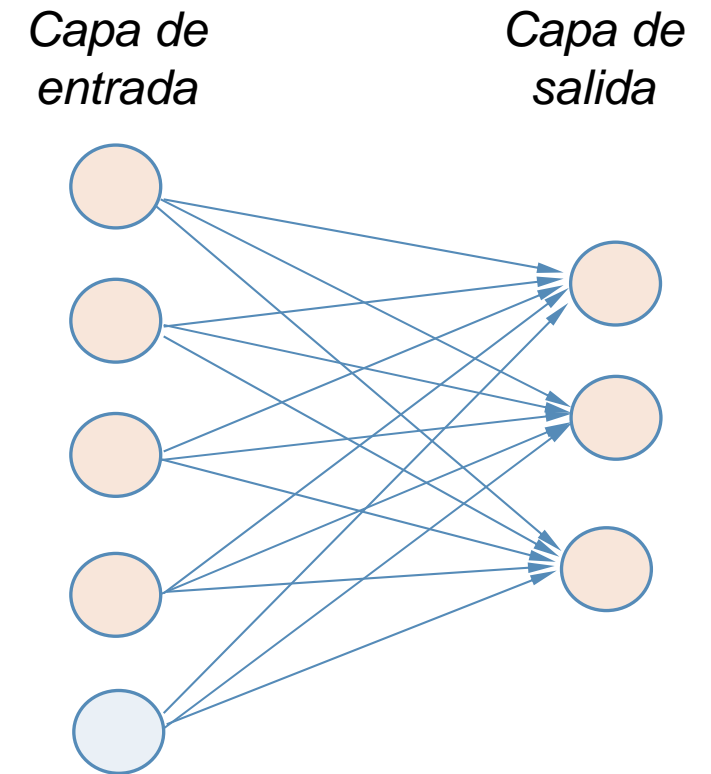
# Ejemplo

36

- 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.



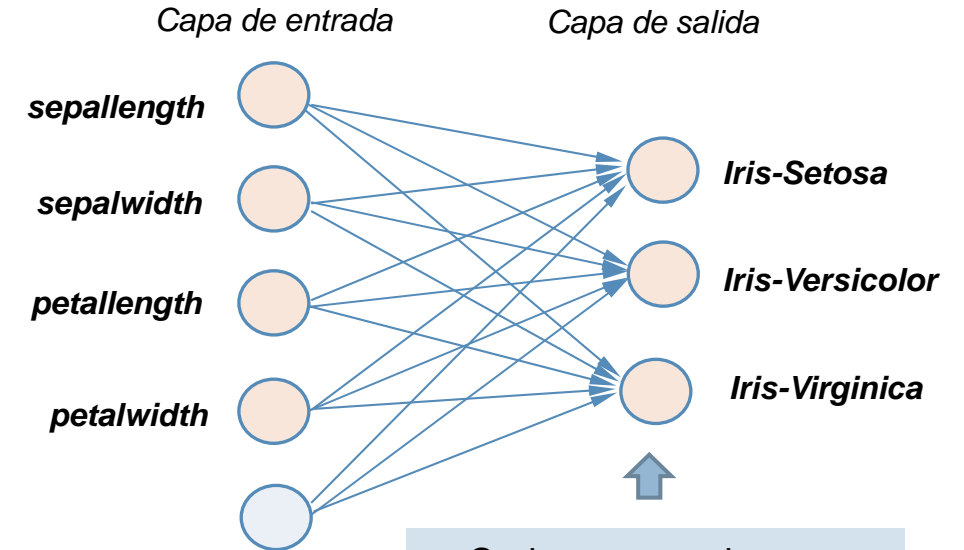
# Clasificación de flores de Iris

**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]]
```

**Y**

```
[[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 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]]
```

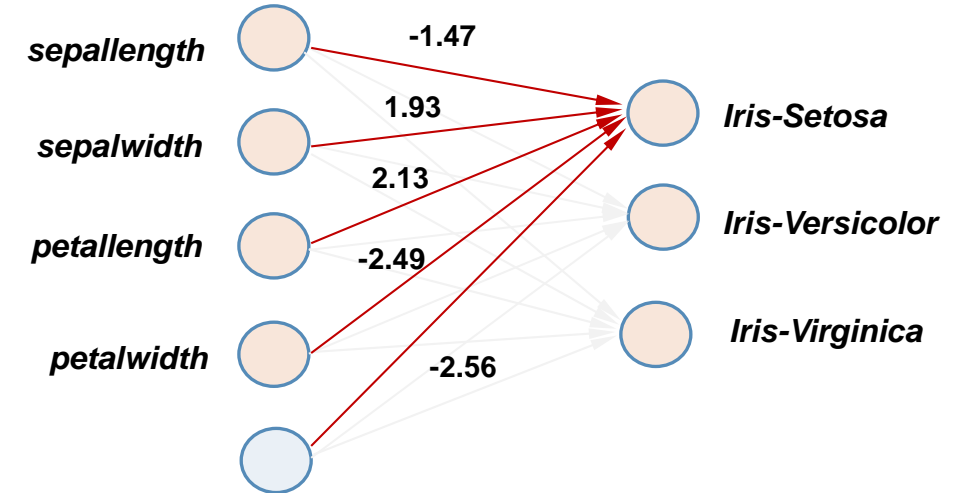
**Y**

```
[[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

Capa de entrada

Capa de salida



```
[[-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**

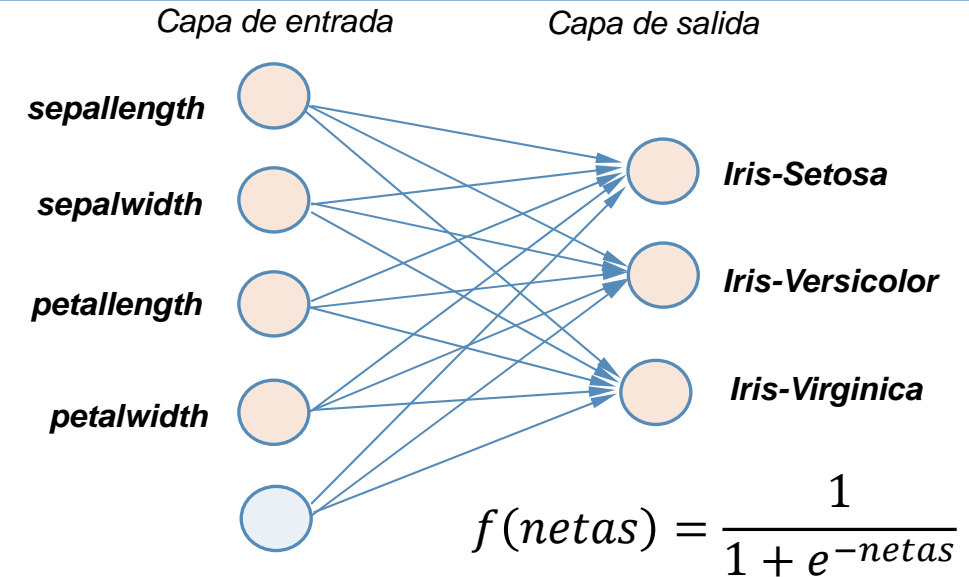
# Clasificación de flores de Iris

**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]]
```

**Y**

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



Ingresar el primer ejemplo a la red y  
calcular su salida

# 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]]

**Y**

[[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]]



**$x^T$**

\*

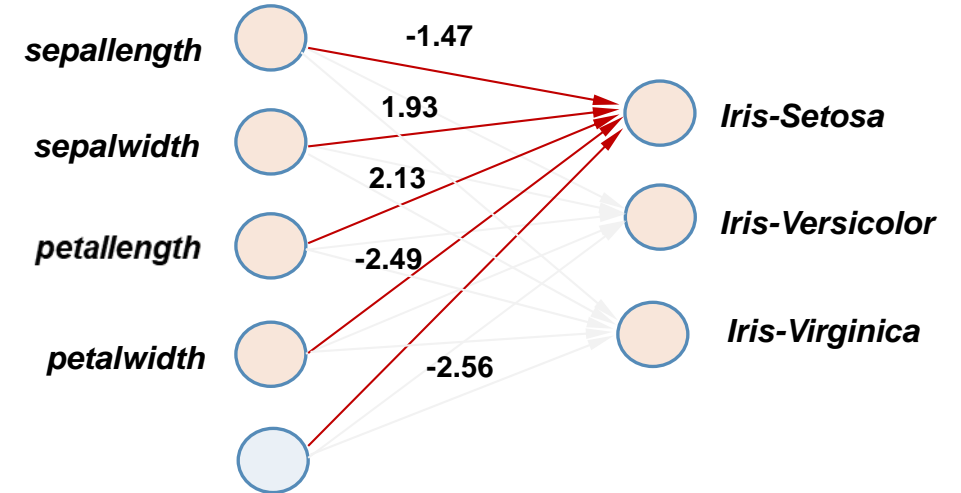
[ [-1.73],  
[-0.05],  
[-1.38],  
[-1.31]]

+

**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]]

**Y**

[[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]]

+

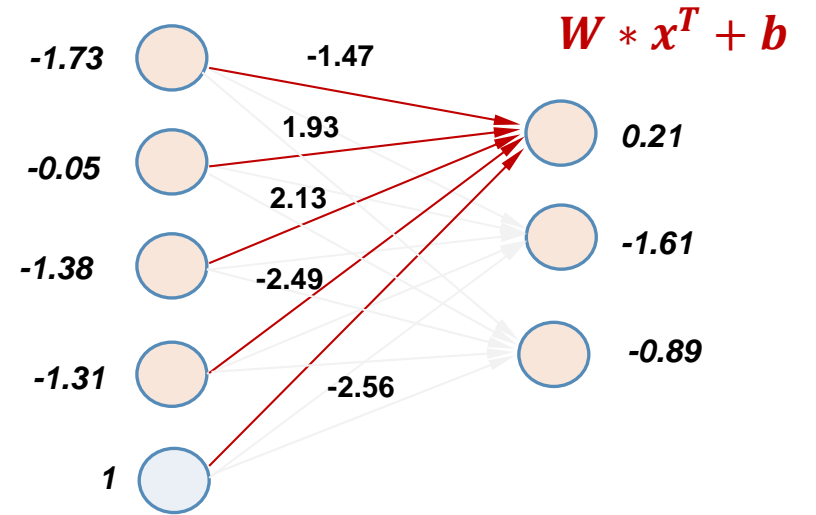
**b**

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

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

=

[ [ 0.21],  
 [ -1.61],  
 [ -0.89]]



# 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]]
```

**Y**

```
[[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]]
```

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

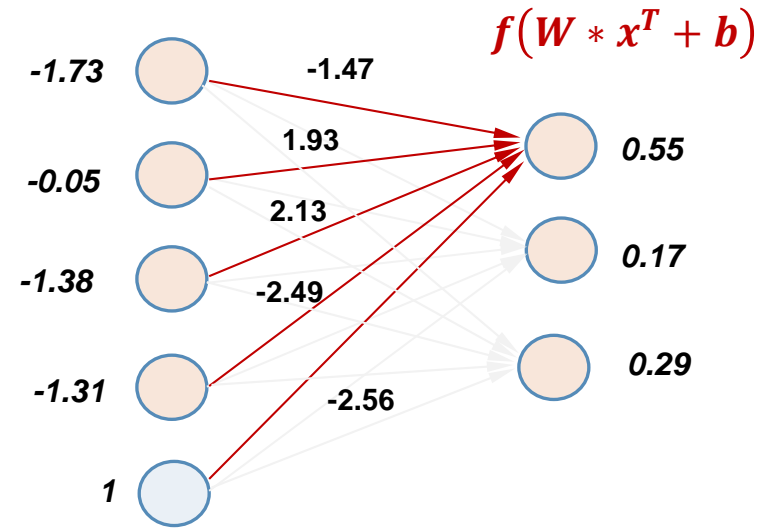
```
[[ 0.21],
 [-1.61],
 [-0.89]]
```

**salidas =  $f(netas)$  =**

```
[[ 1/(1+exp(-0.21))],
 [ 1/(1+exp(1.61))],
 [ 1/(1+exp(0.89))]]
```

**=**

```
[[0.5521],
 [0.1669],
 [0.2921]]
```



Calculamos el error cometido en cada neurona

# 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]]
```


**Y**

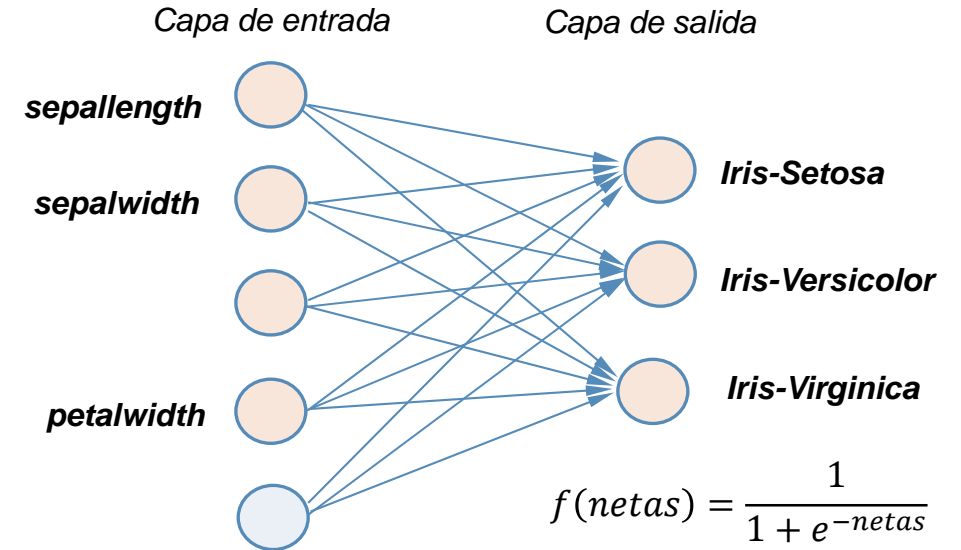
```
[1,0,0],  
[0,1,0],  
[0,0,1],  
[1,0,0],  
[0,0,1]]
```

- Error en la respuesta de la red para este ejemplo

```
ErrorSalida = y.T - salidas
```

```
ErrorSalida =  $\begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix}$  -  $\begin{bmatrix} 0.5521 \\ 0.1669 \\ 0.2921 \end{bmatrix}$  =  $\begin{bmatrix} 0.4479 \\ -0.1669 \\ -0.2921 \end{bmatrix}$ 
```

 **salidas**



# 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]]
```

**Y**

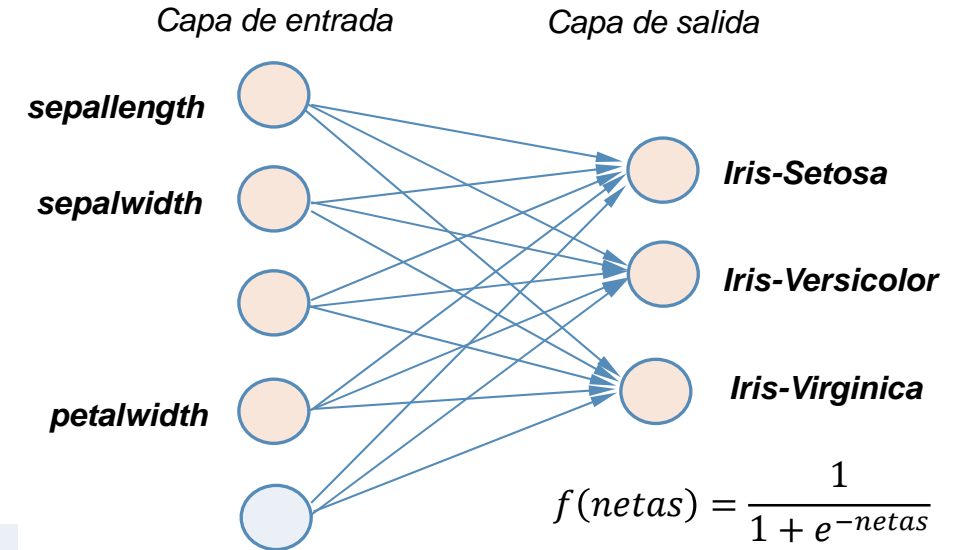
```
[[1, 0, 0],
 [0, 1, 0],
 [0, 0, 1],
 [1, 0, 0],
 [0, 0, 1]]
```

## Factores para corregir W y b

```
delta = ErrorSalida .* derivada_Fun

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

`salidas*(1-salidas)`



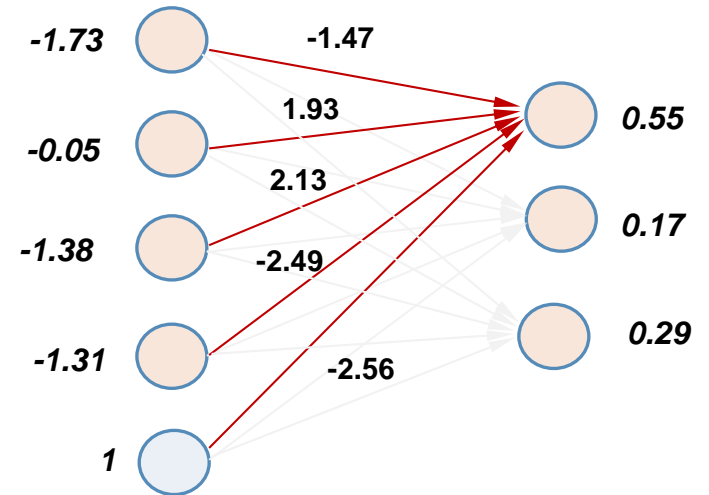
# Clasificación de flores de Iris

**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]]
```

**Y**

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



## Modificación de W y b

$$W = W + \text{alfa} * \text{delta} * X$$

$$W = \begin{bmatrix} -1.47 & 1.93 & 2.13 & -2.49 \\ 0.79 & -1.38 & 3.36 & -3.57 \\ -1.57 & -0.99 & -6.17 & 3.92 \end{bmatrix} + \text{alfa} * \begin{bmatrix} 0.1108 \\ -0.0232 \\ -0.0604 \end{bmatrix} * \begin{bmatrix} -1.73 & -0.05 & -1.38 & -1.31 \end{bmatrix} = \begin{bmatrix} -1.49 & 1.93 & 2.11 & -2.5 \\ 0.79 & -1.38 & 3.36 & -3.57 \\ -1.56 & -0.99 & -6.16 & 3.93 \end{bmatrix}$$

$$b = b + \text{alfa} * \text{delta} = \begin{bmatrix} -2.56 \\ -0.35 \\ -7.03 \end{bmatrix} + \text{alfa} * \begin{bmatrix} 0.1108 \\ -0.0232 \\ -0.0604 \end{bmatrix} = \begin{bmatrix} -2.55 \\ -0.35 \\ -7.04 \end{bmatrix}$$

# Clasificación de flores de Iris

X

Y

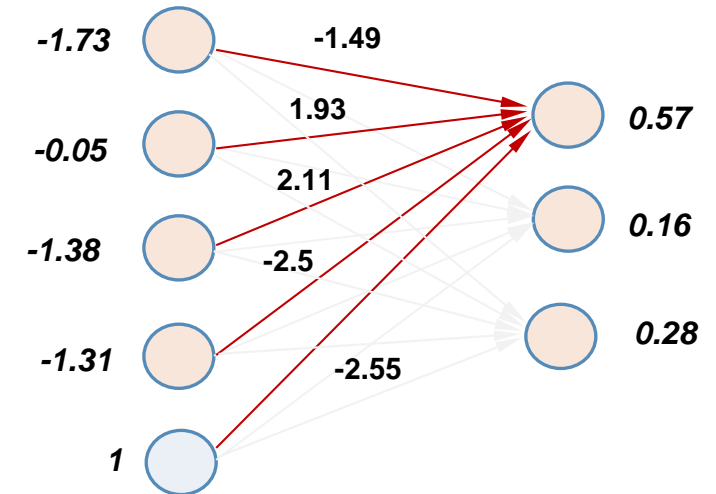
<code>[[-1.73,-0.05,-1.38,-1.31],</code>	<code>[[1,0,0],</code>	←
<code>[-0.37,-1.62, 0.22, 0.18],</code>	<code>[0,1,0],</code>	
<code>[ 1.11,-0.05, 0.93, 1.54],</code>	<code>[0,0,1],</code>	
<code>[-0.99, 0.39,-1.44,-1.31],</code>	<code>[1,0,0],</code>	
<code>[ 1.73, 1.29, 1.46, 1.81]]</code>	<code>[0,0,1]]</code>	

## Modificación de W y b

$$W = W + \text{alfa} * \text{delta} * X$$

$$W = \begin{bmatrix} -1.47 & 1.93 & 2.13 & -2.49 \\ 0.79 & -1.38 & 3.36 & -3.57 \\ -1.57 & -0.99 & -6.17 & 3.92 \end{bmatrix} + \text{alfa} * \begin{bmatrix} 0.1108 \\ -0.0232 \\ -0.0604 \end{bmatrix} * \begin{bmatrix} -1.73 & -0.05 & -1.38 & -1.31 \end{bmatrix} = \begin{bmatrix} -1.49 & 1.93 & 2.11 & -2.5 \\ 0.79 & -1.38 & 3.36 & -3.57 \\ -1.56 & -0.99 & -6.16 & 3.93 \end{bmatrix}$$

$$b = b + \text{alfa} * \text{delta} = \begin{bmatrix} -2.56 \\ -0.35 \\ -7.03 \end{bmatrix} + \text{alfa} * \begin{bmatrix} 0.1108 \\ -0.0232 \\ -0.0604 \end{bmatrix} = \begin{bmatrix} -2.55 \\ -0.35 \\ -7.04 \end{bmatrix}$$



# 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]]
```

**Y**

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

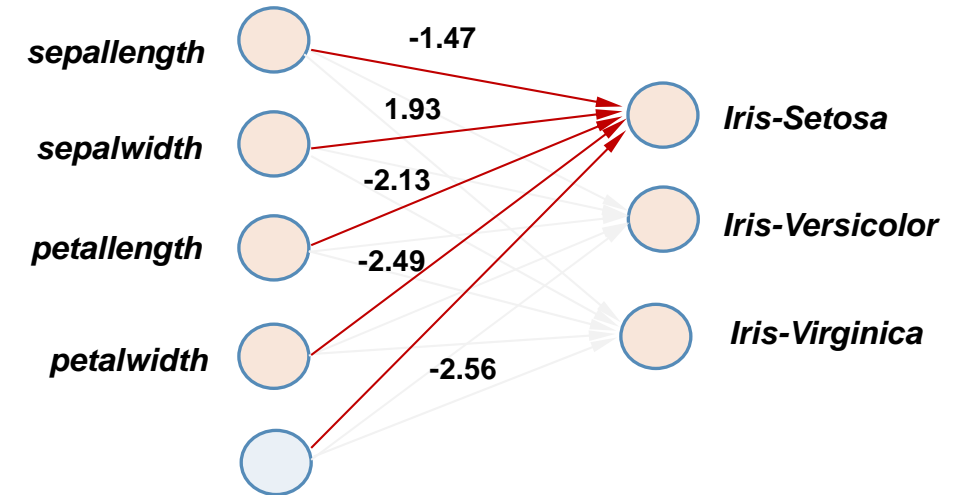
Para obtener el resultado de la red  
debe calcularse

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

siendo  $f$  la función de activación

Capa de entrada

Capa de salida



```
[ [-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]]

**Y**

[[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]]

\*



**$x^T$**

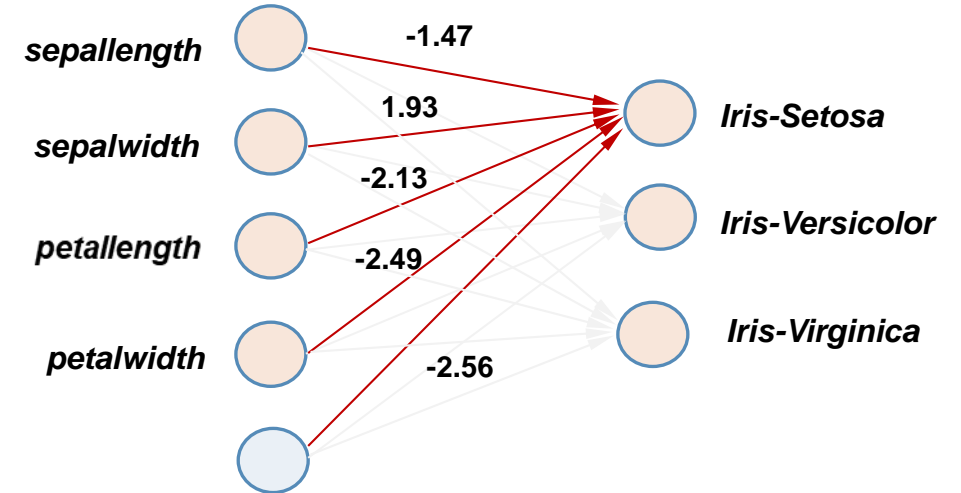
[ [-1.73],  
[-0.05],  
[-1.38],  
[-1.31]]

+

**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]]

**Y**

[ [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]]

+

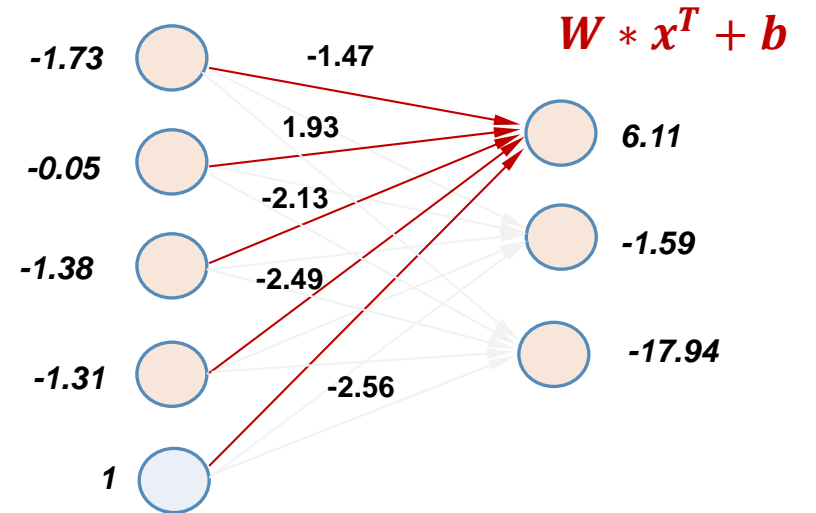
**b**

[ [-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]]
```

**Y**

```
[ [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]]
```

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

=

```
[ [ 1/(1+exp(-6.11))],
  [ 1/(1+exp( 1.59))],
  [ 1/(1+exp(17.94))]]
```

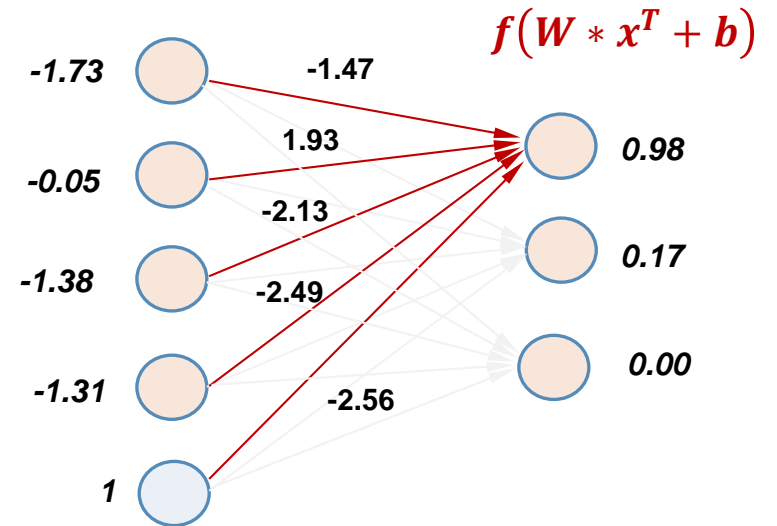
=

```
[ [ 0.98],
  [ 0.17],
  [ 0.00]]
```

Se interpreta  
como

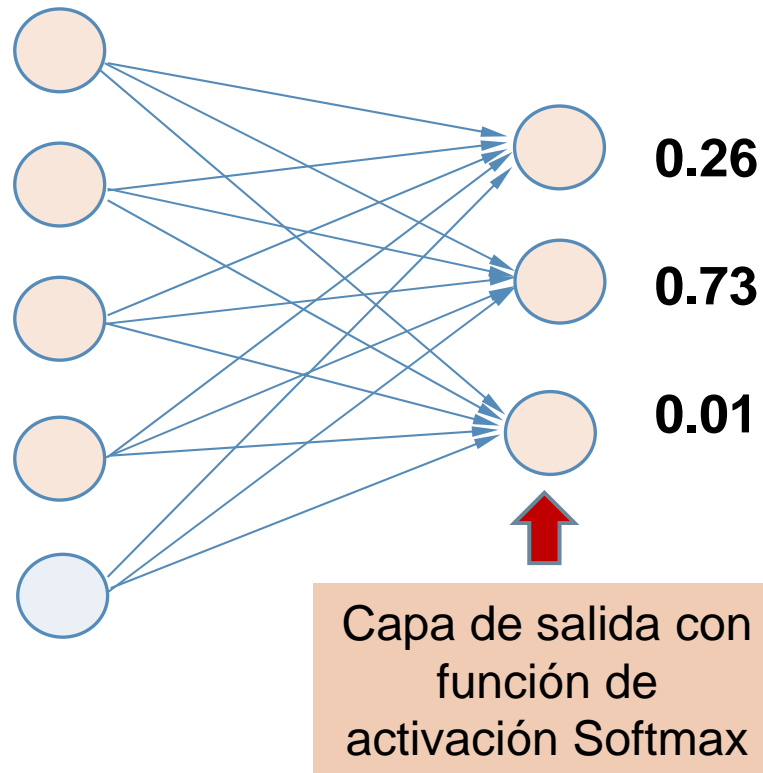


```
[ [ 1 ],
  [ 0 ],
  [ 0 ]]
```



# 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

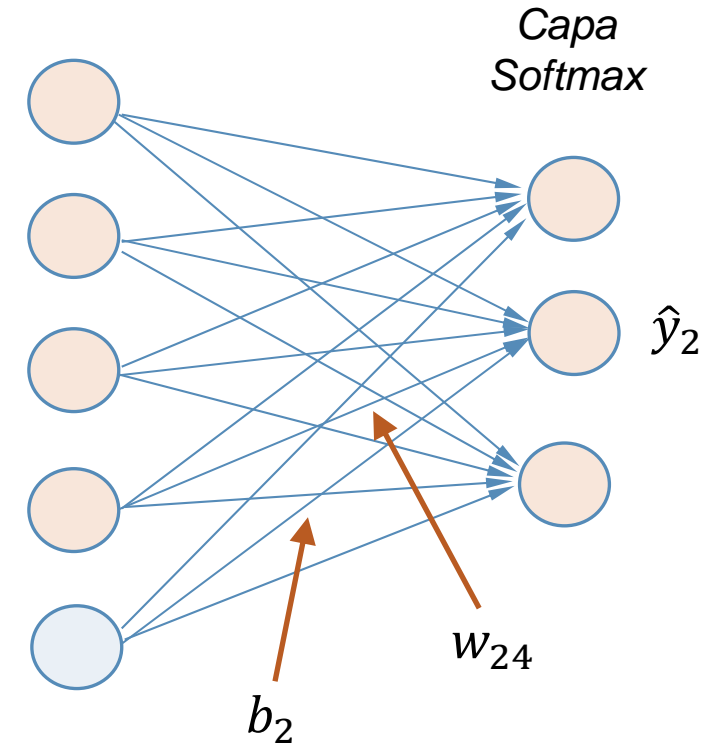
$$neta_j = \sum_i w_{ji} x_i + b_j$$

$$\hat{y}_j = \frac{e^{neta_j}}{\sum_k e^{neta_k}}$$

□ La salida de la capa es una distribución de probabilidad

□  $\hat{y}_j > 0 \quad j = 1..k$

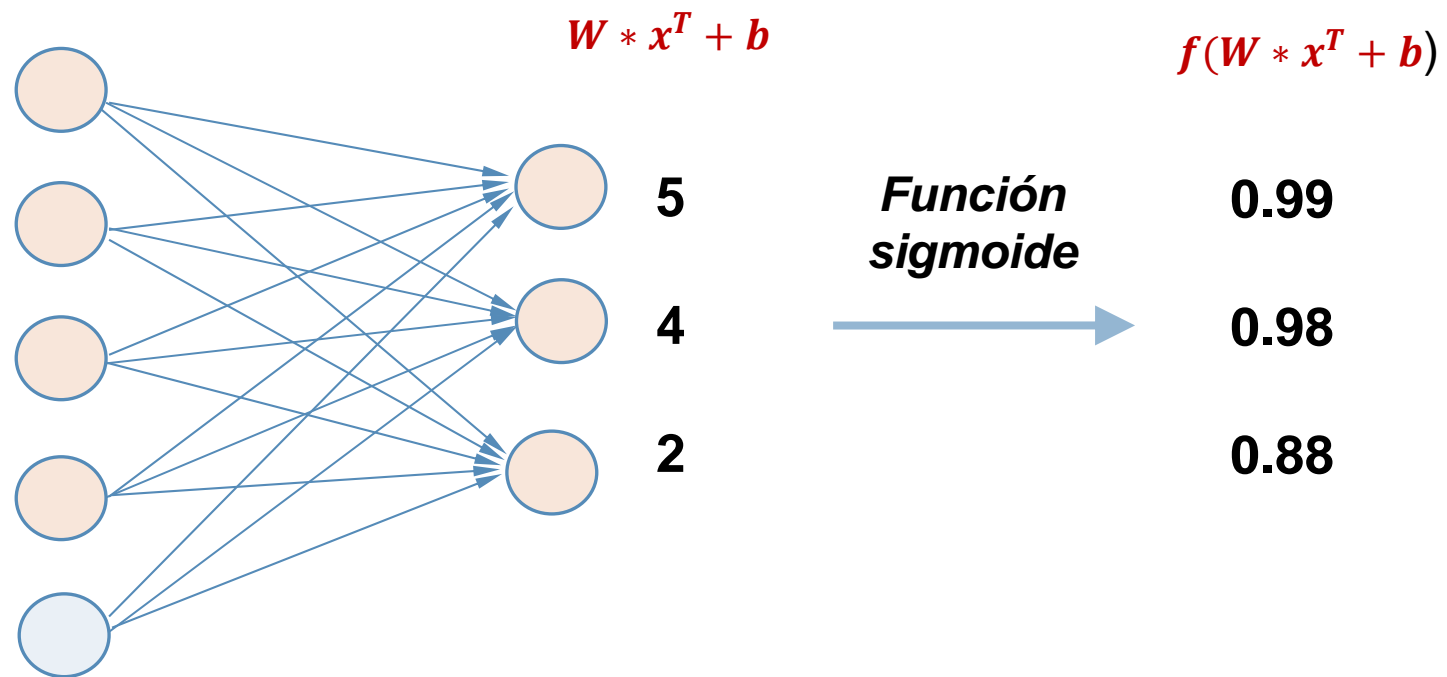
□  $\sum_j \hat{y}_j = 1$



Ver que el incremento en algún  $\hat{y}_j$  producirá disminuciones en el resto

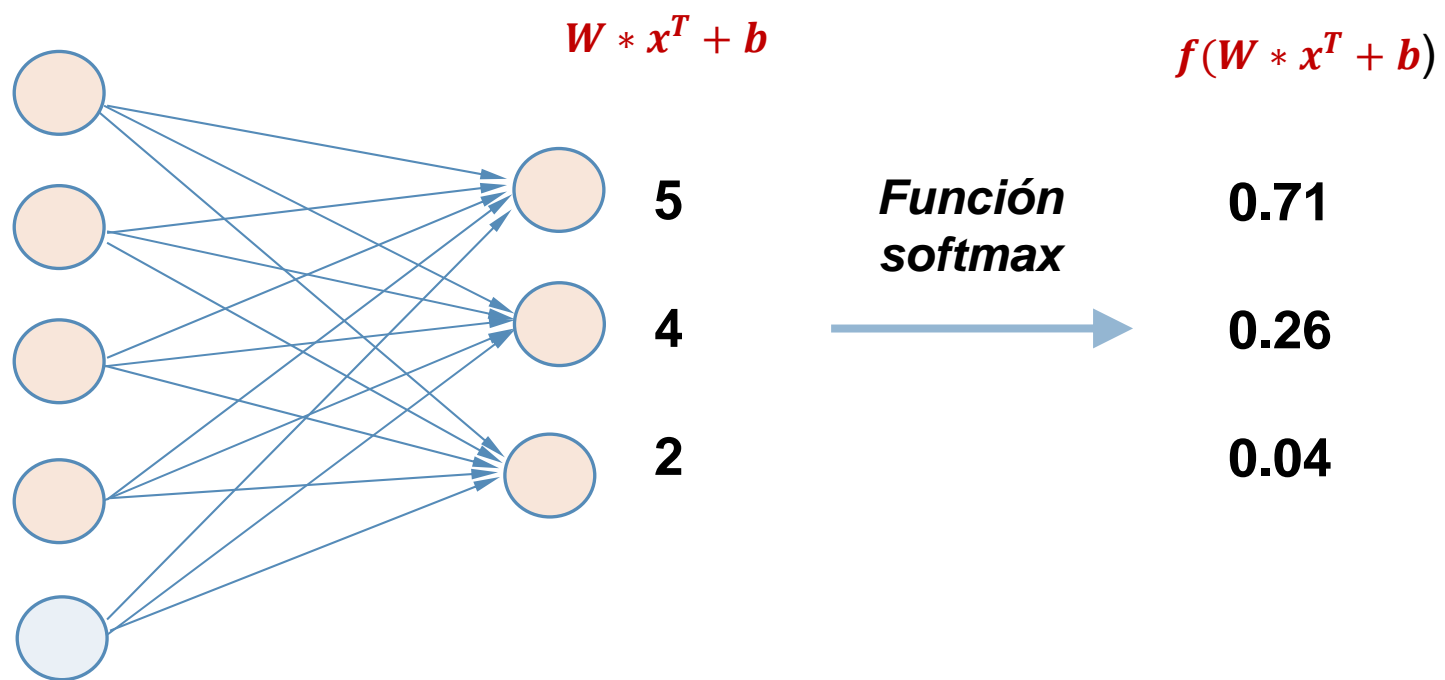
# Función Softmax

## □ Ejemplo

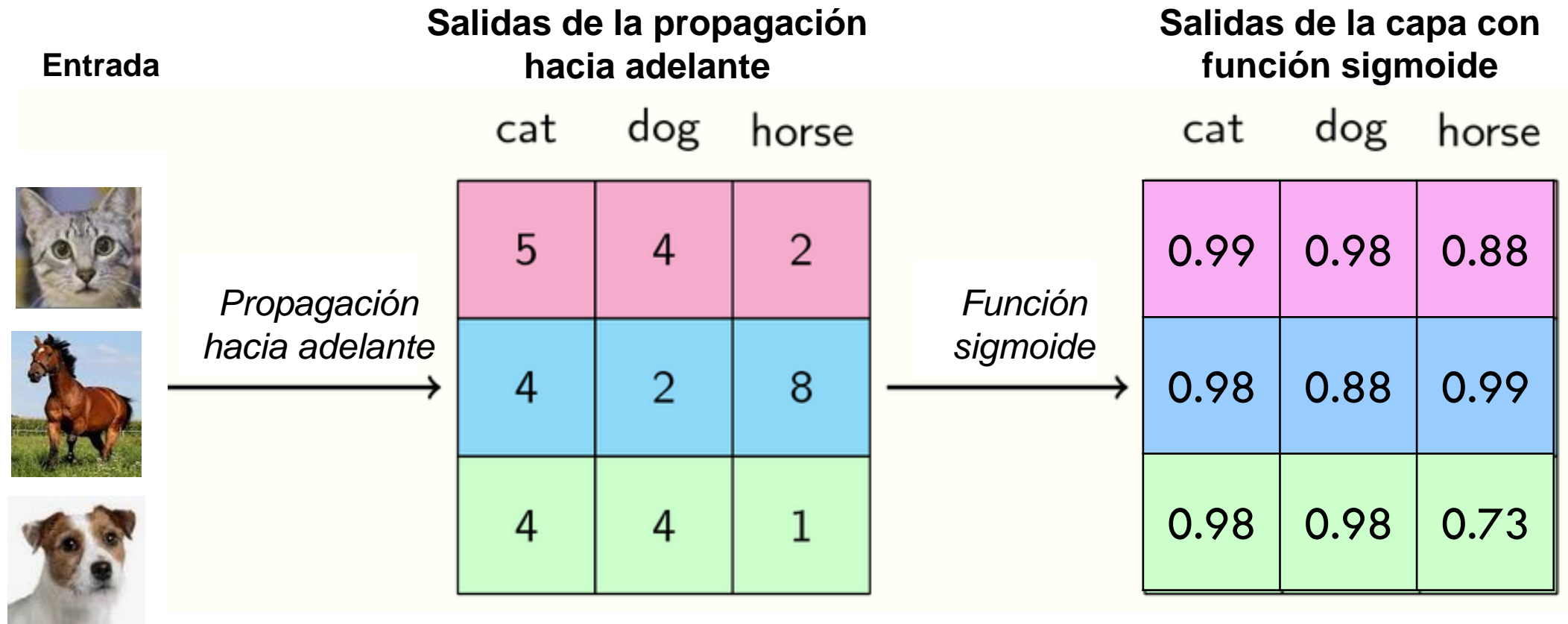


# Función Softmax

## □ Ejemplo

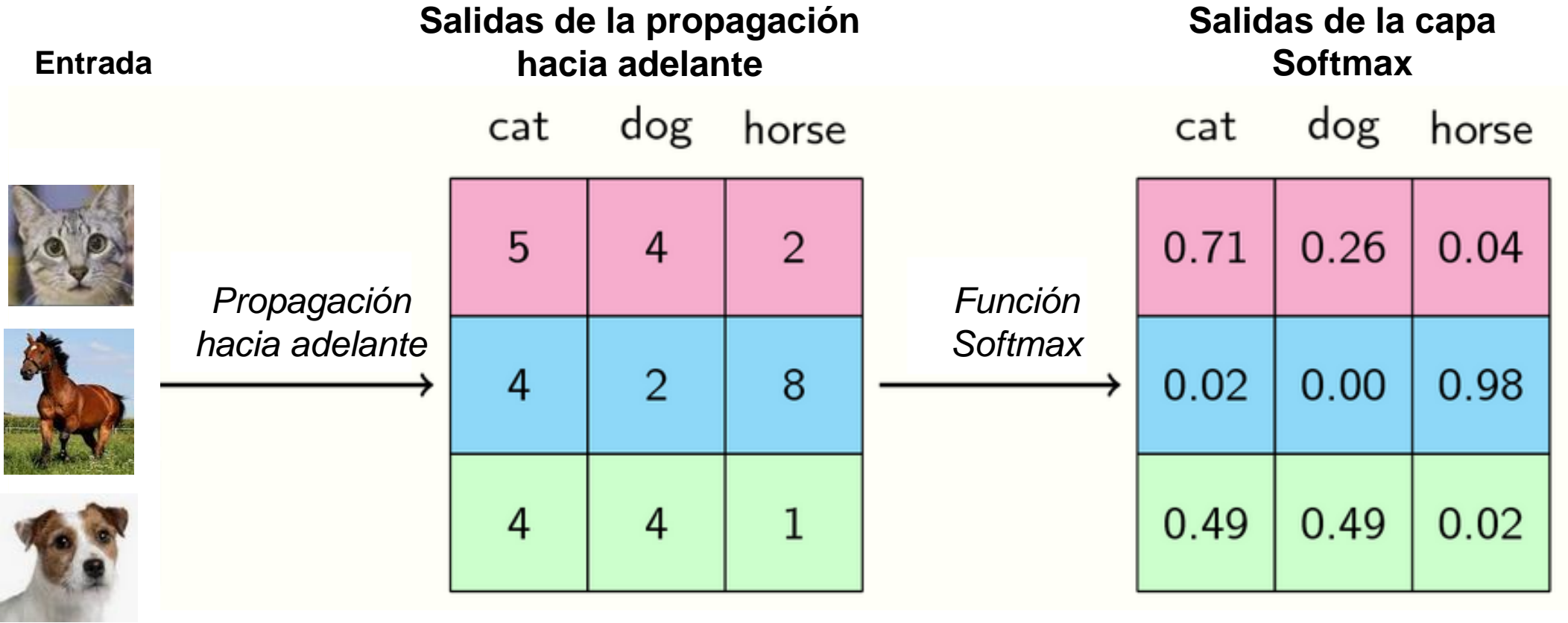


# Función softmax



# 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





# Capa Softmax

$$\hat{y}_j = \frac{e^{neta_j}}{\sum_k e^{neta_k}}$$

## □ Función de costo: Entropía cruzada categórica

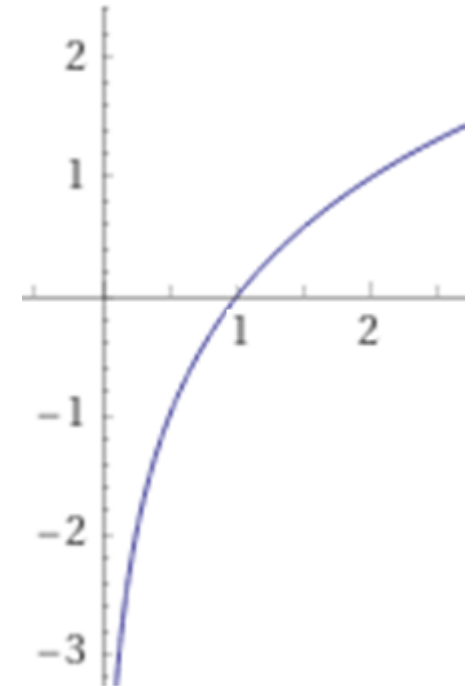
$$C = - \sum_k y_k \ln \hat{y}_k$$

donde  $y$  es un vector binario que vale 1 sólo en la posición correspondiente al valor de clase esperado.

Luego

$$C = - \ln \hat{y}_s$$

*s es la neurona correspondiente al valor de clase esperado*



# Capa Softmax

$$\hat{y}_j = \frac{e^{neta_j}}{\sum_k e^{neta_k}}$$

## □ Función de costo: Entropía cruzada categórica

$$C = -\ln \hat{y}_s$$

*S es la neurona correspondiente al valor de clase esperado*

## □ Derivada de la función de costo




$$\frac{\partial C}{\partial w_{jk}} = -(y_j - \hat{y}_j) x_k$$

$$\frac{\partial C}{\partial b_j} = -(y_j - \hat{y}_j)$$

*Coincide con la derivada de la entropía cruzada binaria*

# Capa softmax

## Función de costo: Entropía cruzada categórica

Entrada	Salida Softmax			Loss, L(a)
	cat	dog	horse	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

La clase correcta está pintada de rojo

$-\log(\hat{y}_s)$  en la clase correcta

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 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.

# ClassRNMulticlase.py

**$Y = \text{nn.predict}(X)$**

□ Parámetros de entrada

▣  $X$  : arreglo de  $N \times M$  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

***RNMulticlase\_IRIS\_RN.ipynb***

# Clasificando flores

- 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.
  
- 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

