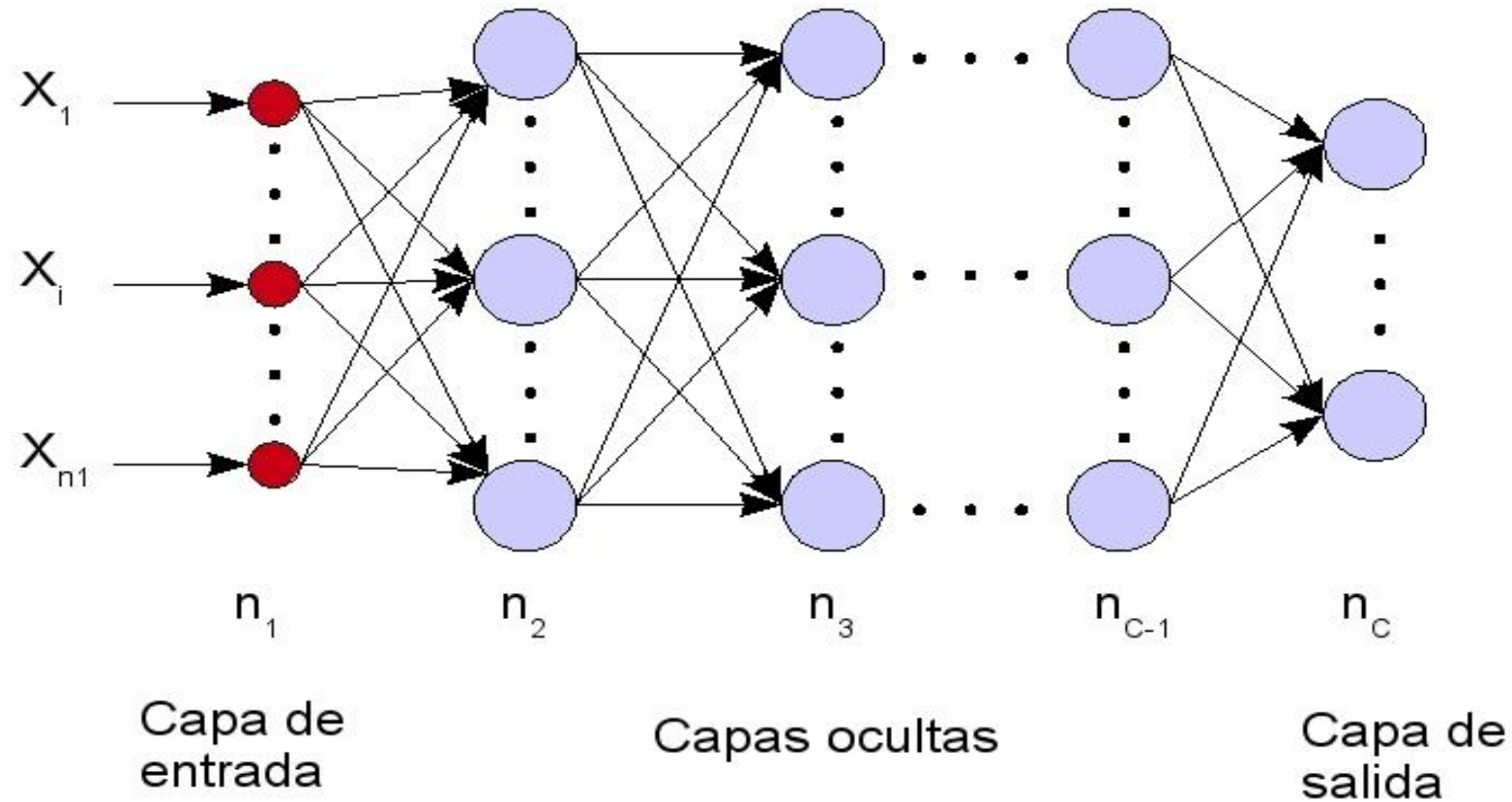


MULTIPERCEPTRÓN Y EL ALGORITMO BACKPROPAGATION

Multiperceptrón - Arquitectura

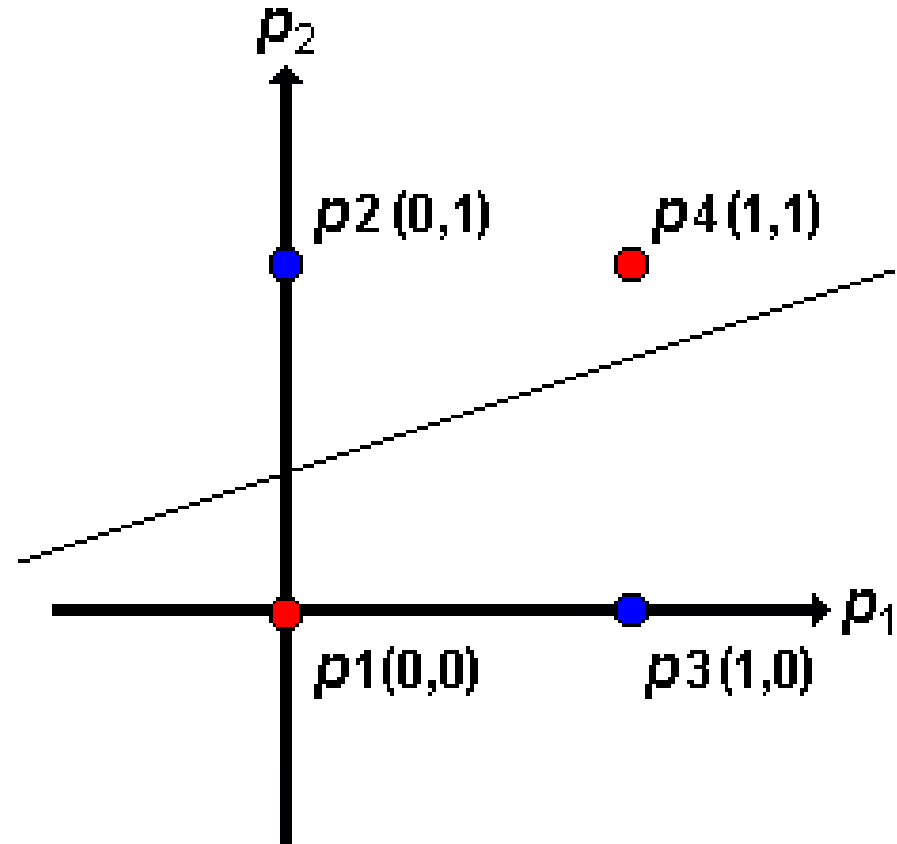
2



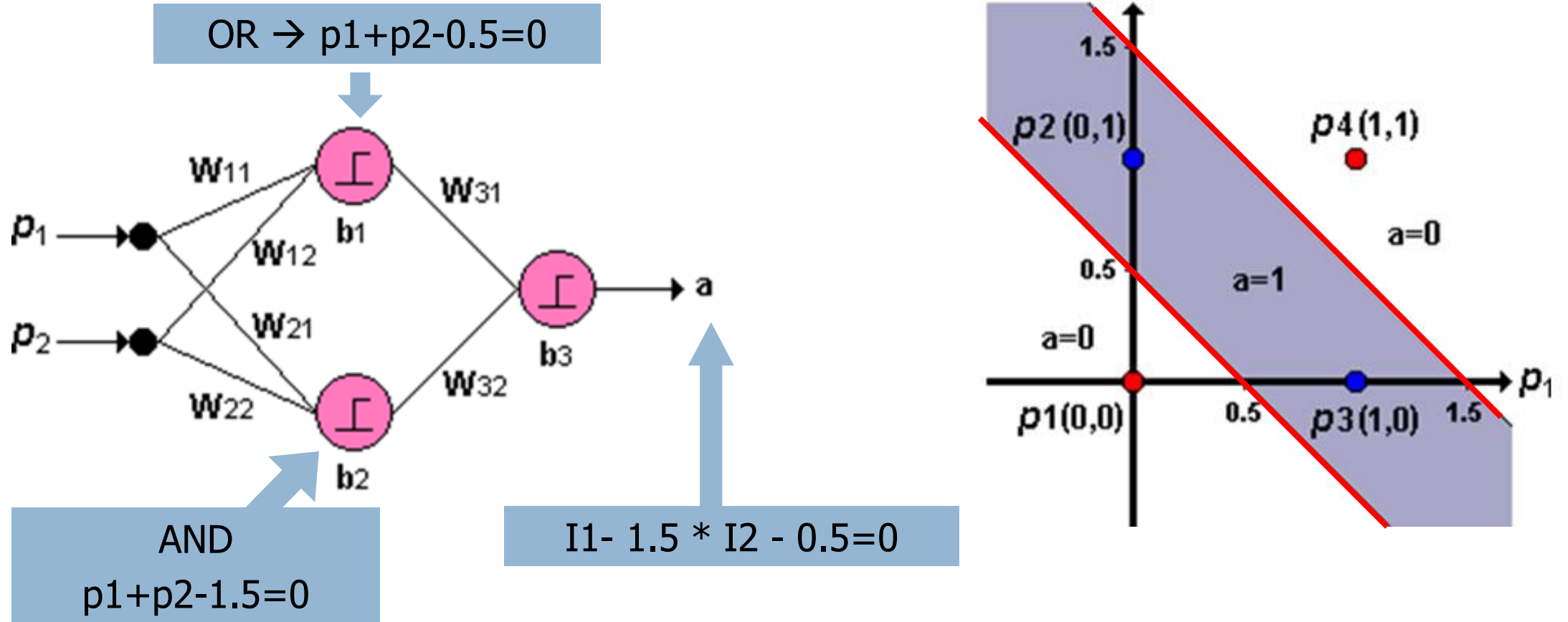
Redes multicapa

3

- Con una sola neurona no se puede resolver el problema del XOR porque no es linealmente separable.



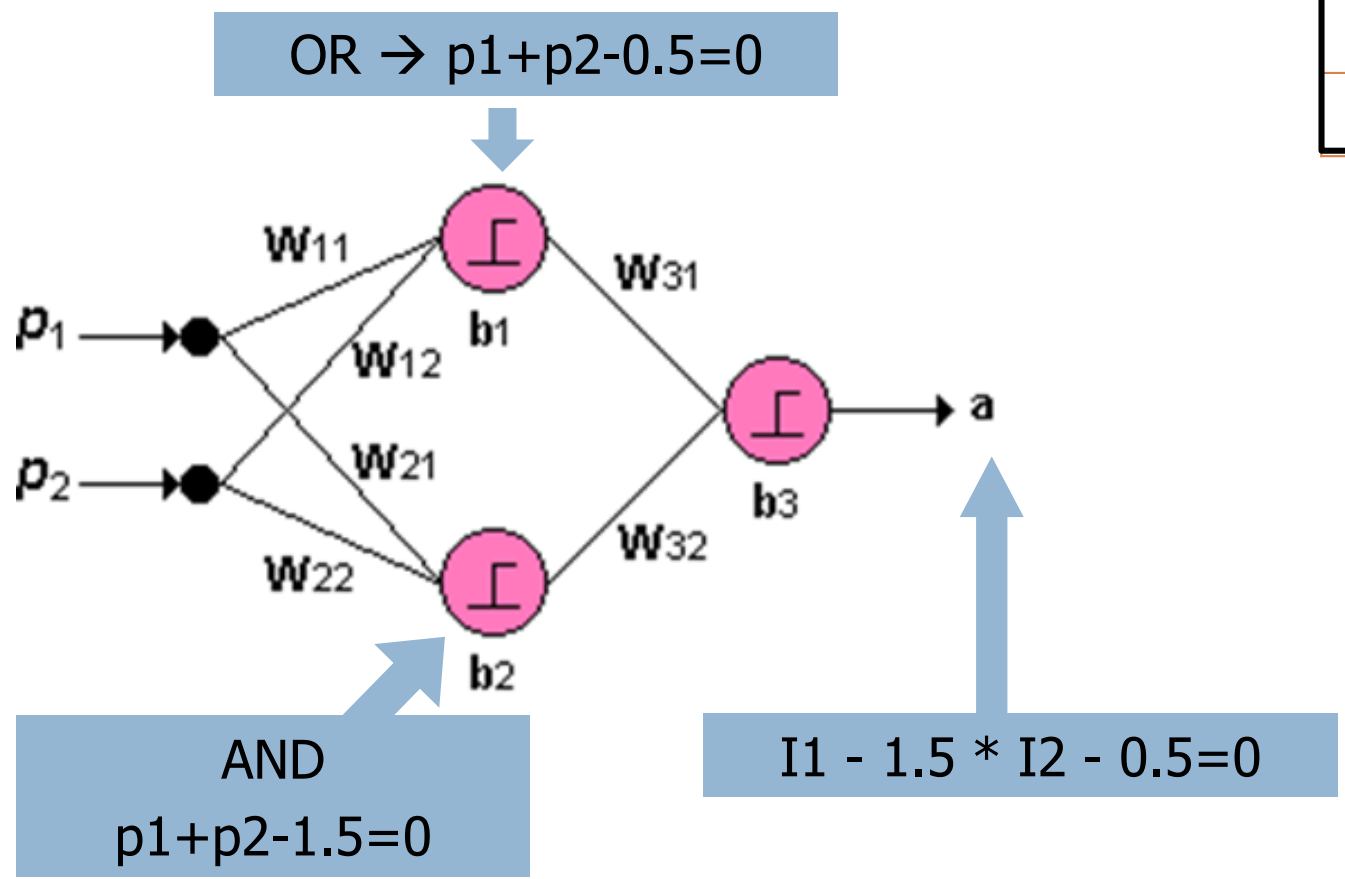
XOR



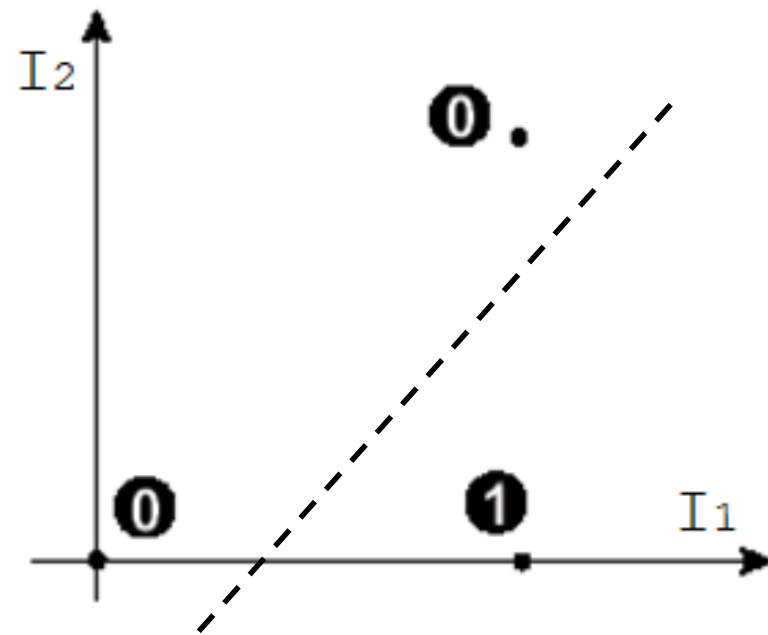
$$w_{11}=1 \quad w_{12}=1 \quad b_1=-0.5 \quad ; \quad w_{21}=1 \quad w_{22}=1 \quad b_2=-1.5 \quad ; \quad w_{31}=1 \quad w_{32}=-1.5 \quad b_3=-0.5$$

XOR

5

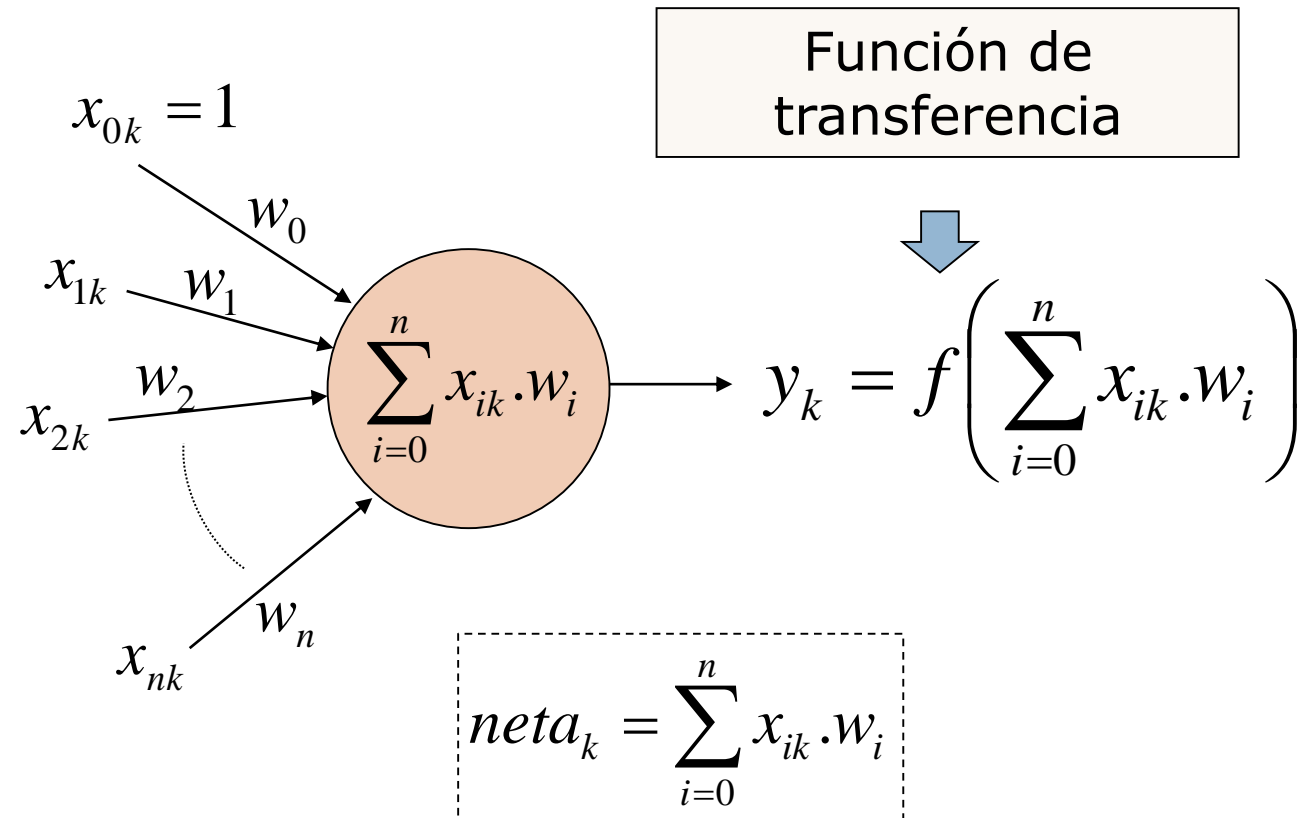


p1	p2	I1 (or)	I2 (AND)	a
1	0	1	0	1
1	1	1	1	0
0	0	0	0	0
0	1	1	0	1



Neurona artificial

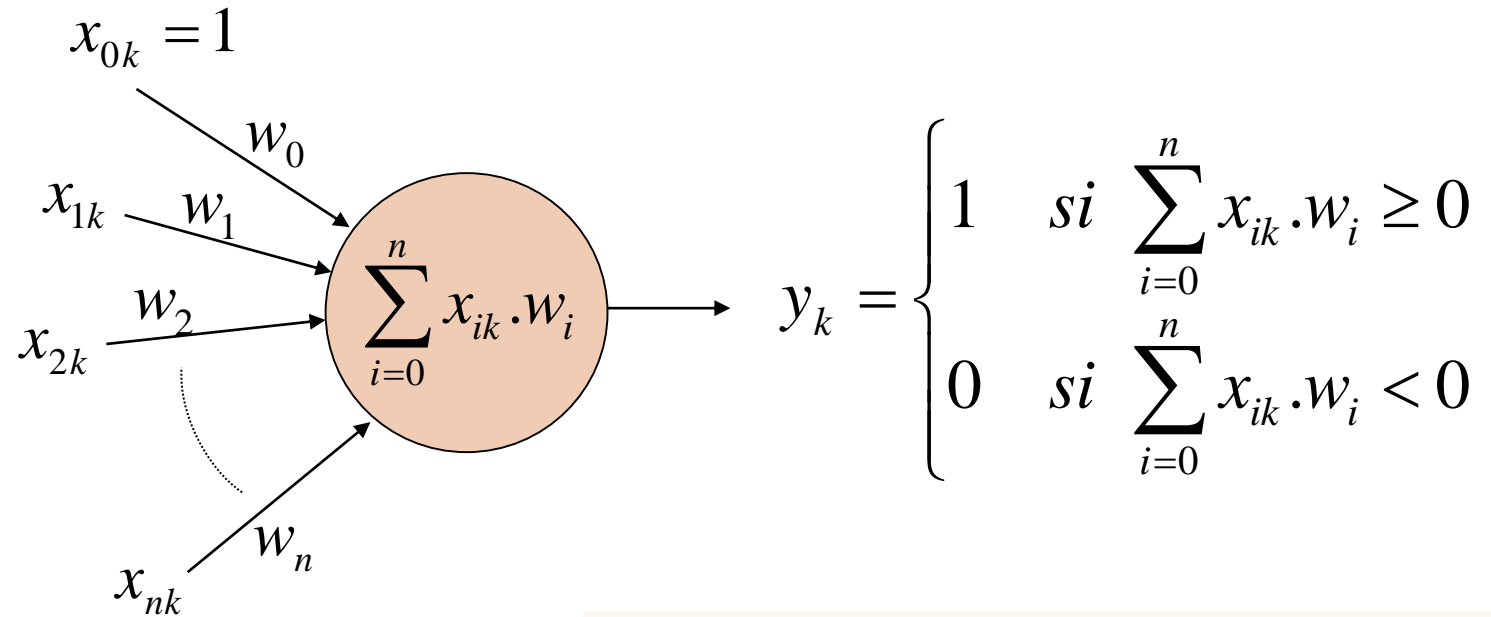
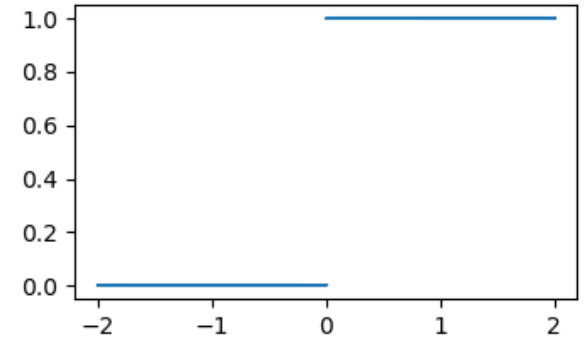
6



Perceptrón

7

$$f(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases}$$

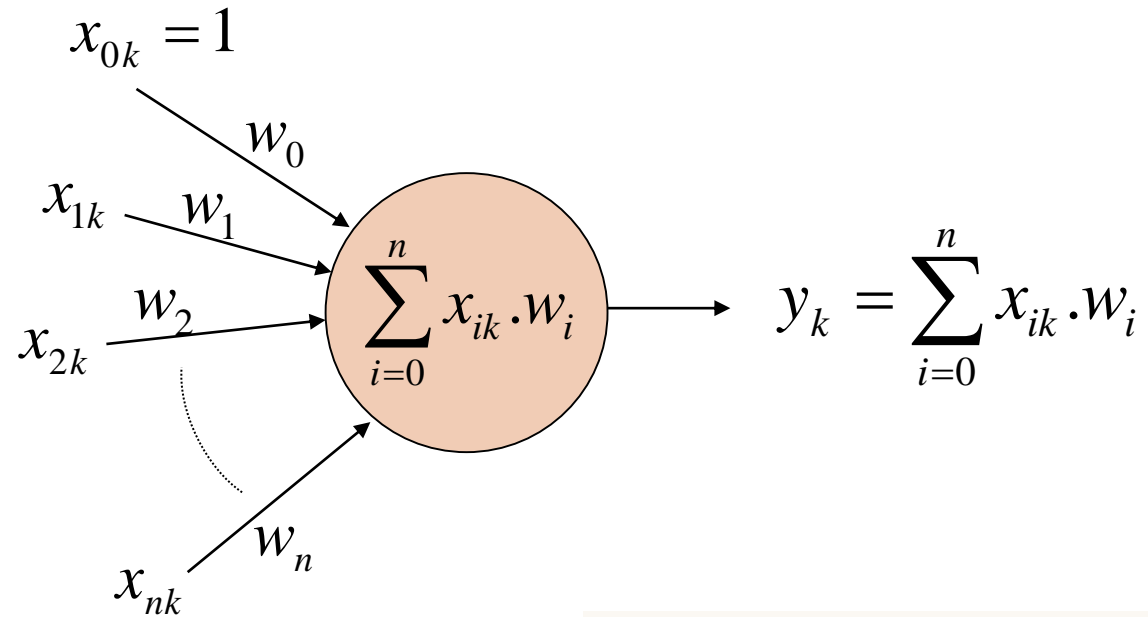
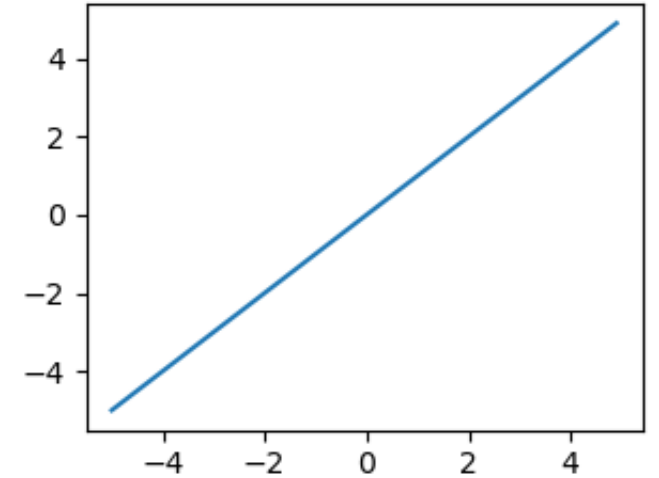


- Actualiza el vector W modificando el ángulo que forma con cada ejemplo X_k

Combinador lineal

8

$$f(x) = x$$

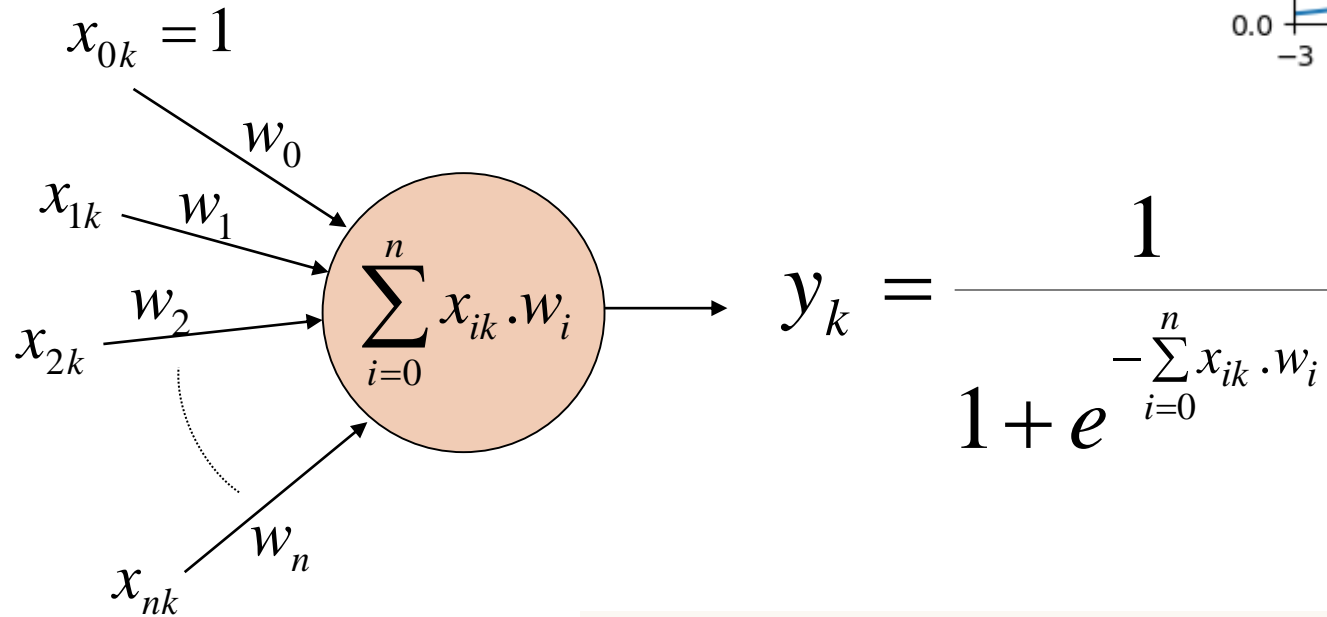
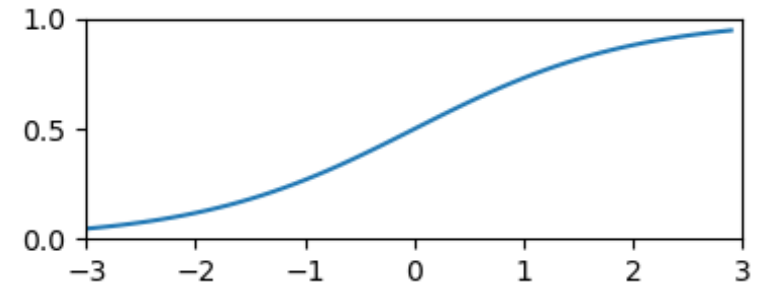


- Utiliza descenso por gradiente para actualizar el vector de pesos.

Neurona no lineal (logsig)

9

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

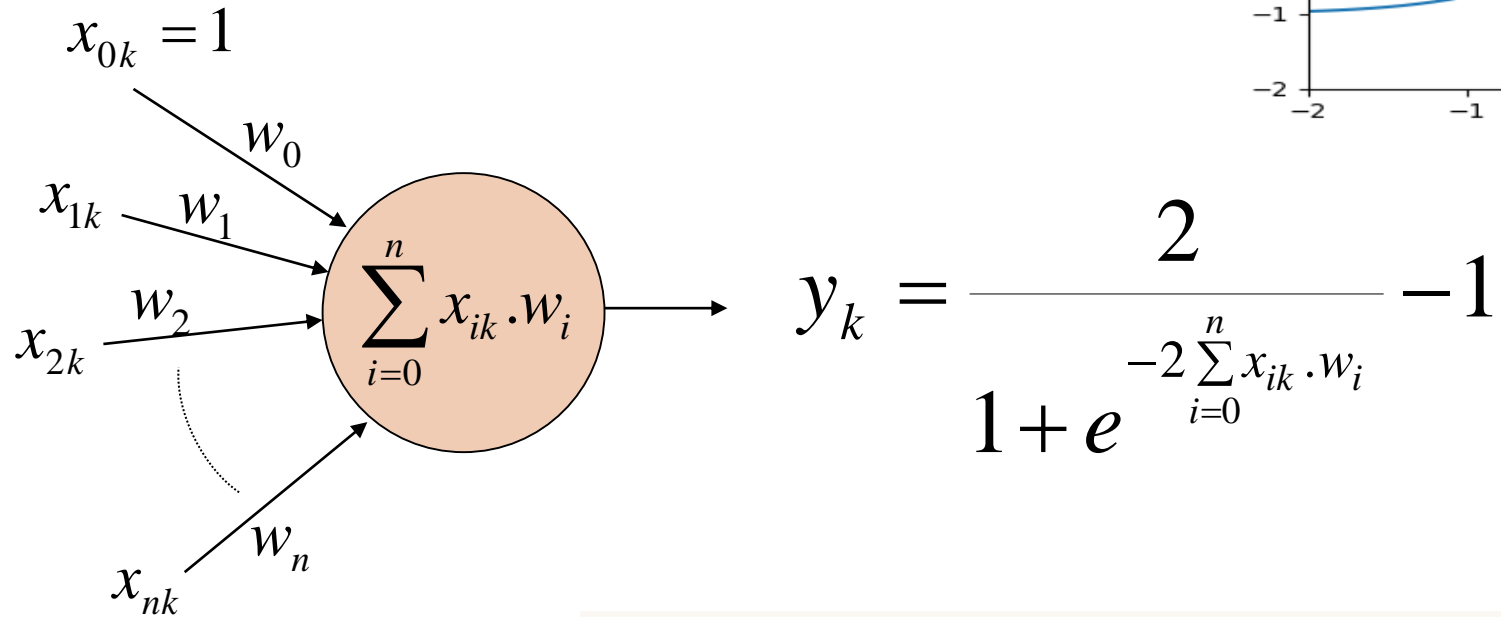
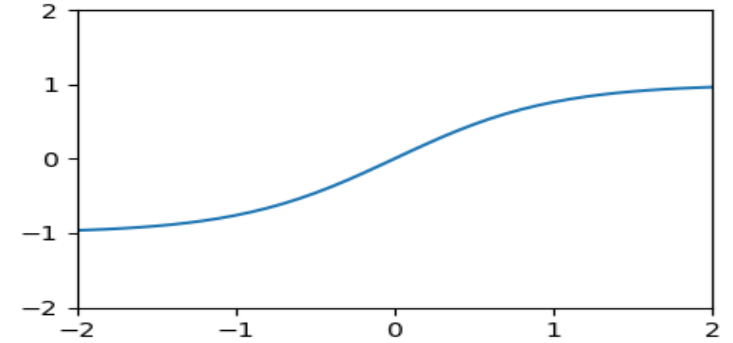


- Utiliza descenso por gradiente para actualizar el vector de pesos.

Neurona no lineal (tansig)

10

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

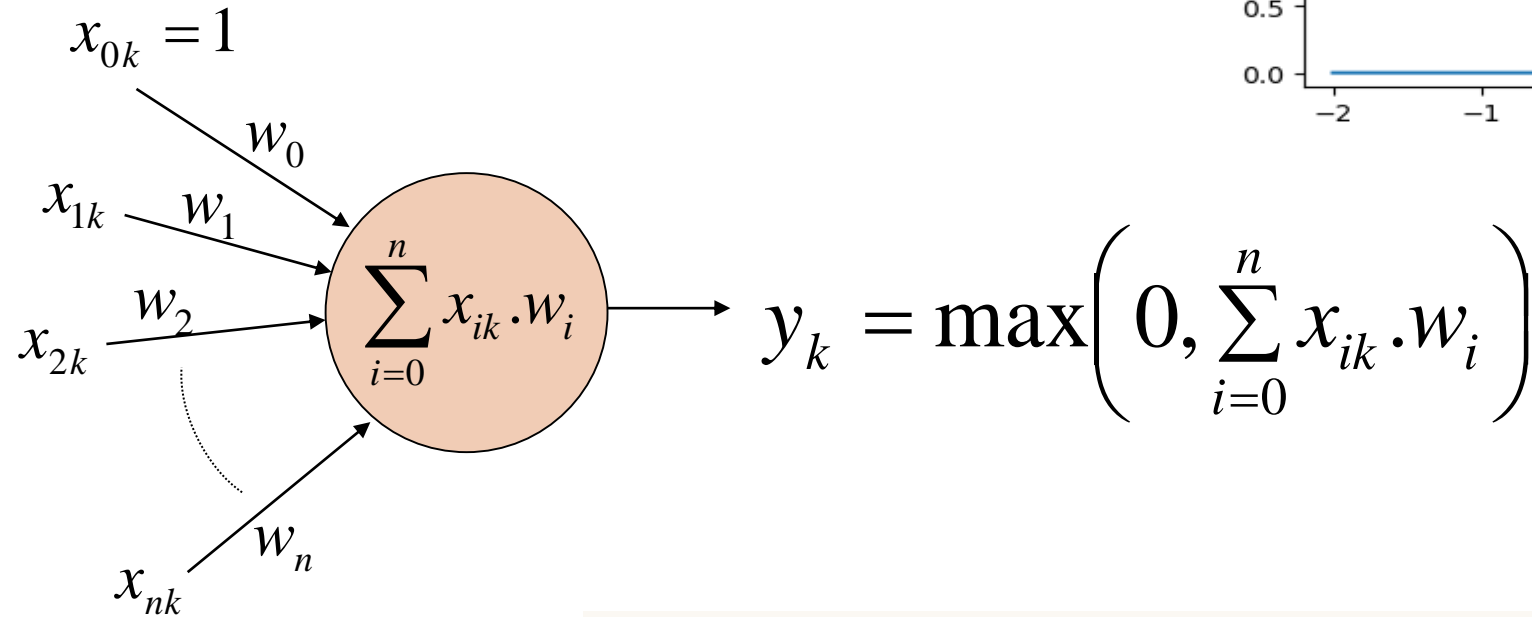
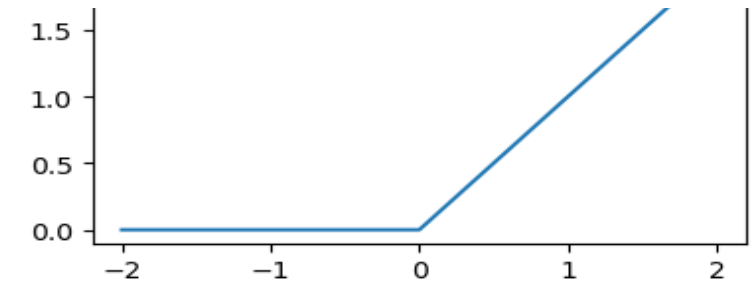


- Utiliza descenso por gradiente para actualizar el vector de pesos.

Neurona no lineal (Relu)

11

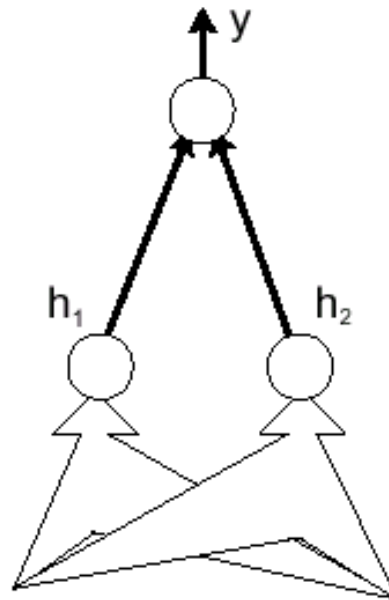
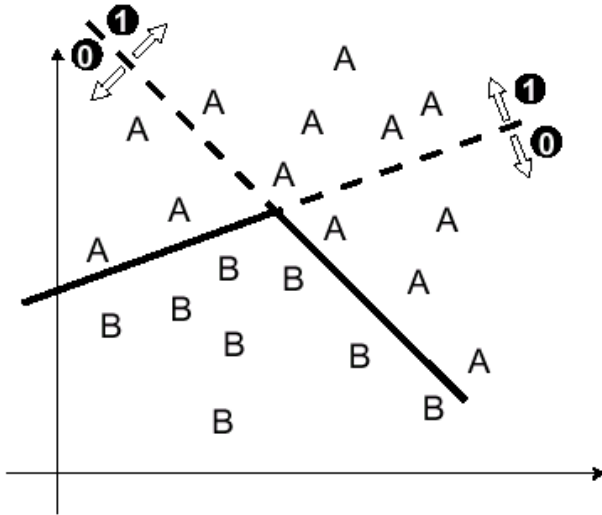
$$f(x) = \max(0, x)$$



- Utiliza descenso por gradiente para actualizar el vector de pesos.

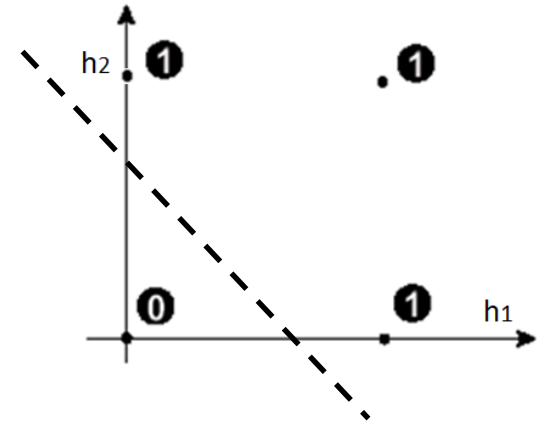
Problema no separable linealmente

12



h_1	h_2	y
0	0	0
0	1	1
1	0	1
1	1	1

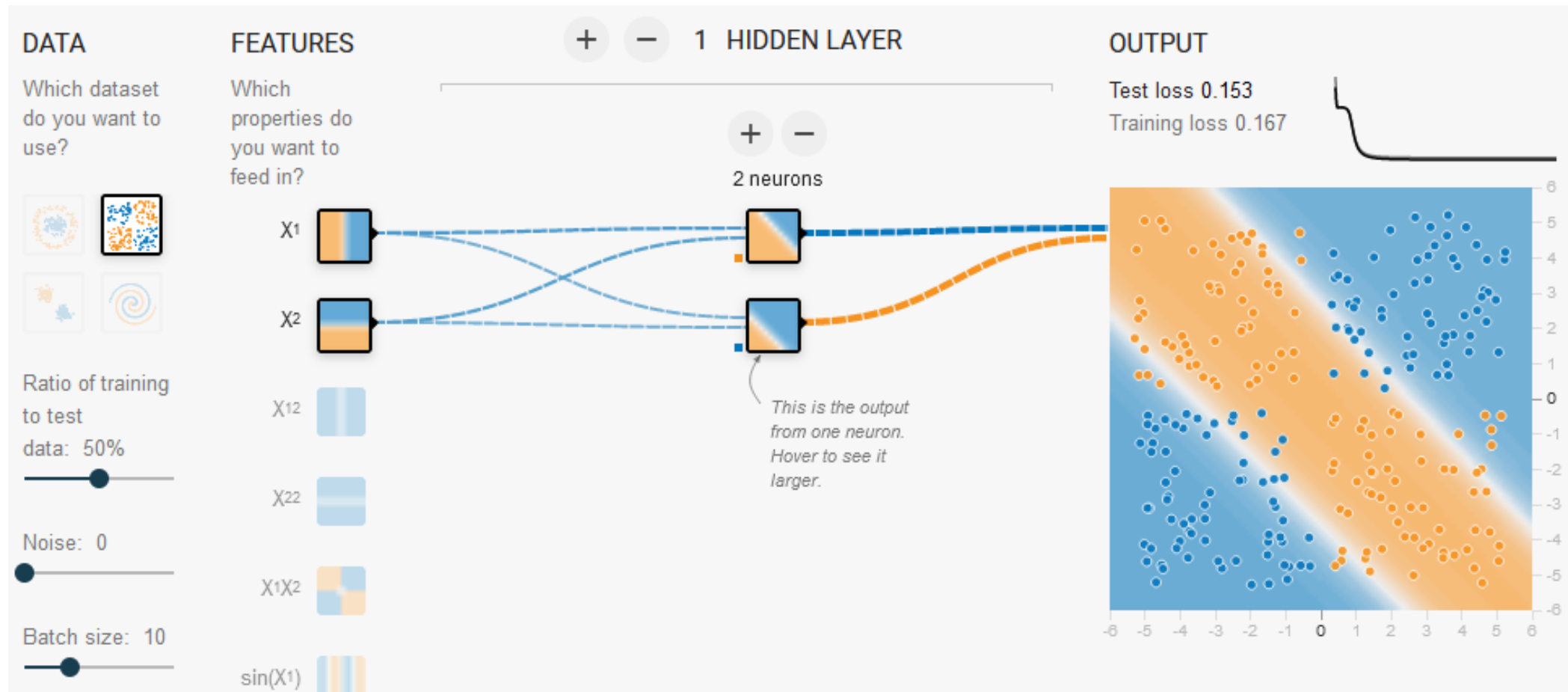
} $A \leftrightarrow 1$



- Se busca obtener un algoritmo más general que permita integrar el aprendizaje entre las dos capas.

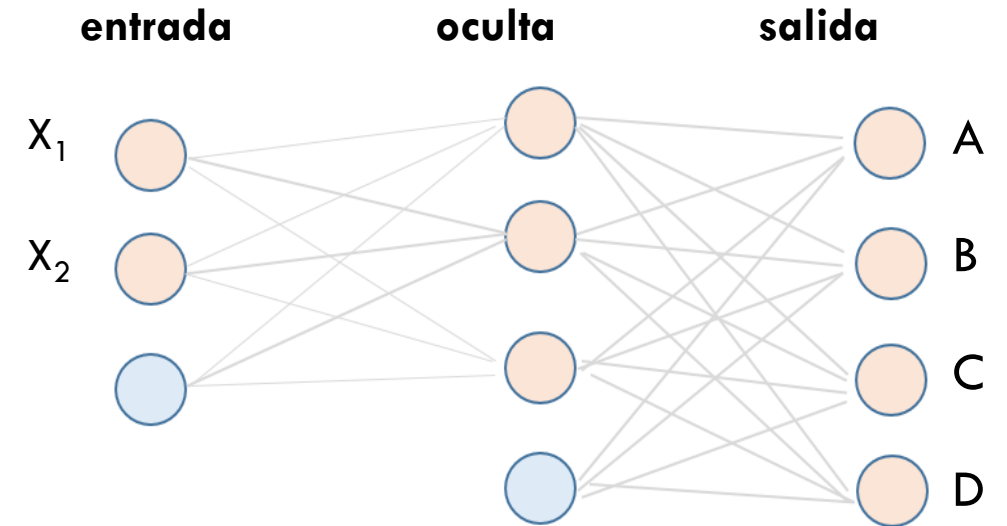
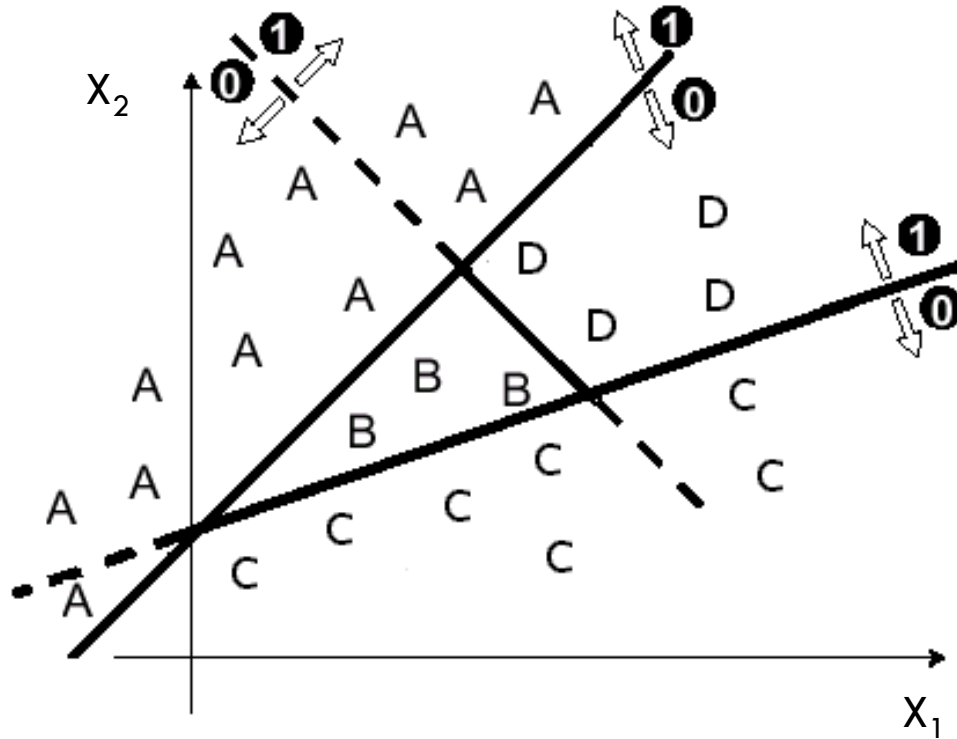
Animación de una RN

Tinker With a Neural Network Right Here in Your Browser



Problema no separable linealmente

14



□ ¿Cuál es el tamaño de cada capa?

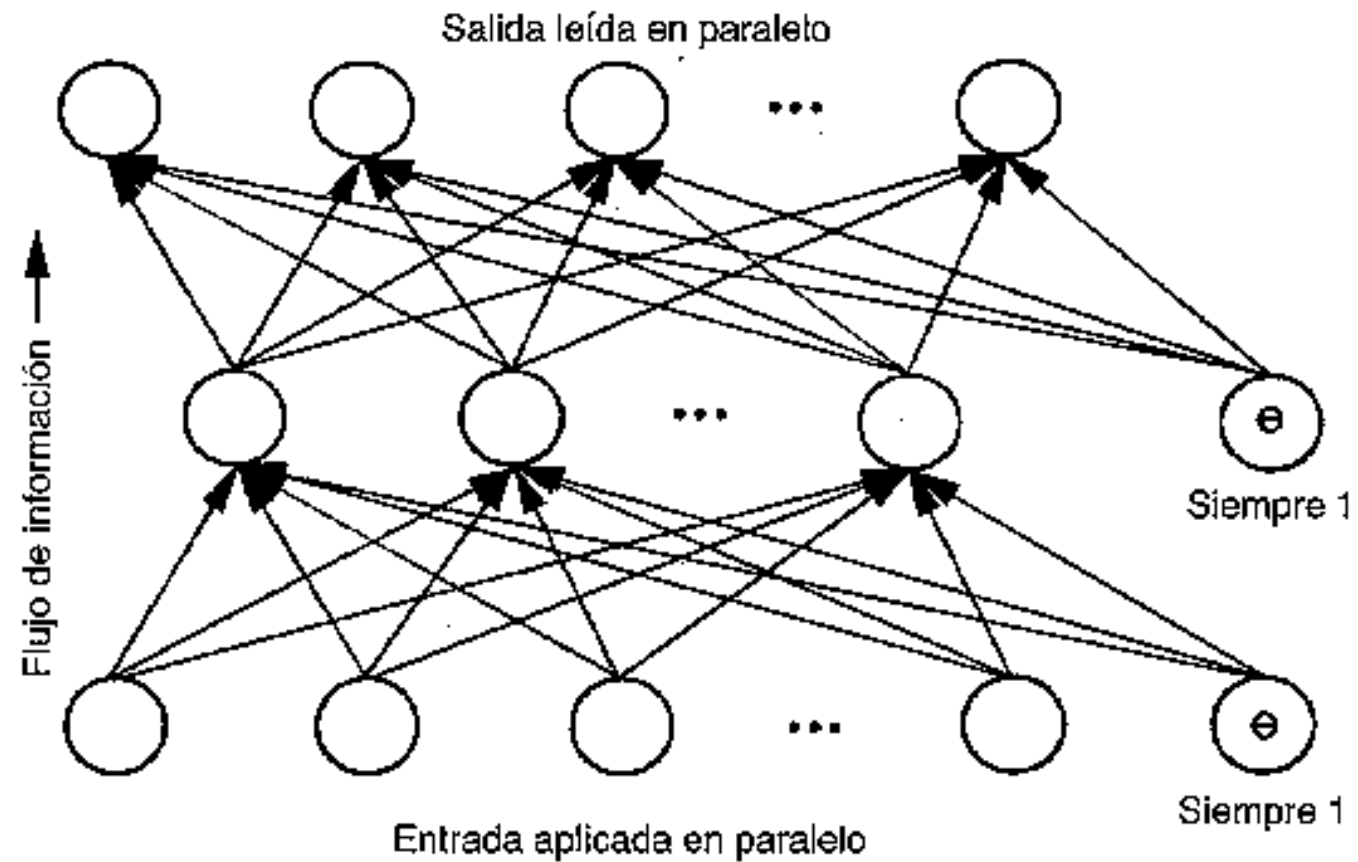
Problema no separable linealmente

15

- La idea es aplicar un descenso en la dirección del gradiente sobre la superficie de error expresada como una función de los pesos.
- Deberán tenerse en cuenta los pesos de los arcos que unen AMBAS capas.
- Dado que el aprendizaje es supervisado, para los nodos de salida se conoce la respuesta esperada a cada entrada. Por lo tanto, puede aplicarse la **regla delta** vista para el Combinador Lineal y la Neurona No Lineal.

Multiperceptrón - Arquitectura

16



Algoritmo backpropagation

17

Dado el siguiente conjunto de vectores

$$\{(x_1, y_1), \dots, (x_p, y_p)\}$$

que son ejemplos de correspondencia funcional

$$y = \phi(x) \quad x \in R^N, y \in R^M$$

se busca entrenar la red para que aprenda una aproximación

$$y' = \phi'(x)$$

Backpropagation. Capa Oculta

18

- Ejemplo de entrada

$$x_p = (x_{p1}, x_{p2}, \dots, x_{pN})^t$$

- Entrada neta de la j-ésima neurona de la capa oculta

$$neta_{pj}^h = \sum_{i=1}^n w_{ji}^h x_{pi} + \theta_j^h$$

- Salida de la j-ésima neurona de la capa oculta

$$i_{pj} = f_j^h(neta_{pj}^h)$$

Backpropagation. Capa de Salida

19

- Entrada neta de la k-ésima neurona de la capa de salida

$$neta_{pk}^o = \sum_{j=1}^L w_{kj}^o i_{pj} + \theta_k^o$$

- Salida de la k-ésima neurona de la capa de salida

$$o_{pk} = f_k^o(neta_{pk}^o)$$

Actualización de pesos

20

- Error en una sola unidad de la capa de salida

$$\delta_{pk} = (y_{pk} - o_{pk})$$

donde

- ▣ y es la salida deseada
- ▣ o es la salida real.
- ▣ p se refiere al p -ésimo vector de entrada
- ▣ k se refiere a la k -ésima unidad de salida

Actualización de pesos

21

- Se busca minimizar

$$E_p = \frac{1}{2} \sum_{k=1}^M \delta_{pk}^2$$

$$E_p = \frac{1}{2} \sum_{k=1}^M (y_{pk} - o_{pk})^2$$

se tomará el valor negativo del gradiente

Actualización de pesos

$$\frac{\partial E_p}{\partial w_{kj}^o} = -(y_{pk} - o_{pk}) \frac{\partial f_k^o}{\partial (neta_{pk}^o)} \frac{\partial (neta_{pk}^o)}{\partial w_{kj}^o}$$

$f_k^o{}'(neta_{pk}^o)$

$\frac{\partial}{\partial w_{kj}^o} \left(\sum_{j=1}^L w_{kj}^o i_{pj} + h_k^o \right) = i_{pj}$

$$\frac{\partial E_p}{\partial w_{kj}^o} = -(y_{pk} - o_{pk}) f_k^o{}'(neta_{pk}^o) i_{pj}$$

Salida de la neurona oculta j

Peso del arco que une la neurona j de la capa oculta y la neurona k de la capa de salida

Actualización de pesos

- Por lo tanto, para la capa de salida se tiene

$$\delta_{pk}^o = (y_{pk} - o_{pk}) f_k^{o'}(neta_{pk}^o)$$

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \alpha \delta_{pk}^o i_{pj}$$

Actualización de pesos

- Corrección para los pesos de los arcos entre la capa de entrada y la oculta

$$\Delta_p w_{ji}^h(t) = \alpha \delta_{pj}^h x_{pi}$$

serán de la forma:

$$\delta_{pj}^h = f_j^{h'}(neta_{pj}^h) \sum_k \delta_{pk}^o w_{kj}^o$$

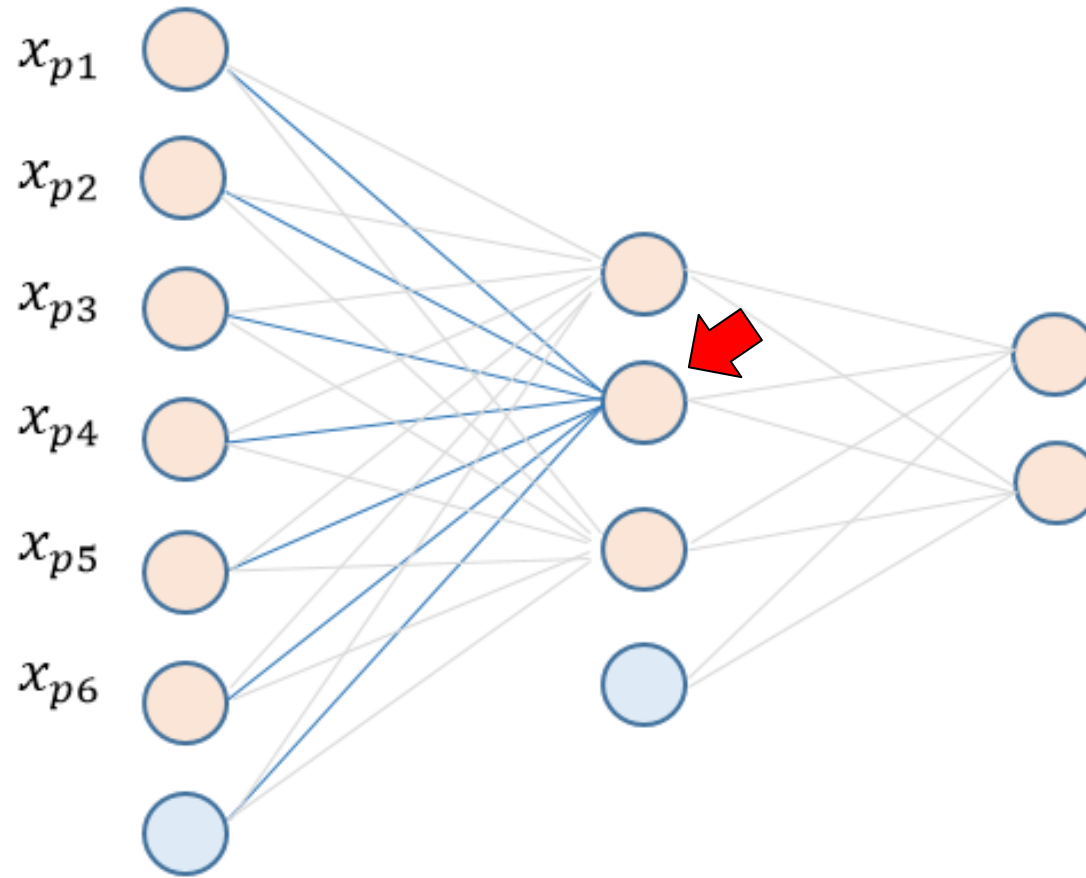
Backpropagation. Entrenamiento

25

- Aplicar un vector de entrada y calcular su salida.
- Calcular el error.
- Determinar en qué dirección (+ o -) debe cambiarse los pesos para reducir el error.
- Determinar la cantidad en que es preciso cambiar cada peso.
- Corregir los pesos de las conexiones.
- Repetir los pasos anteriores para todos los ejemplos hasta reducir el error a un valor aceptable.

□ Propagar el ejemplo de entrada a través de la capa oculta

26

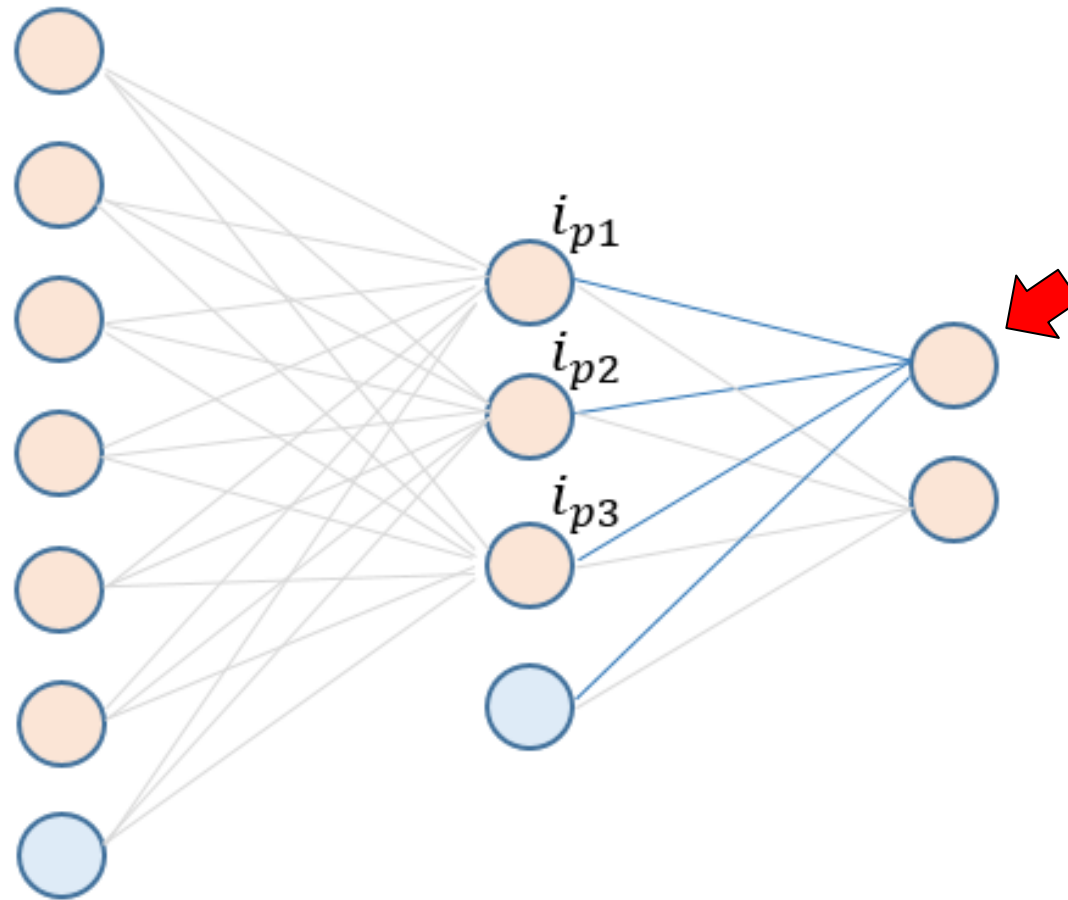


$$neta_{pj}^h = \sum_{i=1}^n w_{ji}^h x_{pi} + \theta_j^h$$

$$i_{pj} = f_j^h(neta_{pj}^h)$$

- Propagar las salidas de la capa oculta hacia la capa de salida

27

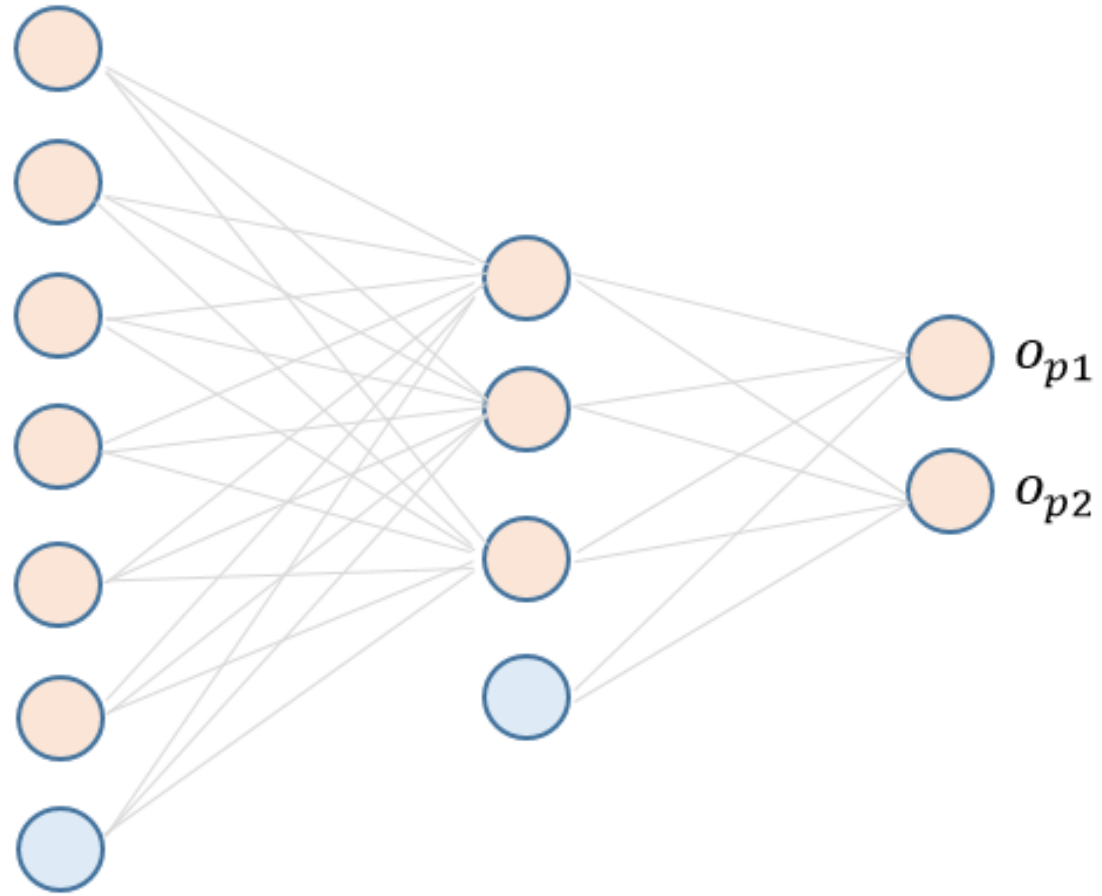


$$neta_{pk}^o = \sum_{j=1}^L w_{kj}^o i_{pj} + \theta_k^o$$

$$o_{pk} = f_k^o(neta_{pk}^o)$$

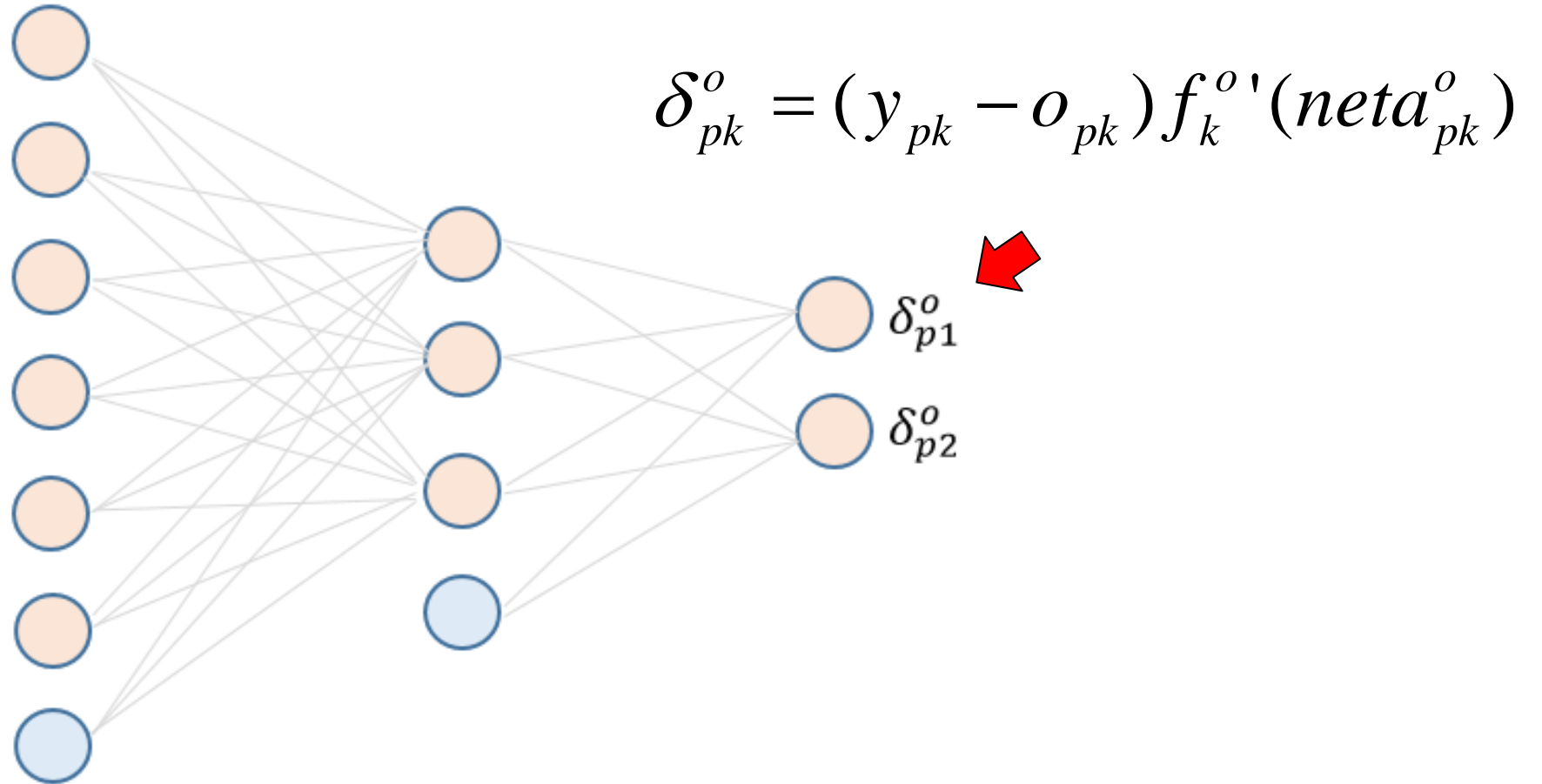
- Se obtienen los valores de salida

28



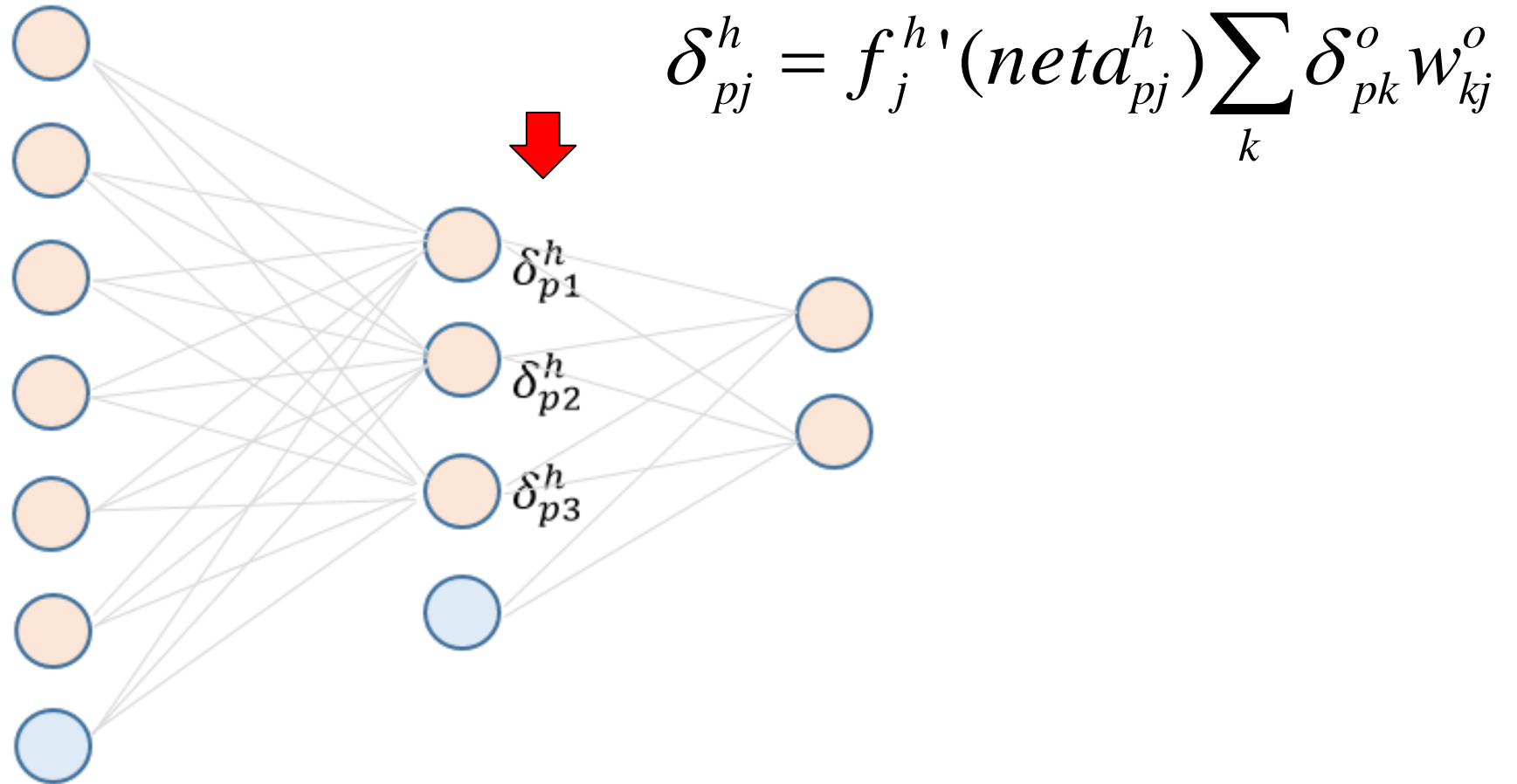
- Se calcula la corrección que se realizará al vector de pesos que llega a cada neurona de salida

29



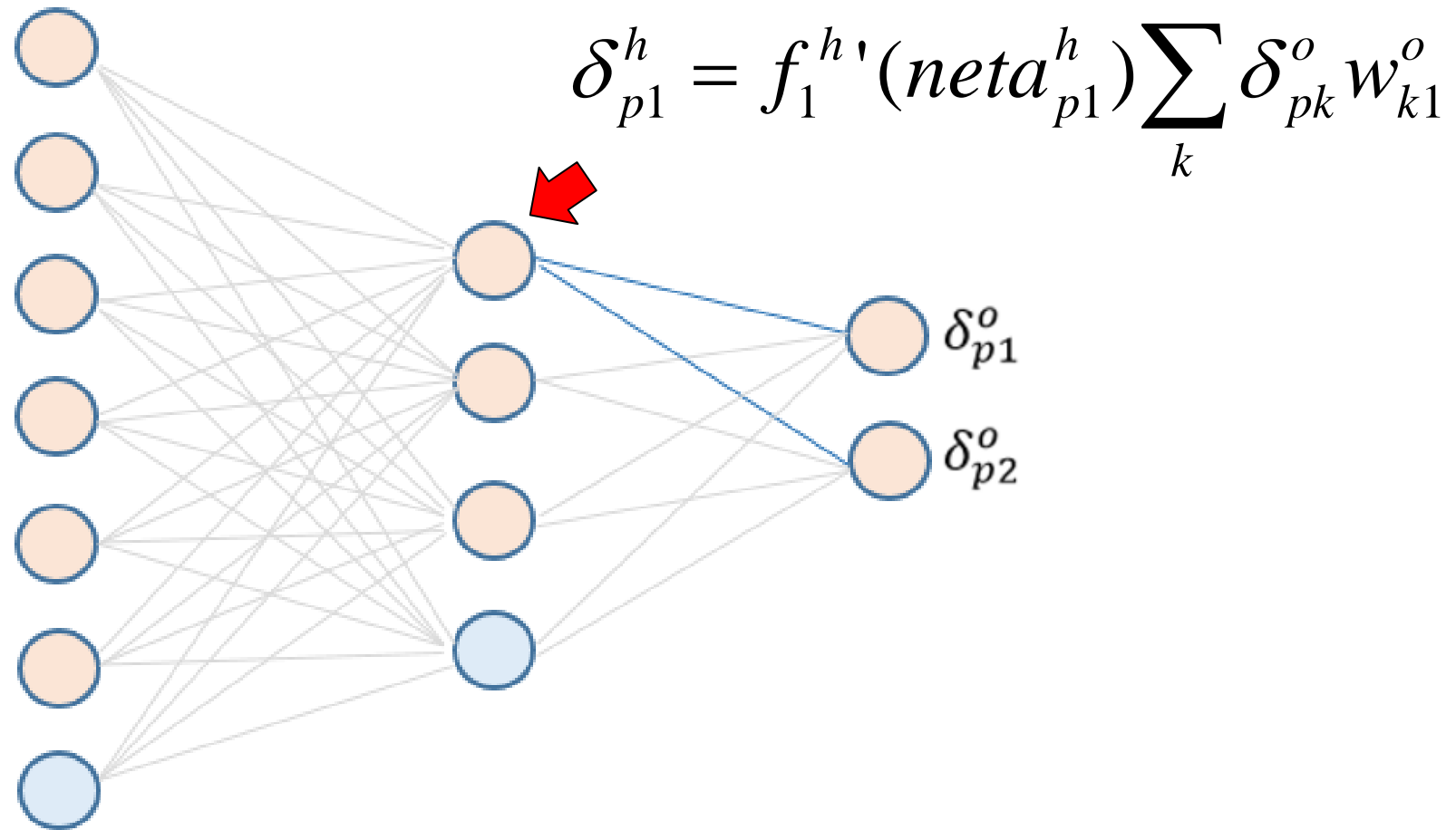
- Se calcula la corrección que se realizará al vector de pesos que llega a cada neurona oculta

30



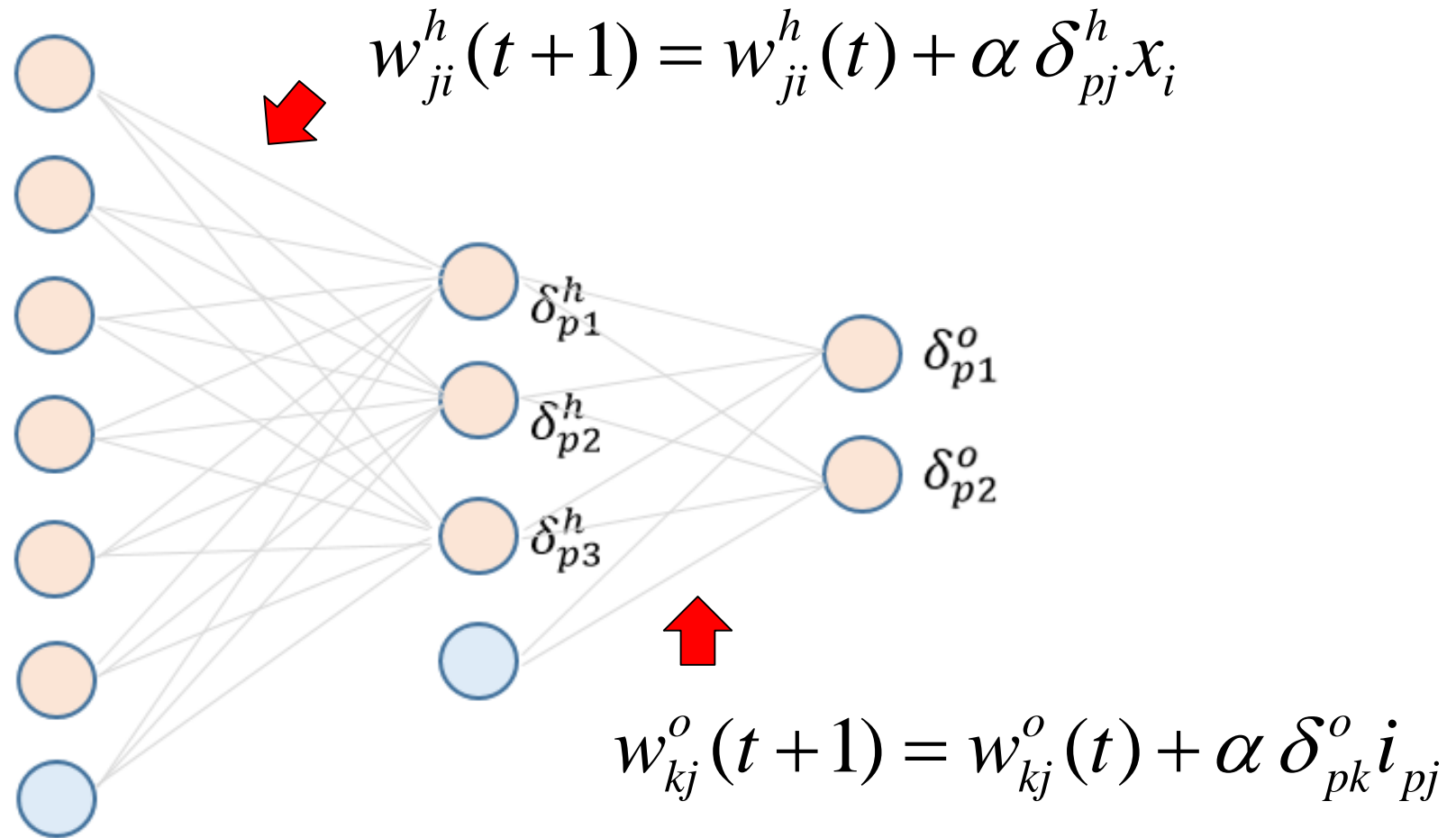
- Se calcula la corrección que se realizará al vector de pesos que llega a cada neurona de la capa oculta

31



- Se actualizan ambas matrices de pesos

33



Backpropagation. Resumen

34

- Aplicar el vector de entrada

$$x_p = (x_{p1}, x_{p2}, \dots, x_{pN})^t$$

- Calcular los valores netos de las unidades de la capa oculta

$$neta_{pj}^h = \sum_{i=1}^n w_{ji}^h x_{pi} + \theta_j^h$$

- Calcular las salidas de la capa oculta

$$i_{pj} = f_j^h(neta_{pj}^h)$$

Backpropagation. Resumen

35

- Calcular los valores netos de las unidades de la capa de salida

$$neta_{pk}^o = \sum_{j=1}^L w_{kj}^o i_{pj} + \theta_k^o$$

- Calcular las salidas

$$o_{pk} = f_k^o(neta_{pk}^o)$$

Backpropagation. Resumen

36

- Calcular los términos de error para las unidades de salida

$$\delta_{pk}^o = (y_{pk} - o_{pk}) f_k^{o'}(neta_{pk}^o)$$

- Calcular los términos de error para las unidades ocultas

$$\delta_{pj}^h = f_j^{h'}(neta_{pj}^h) \sum_k \delta_{pk}^o w_{kj}^o$$

Backpropagation. Resumen

37

- Se actualizan los pesos de la capa de salida

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \alpha \delta_{pk}^o i_{pj}$$

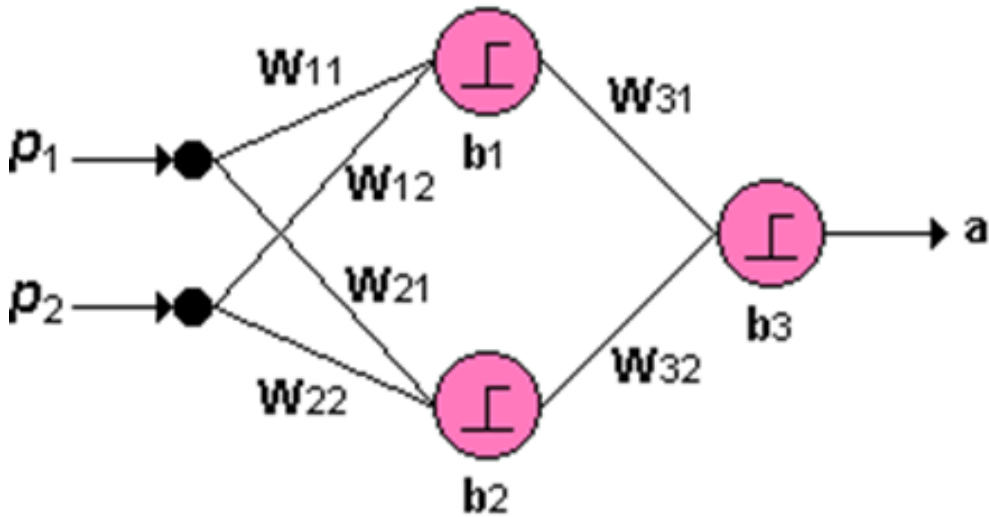
- Se actualizan los pesos de la capa oculta

$$w_{ji}^h(t+1) = w_{ji}^h(t) + \alpha \delta_{pj}^h x_i$$

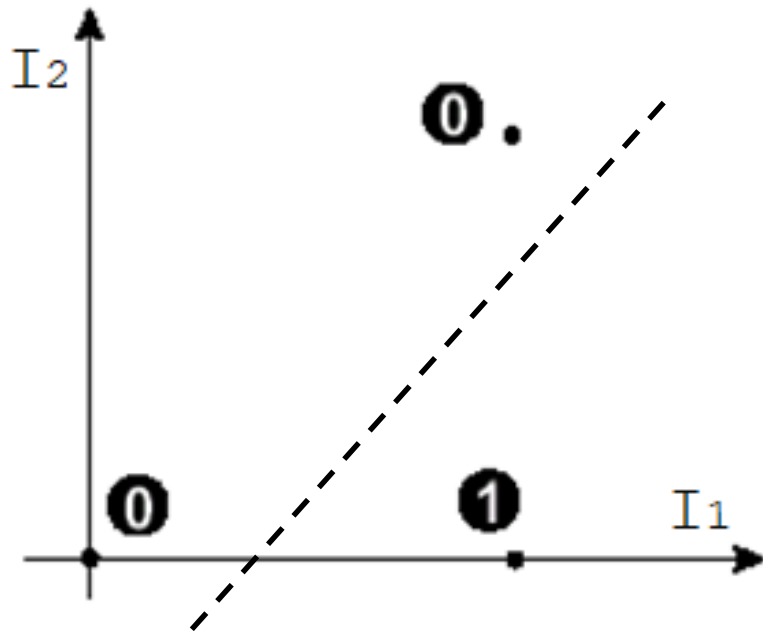
- Repetir hasta que el error resulte aceptable

XOR

38



p_1	p_2	I_1 (or)	I_2 (AND)	a
1	0	1	0	1
1	1	1	1	0
0	0	0	0	0
0	1	1	0	1



Problema del XOR

39

```
import numpy as np
from graficaMLP import dibuPtos_y_2Rectas
from Funciones import evaluar, evaluarDerivada

X = np.array([ [-1, -1], [-1, 1], [1, -1], [1, 1]])
Y = np.array([-1, 1, 1, -1]).reshape(-1,1)

entradas = X.shape[1]
ocultas = 2
salidas = Y.shape[1]
```

```
In [13]: X
```

```
Out[13]:
```

```
array([[ -1,  -1],
       [ -1,   1],
       [  1,  -1],
       [  1,   1]])
```

```
In [14]: Y
```

```
Out[14]:
```

```
array([[ 1],
       [-1],
       [-1],
       [ 1]])
```

Pesos iniciales

40

```
W1 = np.random.uniform(-0.5,0.5,[ocultas, entradas])
b1 = np.random.uniform(-0.5,0.5, [ocultas,1])
W2 = np.random.uniform(-0.5,0.5,[salidas, ocultas])
b2 = np.random.uniform(-0.5,0.5, [salidas,1])
```

```
In [17]: W1
Out[17]: array([[ -0.09705477, -0.48156505],
               [  0.14081924,  0.17185576]])

In [18]: b1
Out[18]: array([[ 0.28208473],
               [ 0.07973888]])

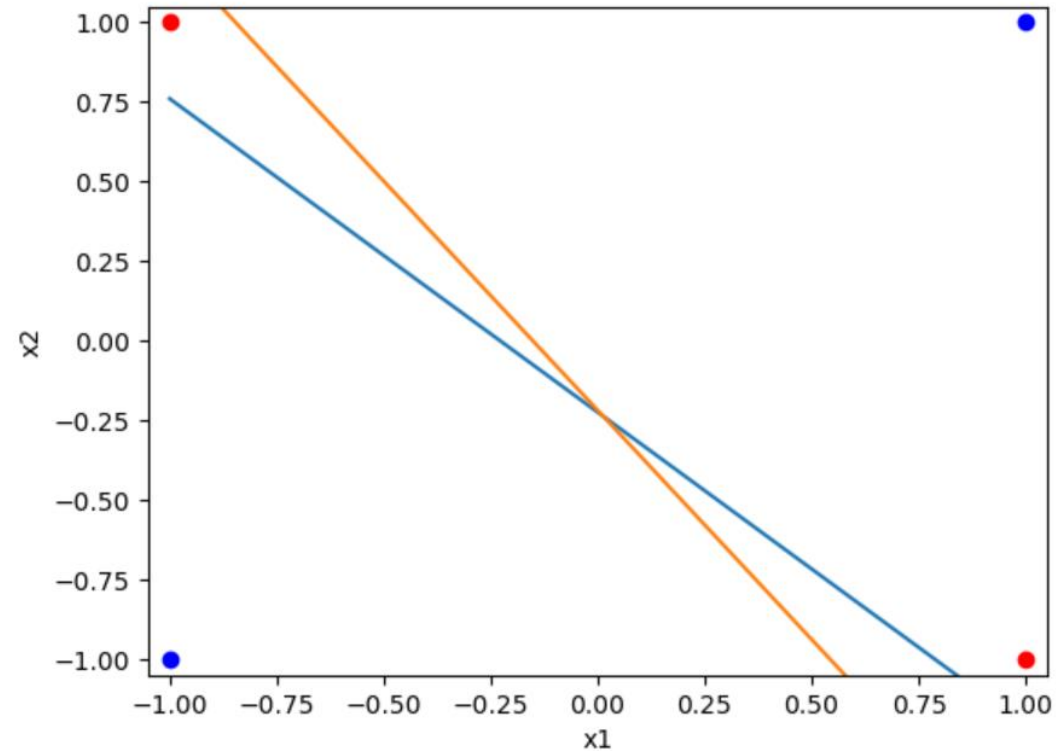
In [19]: W2
Out[19]: array([[ -0.48098277,  0.45515249]])

In [20]: b2
Out[20]: array([[ 0.15708682]])
```

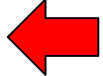
Graficar W1 y b1

41

```
ph = dibuPtos_y_2Rectas(X,Y, W1, b1, ph)
```




```
alfa = 0.15
CotaError = 0.001
MAX_ITERA = 300
ite = 0

while ( abs(ErrorAVG-ErrorAnt)>Cota) and ( ite < MAX_ITERA ):
    for p in range(len(P)):    #para cada ejemplo
        # propagar el ejemplo hacia adelante 
        # calcular los errores en ambas capas
        # corregir los todos los pesos

    # Recalcular AVGError
    ite = ite + 1
    print(ite, AVGError)
# Graficar las rectas
```

Calcular la salida para el ejemplo p

funciones de activación de cada capa

FunH = 'sigmoid'

FunO = 'tanh'

propagar el ejemplo hacia adelante

netasH = W1 @ X[p:p+1,:].T + b1

salidasH = evaluar(FunH, netasH)

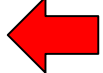
netasO = W2 @ salidasH + b2

salidasO = evaluar(FunO, netasO)

```
alfa = 0.15
CotaError = 0.001
MAX_ITERA = 300
ite = 0

while ( AVGError > CotaError ) and ( ite < MAX_ITERA ) :
    for p in range(len(X)) :    #para cada ejemplo
        # propagar el ejemplo hacia adelante
        # calcular los errores en ambas capas
        # corregir los todos los pesos

    # Recalcular AVGError
    ite = ite + 1
    print(ite, AVGError)
    # Graficar las rectas
```



Términos de error

- Calcular los términos de error para las unidades de salida

$$\delta_{pk}^o = (y_{pk} - o_{pk}) f_k^{o'}(neta_{pk}^o)$$

- Calcular los términos de error para las unidades ocultas

$$\delta_{pj}^h = f_j^{h'}(neta_{pj}^h) \sum_k \delta_{pk}^o w_{kj}^o$$

Funciones de Activación

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

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

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

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

```
Fun = 'sigmoid'
```

```
netas = W1 @ X.T + b1
```

```
i = evaluar(FunH, netas)
```

```
derivH = evaluarDerivada(FunH,i)
```

```
Fun = 'tanh'
```

```
netas = W1 @ X.T + b1
```

```
i = evaluar(FunH, netas)
```

```
derivH = evaluarDerivada(FunH,i)
```

Calcular errores de cada capa

```
# calcular los errores en ambas capas
```

```
ErrorSalida = Y[p:p+1,:].T - salidasO
```

```
deltaO = ErrorSalida * evaluarDerivada(FunO,salidasO)
```

```
deltaH = evaluarDerivada(FunH,salidasH) * (W2.T @ deltaO)
```

Actualización de los pesos

corregir todos los pesos

```
W1 = W1 + alfa * deltaH @ X[p:p+1, :]
```

```
b1 = b1 + alfa * deltaH
```

```
W2 = W2 + alfa * deltaO @ salidasH.T
```

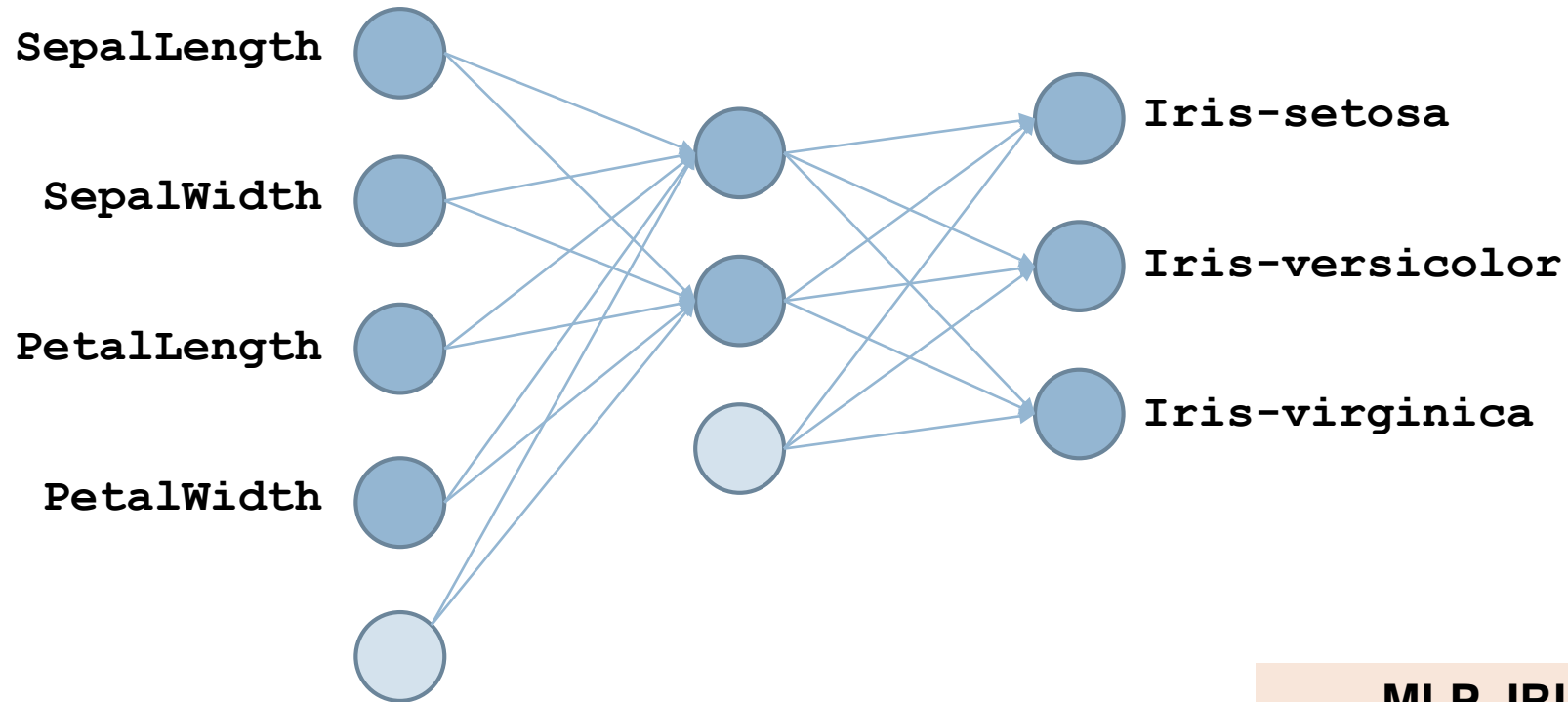
```
b2 = b2 + alfa * deltaO
```

Ejemplo: Clasificación de flores de Iris

Id	sepalength	sepalwidth	petallength	petalwidth	class
1	5,1	3,5	1,4	0,2	Iris-setosa
2	4,9	3,0	1,4	0,2	Iris-setosa
...
95	5,6	2,7	4,2	1,3	Iris-versicolor
96	5,7	3,0	4,2	1,2	Iris-versicolor
97	5,7	2,9	4,2	1,3	Iris-versicolor
...
149	6,2	3,4	5,4	2,3	Iris-virginica
150	5,9	3,0	5,1	1,8	Iris-virginica

<https://archive.ics.uci.edu/ml/datasets/Iris>

Ejemplo: Clasificación de flores de Iris



MLP_IRIS.ipynb

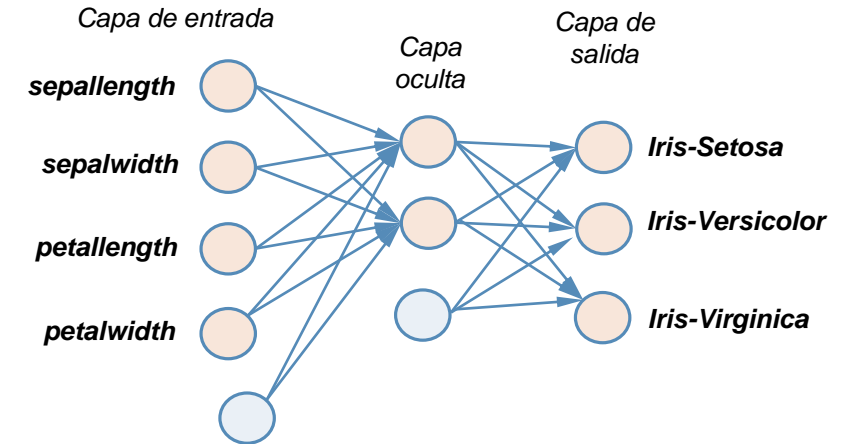
Clasificación de flores de Iris

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]]
```



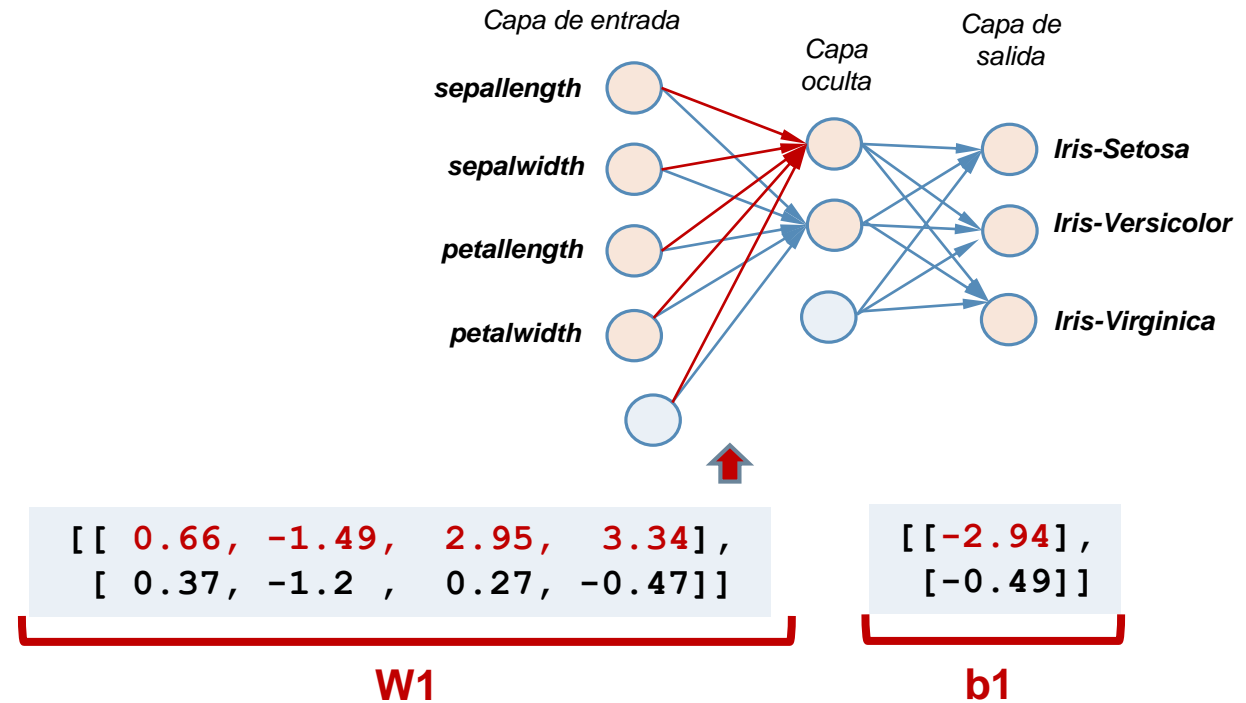
Clasificación de flores de Iris

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]]
```



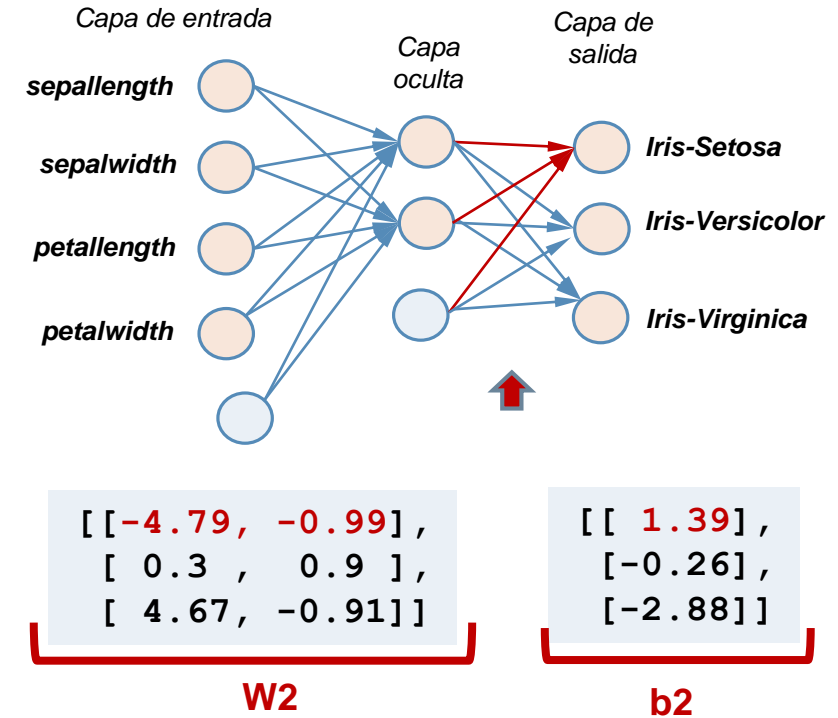
Clasificación de flores de Iris

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]]
```



Clasificación de flores de Iris

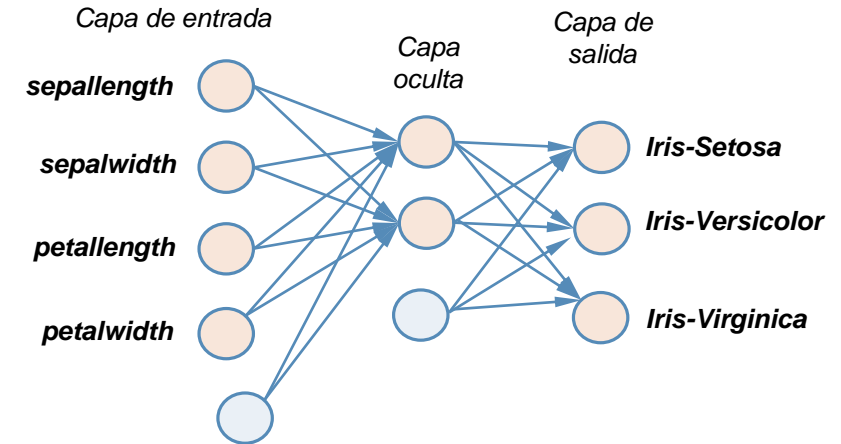
X

`[[-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]]`

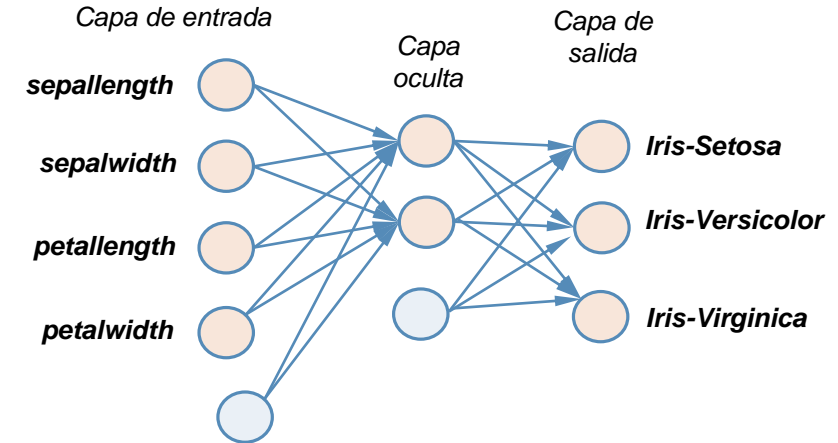
`FunH='tanh' ; FunO='sigmoid'`



Ingresar el primer ejemplo a la red y
calcular su salida

Calculando la salida de la capa oculta

$$\begin{array}{c} \mathbf{X} \\ \left[\begin{array}{cccc} [-1.73, -0.05, -1.38, -1.31], \\ [-0.37, -1.62, 0.22, 0.18], \\ [1.11, -0.05, 0.93, 1.54], \\ [-0.99, 0.39, -1.44, -1.31], \\ [1.73, 1.29, 1.46, 1.81] \end{array} \right] \end{array} \quad \leftarrow \quad \begin{array}{c} \mathbf{Y} \\ \left[\begin{array}{ccc} [1, 0, 0], \\ [0, 1, 0], \\ [0, 0, 1], \\ [1, 0, 0], \\ [0, 0, 1] \end{array} \right] \end{array}$$



□ Salida de la capa oculta

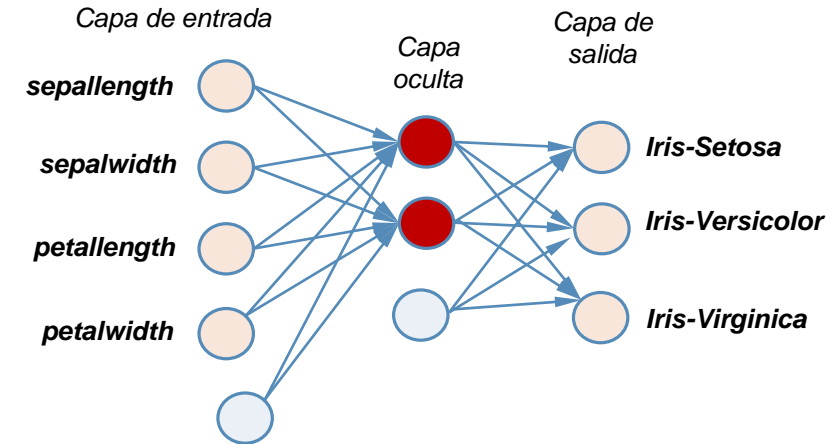
$$\text{netasH} = \mathbf{W1} * \mathbf{x}^T + \mathbf{b1}$$

$$\begin{array}{c} \mathbf{W1} \\ \left[\begin{array}{cccc} [0.66, -1.49, 2.95, 3.34], \\ [0.37, -1.2, 0.27, -0.47] \end{array} \right] \end{array} * \begin{array}{c} \mathbf{x}^T \\ \left[\begin{array}{c} [-1.73], \\ [-0.05], \\ [-1.38], \\ [-1.31] \end{array} \right] \end{array} + \begin{array}{c} \mathbf{b1} \\ \left[\begin{array}{c} [-2.94], \\ [-0.49] \end{array} \right] \end{array} = \begin{array}{c} \left[\begin{array}{c} [-12.4537], \\ [-0.827] \end{array} \right] \end{array}$$

$$\text{FunH} = \text{'tanh'} ; \text{FunO} = \text{'sigmoid'}$$

Calculando la salida de la capa oculta

$$\begin{array}{c} \mathbf{X} \\ \left[\begin{array}{cccc} [-1.73, -0.05, -1.38, -1.31], \\ [-0.37, -1.62, 0.22, 0.18], \\ [1.11, -0.05, 0.93, 1.54], \\ [-0.99, 0.39, -1.44, -1.31], \\ [1.73, 1.29, 1.46, 1.81] \end{array} \right] \end{array} \quad \leftarrow \quad \begin{array}{c} \mathbf{Y} \\ \left[\begin{array}{ccc} [1, 0, 0], \\ [0, 1, 0], \\ [0, 0, 1], \\ [1, 0, 0], \\ [0, 0, 1] \end{array} \right] \end{array}$$



□ Salida de la capa oculta

$$\text{netasH} = \mathbf{W1} * \mathbf{x}^T + \mathbf{b1}$$

$$\begin{array}{c} \left[\begin{array}{cccc} [0.66, -1.49, 2.95, 3.34], \\ [0.37, -1.2, 0.27, -0.47] \end{array} \right] * \begin{array}{c} \mathbf{x}^T \\ \left[\begin{array}{c} [-1.73], \\ [-0.05], \\ [-1.38], \\ [-1.31] \end{array} \right] \end{array} + \begin{array}{c} \mathbf{b1} \\ \left[\begin{array}{c} [-2.94], \\ [-0.49] \end{array} \right] \end{array} = \begin{array}{c} \left[\begin{array}{c} [-12.4537] \\ [-0.827] \end{array} \right] \end{array}
 \end{array}$$

FunH='tanh' ; FunO='sigmoid'

$$\text{salidasH} = 2 / (1 + \text{np.exp}(-\text{netasH})) - 1 = \begin{array}{c} \left[\begin{array}{c} [-0.99999219] \\ [-0.39144044] \end{array} \right] \end{array}$$

Calculando la salida de la red (capa de salida)

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]]
```

□ Salida de red

```
netasO = W2 * salidasH + b2
```

```
[[-4.79, -0.99],  
 [ 0.3 ,  0.9 ],  
 [ 4.67, -0.91]]
```

W2

salidasH

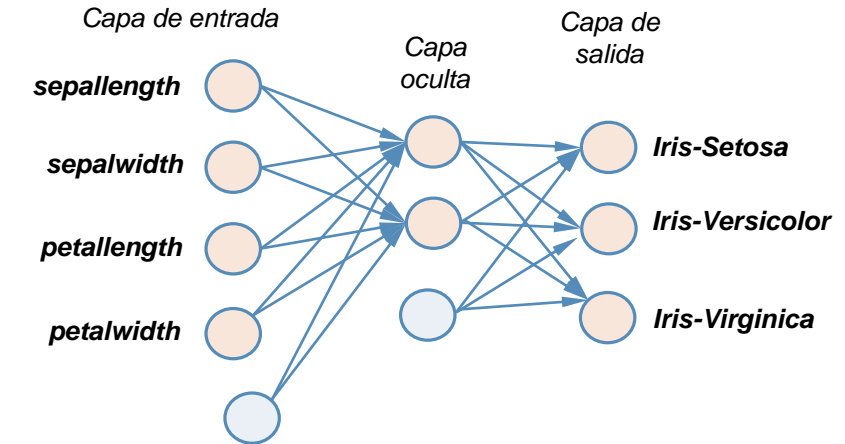
```
* [[-0.99999219]  
   [-0.39144044]]
```

+

```
[[ 1.39],  
 [-0.26],  
 [-2.88]]
```

b2

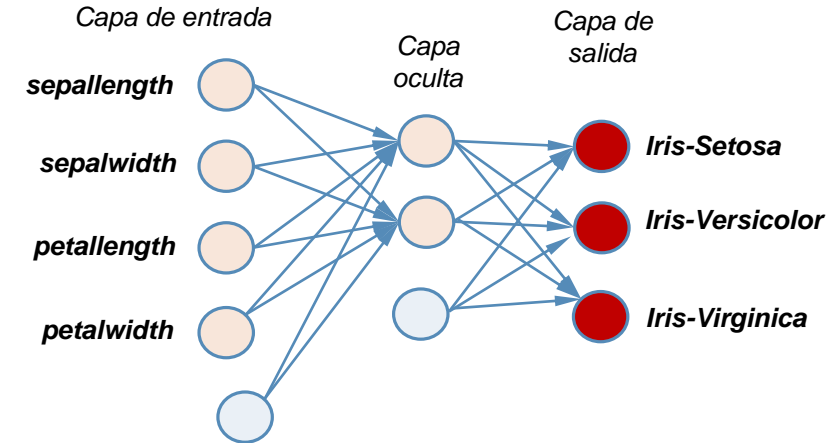
```
= [[ 6.56748865]  
   [-0.91229406]  
   [-7.19375274]]
```



FunH='tanh' ; FunO='sigmoid'

Calculando la salida de la red (capa de salida)

$$\begin{array}{c} \text{X} \\ \begin{bmatrix} [-1.73, -0.05, -1.38, -1.31], \\ [-0.37, -1.62, 0.22, 0.18], \\ [1.11, -0.05, 0.93, 1.54], \\ [-0.99, 0.39, -1.44, -1.31], \\ [1.73, 1.29, 1.46, 1.81] \end{bmatrix} \end{array} \quad \begin{array}{c} \text{Y} \\ \begin{bmatrix} [1, 0, 0], \\ [0, 1, 0], \\ [0, 0, 1], \\ [1, 0, 0], \\ [0, 0, 1] \end{bmatrix} \end{array} \leftarrow$$



□ Salida de red

$$\text{netasO} = \text{W2} * \text{salidasH} + \text{b2}$$

FunH='tanh' ; FunO='sigmoid'

$$\begin{array}{c} \text{salidasH} \\ \begin{bmatrix} [-4.79, -0.99], \\ [0.3, 0.9], \\ [4.67, -0.91] \end{bmatrix} \end{array} \quad \begin{array}{c} * \\ \begin{bmatrix} [-0.99999219] \\ [-0.39144044] \end{bmatrix} \end{array} \quad + \quad \begin{array}{c} \text{b2} \\ \begin{bmatrix} [1.39], \\ [-0.26], \\ [-2.88] \end{bmatrix} \end{array} \quad = \quad \begin{array}{c} \text{netasO} \\ \begin{bmatrix} [6.56748865] \\ [-0.91229406] \\ [-7.19375274] \end{bmatrix} \end{array}$$

$$\text{salidasO} = 1 / (1 + \text{np.exp}(-\text{netasO})) = \begin{bmatrix} [9.98596651\text{e-}01] \\ [2.81401722\text{e-}01] \\ [7.50700367\text{e-}04] \end{bmatrix} \leftarrow$$

Error de la capa de salida

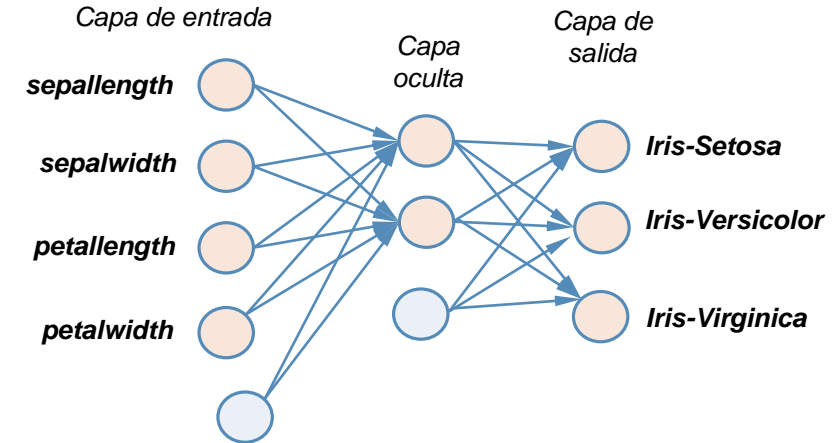
X	Y
$\begin{bmatrix} -1.73, -0.05, -1.38, -1.31 \\ -0.37, -1.62, 0.22, 0.18 \\ 1.11, -0.05, 0.93, 1.54 \\ -0.99, 0.39, -1.44, -1.31 \\ 1.73, 1.29, 1.46, 1.81 \end{bmatrix}$	$\begin{bmatrix} \boxed{1, 0, 0}, \\ 0, 1, 0, \\ 0, 0, 1, \\ 1, 0, 0, \\ 0, 0, 1 \end{bmatrix}$ ←

- Error en la respuesta de la red para este ejemplo

$$\text{ErrorSalida} = y^T - \text{salidasO}$$

$$\text{ErrorSalida} = \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix} - \begin{bmatrix} 9.98596651e-01 \\ 2.81401722e-01 \\ 7.50700367e-04 \end{bmatrix} = \begin{bmatrix} 0.00140335 \\ -0.28653063 \\ -0.0007507 \end{bmatrix}$$

↑
salidasO



FunH='tanh' ; FunO='sigmoid'

Factores de corrección de los pesos

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

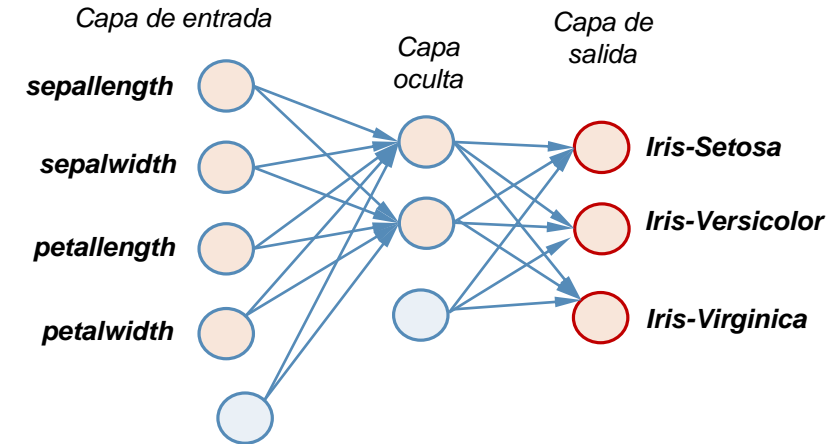
```
[[1,0,0],
 [0,1,0],
 [0,0,1],
 [1,0,0],
 [0,0,1]]
```

Factores para corregir W2 y b2

```
deltaO = ErrorSalida .* derivada_FunO
```

```
deltaO = [[ 0.00140335]
 [-0.28653063]
 [-0.0007507 ]] .* [[0.00140138]
 [0.20443083]
 [0.00075014]] = [[ 1.96662772e-06]
 [-5.85756944e-02]
 [-5.63128256e-07]]
```

`salidasO*(1-salidasO)`



FunH='tanh' ; FunO='sigmoid'

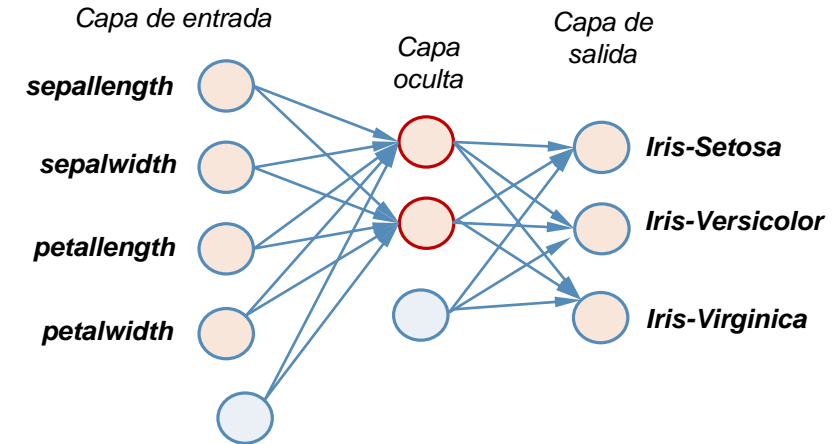
Factores de corrección de los pesos

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]]
```



FunH='tanh' ; FunO='sigmoid'

Factores para corregir W1 y b1

$(1 - \text{salidasH}^2)$

$\text{deltaH} = \text{deriv_FunH} .* (\text{W2.T} @ \text{deltaO})$

$\text{deltaH} = \begin{bmatrix} 1.56128940\text{e-}05 \\ 8.46774379\text{e-}01 \end{bmatrix} .* \begin{bmatrix} [-4.79 & 0.31 & 4.67] \\ [-0.99 & 0.9 & -0.91] \end{bmatrix} @ \begin{bmatrix} 1.96662772\text{e-}06 \\ -5.85756944\text{e-}02 \\ -5.63128256\text{e-}07 \end{bmatrix}$

$\text{deltaH} = \begin{bmatrix} -2.74548968\text{e-}07 \\ -4.46415722\text{e-}02 \end{bmatrix}$

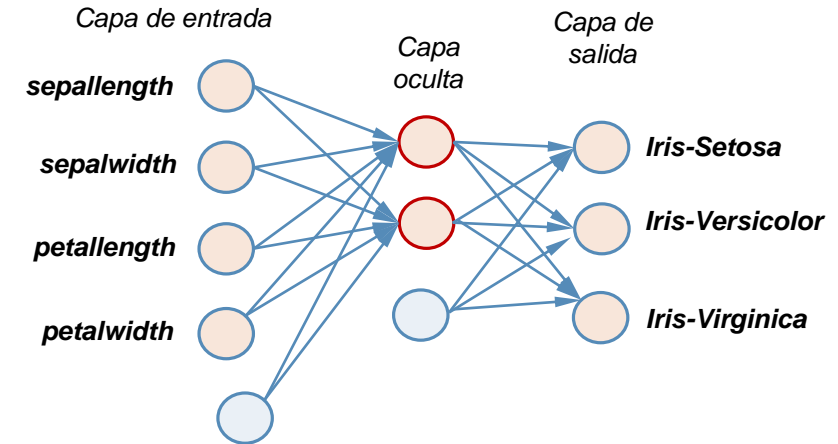
Corrigiendo de los pesos

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 W2 y b2

FunH='tanh' ; FunO='sigmoid'

`W2 = W2 + alfa * deltaO @ salidasH.T`

$$W2 = \begin{bmatrix} -4.79 & -0.99 \\ 0.31 & 0.9 \\ 4.67 & -0.91 \end{bmatrix} + \text{alfa} * \left(\begin{bmatrix} 1.96662439e-06 \\ -5.69035911e-02 \\ -5.63127983e-07 \end{bmatrix} @ \begin{bmatrix} -0.999 & -0.39144 \end{bmatrix} \right) = \begin{bmatrix} -4.79 & -0.99 \\ 0.31 & 0.9 \\ 4.67 & -0.91 \end{bmatrix}$$

$$b2 = b2 + \text{alfa} * \text{deltaO} = \begin{bmatrix} 1.39 \\ -0.26 \\ -2.88 \end{bmatrix} + \text{alfa} * \begin{bmatrix} 1.96662439e-06 \\ -5.69035911e-02 \\ -5.63127983e-07 \end{bmatrix} = \begin{bmatrix} 1.39 \\ -0.26 \\ -2.88 \end{bmatrix}$$

Multiperceptrón en Python

```
from sklearn.neural_network import MLPClassifier

clf = MLPClassifier( solver='sgd',
                    learning_rate_init=0.15,
                    hidden_layer_sizes=(5),
                    max_iter=700, verbose=False,
                    tol=1.0e-09, activation = 'tanh')

clf.fit(X_train,T_train)
y_pred= clf.predict(X_train)
```

Matriz de Confusión

	Predice Clase 1	Predice Clase 2	Recall
True Clase 1	A	B	$A/(A+B)$
True Clase 2	C	D	$D/(C+D)$
Precision	$A/(A+C)$	$D/(B+D)$	$(A+D)/(A+B+C+D)$ accuracy

- Los **aciertos** del modelo están sobre la **diagonal** de la matriz.
- **Precision**: la proporción de **predicciones correctas** sobre **una clase**.
- **Recall**: la proporción de **ejemplos** de **una clase** que son **correctamente clasificados**.
- **Accuracy**: la performance general del modelo, sobre **todas las clases**. Es la cantidad de **aciertos** sobre el **total** de ejemplos.

Sklearn.metrics.accuracy_score

```
from sklearn import metrics

Y_train = [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
Y_pred  = [0, 2, 1, 3, 0, 1, 2, 0, 0, 1, 2, 3]

aciertos = metrics.accuracy_score(Y_train, Y_pred)
print("%% accuracy = %.3f" % aciertos)
```


Sklearn.metrics.accuracy_score

```
from sklearn import metrics

Y_train = [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
Y_pred  = [0, 2, 1, 3, 0, 1, 2, 0, 0, 1, 2, 3]

aciertos = metrics.accuracy_score(Y_train, Y_pred)
print("%% accuracy = %.3f" % aciertos)
```

Rtas esperadas

Rtas obtenidas

- De los 12 valores sólo 9 fueron identificados correctamente.
- La tasa de aciertos es $9/12 = 0.75$

Sklearn.metrics.confusion_matrix

```
Y_train = [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
Y_pred  = [0, 2, 1, 3, 0, 1, 2, 0, 0, 1, 2, 3]
```

```
MM = metrics.confusion_matrix(Y_train, Y_pred)
print("Matriz de confusión:\n%s" % MM)
```

```
Matriz de confusión:
 0 [[3 0 0 0]
 1 [0 2 1 0]
 2 [0 1 2 0]
 3 [1 0 0 2]]
    0  1  2  3
  PREDICE
```

Esperaba obtener 1
como respuesta pero
la red respondió 2

Sklearn.metrics.confusion_matrix

```
Y_train = [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
Y_pred  = [0, 2, 1, 3, 0, 1, 2, 0, 0, 1, 2, 3]
```

```
MM = metrics.confusion_matrix(Y_train, Y_pred)
print("Matriz de confusión:\n%s" % MM)
```

```
Matriz de confusión:
 0 [[3 0 0 0]
 1 [0 2 1 0]
 2 [0 1 2 0]
 3 [1 0 0 2]]
    0  1  2  3
  PREDICE
```

La respuesta correcta
es 2 pero la red
respondió 1

Sklearn.metrics.confusion_matrix

```
Y_train = [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
Y_pred  = [0, 2, 1, 3, 0, 1, 2, 0, 0, 1, 2, 3]
```

```
MM = metrics.confusion_matrix(Y_train, Y_pred)
print("Matriz de confusión:\n%s" % MM)
```

```
Matriz de confusión:
 0 [[3 0 0 0]
 1 [0 2 1 0]
 2 [0 1 2 0]
 3 [1 0 0 2]]
   0  1  2  3
  PREDICE
```

Esperaba un 3 pero la red respondió 0

Sklearn.metrics.confusion_matrix

```
Y_train = [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
Y_pred  = [0, 2, 1, 3, 0, 1, 2, 0, 0, 1, 2, 3]
```

```
MM = metrics.confusion_matrix(Y_train, Y_pred)
print("Matriz de confusión:\n%s" % MM)
```

```

      Matriz de confusión:
TRUE 0 [[3 0 0 0]
      1 [0 2 1 0]
      2 [0 1 2 0]
      3 [1 0 0 2]]
      0 1 2 3
      PREDICE
```

Los valores fuera de la diagonal principal son errores

Sklearn.metrics.classification_report

```
Y_train = [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
Y_pred  = [0, 2, 1, 3, 0, 1, 2, 0, 0, 1, 2, 3]
report = metrics.classification_report(Y_train, Y_pred)
print("Resultado de la clasificación:\n%s" % report)
```

Resultado de la clasificación:

	precision	recall	f1-score	support
0	0.75	1.00	0.86	3
1	0.67	0.67	0.67	3
2	0.67	0.67	0.67	3
3	1.00	0.67	0.80	3
accuracy			0.75	12
macro avg	0.77	0.75	0.75	12
weighted avg	0.77	0.75	0.75	12

Matriz de confusión:

```
[[3 0 0 0]
 [0 2 1 0]
 [0 1 2 0]
 [1 0 0 2]]
```

Sklearn.metrics.classification_report

```
Y_train = [0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3]
Y_pred  = [0, 2, 1, 3, 0, 1, 2, 0, 0, 1, 2, 3]
report = metrics.classification_report(Y_train, Y_pred)
print("Resultado de la clasificación:\n%s" % report)
```

Resultado de la clasificación:

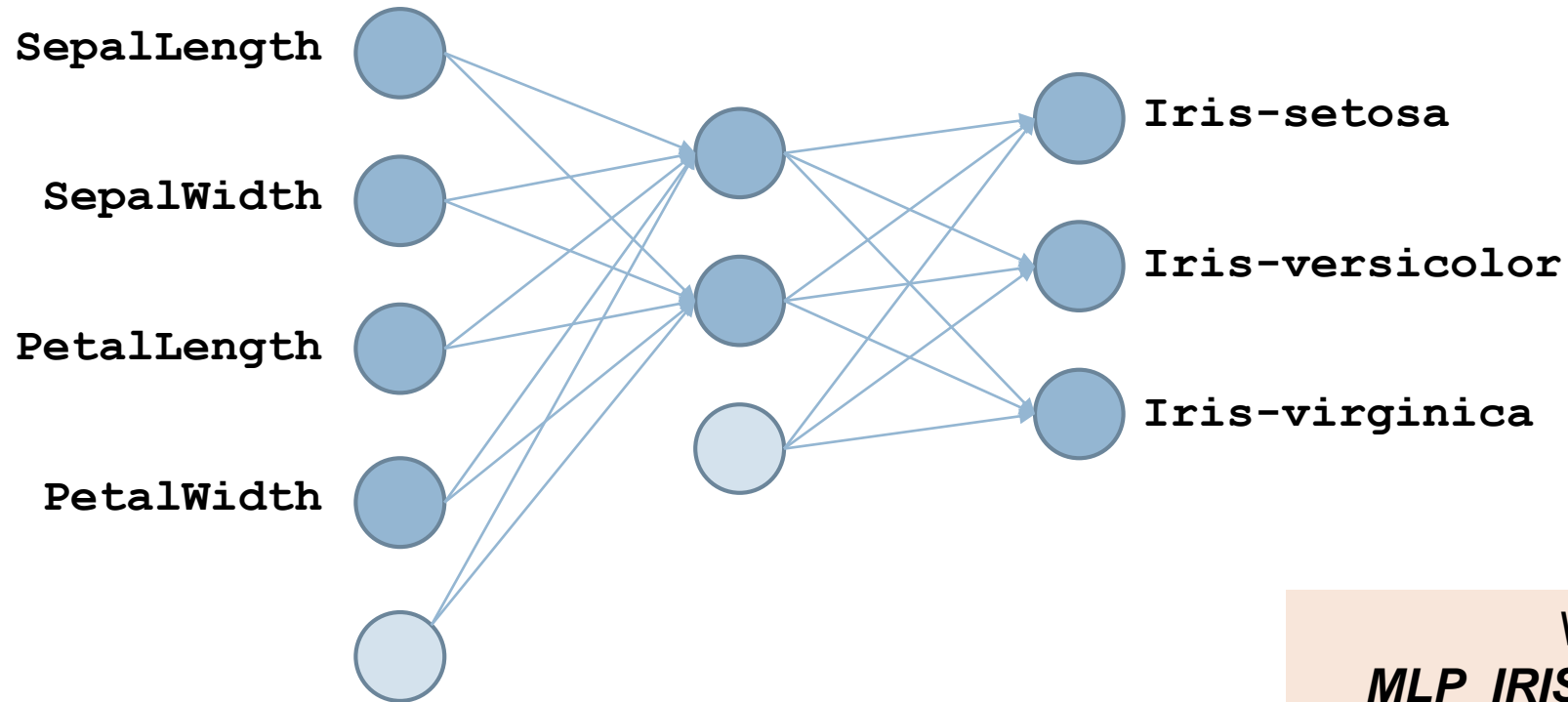


	precision	recall	f1-score	support
0	0.75	1.00	0.86	3
1	0.67	0.67	0.67	3
2	0.67	0.67	0.67	3
3	1.00	0.67	0.80	3
accuracy			0.75	12
macro avg	0.77	0.75	0.75	12
weighted avg	0.77	0.75	0.75	12

F1-score

$$F1 = 2 * \frac{\text{precision} * \text{recall}}{\text{precisión} + \text{recall}}$$

Ejemplo: Clasificación de flores de Iris



Ver
MLP_IRIS_RN.ipynb

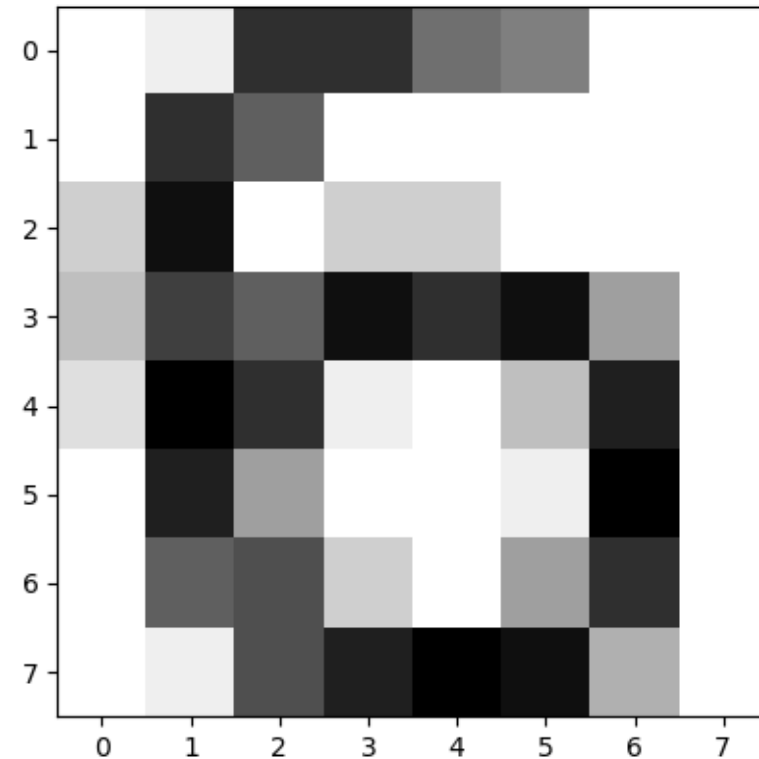
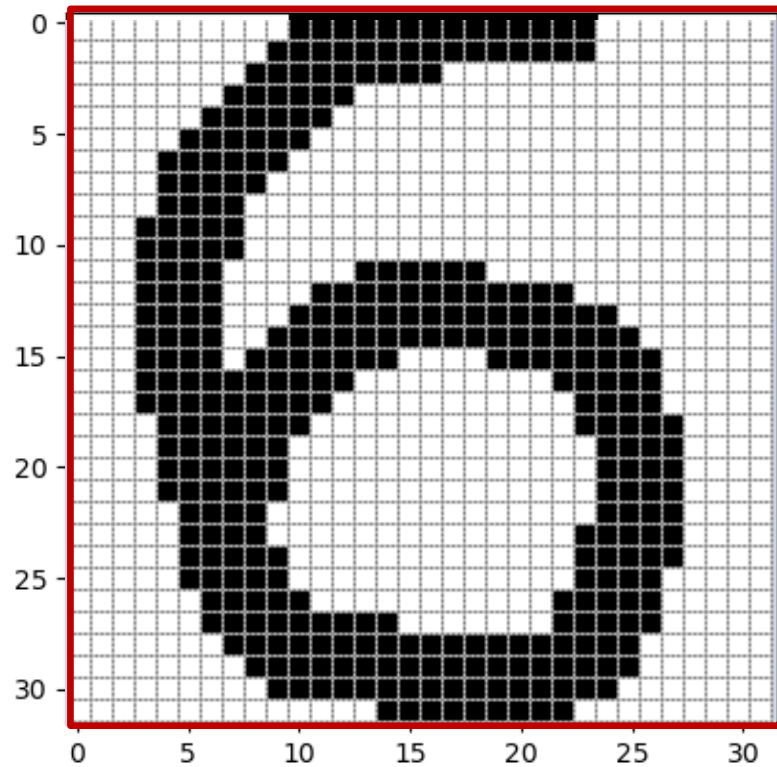
Reconocedor de dígitos escritos a mano

- Se desea entrenar un multiperceptrón para reconocer dígitos escritos a mano. Para ello se dispone de los mapas de bits correspondientes a 3823 dígitos escritos a mano por 30 personas diferentes en el archivo “optdigits_train.csv”.
- El desempeño de la red será probado con los dígitos del archivo “optdigits_test.csv” escritos por otras 13 personas.



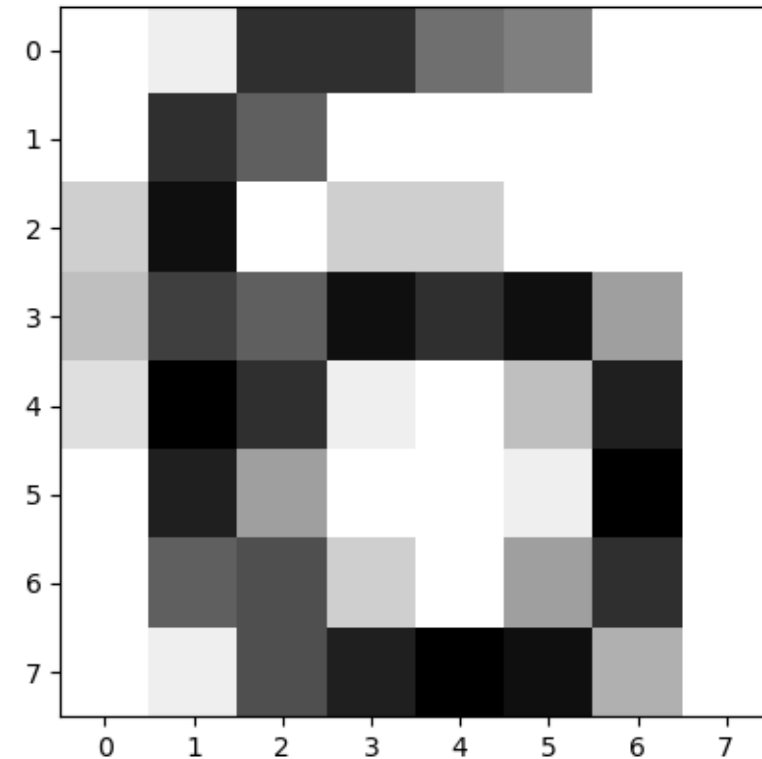
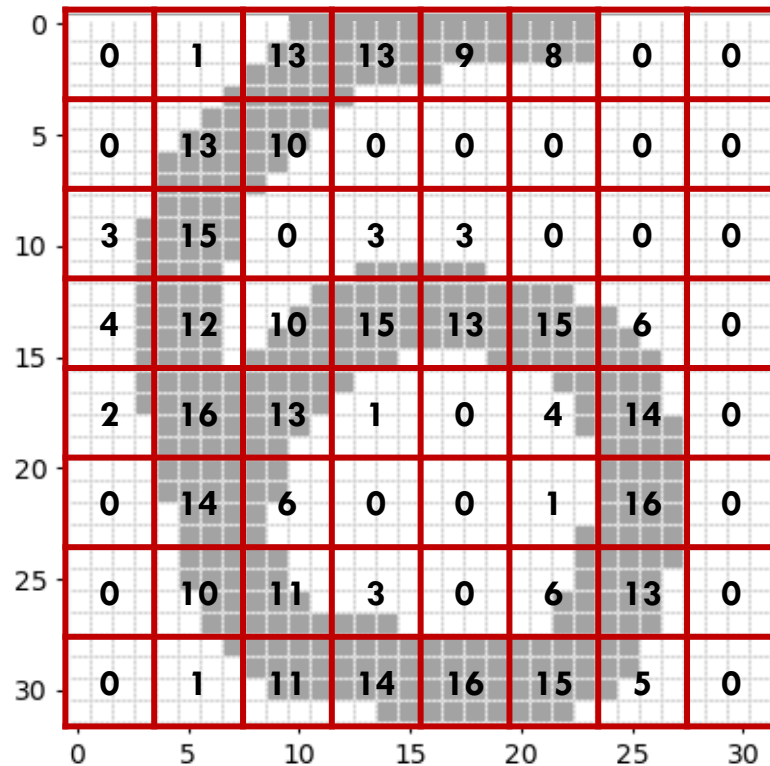
“optdigits_train.csv” y “optdigits_test.csv”

- Cada dígito está representado por una matriz numérica de 8x8



“optdigits_train.csv” y “optdigits_test.csv”

- Cada dígito está representado por una matriz numérica de 8x8



RN para reconocer dígitos manuscritos

```
In [90]: print("ite = %d %% aciertos X_train : %.3f" % (ite,
metrics.accuracy_score(Y_train,Y_pred)))
ite = 200  % aciertos X_train : 0.982
```

```
In [91]: MM = metrics.confusion_matrix(Y_train,Y_pred)
....: print("Matriz de confusión TRAIN:\n%s" % MM)
```

Matriz de confusión TRAIN: im_32x32_a_8x8.ipynb

```
[[375  0  0  0  0  0  0  0  1  0]
 [ 7 382  0  0  0  0  0  0  0  0]
 [ 2  0 378  0  0  0  0  0  0  0]
 [ 7  0  0 380  0  2  0  0  0  0]
 [ 3  0  0  0 383  0  1  0  0  0]
 [ 6  0  0  1  0 369  0  0  0  0]
 [ 2  2  0  0  0  0 373  0  0  0]
 [ 0  0  0  1  0  0  0 386  0  0]
 [17  2  0  0  0  0  0  0 361  0]
 [13  1  0  1  0  1  0  0  0 366]]
```

Entrenamiento
muy lento
¿Se puede
acelerar?

MLP_MNIST_8x8.ipynb

Descenso de gradiente en mini-lotes

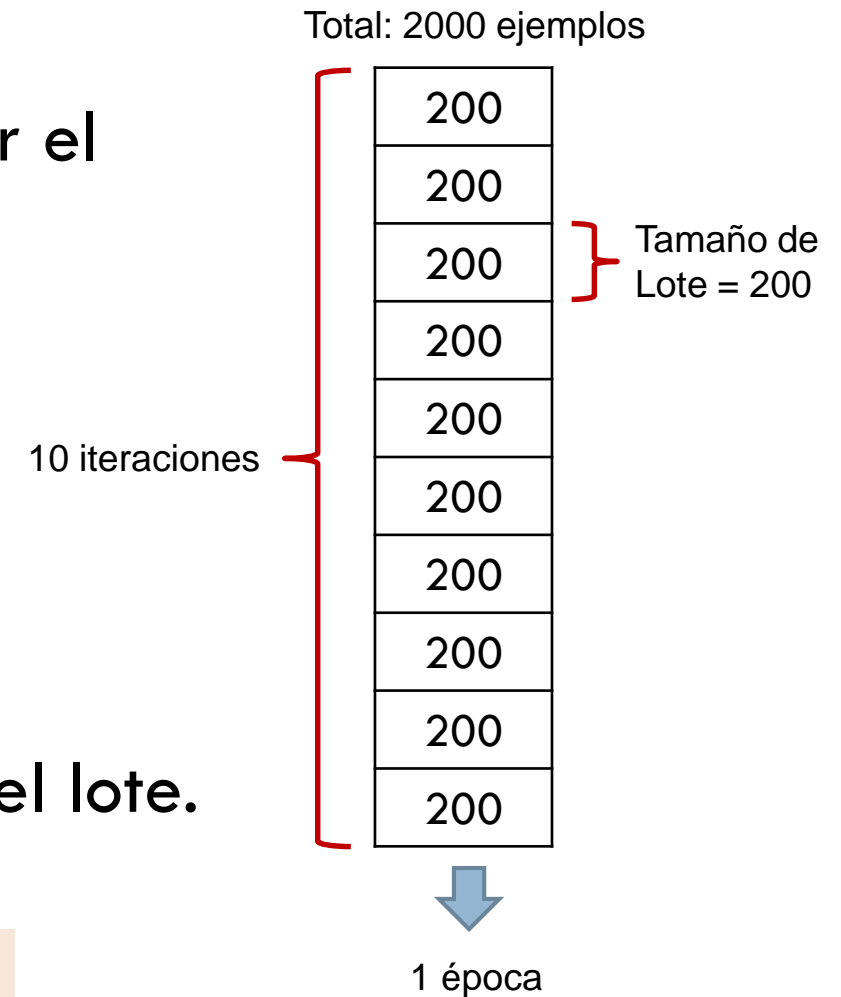
- En lugar de ingresar los ejemplos de a uno, ingresamos N a la red y buscaremos minimizar el error cuadrático promedio del lote.

- La función de costo será

$$C = \frac{1}{N} \sum_{i=1}^N (d_i - f(neta_i))^2$$

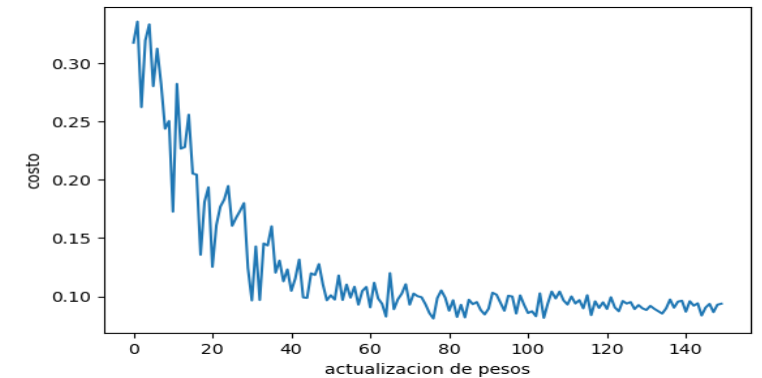
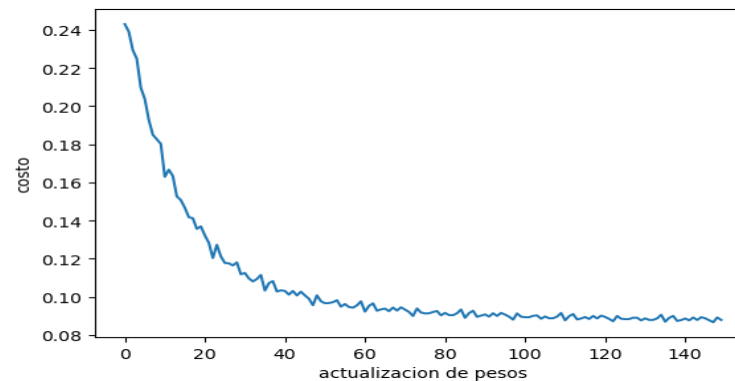
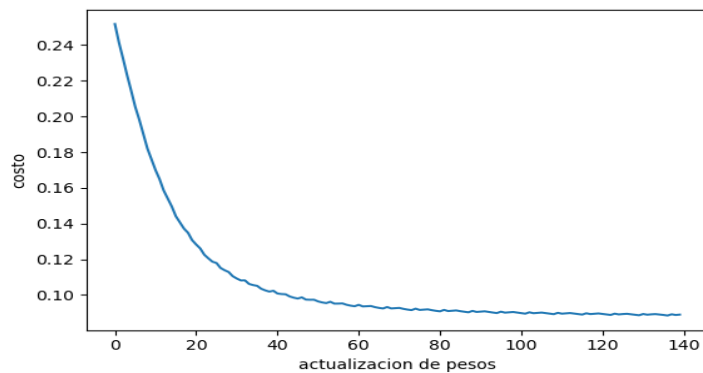
- N es la cantidad de ejemplos que conforman el lote.

MLP_MNIST_RN.ipynb



Descenso de gradiente

Batch	Mini-batch	Stochastic
Ingresa TODOS los ejemplos y luego actualiza los pesos.	Ingresa un LOTE de N ejemplos y luego actualiza los pesos	Ingresa UN ejemplo y luego actualiza los pesos
Bueno para pocos ejemplos	Bueno p/funciones de error convexas	Bueno para data sets grandes
$C = \frac{1}{M} \sum_{i=1}^M (d_i - f(neta_i))^2$	$C = \frac{1}{N} \sum_{i=1}^N (d_i - f(neta_i))^2 \quad N \ll M$	$C = (d - f(neta))^2$



Multiperceptrón en Python

```
from sklearn.neural_network import MLPClassifier

clf = MLPClassifier(solver='sgd', learning_rate_init=0.05,
                    hidden_layer_sizes=(15,), random_state=1,
                    max_iter=2000,
                    verbose=False, tol=1.0e-05,
                    batch_size=200,
                    activation='logistic')

history = clf.fit(X_train, Y_train)

y_pred= clf.predict(X_train)
```

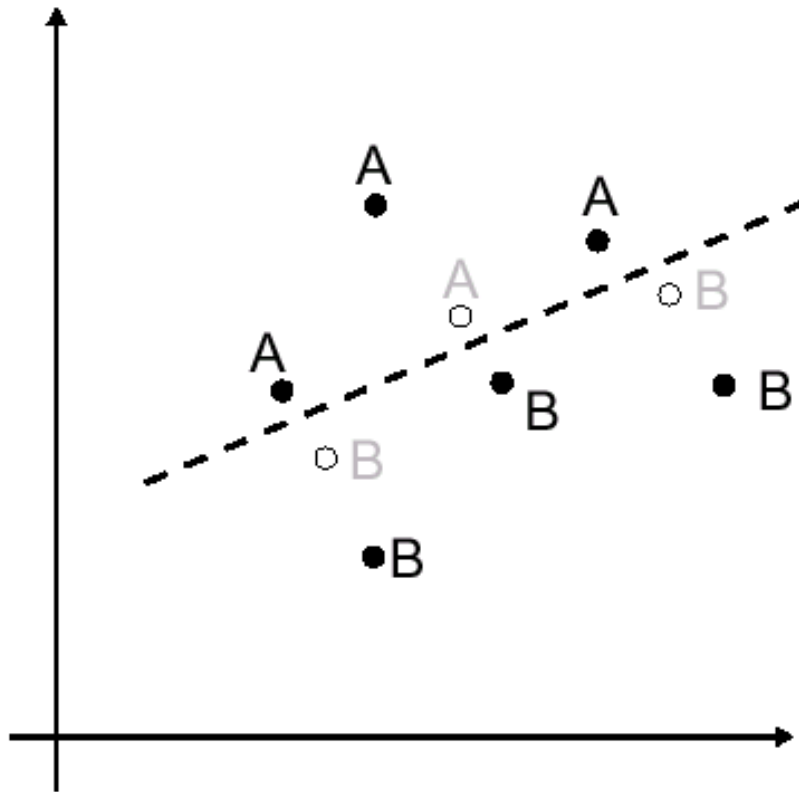
Usa
Entropía cruzada
como función de costo

Problemas

- La capacidad de generalización de la red está relacionada con la cantidad de neuronas de la capa oculta.
- El descenso por la técnica del gradiente tiene el problema de caer en un mínimo local.

Capacidad de generalización

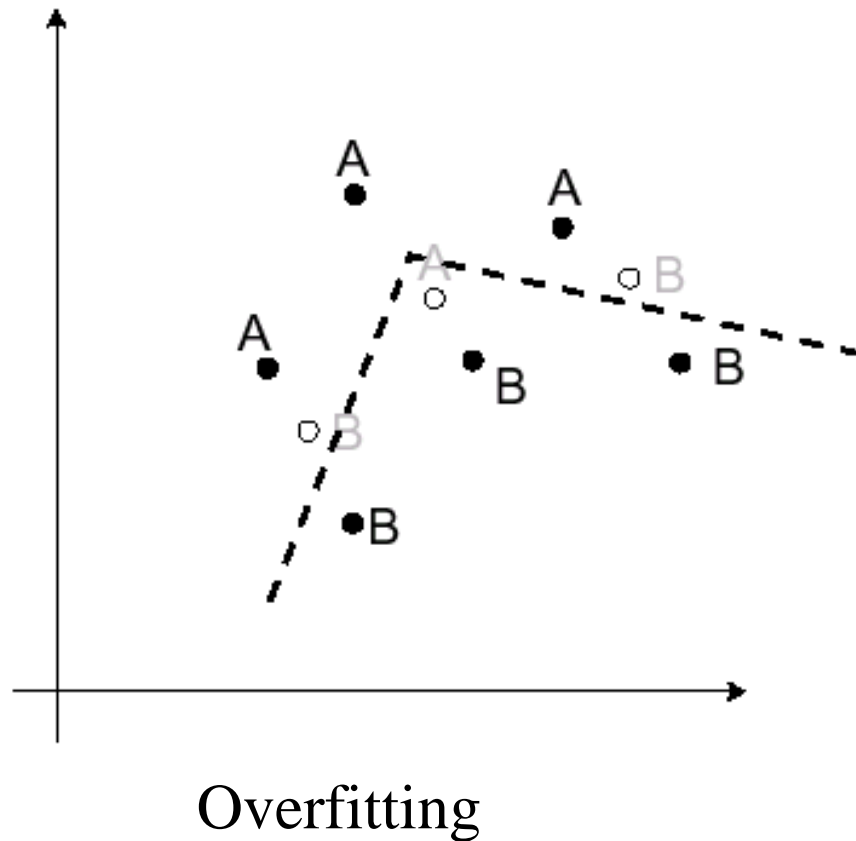
82



- RN formada por una única neurona.
- Los puntos sólidos corresponden a los ejemplos de entrenamiento y los demás a testeo.
- La clasificación es correcta.
- En este caso se dice que la red ha **generalizado** la información correctamente.

Sobreaajuste de la superficie de decisión

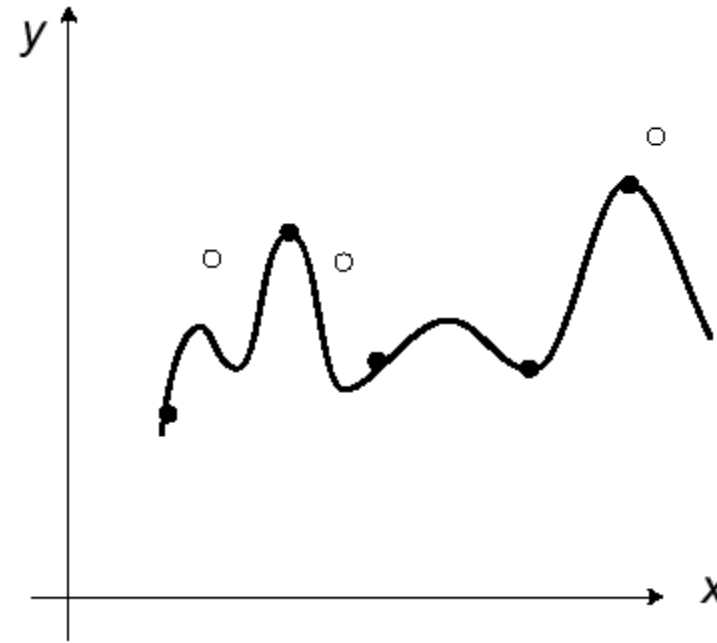
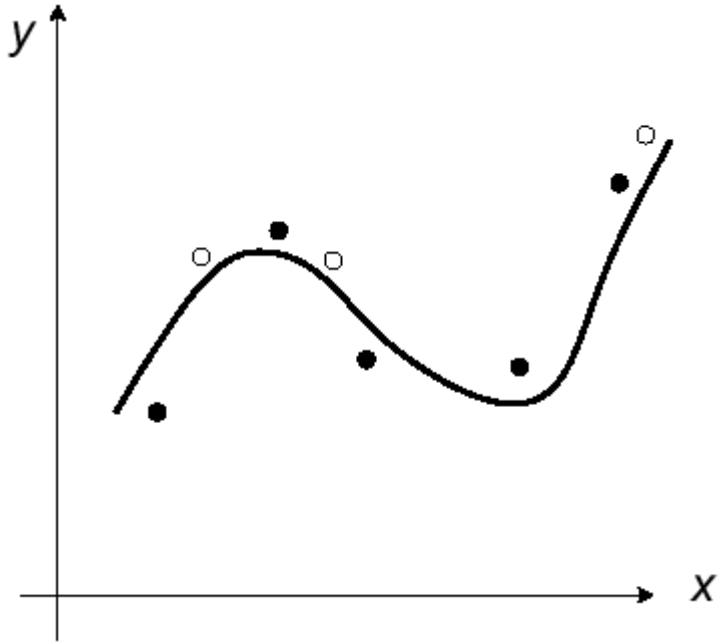
83



- RN que utiliza dos neuronas ocultas.
- Cada hiperplano busca la mayor proximidad a los ejemplos.
- Algunos ejemplos se clasifican incorrectamente.
- En este caso se dice que la red NO ha **generalizado** la información correctamente.

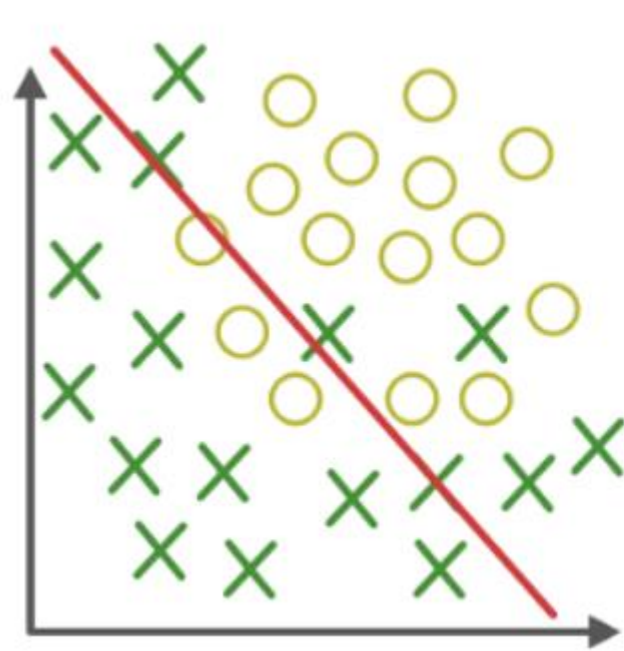
Sobreajuste de la superficie de decisión

84

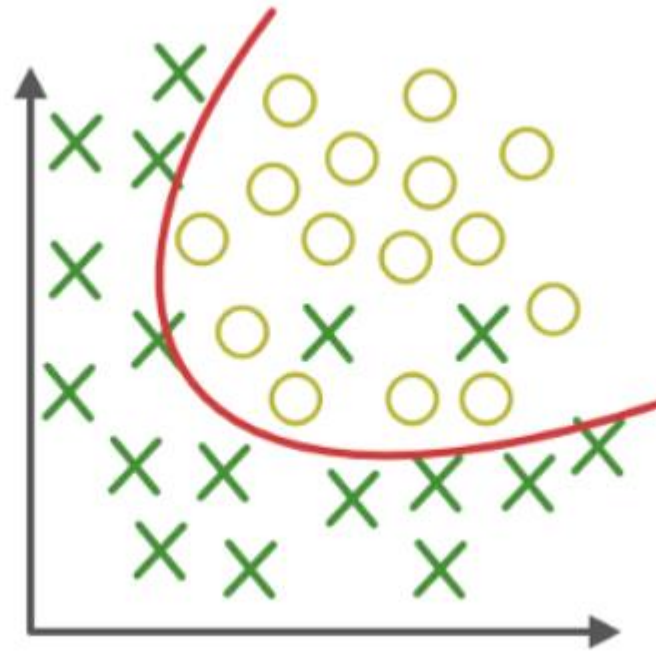


- A mayor cantidad de neuronas en la capa oculta, la red puede variar más rápido en respuesta a los cambios de la entrada.

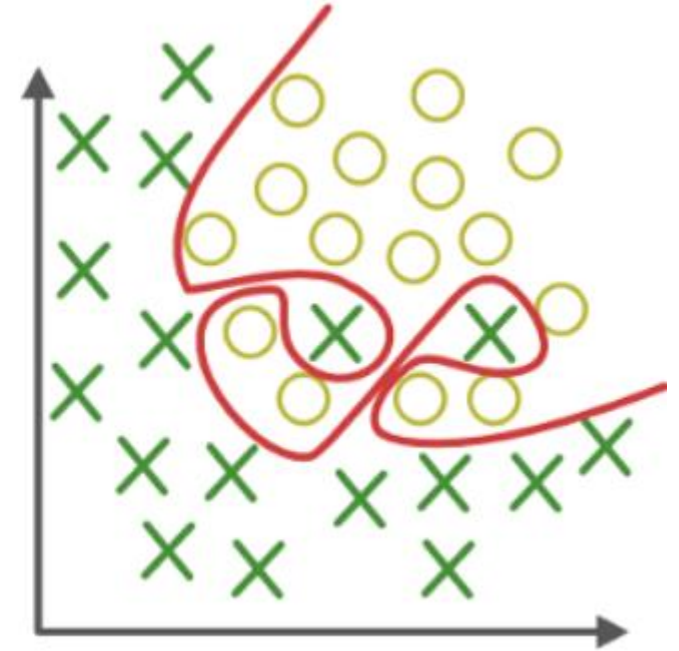
Capacidad de generalización de la red



Underfitting
(demasiado simple)



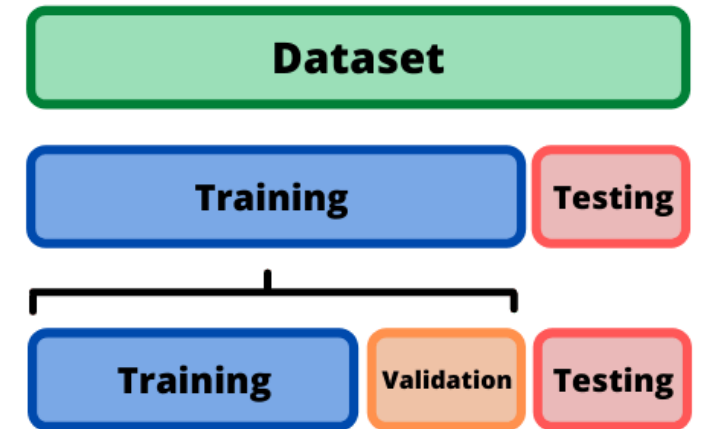
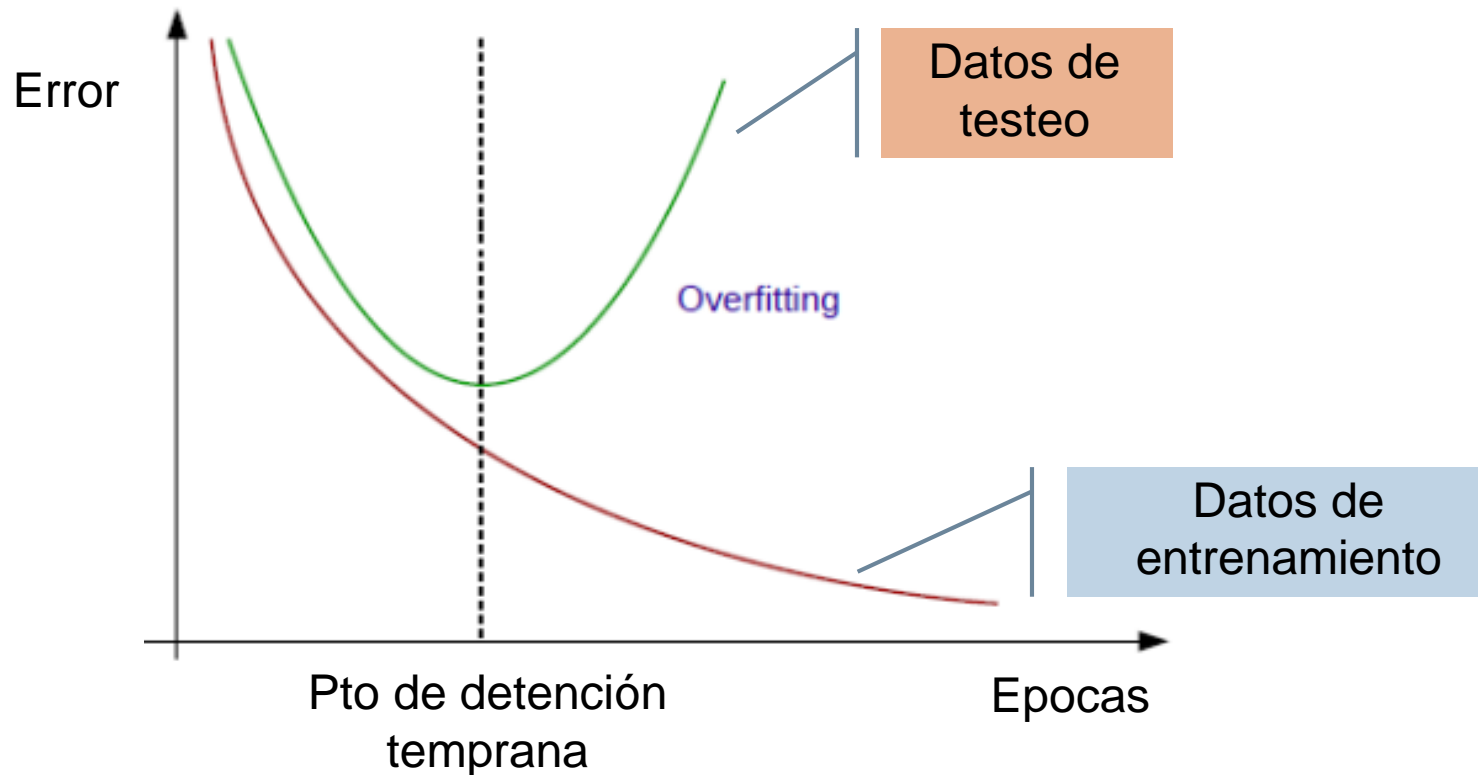
Generalización
correcta



Overfitting
(demasiados
parámetros)

Sobreajuste

□ Parada temprana (early-stopping)



Reducción del sobreajuste

- Si lo que se busca es reducir el sobreajuste puede probar
 - ▣ Incrementar la cantidad de ejemplos de entrenamiento.
 - ▣ Reducir la complejidad del modelo, es decir usar menos pesos (menos capas o menos neuronas por capa).
 - ▣ Aplicar una técnica de regularización
 - Regularización L2
 - Regularización L1
 - Dropout

Tienen por objetivo que los pesos de la red se mantengan pequeños

Sobreaajuste - Regularización L2

- También conocida como técnica de decaimiento de pesos

$$C = C_o + \frac{\lambda}{2} \sum_k w_k^2$$

donde C_o es la función de costo original sin regularizar

- La derivada de la función de costo regularizada será

$$\frac{\partial C}{\partial w_k} = \frac{\partial C_o}{\partial w_k} + \lambda w_k$$

Sobreajuste - Regularización L2

Función de costo regularizada

$$C = C_0 + \frac{\lambda}{2} \sum_k w_k^2$$

Derivada

$$\frac{\partial C}{\partial w_k} = \frac{\partial C_0}{\partial w_k} + \lambda w_k$$

- Actualización de los pesos

$$w_k = w_k - \alpha \frac{\partial C_0}{\partial w_k} - \lambda w_k$$

$$w_k = (1 - \lambda) w_k - \alpha \frac{\partial C_0}{\partial w_k}$$

*La regularización
no se aplica al
bias.*

Sobreajuste - Regularización L1

Función de costo regularizada

$$C = C_0 + \lambda \sum_k |w_k|$$

Derivada

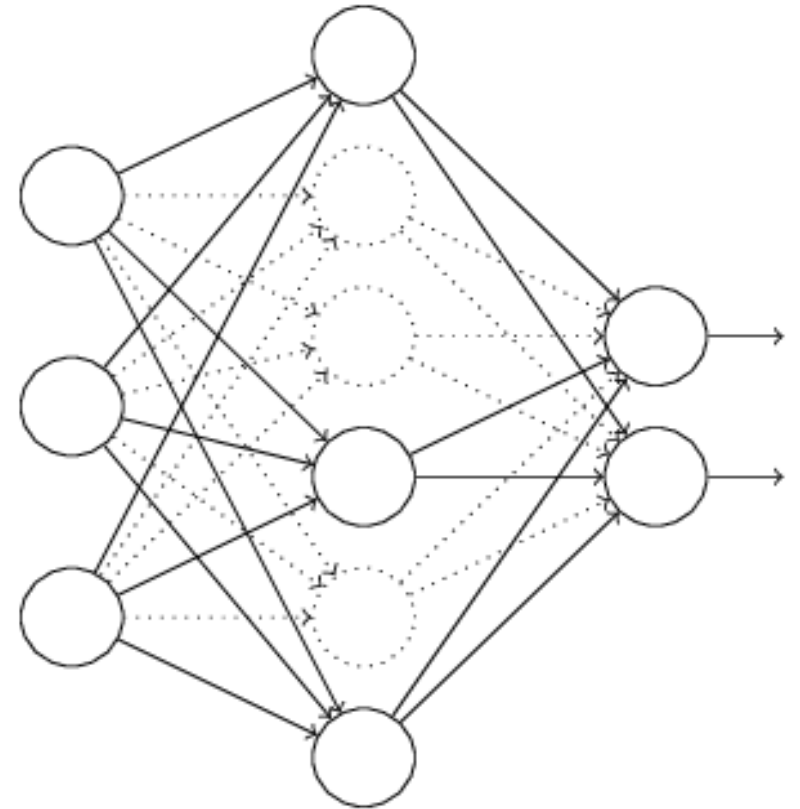
$$\frac{\partial C}{\partial w_k} = \frac{\partial C_0}{\partial w_k} + \lambda \operatorname{sign}(w_k)$$

- Actualización de los pesos

$$w_k = w_k - \alpha \frac{\partial C_0}{\partial w_k} - \lambda \operatorname{sign}(w_k)$$

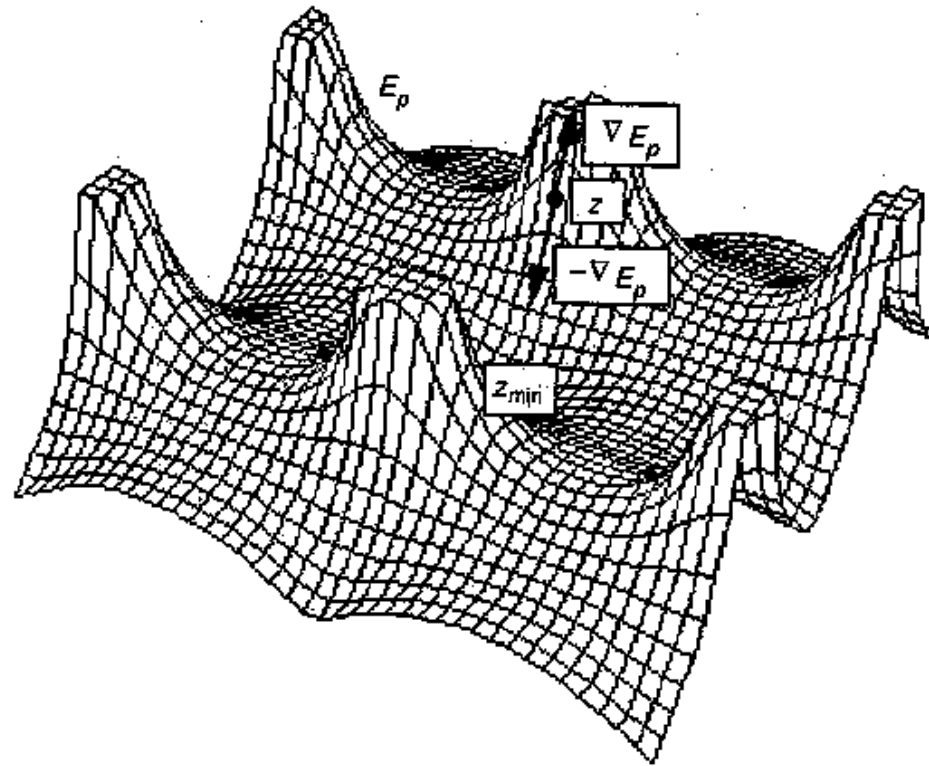
Sobreajuste - Dropout

- No modifica la función de costo sino la arquitectura de la de la red.
- Proceso
 - ▣ Selecciona aleatoriamente las neuronas que no participarán en la próxima iteración y las “borra” temporalmente.
 - ▣ Actualiza los pesos (del mini lote si corresponde).
 - ▣ Restaura las neuronas “borradas”.
 - ▣ Repite hasta que se estabilice.



¿Mínimo local o global?

92



- El problema se resuelve utilizando la dirección del gradiente junto con un componente aleatorio que permita salir de los mínimos locales.

Velocidad de aprendizaje

- α maneja la velocidad de aprendizaje.
- Si su valor se incrementa demasiado, la red puede desestabilizarse.
- Una forma de solucionar esto es incorporar, a la modificación de los pesos, un término que incluya una proporción del último cambio realizado. Este término se denomina **momento**.

Término de momento

- La modificación de los pesos se realizará de la siguiente forma:

$$v = \mu v - \alpha \nabla C$$

$$w = w + v$$

- El parámetro μ representa la contribución del término de momento.