



UNIVERSIDADE DA CORUÑA

# SISTEMAS CONEXIONISTAS ALIMENTADOS HACIA DELANTE

## INTRODUCCIÓN A LAS REDES DE NEURONAS ARTIFICIALES

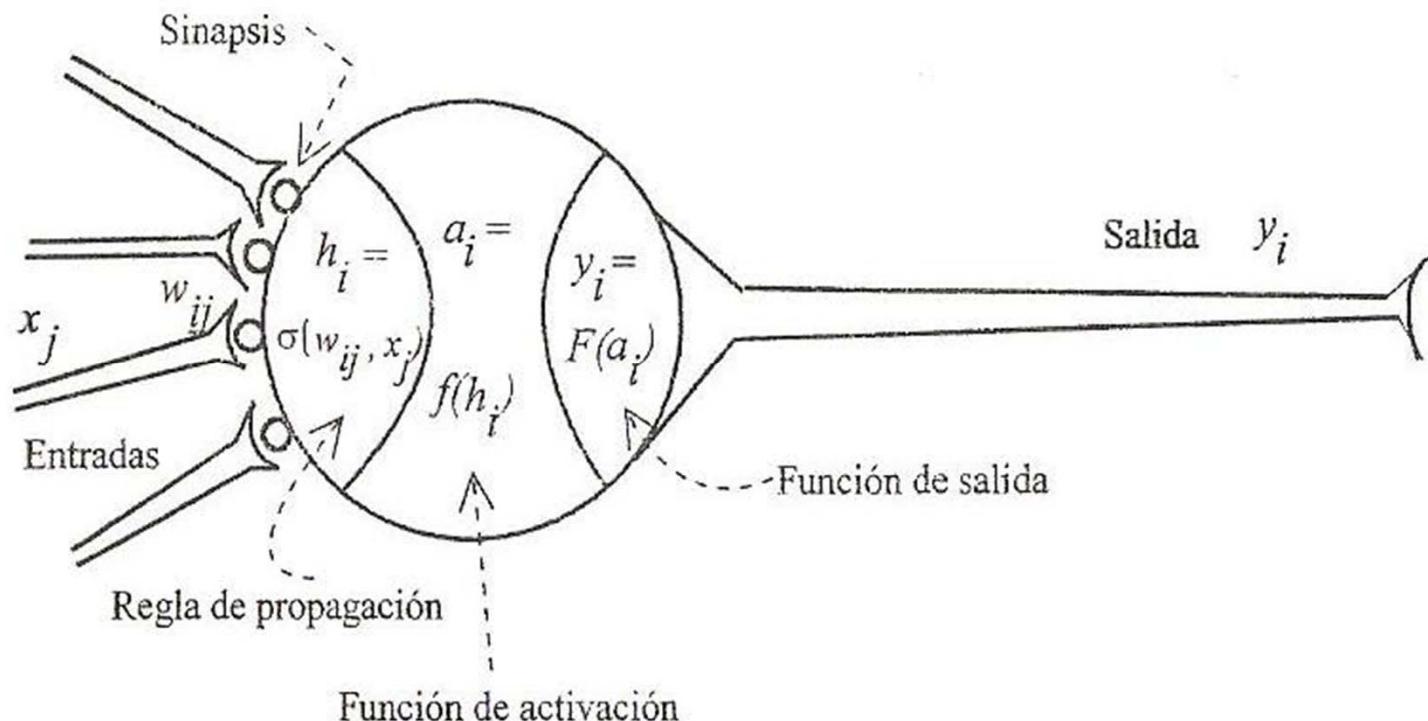


# SCx ALIMENTADOS HACIA DELANTE

- 1. La neurona artificial
- 2. Adaline
- 3. Perceptrón
  - 2.1. Estructura y aprendizaje
  - 2.2. Funciones de transferencia
  - 2.3. Entrenamiento y control de convergencia
  - 2.4. Sobreentrenamiento
- 4. Aplicaciones

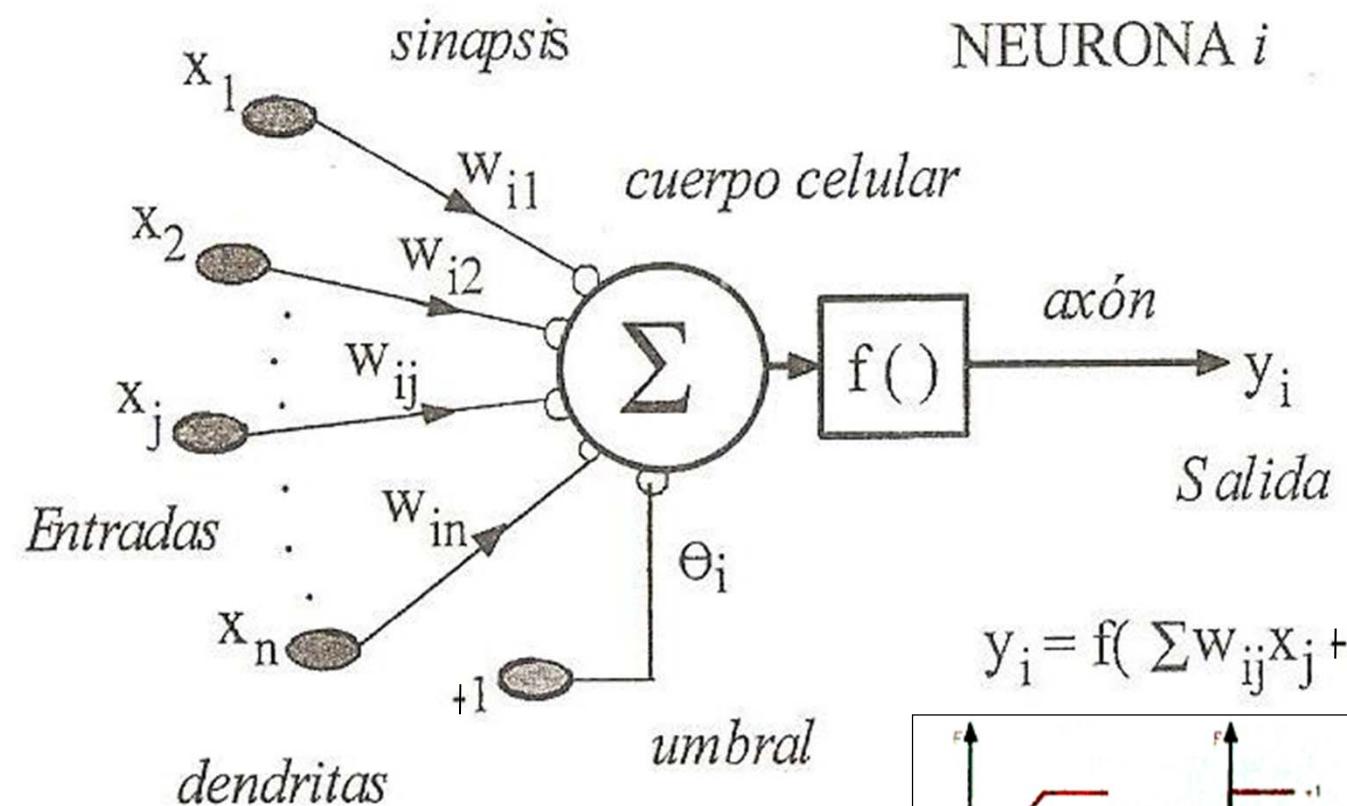
# LA NEURONA ARTIFICIAL

- Neurona i:

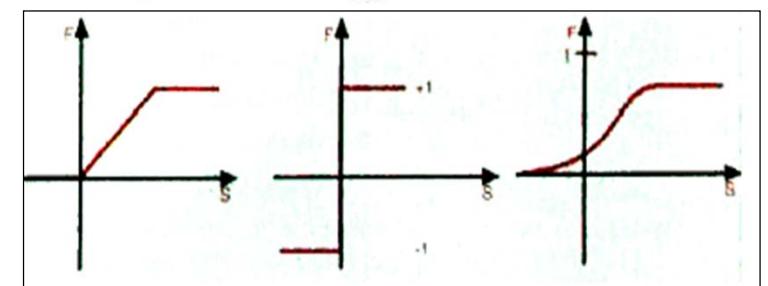


# LA NEURONA ARTIFICIAL

- Neurona i:

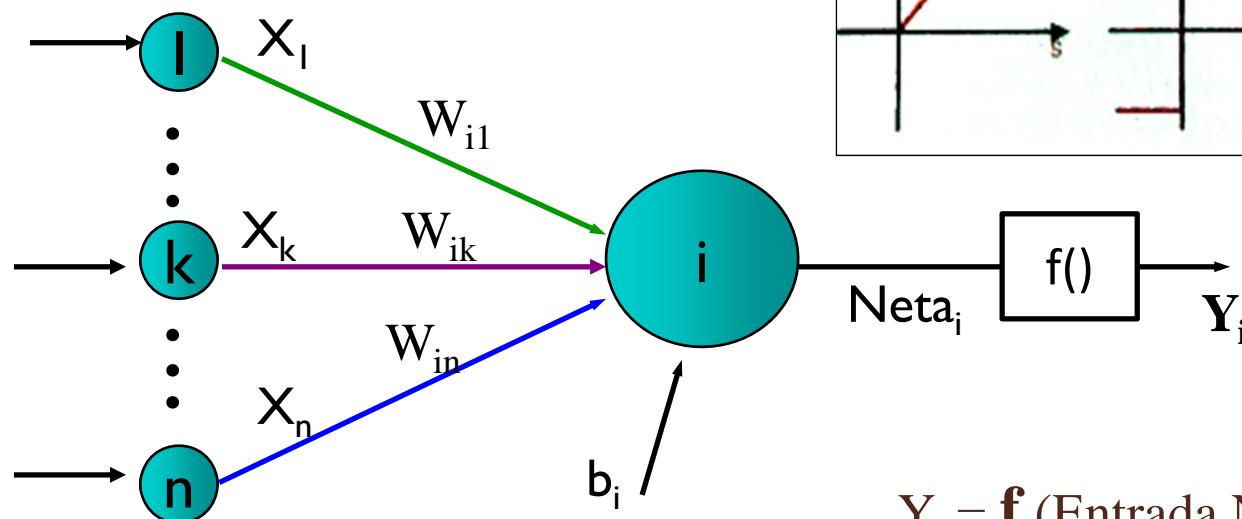


$$y_i = f( \sum w_{ij} x_j + \theta_i )$$

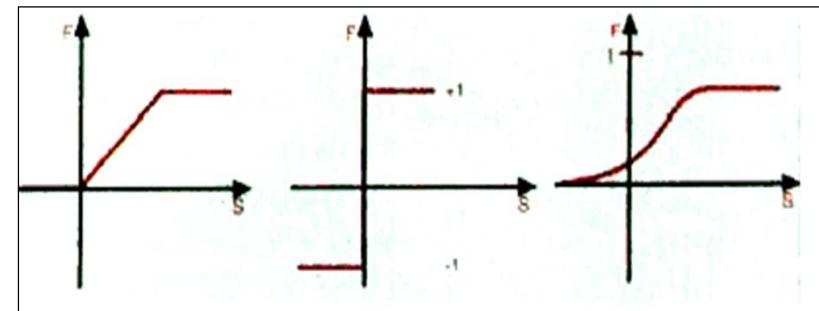


# LA NEURONA ARTIFICIAL

- Neurona i:



Función de Activación o Transferencia



$$Y_i = f(\text{Entrada Neta } i)$$

$$\text{Entrada Neta } i = \sum_{j=1}^n (X_j \ W_{ij}) + b_i$$

$$Y_i = f \left( \sum_{j=1}^n (X_j \ W_{ij}) + b_i \right)$$

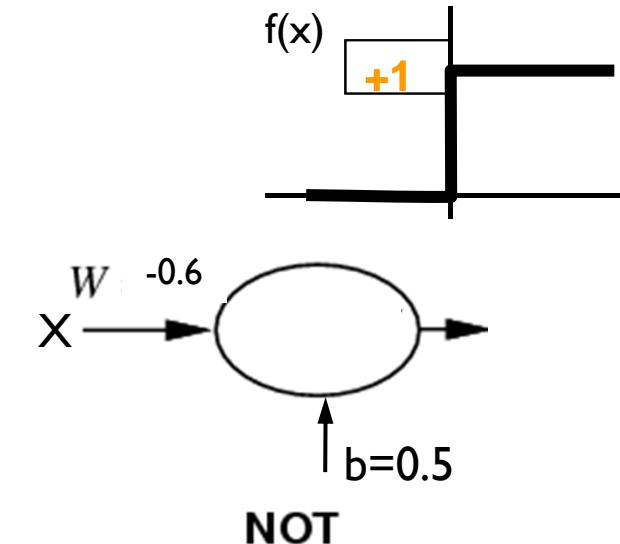
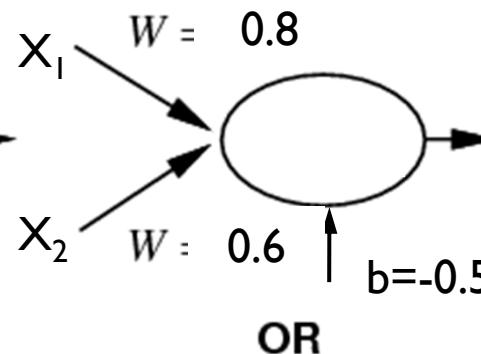
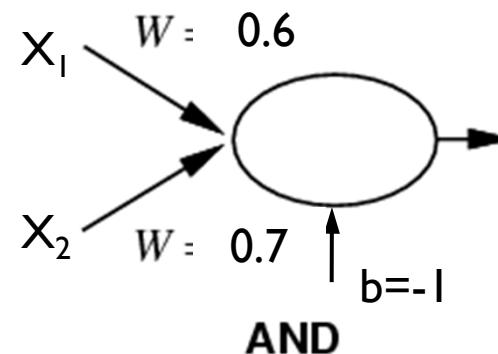


# LA NEURONA ARTIFICIAL

- También llamada Elemento de Procesado (EP o PE)
- Valores de entrada y salida:
  - Las señales de e/s de una RNA son números reales
  - Estos números deben encontrarse dentro de un intervalo
    - típicamente entre  $[0,1]$  o  $[-1,1]$
  - La codificación más simple es la binaria
- Conexiones:
  - Las unidades son conectadas a través de conexiones
  - **Codifican el conocimiento de la red**
  - Las conexiones poseen valores asociados (pesos)
  - Tipos de conexiones
    - excitatorias  $w_{ij} > 0$
    - inhibitorias  $w_{ij} < 0$
    - inexistentes  $w_{ij} = 0$
- Bias:
  - Predisposición de una neurona a activarse
  - Valor constante, similar a un peso, que no recibe entrada (o entrada=1)
- Función de transferencia:
  - Varias disponibles

# LA NEURONA ARTIFICIAL

- Ejemplo:
  - Puertas lógicas



<b><math>x_1</math></b>	<b><math>x_2</math></b>	<b><math>Y</math></b>
0	0	$f(0*0.6 + 0*0.7 - 1) = f(-1) = 0$
0	1	$f(0*0.6 + 1*0.7 - 1) = f(-0.3) = 0$
1	0	$f(1*0.6 + 0*0.7 - 1) = f(-0.4) = 0$
1	1	$f(1*0.6 + 1*0.7 - 1) = f(0.3) = 1$

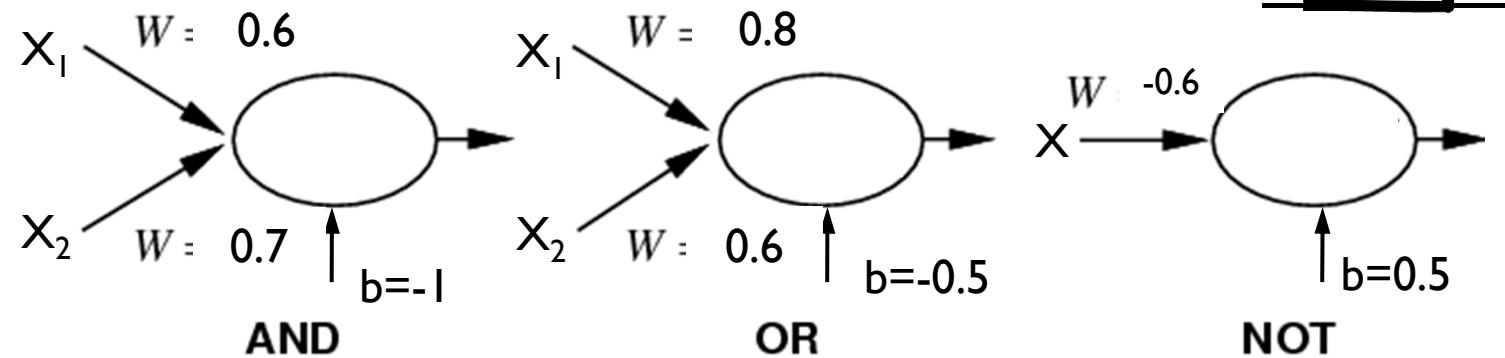
<b><math>x_1</math></b>	<b><math>x_2</math></b>	<b><math>Y</math></b>
0	0	$f(0*0.8 + 0*0.6 - 0.5) = f(-0.5) = 0$
0	1	$f(0*0.8 + 1*0.6 - 0.5) = f(0.1) = 1$
1	0	$f(1*0.8 + 0*0.6 - 0.5) = f(0.3) = 1$
1	1	$f(1*0.8 + 1*0.6 - 0.5) = f(0.9) = 1$

<b><math>x</math></b>	<b><math>Y</math></b>
0	$f(1*(-0.6) + 0.5) = f(0.5) = 1$
1	$f(1*(-0.6) + 0.5) = f(-0.1) = 0$

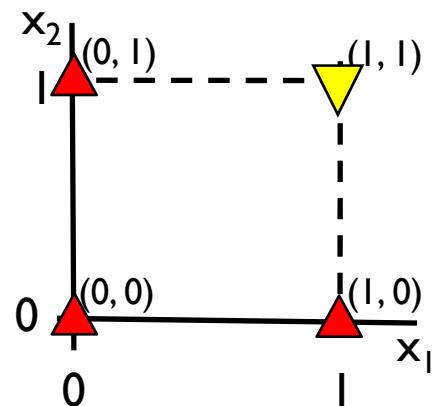
Diferentes valores de pesos y bias permiten obtener puertas lógicas

# LA NEURONA ARTIFICIAL

- Ejemplo:
  - Puertas lógicas

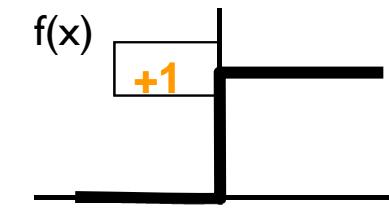
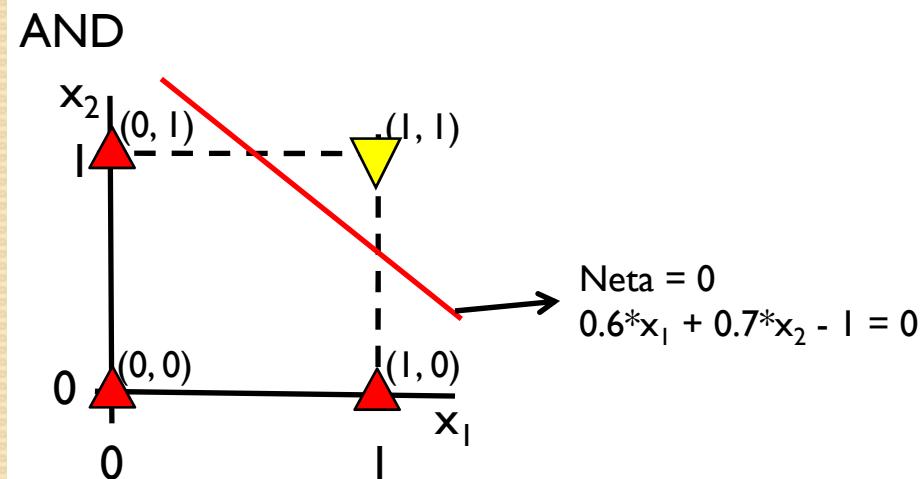
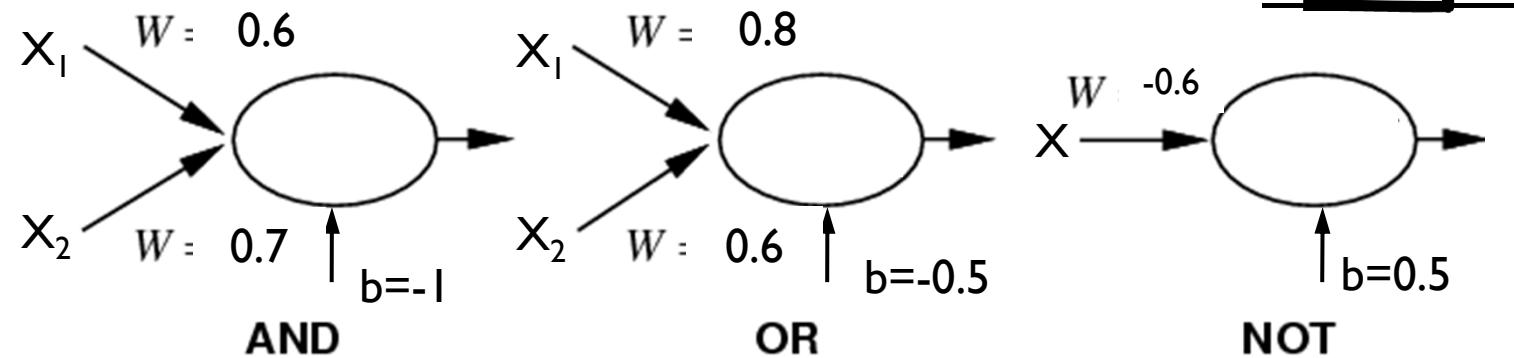


AND



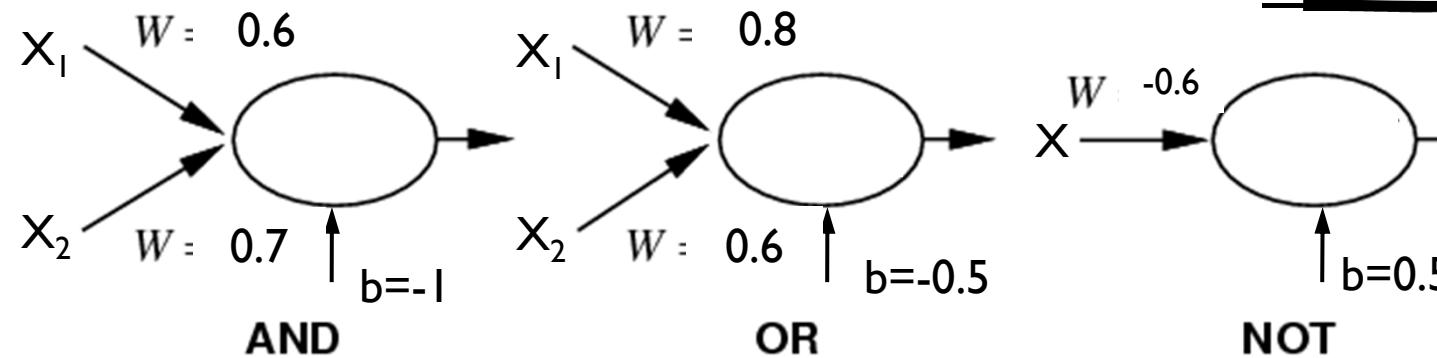
# LA NEURONA ARTIFICIAL

- Ejemplo:
  - Puertas lógicas

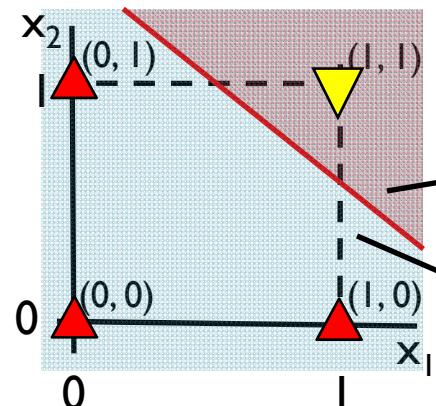


# LA NEURONA ARTIFICIAL

- Ejemplo:
  - Puertas lógicas



AND



Neta > 0

$$0.6x_1 + 0.7x_2 - 1 > 0$$

Neta < 0

$$0.6x_1 + 0.7x_2 - 1 < 0$$

$$f(\text{Neta}) = 1$$

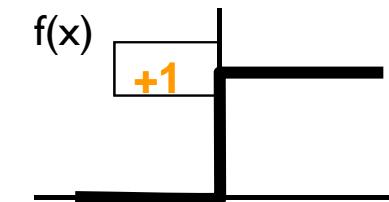
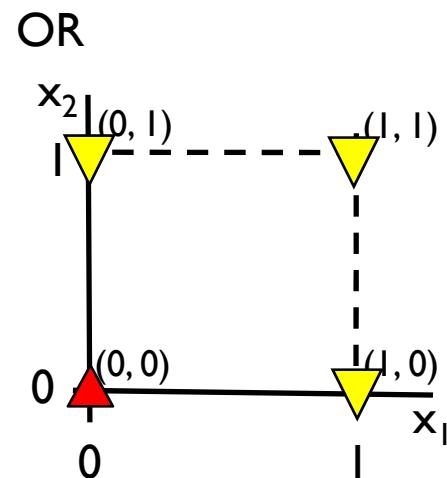
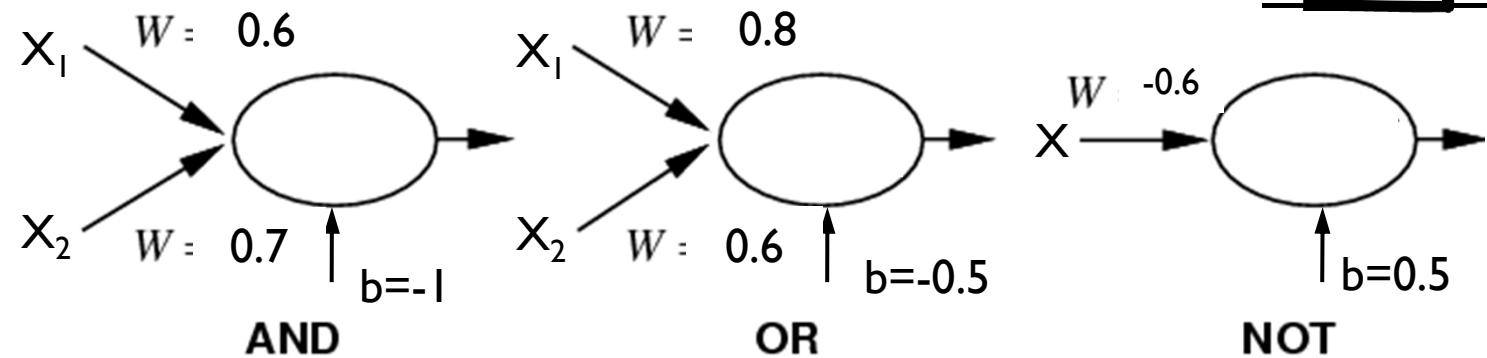
$$f(0.6x_1 + 0.7x_2 - 1) = 1$$

$$f(\text{Neta}) = 0$$

$$f(0.6x_1 + 0.7x_2 - 1) = 0$$

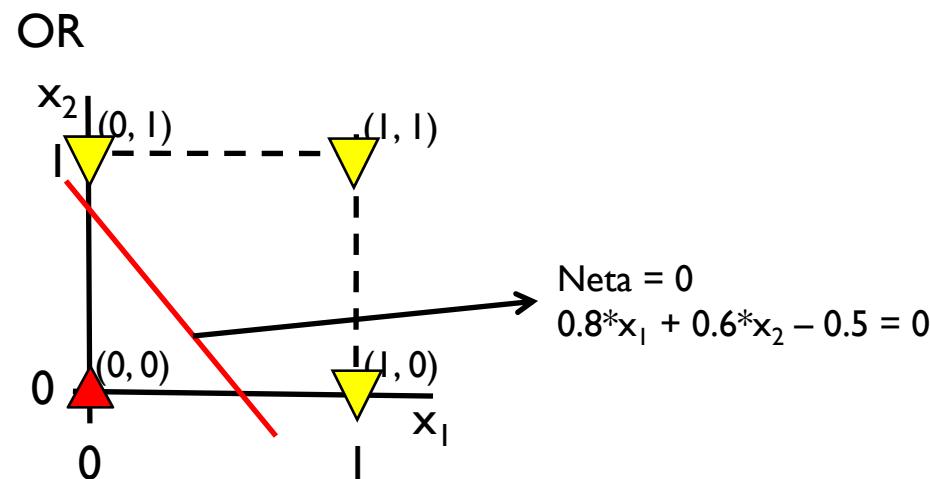
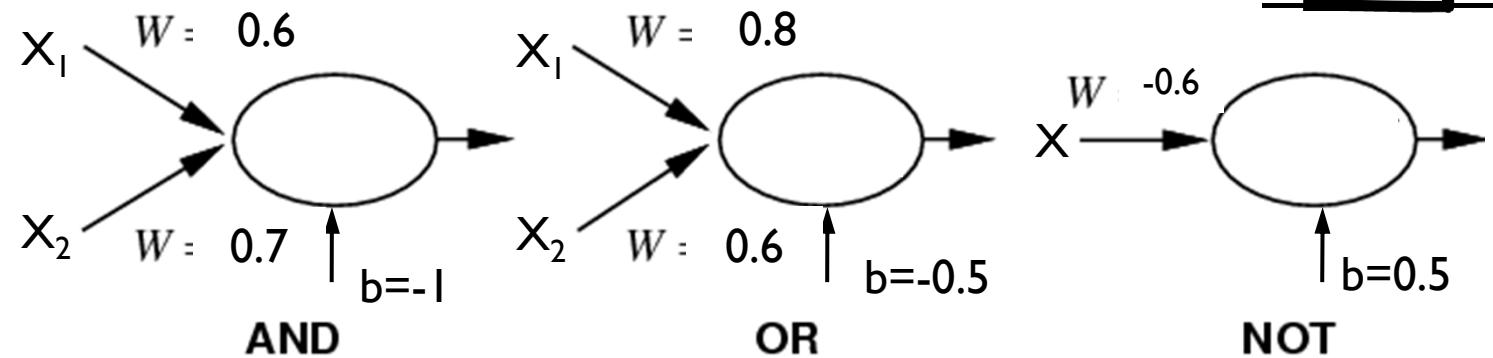
# LA NEURONA ARTIFICIAL

- Ejemplo:
  - Puertas lógicas



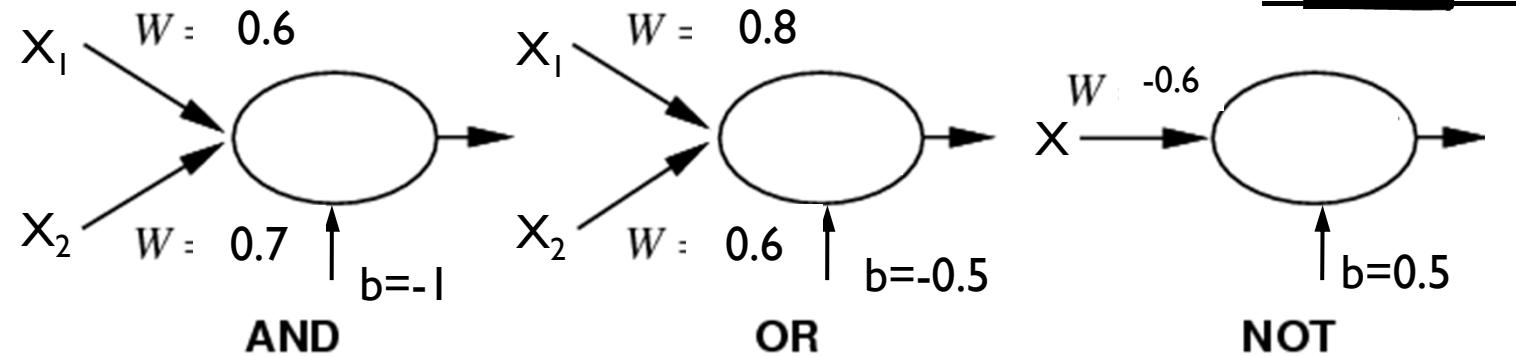
# LA NEURONA ARTIFICIAL

- Ejemplo:
  - Puertas lógicas

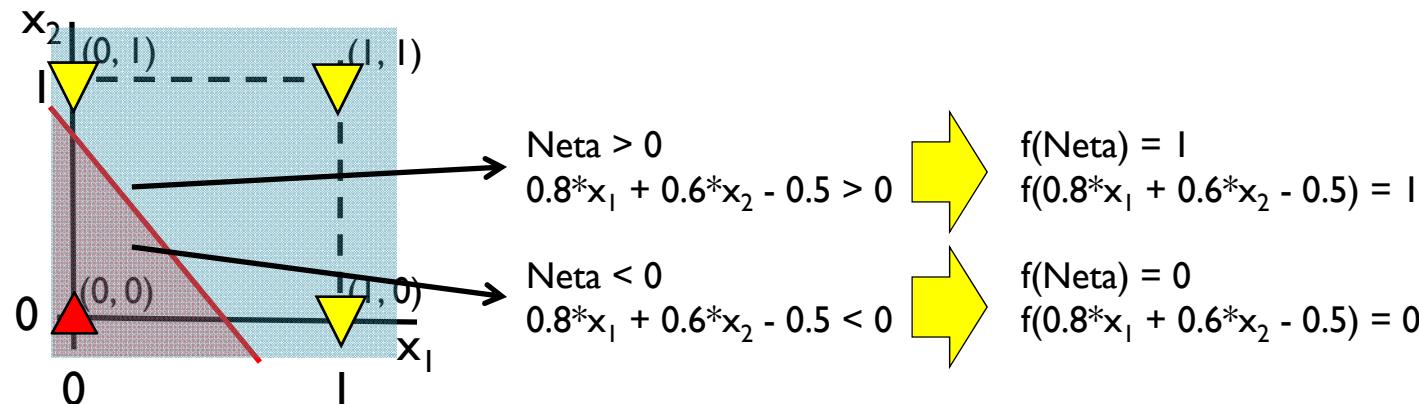


# LA NEURONA ARTIFICIAL

- Ejemplo:
  - Puertas lógicas



OR





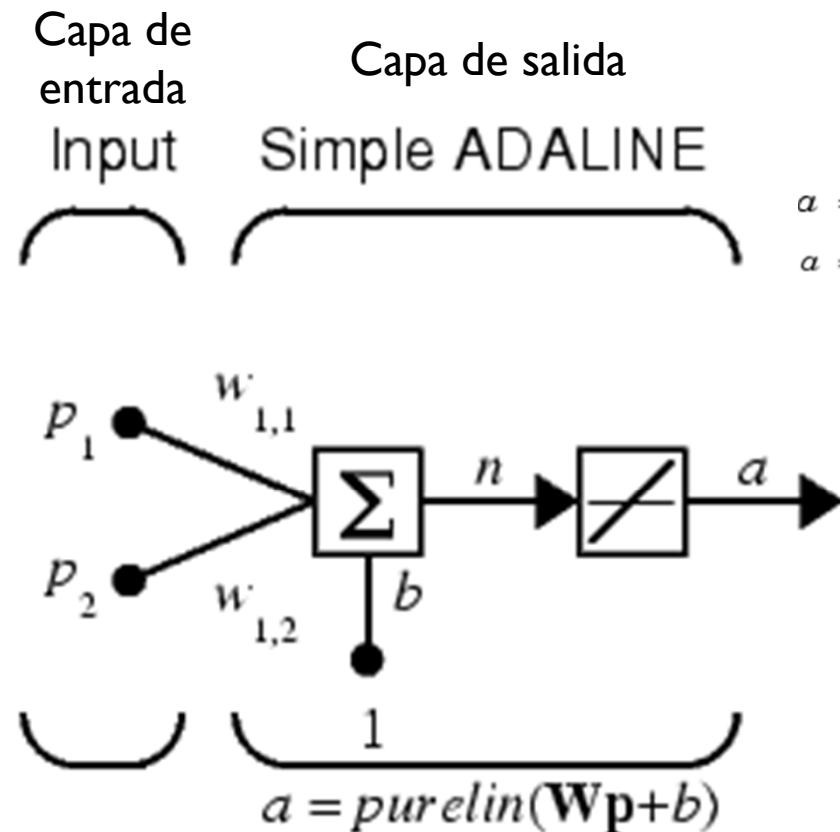
# LA NEURONA ARTIFICIAL

- ¿Cómo hacer que una neurona, ante unas determinadas entradas, emita la salida que se deseé?
  - Modificar los valores de los pesos de las conexiones y bias
  - Proceso denominado **aprendizaje o entrenamiento**
  - Se necesita un conjunto de patrones. Para cada patrón:
    - Entrada
    - Salida que debe dar la red (salida deseada)
  - A partir de estos patrones, se fija el valor de los pesos y bias
  - A este conjunto de patrones se denomina **conjunto de entrenamiento**
  - Ejemplo de conjunto de entrenamiento para conseguir una puerta lógica AND:

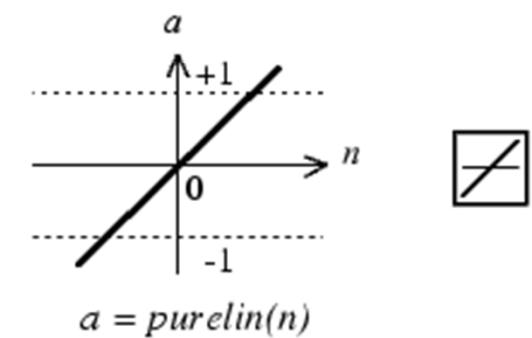
<b><math>x_1</math></b>	<b><math>x_2</math></b>	<b>Salida deseada</b>
0	0	0
0	1	0
1	0	0
1	1	1

# ADALINE

- Adaptive Linear Element
- Modelo básico de RNA



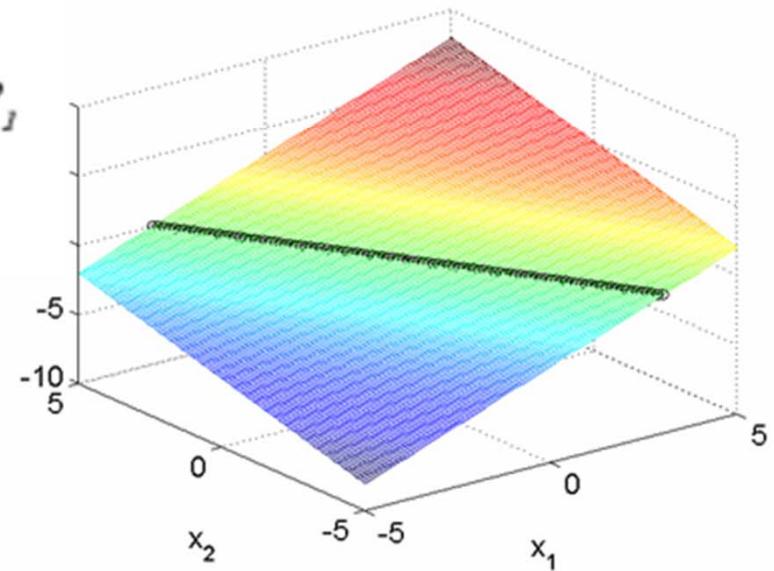
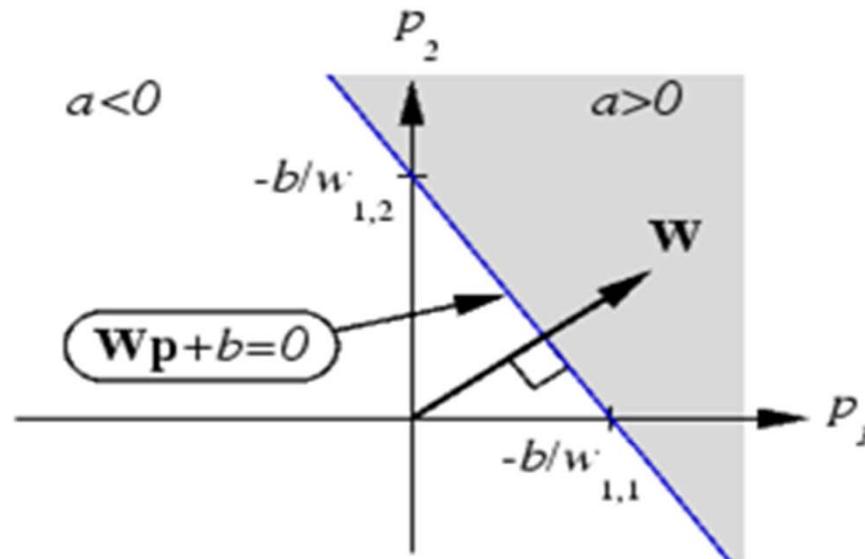
$$a = w_{1,1}p_1 + w_{1,2}p_2 + b$$
$$a = \text{purelin}(n) = \text{purelin}(\mathbf{W}\mathbf{p} + b) = \mathbf{W}\mathbf{p} + b$$

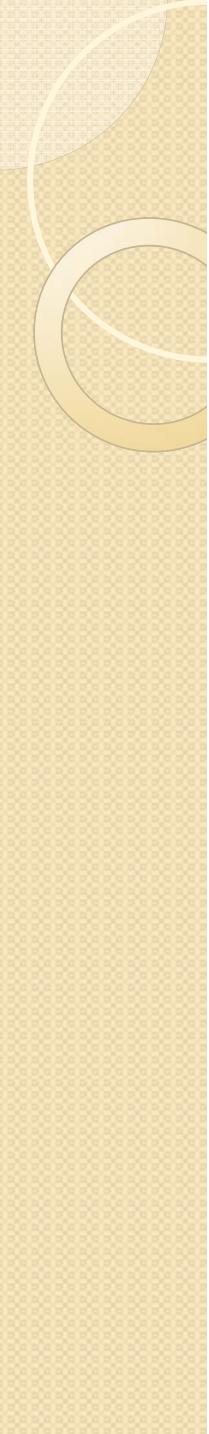


Linear Transfer Function

# ADALINE

- Adaptive Linear Element
- Modelo básico de RNA





## ADALINE

- Widrow y Hoff propusieron un método computacionalmente eficiente denominado LMS (least mean square) para determinar parámetros del Adaline (1960).
  - también llamado Regla Delta o regla de Widrow-Hoff
  - similar a aplicar gradiente descendente
  - muy intuitivo



# ADALINE

- LMS o Regla Delta:
  - Algoritmo de aprendizaje supervisado por corrección de error
    - **Aprendizaje:** fija los valores de los pesos de las conexiones y de las bias
    - **Supervisado:** necesita un “tutor” o “supervisor” que, para cada salida, diga qué error comete con respecto a la deseada
      - Para cada salida, se necesita saber cuál es el valor deseado, para calcular ese error
      - Necesarios patrones, cada uno un par de  
 $\langle$ entradas $\rangle/\langle$ salida deseada $\rangle$
    - **Por corrección de error:** minimiza el error cuadrático medio (ECM) sobre todos los patrones de entrenamiento
      - Este ECM se calcula, para cada patrón, a partir de la salida que emite la red y la salida deseada para ese patrón

# ADALINE

- LMS o Regla Delta:

- Sea el conjunto de entrenamiento ( $X, D$ )
    - $X$ : entradas (conjunto de  $L$  vectores de dimensión  $n$ )
    - $D$ : salidas deseadas (conjunto de  $L$  vectores de dimensión 1)
  - Para cada patrón  $k$ , la RNA emite una salida  $y_k$

$$y_k = \sum_{j=0}^n w_j \cdot x_j$$

- Error para el patrón  $k$ :
    - $E_k = d_k - y_k$  (salida deseada menos salida obtenida)
    - Error cuadrático (para el patrón  $k$ ):  $E_k = \frac{1}{2}(d_k - y_k)^2$
    - Error cuadrático medio (para todos los patrones):

$$E = \sum_{k=1}^L E_k = \frac{1}{2} \sum_{k=1}^L (d_k - y_k)^2$$

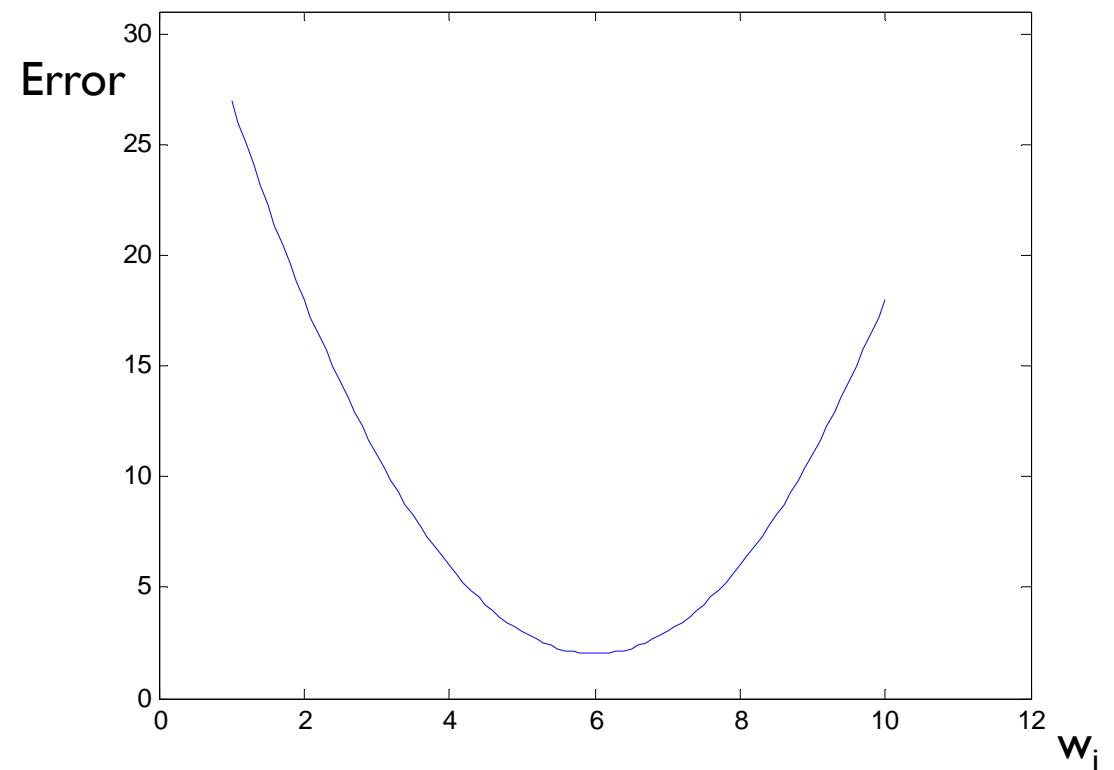


# ADALINE

- LMS o Regla Delta:
  - Para minimizar el error según W:
    - Se deriva con respecto a W
      - Minimización de gradiente
    - La superficie de error es desconocida
      - Pero se tiene información local a través del gradiente
        - Hacia dónde está el mínimo

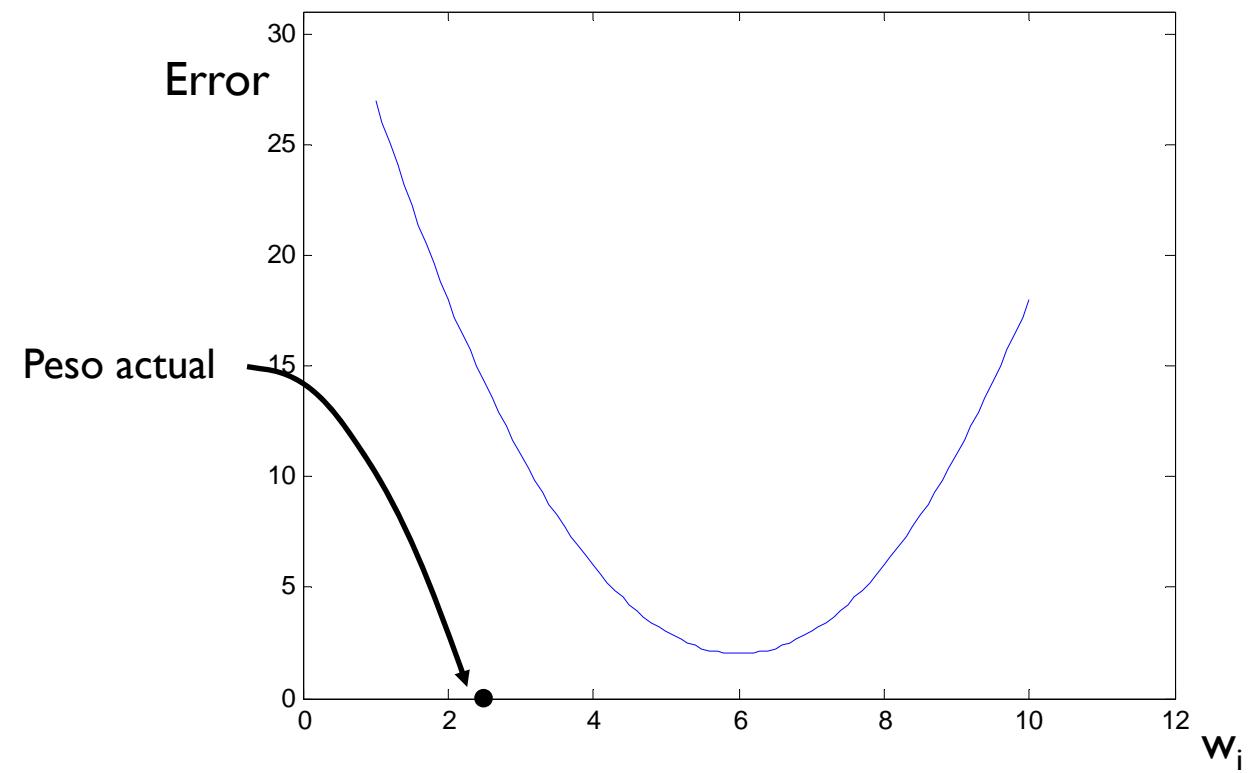
# ADALINE

- LMS o Regla Delta:
  - Gradiente descendente



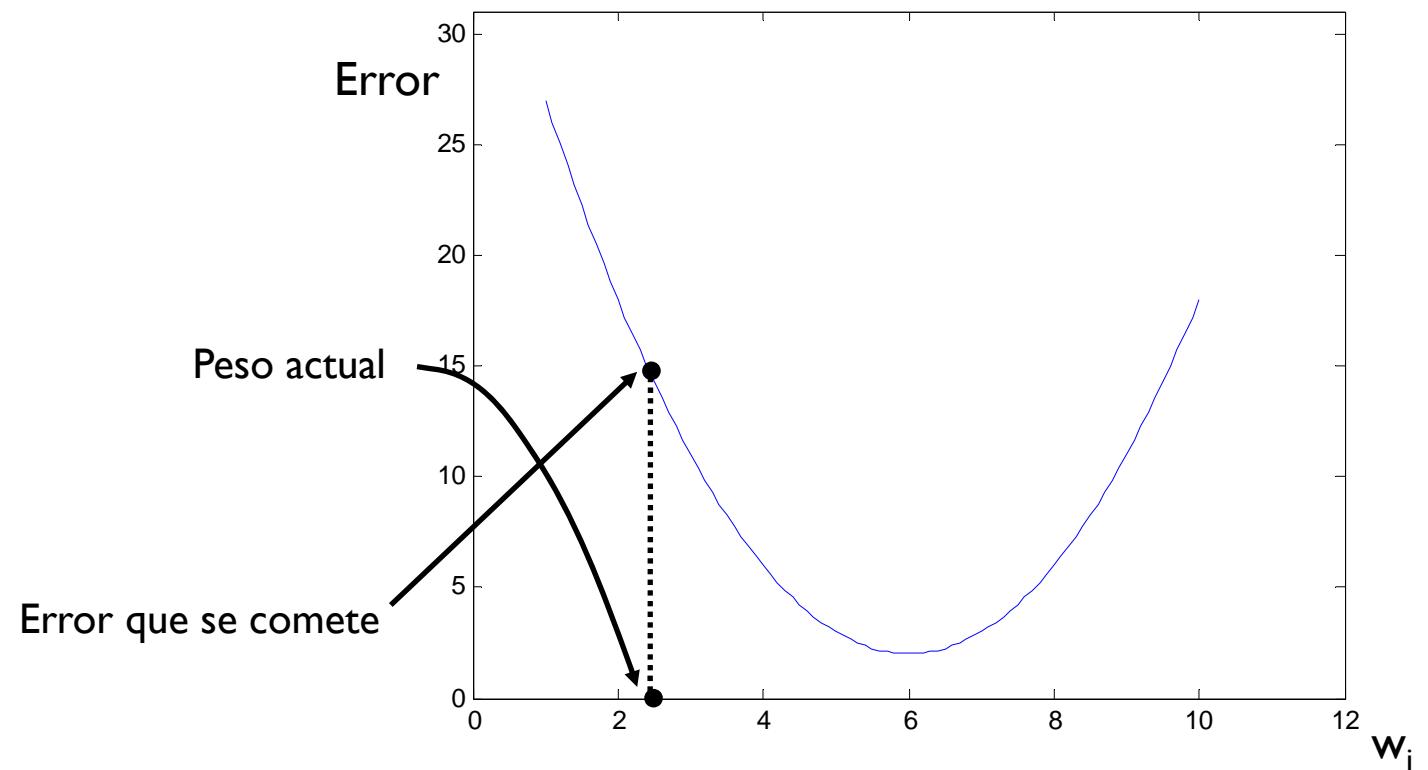
# ADALINE

- LMS o Regla Delta:
  - Gradiente descendente



# ADALINE

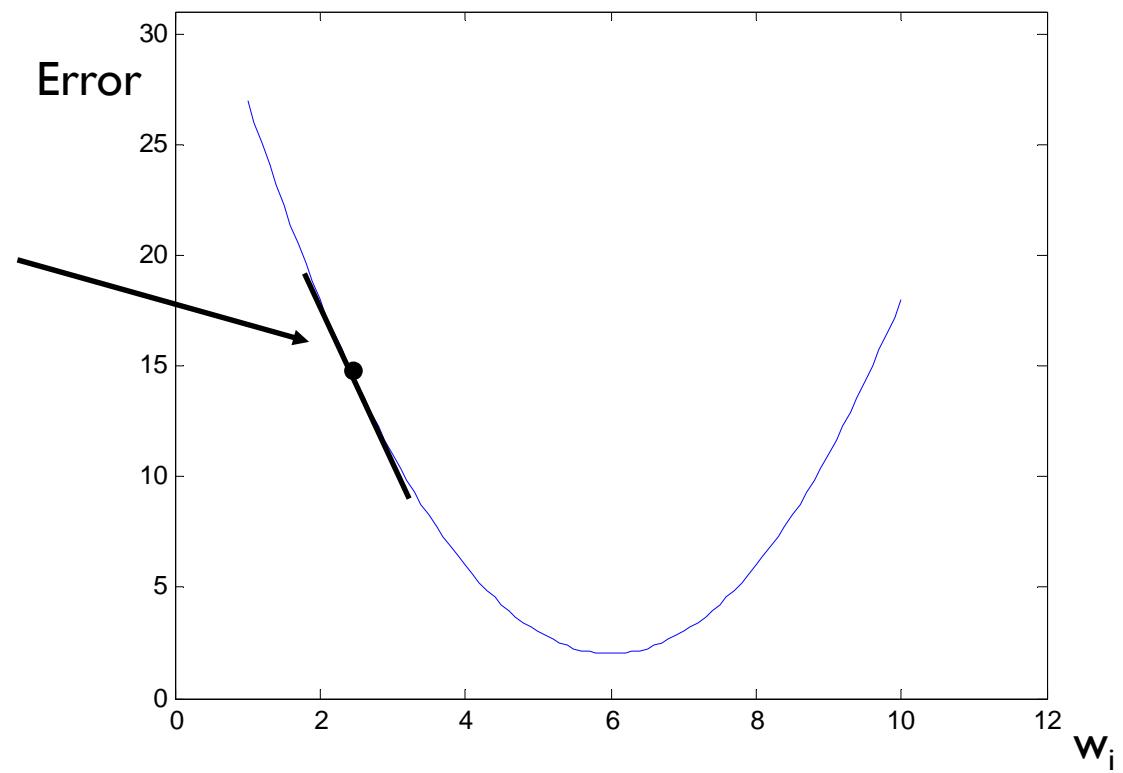
- LMS o Regla Delta:
  - Gradiente descendente



# ADALINE

- LMS o Regla Delta:
  - Gradiente descendente

Derivada en el error:  
pendiente de la tangente

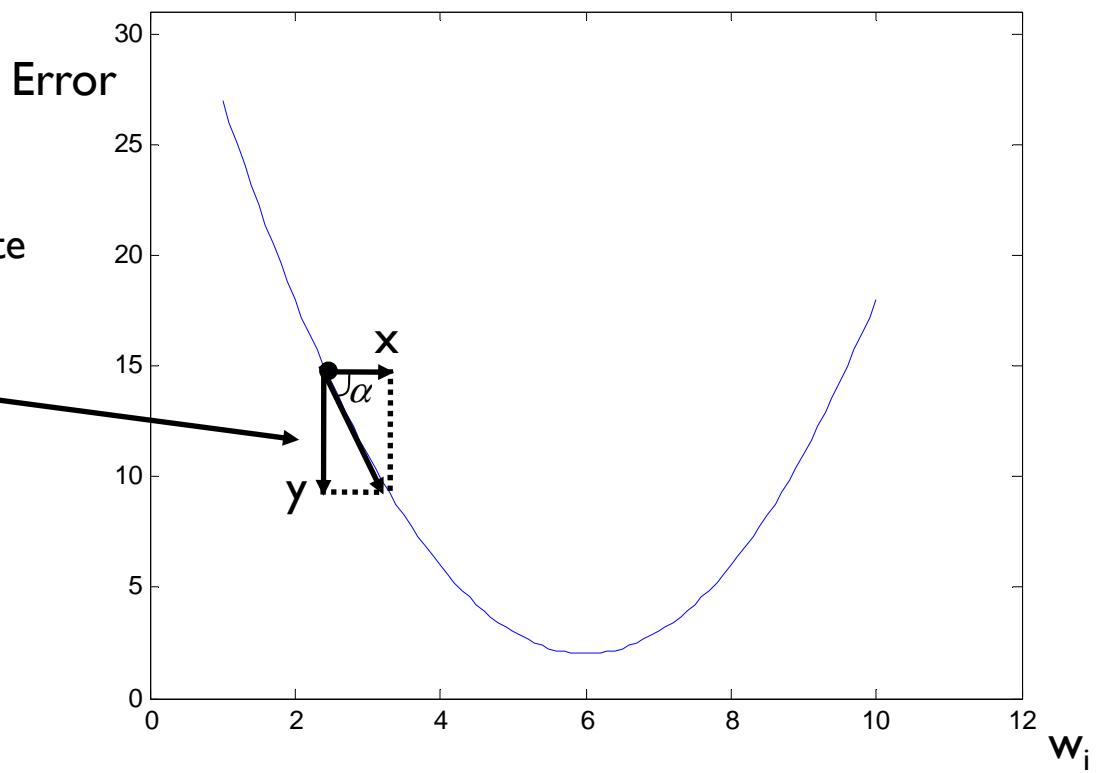


# ADALINE

- LMS o Regla Delta:
  - Gradiente descendente

Derivada en el error: pendiente de la tangente

$$\operatorname{tg}(\alpha) = \frac{y}{x}$$

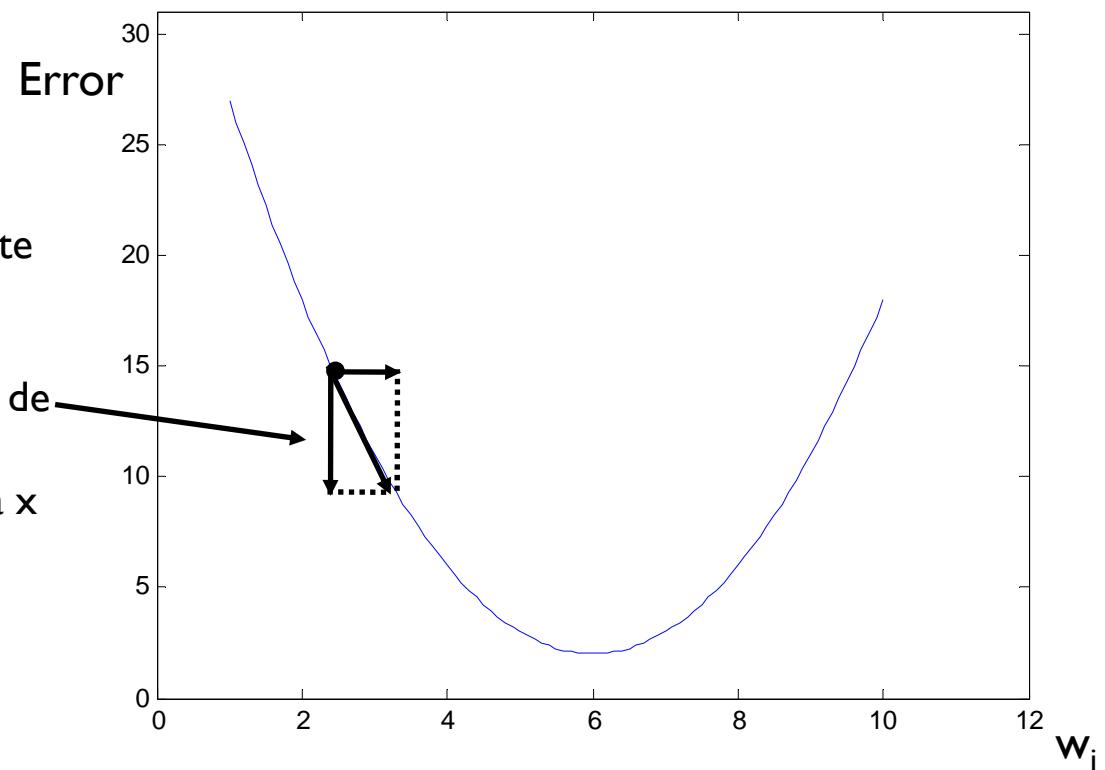


# ADALINE

- LMS o Regla Delta:
  - Gradiente descendente

Derivada en el error: pendiente de la tangente

Cuanto mayor sea la magnitud de la pendiente, mayor será la componente y en relación a la x



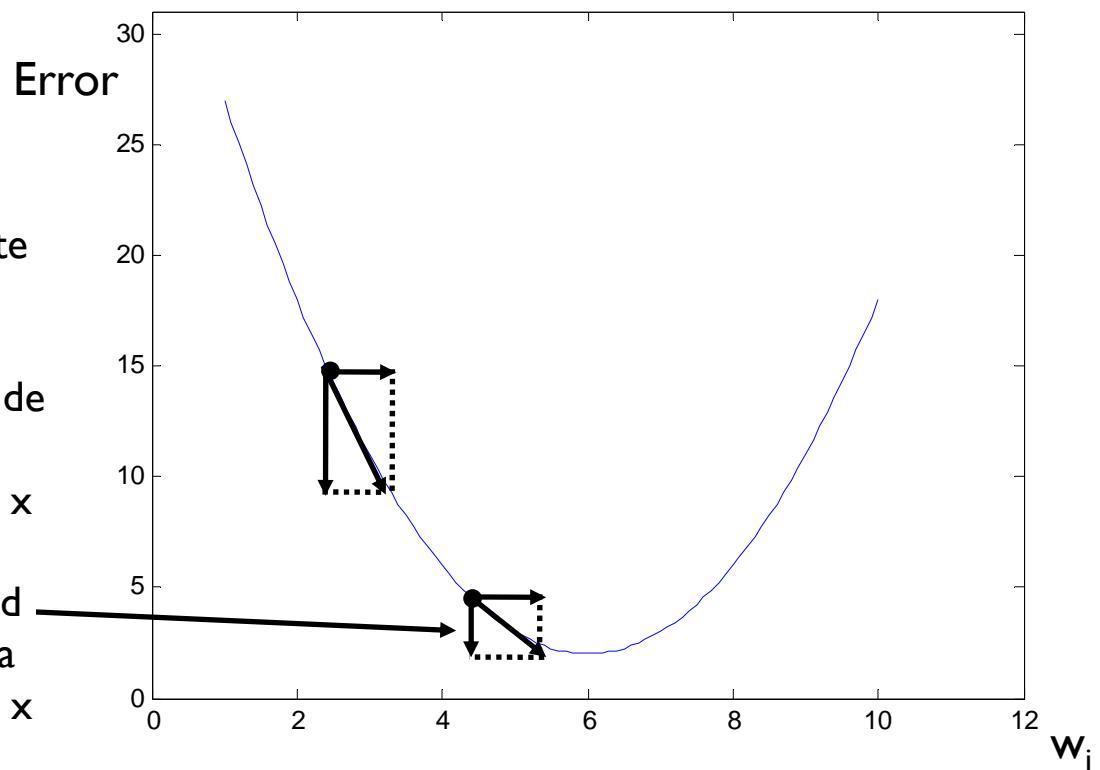
# ADALINE

- LMS o Regla Delta:
  - Gradiente descendente

Derivada en el error: pendiente de la tangente

Cuanto mayor sea la magnitud de la pendiente, mayor será la componente y en relación a la x

Cuanto menor sea la magnitud de la pendiente, menor será la componente y en relación a la x



# ADALINE

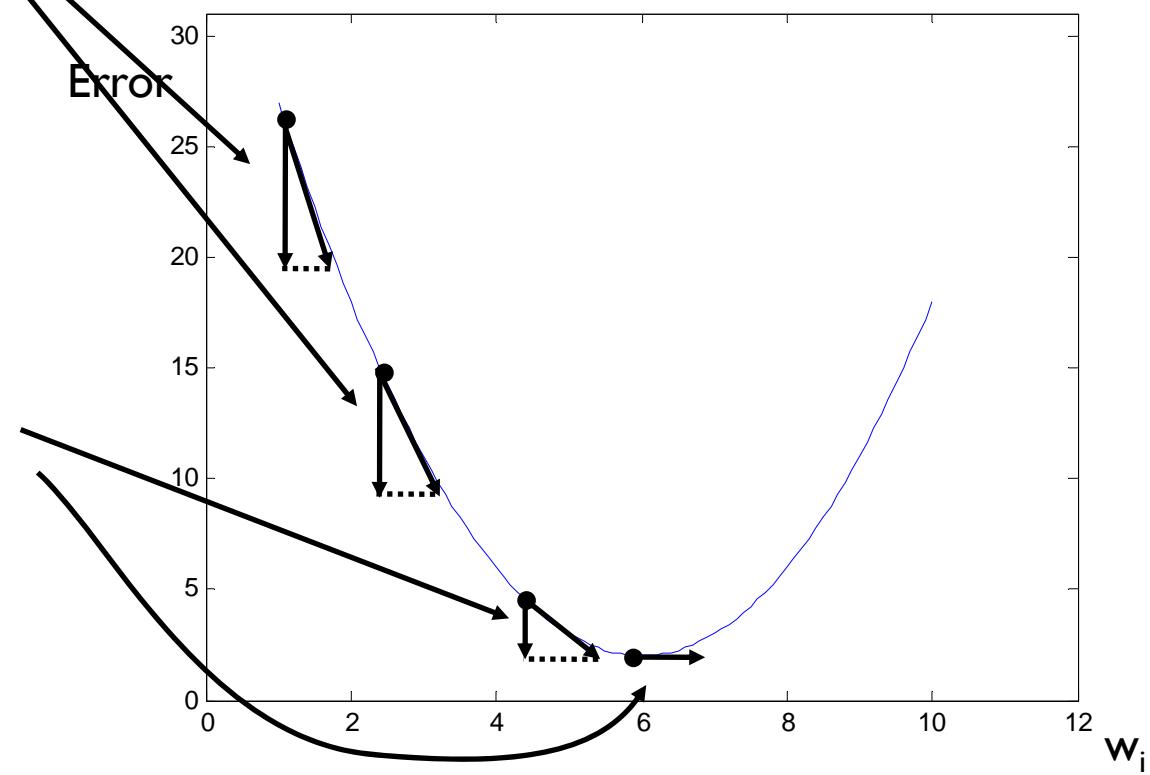
- LMS o Regla Delta:
  - Gradiente descendente

Cuanto más alejado se esté de un mínimo (o máximo), mayor será la magnitud de la pendiente

  
La modificación del peso será mayor

Cuanto más cerca se esté de un mínimo (o máximo), menor será la magnitud de la pendiente

  
La modificación del peso será menor



# ADALINE

- LMS o Regla Delta:
  - Gradiente descendente

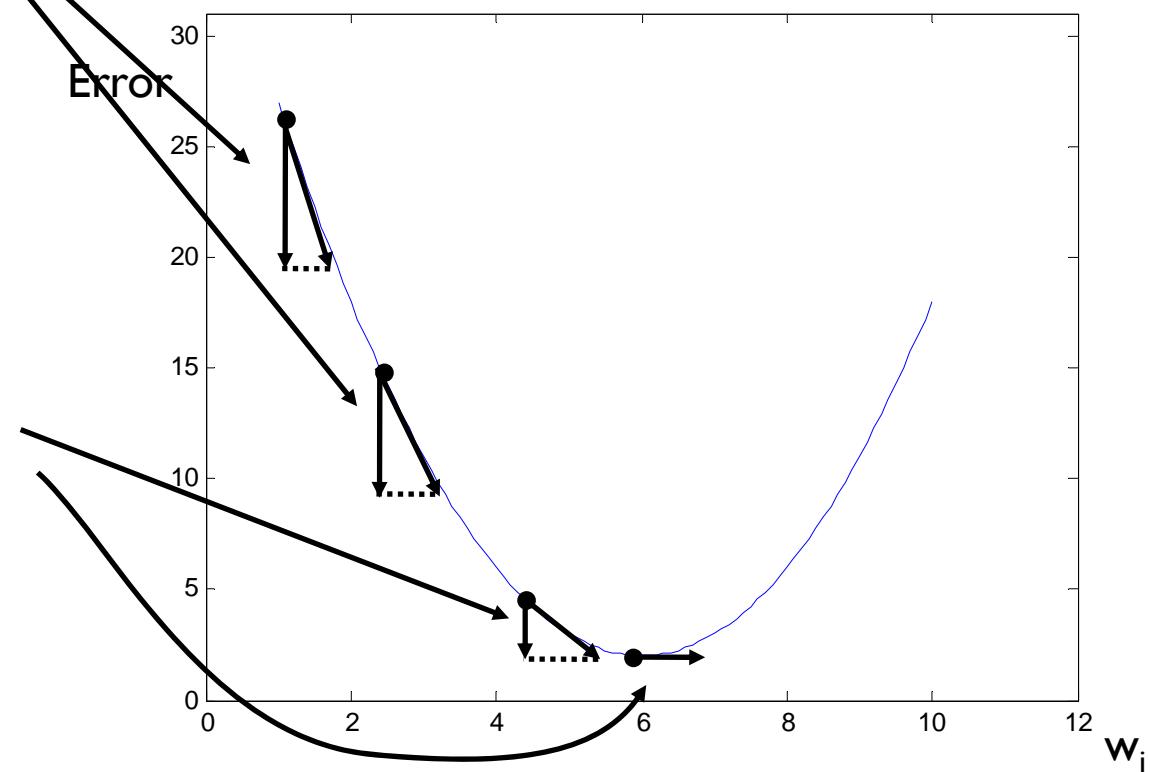
Cuanto más alejado se esté de un mínimo (o máximo), mayor será la magnitud de la pendiente

La modificación del peso será mayor

Cuanto más cerca se esté de un mínimo (o máximo), menor será la magnitud de la pendiente

La modificación del peso será menor

La modificación del peso se realiza en función de la pendiente

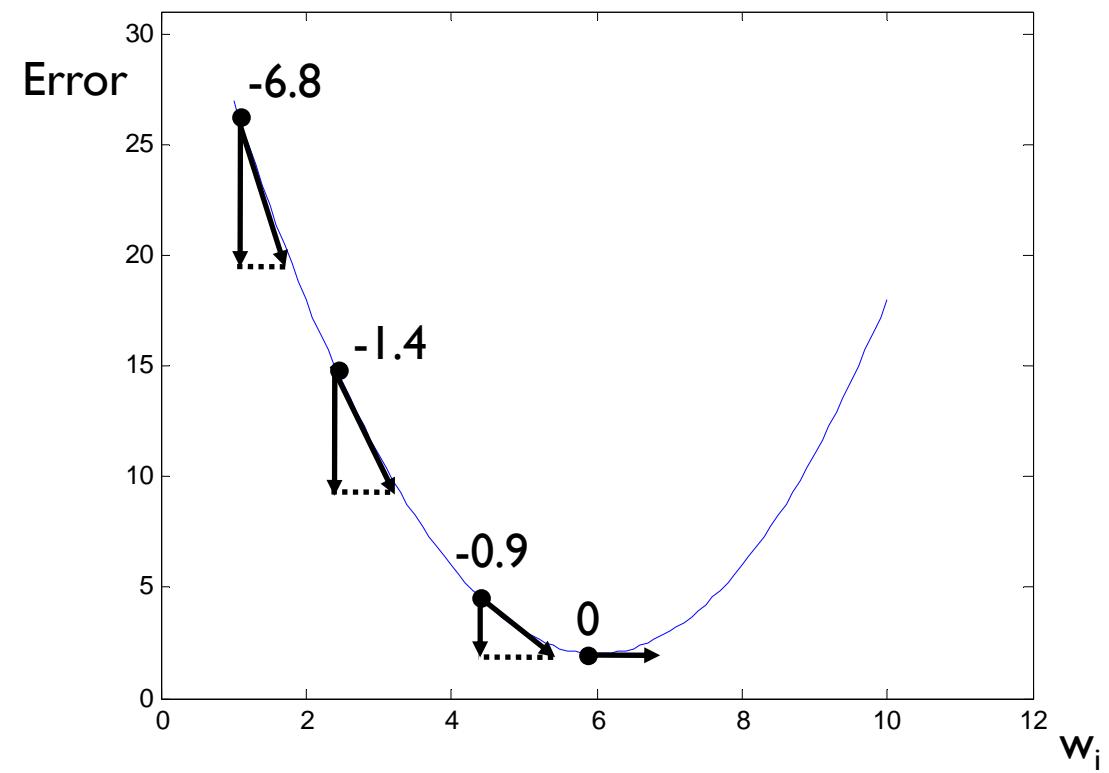


# ADALINE

- LMS o Regla Delta:
  - Gradiente descendente

La modificación del peso se realiza en función de la pendiente

Pendientes:



# ADALINE

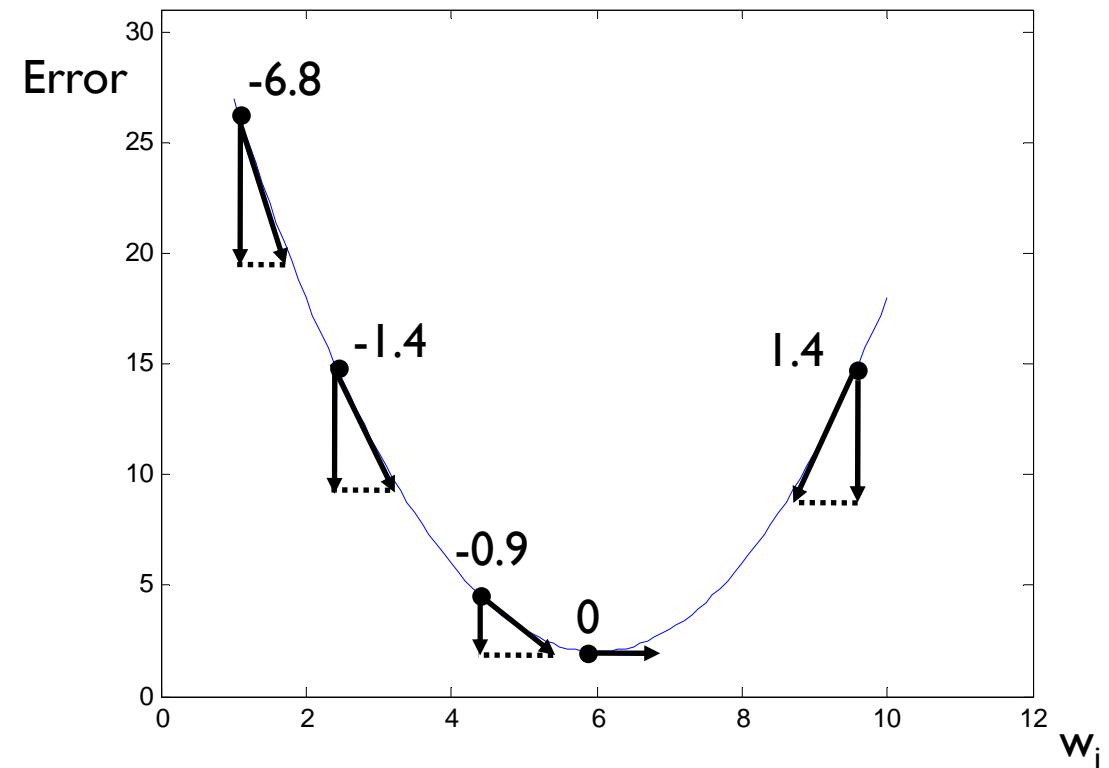
- LMS o Regla Delta:
  - Gradiente descendente

La modificación del peso se realiza en función de la pendiente

Signo de la pendiente:

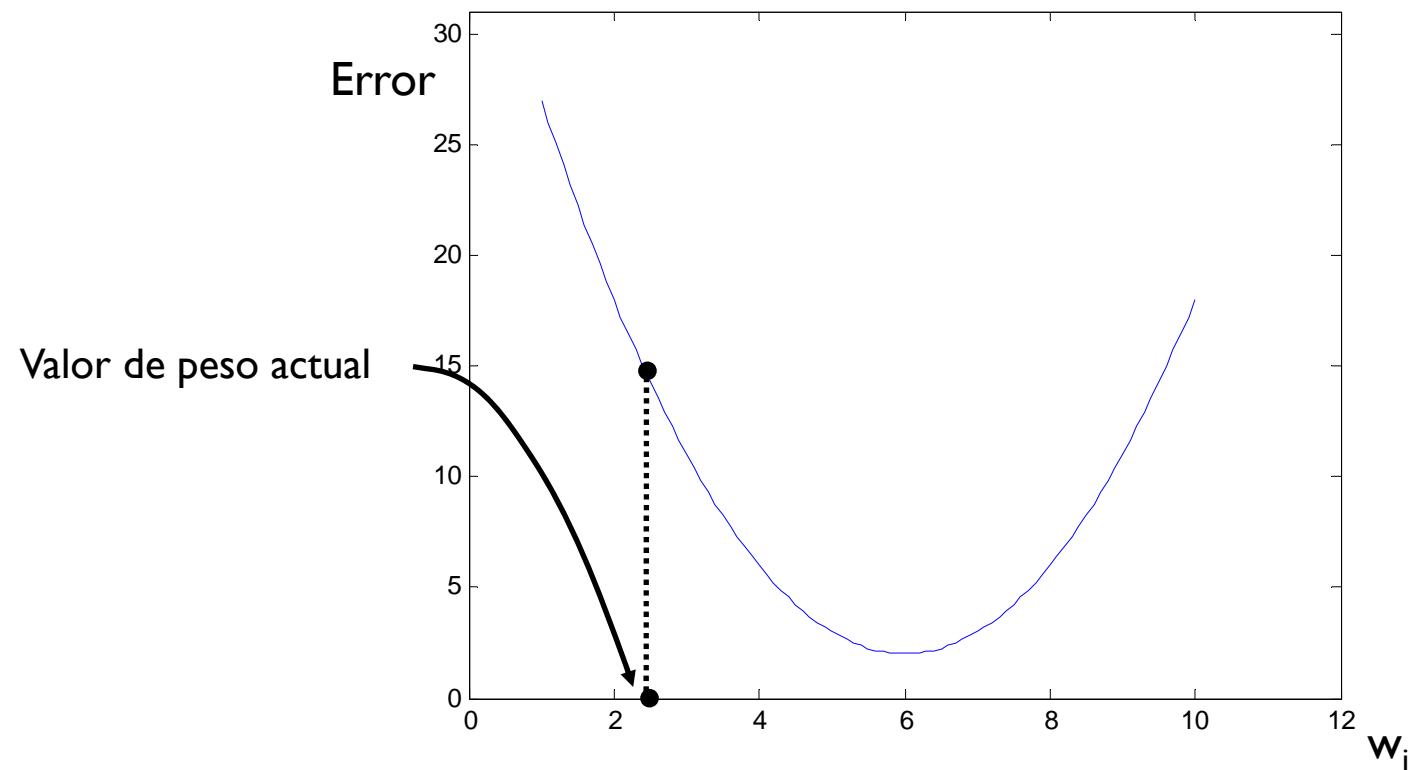
Necesario moverse en sentido contrario

Pendientes:



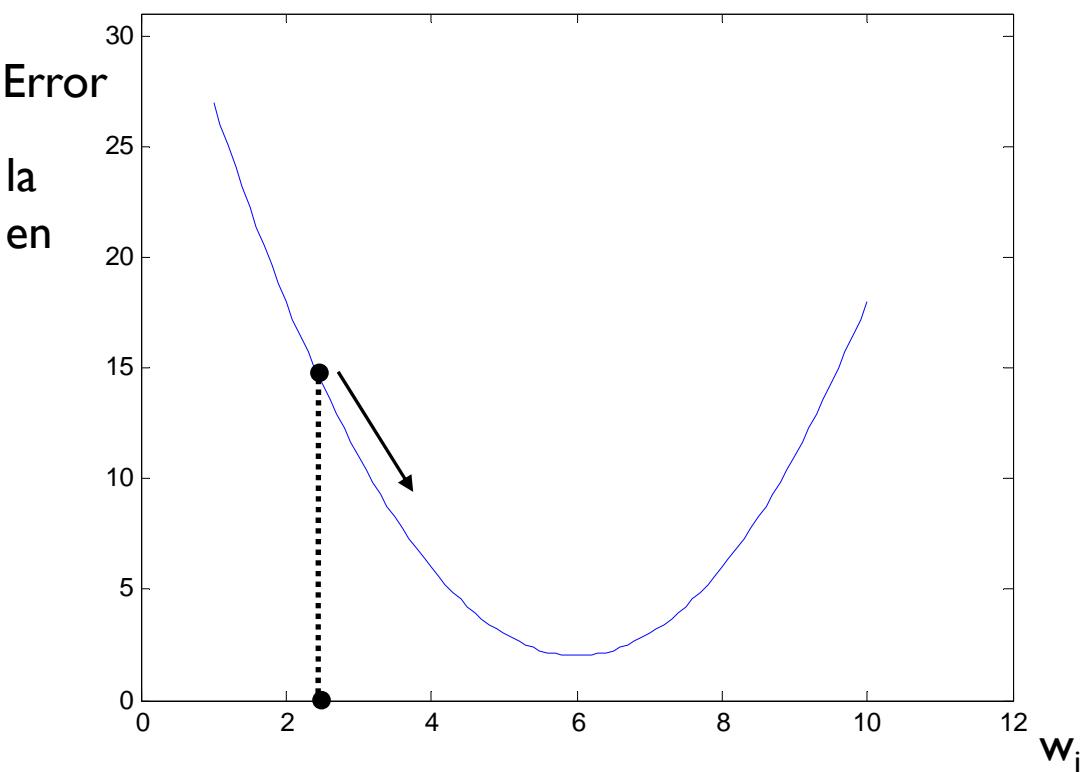
# ADALINE

- LMS o Regla Delta: Proceso:
  1. Se inicializan los pesos de forma aleatoria



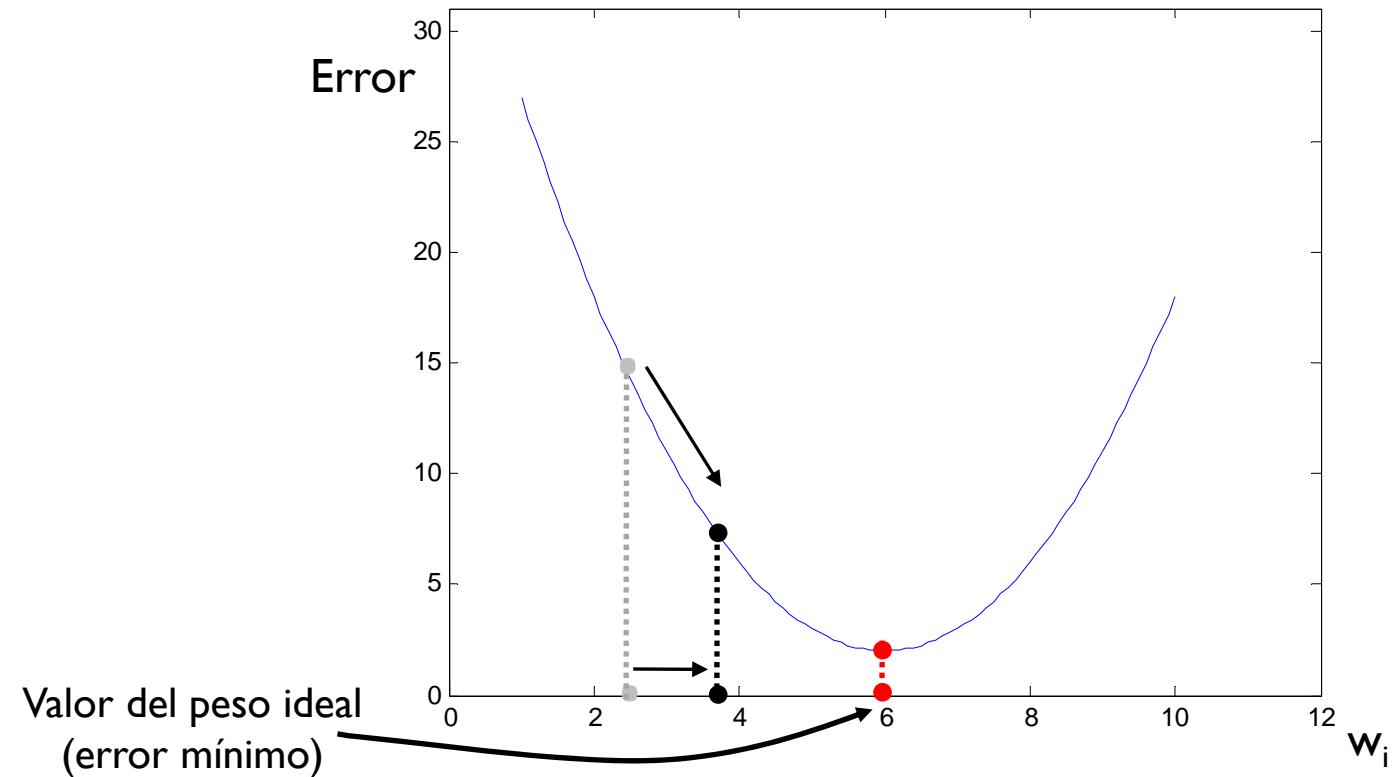
# ADALINE

- LMS o Regla Delta: Proceso:
  1. Se inicializan los pesos de forma aleatoria
  2. Se determina la dirección de la pendiente más pronunciada en dirección hacia abajo (gradiente descendente)
    - Para ello, se utiliza la derivada del error en ese punto



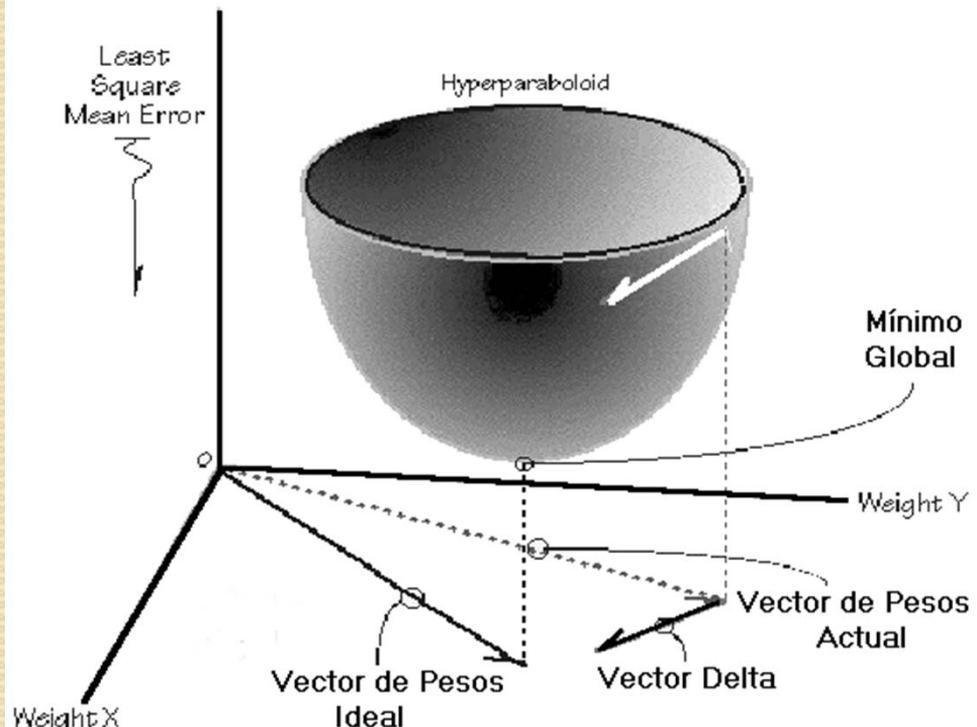
# ADALINE

- LMS o Regla Delta: Proceso:
  3. Se modifican los pesos para situarse un poco más abajo en la superficie



# ADALINE

- LMS o Regla Delta: Proceso:
  - Sea  $W(t)$  el vector de pesos en el instante  $t$ ,
    - En el instante  $t+1$ :  $W(t+1) = W(t) + \Delta W(t)$



- Magnitud del cambio  $\Delta W(t)$ :
  - Según la derivada del error

$$\frac{\nabla E}{\nabla w}$$

se aplica un cambio

# ADALINE

- LMS o Regla Delta: Proceso:
    - La idea es realizar un cambio en los pesos proporcional a la derivada del error, medida en el patrón actual k, respecto del peso j:
- $$\Delta_k w_j = -\mu \frac{\nabla E_k}{\nabla w_j}$$
- $$E_k = \frac{1}{2} (d_k - y_k)^2$$
- $$y_k = \sum_{j=0}^n w_j \cdot x_j$$
- La regla de aprendizaje del ADALINE se justifica mediante la aplicación del gradiente descendente (derivada del error):
$$\frac{\nabla E_k}{\nabla w_j} = \frac{\nabla (\frac{1}{2} (d_k - y_k)^2)}{\nabla w_j} = \frac{1}{2} \cdot \frac{\nabla ((d_k - y_k)^2)}{\nabla w_j} = 2 \cdot \frac{1}{2} \cdot (d_k - y_k) \frac{\nabla (d_k - y_k)}{\nabla w_j} =$$
$$= -(d_k - y_k) \frac{\nabla y_k}{\nabla w_j} = -(d_k - y_k) \frac{\nabla \sum_i w_i \cdot x_{ik}}{\nabla w_j} = -(d_k - y_k) \frac{\nabla (w_j \cdot x_{jk})}{\nabla w_j} = -(d_k - y_k) x_{jk}$$

# ADALINE

- LMS o Regla Delta: Proceso:
  - Si los pesos se mueven en dirección  $(y-d)x$ ,  
(dada por  $\frac{\nabla E_k}{\nabla w_j}$ ), se incrementa el error
  - Por tanto, hay que moverse en dirección  $(d-y)x$   
(dada por  $-\frac{\nabla E_k}{\nabla w_j}$ ) para decrementar el error
  - Por lo tanto, la modificación de pesos viene dada por:

$$\Delta_k w_j = -\mu \frac{\nabla E_k}{\nabla w_j} \quad \Rightarrow \quad \Delta_k w_j = \mu \cdot (d_k - y_k) \cdot x_{jk}$$
$$\frac{\nabla E_k}{\nabla w_j} = -(d_k - y_k) \cdot x_{jk}$$

# ADALINE

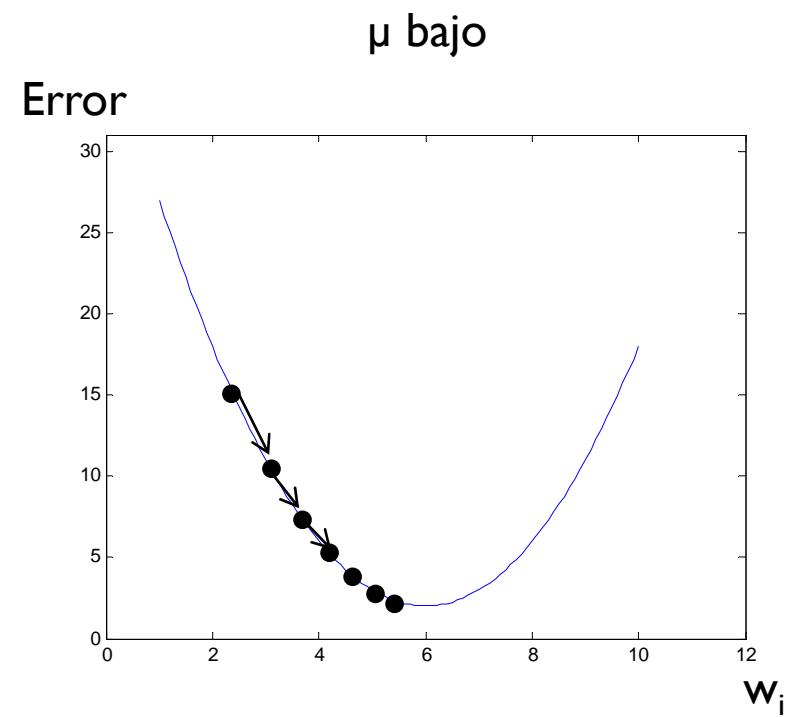
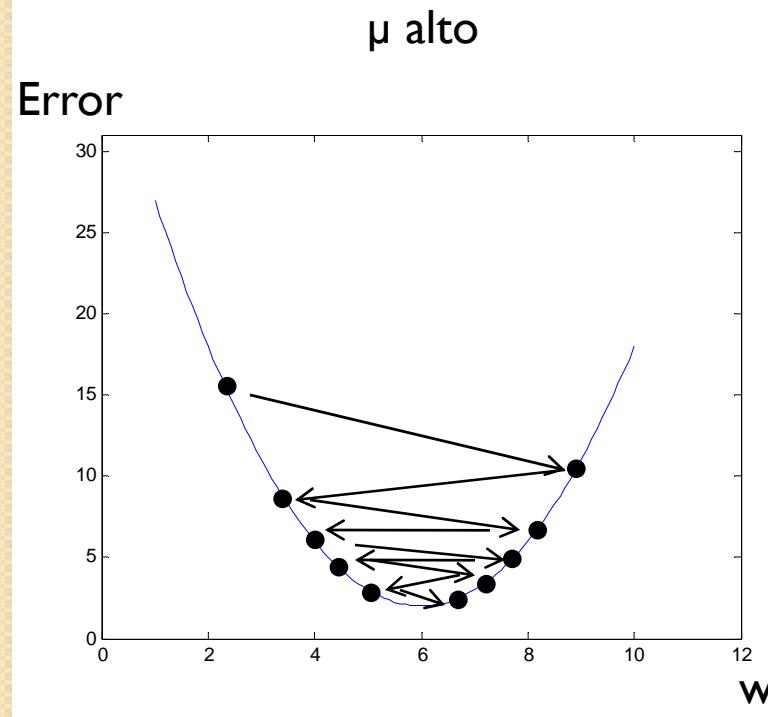
- LMS o Regla Delta: Proceso:
  - Modificación del peso j, para un patrón k, en el instante t:

$$w_j(t+1) = w_j(t) + \mu \cdot (d_k - y_k) \cdot x_{jk}$$

$\mu$ : tasa o velocidad de aprendizaje

# ADALINE

- LMS o Regla Delta:
  - Tasa de aprendizaje:
    - Controla la convergencia del entrenamiento





# ADALINE

- LMS o Regla Delta: Algoritmo:
  1. Se inicializan los pesos de forma aleatoria
  2. Se aplica un patrón de entrada
  3. Se computa la salida que emite la red
  4. Se calcula el error cometido para dicho patrón
  5. Se actualizan las conexiones mediante la ecuación anterior
  6. Se repiten los pasos del 2 al 5 para todos los patrones de entrenamiento
  7. Si el ECM es un valor aceptable, se termina el proceso
    - Si no, se vuelve al paso 2
- Variante: calcular el ECM sobre todos los patrones y actualizar los pesos según ese error
  - Es decir, se actualizan los pesos una vez introducidos todos los patrones
  - De esta manera, el algoritmo no es dependiente del orden de introducción de los patrones

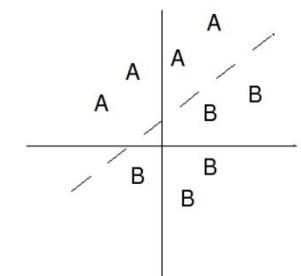
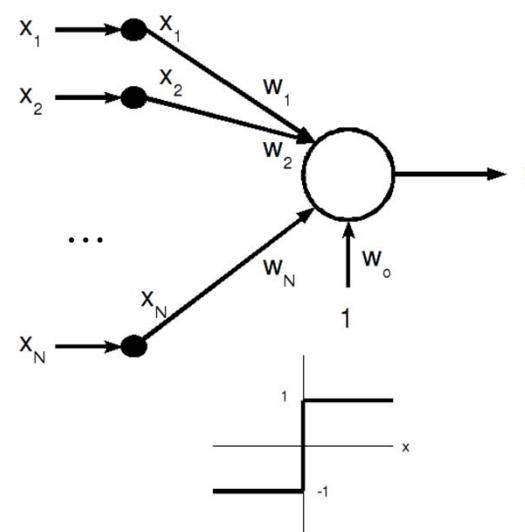


# ADALINE

- Aplicaciones:
  - Procesado de señales
    - Diseño de filtros capaces de eliminar ruido en señales portadores de información
    - RNA de predicción de valores futuros en señales
- Conclusiones:
  - ADALINE: una sola capa de PE lineales pueden realizar aproximaciones a funciones lineales o asociación de patrones
  - Se puede entrenar con el algoritmo LMS
    - Si hay más de una neurona, se puede aplicar LMS sobre todas de forma sucesiva
  - Limitaciones: sólo aproximaciones lineales

# PERCEPTRÓN

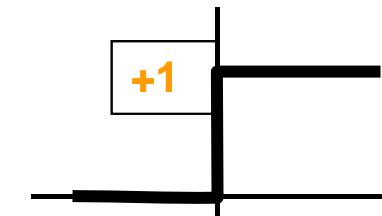
- Primer modelo de RNA desarrollado por Rosenblatt en 1958
  - Anterior al ADALINE
- Gran interés en los años 60, por su capacidad para aprender a clasificar patrones
- Estructura: capa de entrada y un único elemento en la capa de salida
- Diferencias con ADALINE:
  - Función de transferencia umbral (*hardlimiter*)
  - Algoritmo de entrenamiento



# PERCEPTRÓN

- Capacidad de representación bastante limitada: un único EP en la capa de salida
- Capaz de resolver problemas linealmente separables.
  - Entradas bidimensionales: separación a través de una línea
  - Entradas tridimensionales: separación a través de un plano
  - Entradas n-dimensionales: separación a través de un hiperplano
- Ejemplo: función OR:

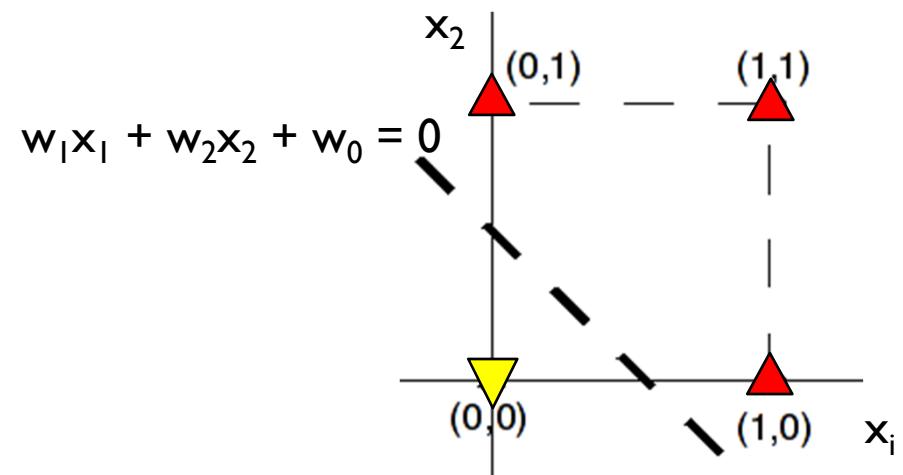
X <sub>1</sub>	X <sub>2</sub>	Y
0	0	0
0	1	1
1	0	1
1	1	1



- Salida de la red:  $y = f(w_0 + \sum_i w_i x_i) = f(w_0 + w_1 x_1 + w_2 x_2)$
- Si  $w_0 + w_1 x_1 + w_2 x_2 > 0$  la salida será 1
  - En caso contrario, la salida será 0

# PERCEPTRÓN

- El sumatorio que se le pasa como entrada a la función de transferencia representa la recta que separará las dos clases.
  - $w_1x_1 + w_2x_2 + w_0 = 0$
- Representados los patrones según las entradas:
  - 4 patrones:
    - (0,0) (0,1) (1,0) (1,1)





# PERCEPTRÓN

- Reglas para la actualización de pesos:

- 1.  $w_i(t + 1) = w_i(t)$  cuando la salida es correcta

- $w_i(t + 1) = w_i(t) + x_i(t)$  si la salida es -1 y debería ser 1

- $w_i(t + 1) = w_i(t) - x_i(t)$  si la salida es 1 y debería ser -1

- 2.  $w_i(t + 1) = w_i(t)$  cuando la salida es correcta.

- $w_i(t + 1) = w_i(t) + \mu x_i(t)$  si la salida es -1 y debería ser 1

- $w_i(t + 1) = w_i(t) - \mu x_i(t)$  si la salida es 1 y debería ser -1

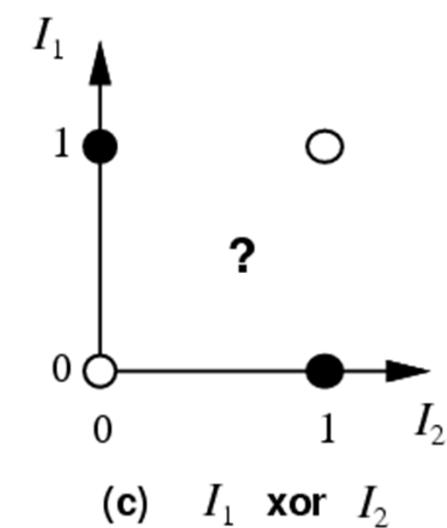
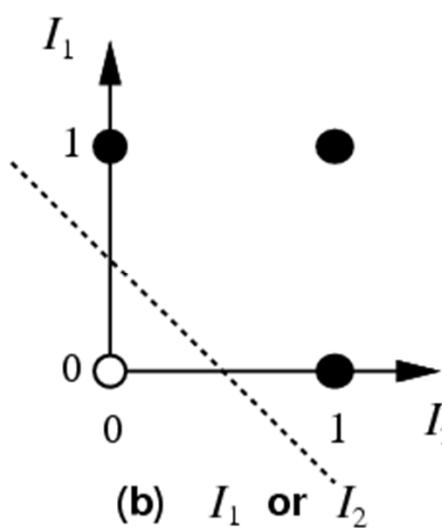
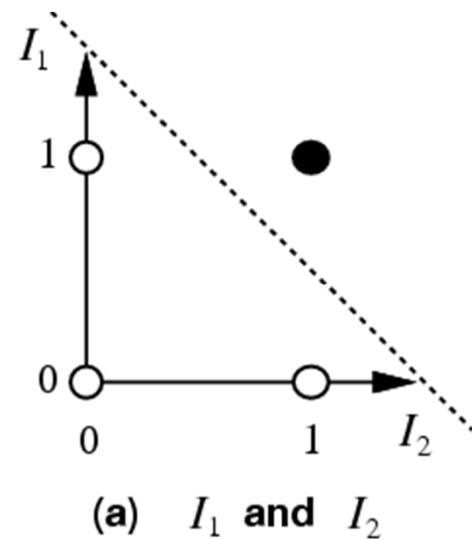
- 3. La más utilizada es la regla delta

- $w_i(t + 1) = w_i(t) + \mu(d(t) - y(t))x_i(t)$

- Con esta regla de aprendizaje se obtiene una convergencia finita si el conjunto de entrenamiento es **linealmente separable**

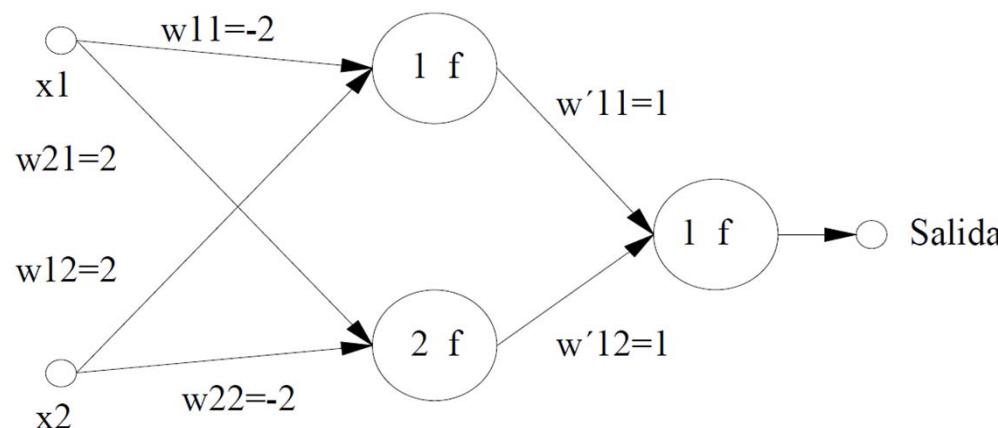
# PERCEPTRÓN

- El problema del XOR (separabilidad lineal)
  - Los patrones para los problemas AND y OR son separables mediante una línea
  - Para el problema del XOR, los patrones no se pueden separar mediante una línea



# PERCEPTRÓN MULTICAPA

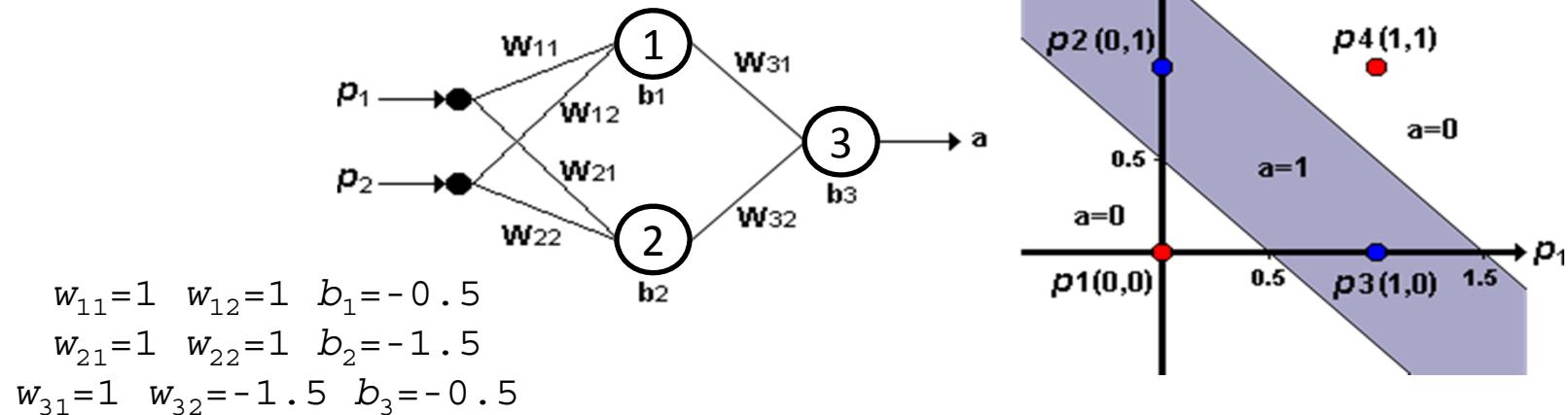
- Solución: añadir más capas:
  - Perceptrón multicapa (*Multilayer Perceptron, MLP*)
  - Se conectan más neuronas a la salida de otras
    - Cada neurona recibe entradas, y computa su salida mediante los pesos y la función de transferencia
  - Se organizan en capas



X1	X2	Salida neurona oculta 1	Salida neurona oculta 2	Salida
0	0	0	0	0
0	1	1	0	1
1	0	0	1	1
1	1	0	0	0

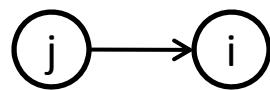
# PERCEPTRÓN MULTICAPA

- Solución: añadir más capas:
  - Perceptrón multicapa (*Multilayer Perceptron, MLP*)



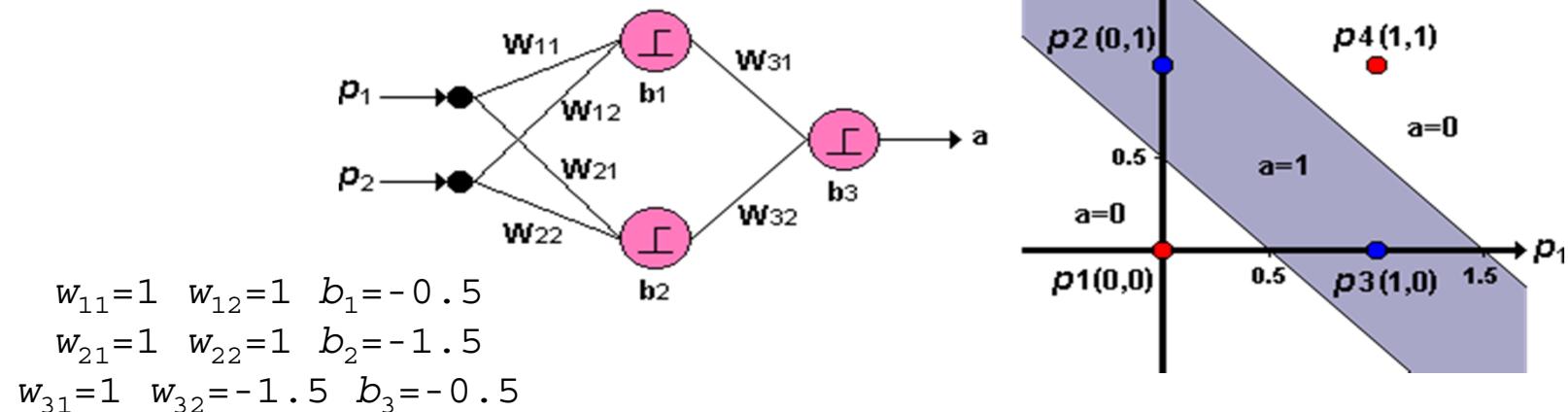
- Notación:

- $W_{ij}$ :



# PERCEPTRÓN MULTICAPA

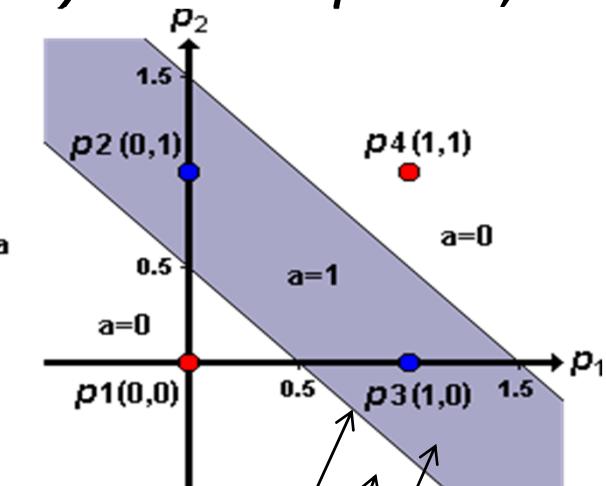
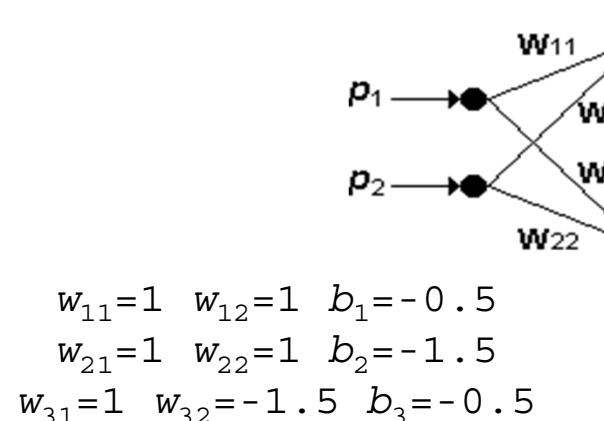
- Solución: añadir más capas:
  - Perceptrón multicapa (*Multilayer Perceptron, MLP*)



X1	X2	Neta neurona oculta 1	Salida neurona oculta 1	Neta neurona oculta 2	Salida neurona oculta 2	Neta neurona salida	Salida
0	0	$0*I + 0*I - 0.5 = -0.5$	0	$0*I + 0*I - 1.5 = -1.5$	0	$0*I - 0*1.5 - 0.5 = -0.5$	0
0	1	$0*I + 1*I - 0.5 = 0.5$	1	$0*I + 1*I - 1.5 = -0.5$	0	$1*I - 0*1.5 - 0.5 = 0.5$	1
1	0	$1*I + 0*I - 0.5 = 0.5$	1	$1*I + 0*I - 1.5 = -0.5$	0	$1*I - 0*1.5 - 0.5 = 0.5$	1
1	1	$1*I + 1*I - 0.5 = 1.5$	1	$1*I + 1*I - 1.5 = 0.5$	1	$1*I - 1*1.5 - 0.5 = -1$	0

# PERCEPTRÓN MULTICAPA

- Solución: añadir más capas:
  - Perceptrón multicapa (*Multilayer Perceptron, MLP*)



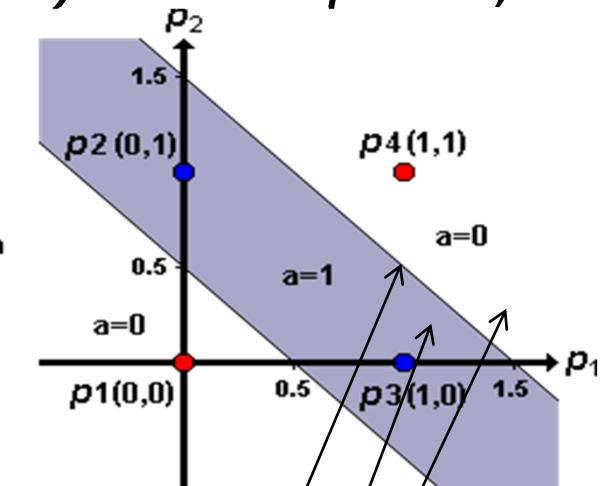
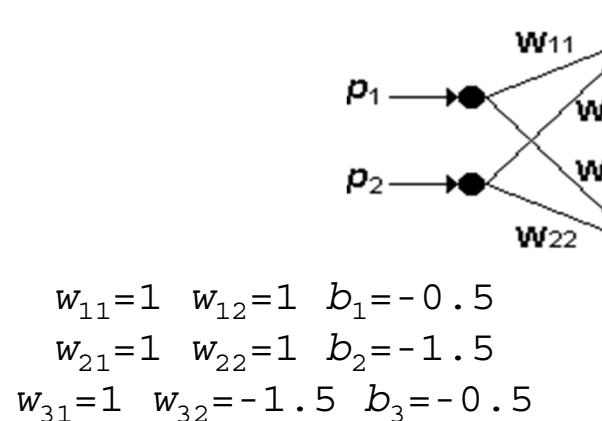
Neurona 1: "OR"

$$x_1 * w_{11} + x_2 * w_{12} + b_1 = 0 \\ x_1 + x_2 - 0.5 = 0$$

$$x_1 + x_2 - 0.5 < 0 \text{ (Salida: 0)} \\ x_1 + x_2 - 0.5 > 0 \text{ (Salida: 1)}$$

# PERCEPTRÓN MULTICAPA

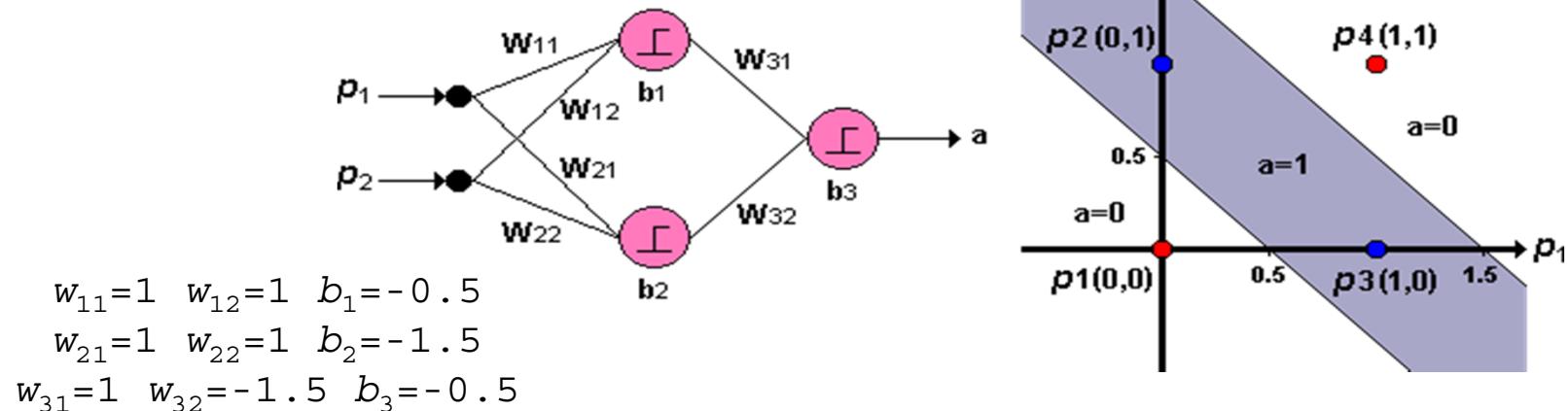
- Solución: añadir más capas:
  - Perceptrón multicapa (*Multilayer Perceptron, MLP*)



$$x_1 + x_2 - 1.5 < 0 \text{ (Salida: 0)}$$
$$x_1 + x_2 - 1.5 > 0 \text{ (Salida: 1)}$$

# PERCEPTRÓN MULTICAPA

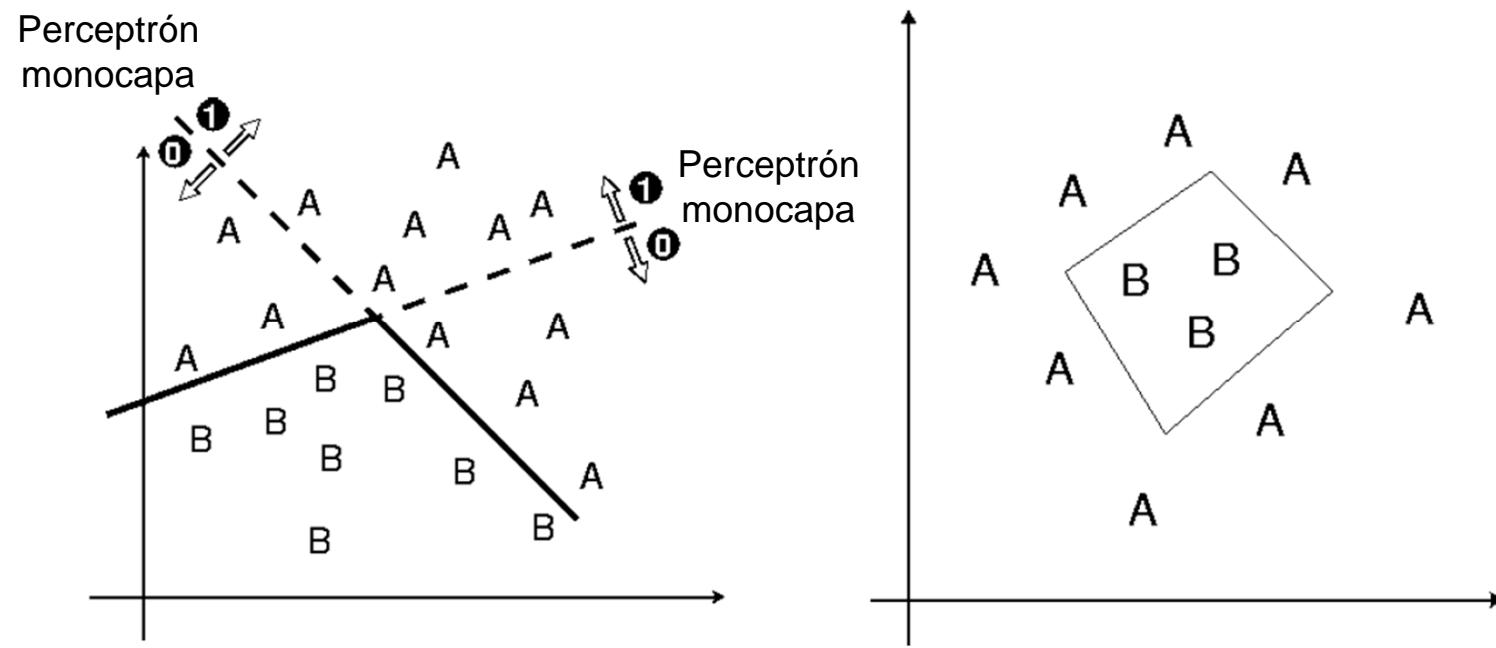
- Solución: añadir más capas:
  - Perceptrón multicapa (*Multilayer Perceptron, MLP*)



Salida neurona oculta 1 Entrada 1 neurona de salida	Salida neurona oculta 2 Entrada 2 neurona de salida	Neta neurona salida	Salida
0	0	$0*1 - 0*1.5 - 0.5 = -0.5$	0
0	1	Caso no existe	-
1	0	$1*1 - 0*1.5 - 0.5 = 0.5$	1
1	1	$1*1 - 1*1.5 - 0.5 = -1$	0

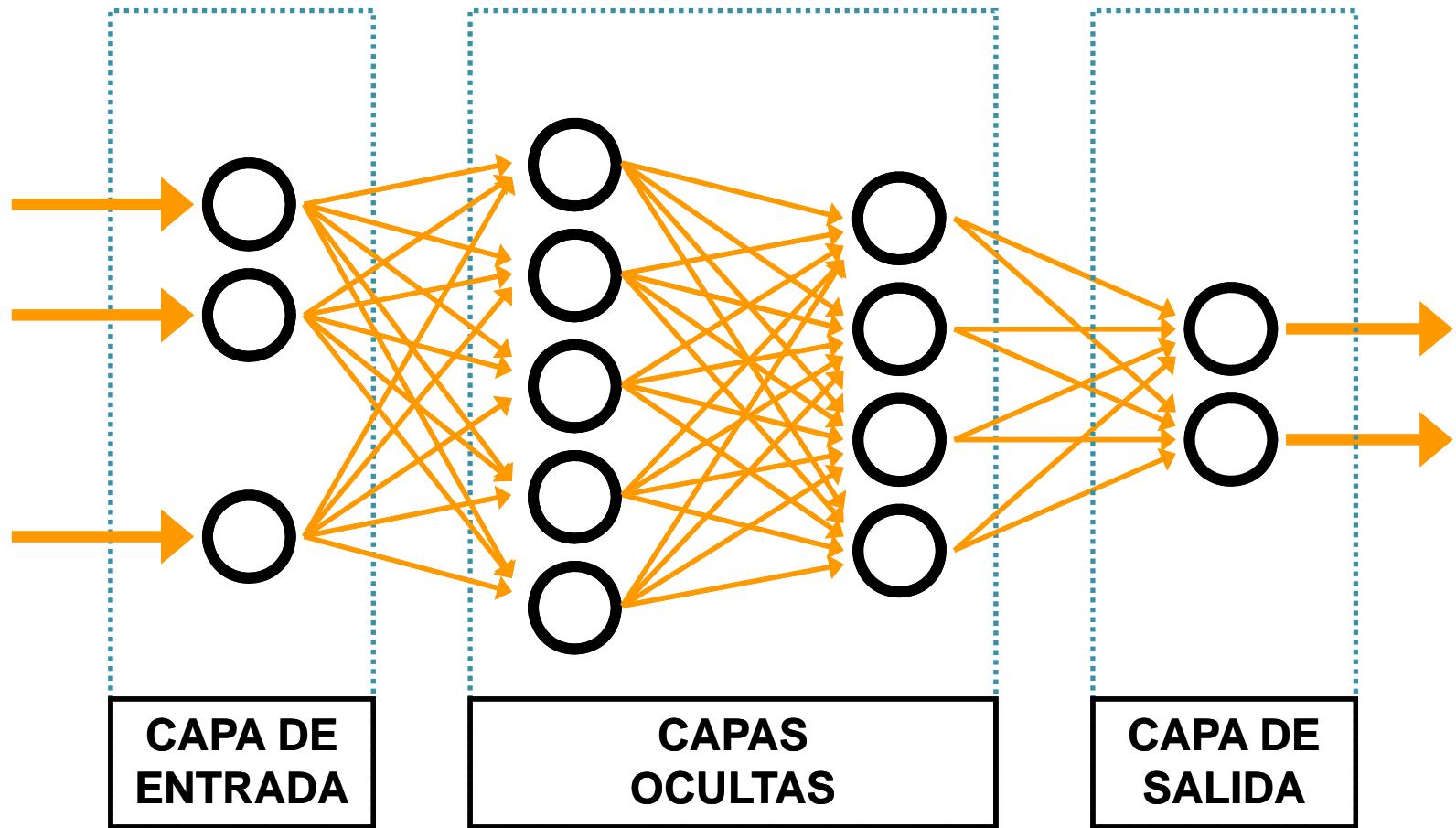
# PERCEPTRÓN MULTICAPA

- Solución: añadir más capas:
  - Perceptrón multicapa (*Multilayer Perceptron, MLP*)
    - Este tipo de estructuras se introducen para resolver problemas que no son linealmente separables



# PERCEPTRÓN MULTICAPA

- Arquitectura general:





# PERCEPTRÓN MULTICAPA

- Arquitectura general:
  - Capa de entrada:
    - Neuronas de entrada: no computan nada, solo almacenan las entradas para pasarlas a la siguiente capa
  - Capa(s) oculta(s)
    - Formadas por neuronas ocultas
  - Capa de salida
    - Emiten la salida de la RNA
- Generalmente, cada capa está totalmente conectada a la capa siguiente



# PERCEPTRÓN MULTICAPA

- Arquitectura general:
  - Capa de entrada:
    - Una neurona de entrada por cada entrada
  - Capa de salida
    - Una neurona por cada salida deseada
  - Capa(s) oculta(s)
    - ¿Cuántas capas ocultas?
      - No hay método
      - No más de dos capas ocultas
    - ¿Cuántas neuronas en cada capa?
      - No hay método
      - Ensayo y error

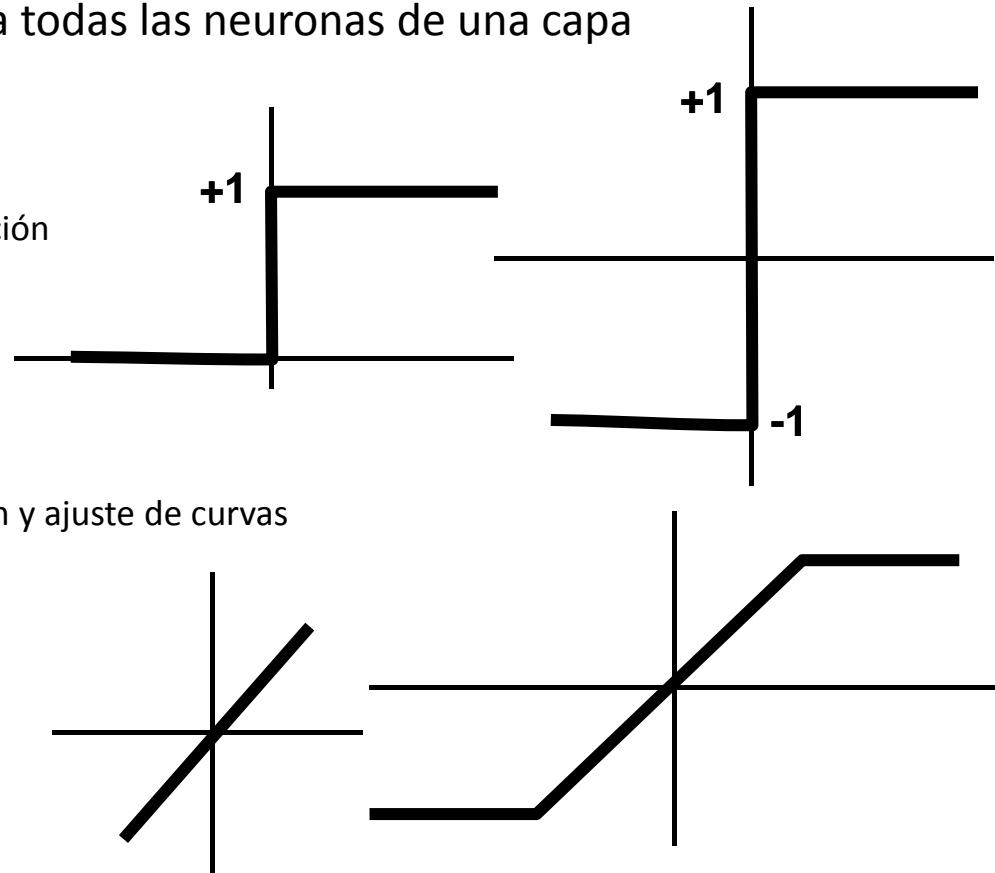


# PERCEPTRÓN MULTICAPA

- Arquitectura general:
  - Conocimiento en la red:
    - Reside en los pesos de las conexiones y bias
    - No centralizado, distribuido
    - Ventajas:
      - Autoorganización
      - Tolerancia a fallos
    - Desventajas:
      - Imposibilidad de explicar su funcionamiento

# PERCEPTRÓN MULTICAPA

- Funciones de transferencia:
  - No están limitadas a ser umbral o lineales
  - Se asume que es la misma para todas las neuronas de la RNA
    - Al menos, la misma para todas las neuronas de una capa
  - Funciones más típicas:
    - Escalón o umbral:
      - Para problemas de clasificación
      - Salidas 0/1 o -1/1:
        - No pertenencia o pertenencia a una clase
    - Lineal y lineal mixta:
      - Para problemas de regresión y ajuste de curvas



# PERCEPTRÓN MULTICAPA

- Funciones de transferencia:

- Funciones más típicas:

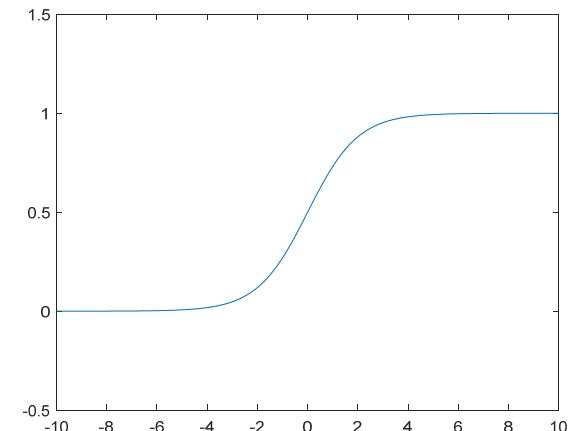
- Logarítmica sigmoidal (logsig):

- Todo tipo de problemas

- Clasificación:

- Uso de umbral

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



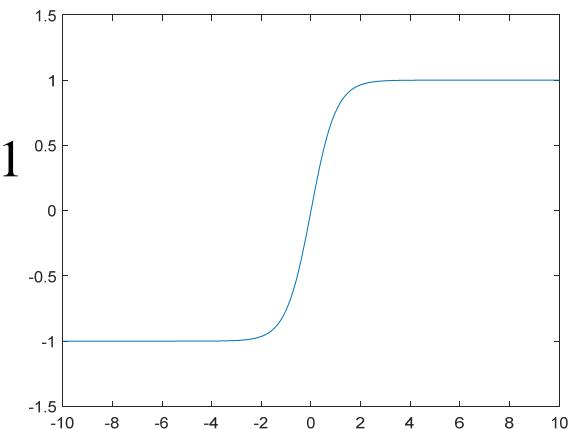
- Tangente sigmoidal hiperbólica (tansig):

- Todo tipo de problemas

- Clasificación

- Uso de umbral

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





# PERCEPTRÓN MULTICAPA

- Funciones de transferencia:
  - Importante:
    - La función de transferencia de las capas internas no puede ser lineal
      - Si lo es, esa capa pierde su capacidad de representación no lineal
    - La función de transferencia en la capa de salida depende del problema a resolver
      - Problemas de regresión: Función lineal
      - Problemas de clasificación: Función logsig o lineal, dependiendo de las clases
        - Si es lineal, se suele aplicar la función *softmax* a la salida de las neuronas de la última capa (ver más adelante)

# PERCEPTRÓN MULTICAPA

- Funciones de transferencia:

compet	Competitive transfer function.	
hardlim	Hard limit transfer function.	
hardlims	Symmetric hard limit transfer function	
logsig	Log sigmoid transfer function.	
poslin	Positive linear transfer function	
purelin	Linear transfer function.	
radbas	Radial basis transfer function.	
satlin	Saturating linear transfer function.	
satlins	Symmetric saturating linear transfer function	
softmax	Soft max transfer function.	
tansig	Hyperbolic tangent sigmoid transfer function.	
tribas	Triangular basis transfer function.	



# PERCEPTRÓN MULTICAPA

- Aproximadores universales:
  - Varios investigadores demostraron que un perceptrón multicapa con suficientes unidades no lineales **puede aprender cualquier tipo de función o relación continua entre un grupo de variables de entrada y salida**
    - **Teoremas de aproximación universal**
  - Esta propiedad convierte a los perceptrones multicapa en herramientas de propósito general, flexibles y no lineales



# PERCEPTRÓN MULTICAPA

- Aproximadores universales:

- Algunas referencias:

- Cybenko, G. (1989) "Approximation by superpositions of a sigmoidal function", *Mathematics of Control, Signals, and Systems*, 2(4), 303–314
    - Funahashi (1989) "On the Approximate Realization of Continuous Mappings by Neural Networks", *Neural Networks* 2, 183-192
    - K. Hornik; M, Stinchcombe; H. White (1989) "Multilayer Feedforward Networks are Universal Approximators", *Neural Networks* 2, 359-366.
    - Kurt Hornik (1991) "Approximation capabilities of multilayer feedforward networks", *Neural Networks*, 4(2), 251–257
    - Leshno, Moshe; Lin, Vladimir Ya.; Pinkus, Allan; Schocken, Shimon (1993). "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". *Neural Networks*. 6 (6): 861–867



# PERCEPTRÓN MULTICAPA

- Aplicaciones:

- Ajuste de funciones y curvas
  - Reproducir una función matemática desconocida
    - Los patrones serían pares de entradas / valor de la función
    - Patrón k:  $(X_{k1}, X_{k2}, \dots, X_{kN}; Y_{k1}, Y_{k2}, \dots, Y_{kM})$  ( $Y_{ki}$  valor real)
- Clasificación
  - Sistema que, ante un nuevo patrón, lo clasifique en una clase entre varias posibles
    - Los patrones serían pares de valores / clase deseada
    - Patrón k:  $(X_{k1}, X_{k2}, \dots, X_{kN}; Y_k)$  ( $Y_k$  clase deseada)
- Regresión
  - Sistema que, ante un nuevo patrón, le de la salida esperada y coherente con los patrones usados en el entrenamiento
    - Los patrones serían pares de entradas / valor deseado
    - Patrón k:  $(X_{k1}, X_{k2}, \dots, X_{kN}; Y_{k1}, Y_{k2}, \dots, Y_{kM})$  ( $Y_{ki}$  valor real)



# PERCEPTRÓN MULTICAPA

- Entrenamiento:
  - No se puede usar la Regla Delta:
    - Pensado para redes sin capas ocultas
    - Se puede computar el error en la capa de salida y modificar los pesos de la capa de salida
    - Pero, ¿cómo se modifican los pesos de las capas anteriores?
      - No se conocen las salidas deseadas para las capas ocultas
      - No se puede computar el error en las capas ocultas
  - La Regla Delta Generalizada o *Backpropagation* o algoritmo de retropropagación del error
    - Generalización de la Regla Delta para RR.NN.AA. de múltiples capas y funciones de transferencia no lineales y diferenciables
    - Ahora hay función de transferencia, no lineal, con la Regla Delta no había



# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - Características
    - Entrenamiento supervisado por corrección de error
    - Aprendizaje off-line
    - Capacidad de generalización
  - Notación:
    - n: número de entradas de cada patrón
    - h: capa oculta; o: capa de salida
    - p: patrón
    - j: PE j en la capa oculta h
    - k: PE k en la capa de salida
    - $w_{ji}^h$ : conexión PE i (capa h-1) con PE j (capa h)
    - $i_{pj}^h$ : salida PE j en la capa oculta h para el patrón p
    - $o_{pj}^o$ : salida PE j en la capa de salida para el patrón p

$$i_{pj}^h = f_j^h(neta_{pj}^h) \quad neta_{pj}^h = \sum_{i=0}^n w_{ji}^h x_{pi} \quad o_{pj}^o = f_j^o(neta_{pj}^o)$$



# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - En la capa de salida puede haber más de un PE
    - No basta con calcular un único valor de error:
      - (calculado en el PE  $k$  para el patrón  $p$ )
    - en cambio, el error que hay que minimizar es la suma de los cuadrados de los errores de todas las unidades de salida
    - ( $n$  salidas, para el patrón  $p$ )

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

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

# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - Es necesario minimizar este error: se deriva:

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

$$\nabla E_p = \frac{\partial E_p}{\partial w_{kj}^o} = -(y_{pk} - o_{pk}) \frac{\partial f_k^o}{\partial (\text{neta}_{pk}^o)} \underbrace{\frac{\partial (\text{neta}_{pk}^o)}{\partial w_{kj}^o}}_{\frac{\partial (\text{neta}_{pk}^o)}{\partial w_{kj}^o} = \frac{\partial}{\partial w_{kj}^o} \left( \sum_i w_{ki}^o i_{pi}^{o-1} \right) = i_{pj}^{o-1}}$$

- Por lo tanto:

$$\nabla E_p = (y_{pk} - o_{pk}) f_k^o (\text{neta}_{pk}^o) i_{pk}^{o-1}$$



# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - Por tanto, los pesos **en la capa de salida** se modifican:
    - Para el peso del EP k de la capa de salida o, desde el EP j, patrón p

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \Delta_p w_{kj}^o(t)$$

$$\Delta_p w_{kj}^o(t) = \mu(y_{pk} - o_{pk}) f_k^o(neta_{pk}^o) i_{pj}^{o-1}$$

- Si llamamos

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

- Entonces:

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

- Condición necesaria:

- Que la función f sea derivable



# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - Por tanto, es posible modificar los pesos de la capa de salida o (conexiones de o-1 a la capa o)
    - Porque se tienen las salidas obtenidas en la capa de salida y la salida deseada  $y$ , con ello, se puede calcular el error
  - En las capas ocultas no hay valores de salidas deseadas
    - No se puede calcular el error ni, por lo tanto, modificar los pesos
  - Solución: propagar el error hacia atrás
    - El error en la capa de salida  $E_p$  está relacionado con la salida de los PE en la capa anterior (o-1)
    - Las actualizaciones de los pesos en esa capa dependen de todos los términos de errores de la capa de salida
    - A esto se refiere el término de retropropagación del error o *backpropagation*



# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:

- El error en la capa de salida  $E_p$  se puede escribir en función de los pesos de la capa de salida  $w_{kj}^o$  (Regla Delta) y de las entradas que recibe esa capa  $i_{pj}^h$ :

$$E_p = \frac{1}{2} \sum_{k=1}^n \delta_{pk}^2 = \frac{1}{2} \sum_k (y_{pk} - o_{pk})^2 = \frac{1}{2} \sum_k (y_{pk} - f^o(neta_{pk}^o))^2 = \frac{1}{2} \sum_k (y_{pk} - f^o(\sum_j w_{kj}^o i_{pj}^h))^2$$

- Sin embargo, esas entradas de la capa de salida  $i_{pj}^h$  son las salidas de las neuronas de la capa oculta j. Por lo tanto, se pueden escribir en función de las entradas que recibe esa neurona oculta j ( $i_{pi}^{h-1}$ ) y de los pesos ( $w_{ji}^h$ )

$$i_{pj}^h = f_j^h(neta_{pj}^h) = f_j^h(\sum_i w_{ji}^h i_{pi}^{h-1})$$

- Por lo tanto, se puede escribir el error en función de los pesos de la capa oculta:

$$E_p = \dots = \frac{1}{2} \sum_k (y_{pk} - f^o(\sum_j w_{kj}^o i_{pj}^h))^2 = \frac{1}{2} \sum_k (y_{pk} - f^o(\sum_j w_{kj}^o f_j^h(\sum_i w_{ji}^h i_{pi}^{h-1})))^2$$

- Y se puede derivar el error con respecto al peso  $w_{ji}^h$  de la neurona oculta j, para modificar ese peso

# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:

- Error: Es función de la salida emitida por la capa de salida:  $E_p = \frac{1}{2} \sum_k (y_{pk} - o_{pk})^2$
- Capa de salida:
  - Salida es función de neta:  $o_{pk} = f^o(neta_{pk}^o)$
  - Neta es función de la salida de la capa oculta:  $neta_{pk}^o = \sum_j w_{kj}^o i_{pj}^h$
- Capa oculta:
  - Salida es función d neta:  $i_{pj}^h = f_j^h(neta_{pj}^h)$
  - Neta es función de los pesos de la capa oculta:  $neta_{pj}^h = \sum_i w_{ji}^h i_{pi}^{h-1}$
- Derivando el error con respecto a los pesos de la capa oculta  $w_{ji}^h$ :

$$\begin{aligned}\frac{\partial E_p}{\partial w_{ji}^h} &= \frac{1}{2} \sum_k \frac{\partial}{\partial w_{ji}^h} (y_{pk} - o_{pk})^2 = -\sum_k (y_{pk} - o_{pk})^2 \frac{\partial o_{pk}}{\partial (neta_{pk}^o)} \frac{\partial (neta_{pk}^o)}{\partial i_{pj}^h} \frac{\partial i_{pj}^h}{\partial neta_{pj}^h} \frac{\partial neta_{pj}^h}{\partial w_{ji}^h} = \\ &= -\sum_k (y_{pk} - o_{pk}) f_k^o(neta_{pk}^o) w_{kj}^o f_j^h(neta_{pj}^h) i_{pi}^{h-1}\end{aligned}$$

y por lo tanto:

$$\begin{aligned}\Delta w_{ji}^h &= \mu \cdot f_j^h(neta_{pj}^h) i_{pi}^{h-1} \underbrace{\sum_k (y_{pk} - o_{pk}) f_k^o(neta_{pk}^o) w_{kj}^o}_{\delta_{pk}^o} \\ \Delta w_{ji}^h &= \mu \cdot f_j^h(neta_{pj}^h) i_{pi}^{h-1} \sum_k \delta_{pk}^o w_{kj}^o\end{aligned}$$



# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - Las actualizaciones de los pesos en esa capa dependen de todos los términos de errores de la capa de salida:

$$\delta_{pj}^h = f_j^h(\text{neto}_{pj}^h) \sum_k \delta_{pk}^o w_{kj}^o$$

- Por lo tanto, para la capa oculta  $h$  (anterior a la capa de salida):

$$w_{ji}^h(t+1) = w_{ji}^h(t) + \mu \delta_{pj}^h i_i^{h-1}$$

# PERCEPTRÓN MULTICAPA

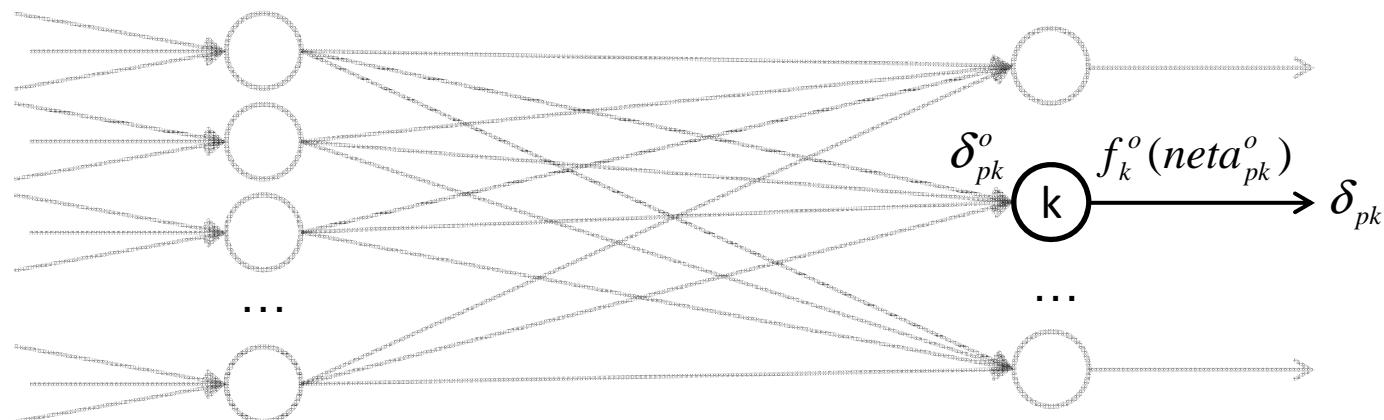
- El algoritmo de *backpropagation*:
  - Comparando ambos términos de modificación de pesos:

	Capa oculta (h) – neurona j	Capa de salida (o) – neurona k
Término de error	$\delta_{pj}^h$ se calcula a partir de la salida que emite el PE j y el error de salida retropropagado hacia esa capa h ( $\sum_k \delta_{pk}^o w_{kj}^o$ ) $\delta_{pj}^h = f_j^h(neta_{pj}^h) \sum_k \delta_{pk}^o w_{kj}^o$	$\delta_{pk}^o$ se calcula a partir de la salida que emite el PE k ( $f_k^o(neta_{pk}^o)$ ) y el error de salida $\delta_{pk}$ $\delta_{pk}^o = \delta_{pk} f_k^o(neta_{pk}^o)$
Variación de los pesos	La variación de los pesos se calcula a partir de ese $\delta_{pj}^h$ y de las entradas que recibe la neurona $w_{ji}^h(t+1) = w_{ji}^h(t) + \mu \delta_{pj}^h i_i^{h-1}$	La variación de los pesos se calcula a partir de ese $\delta_{pk}^o$ y de las entradas que recibe la neurona $w_{kj}^o(t+1) = w_{kj}^o(t) + \mu \delta_{pk}^o i_{pj}^{o-1}$

# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - Comparando ambos términos de modificación de pesos:

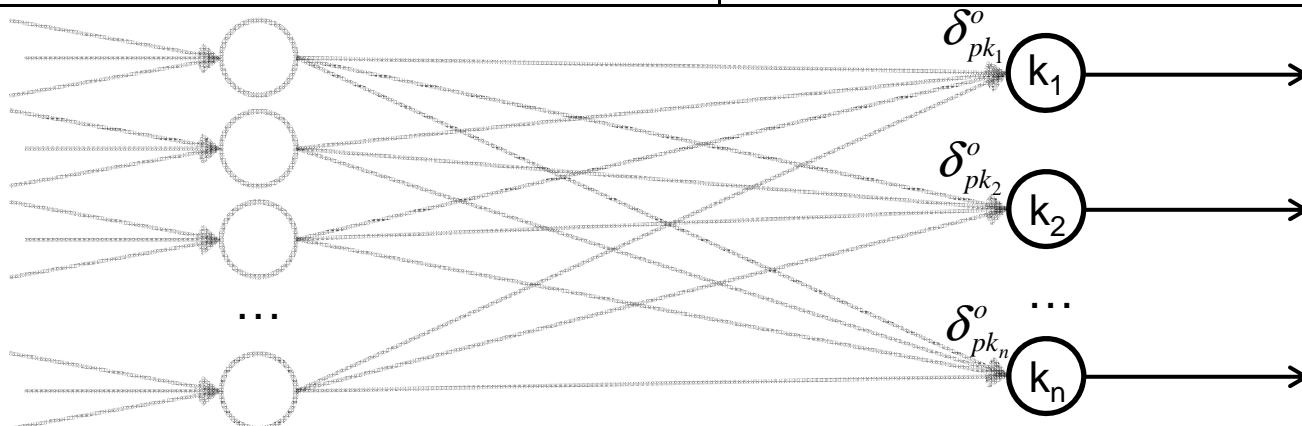
Término de error	Capa oculta (h) – neurona j	Capa de salida (o) – neurona k
		$\delta_{pk}^o$ se calcula a partir de la salida que emite el PE k ( $f_k^o(neta_{pk}^o)$ ) y el error de salida $\delta_{pk}$ $\delta_{pk}^o = \delta_{pk} f_k^o(neta_{pk}^o)$



# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - Comparando ambos términos de modificación de pesos:

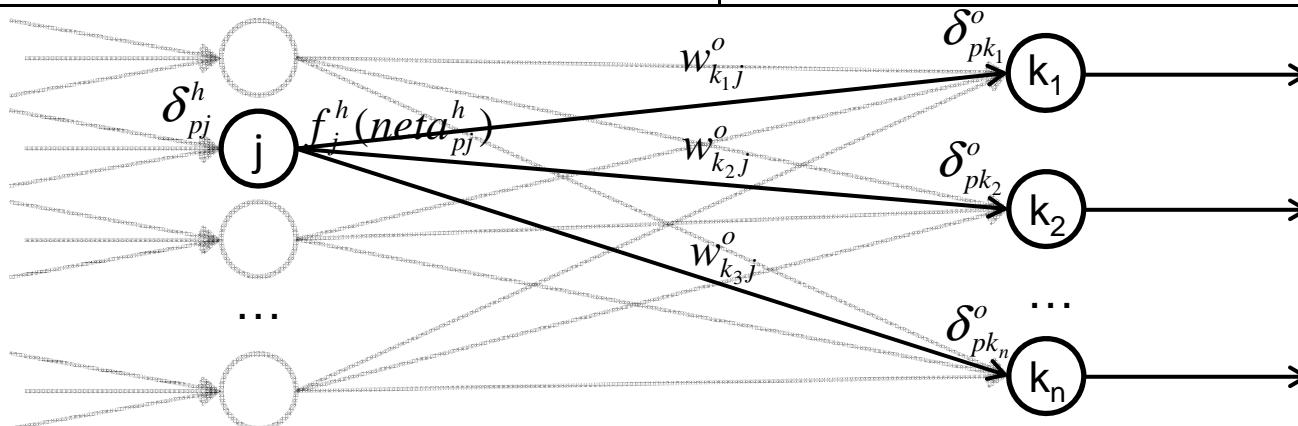
Término de error	Capa oculta (h) – neurona j	Capa de salida (o) – neurona k
		$\delta_{pk}^o$ se calcula a partir de la salida que emite el PE k ( $f_k^o(neta_{pk}^o)$ ) y el error de salida $\delta_{pk}$ $\delta_{pk}^o = \delta_{pk} f_k^o(neta_{pk}^o)$



# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - Comparando ambos términos de modificación de pesos:

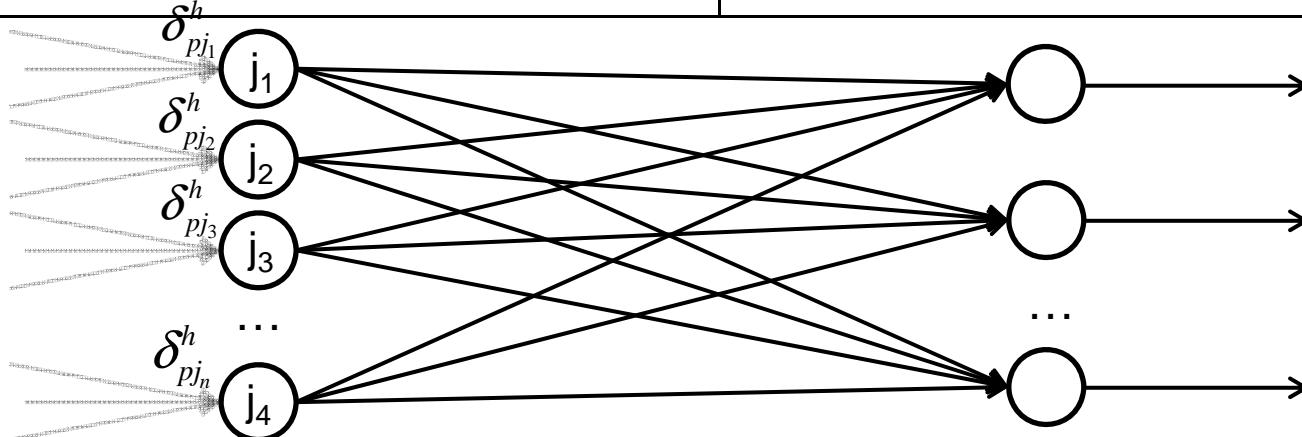
	Capa oculta (h) – neurona j	Capa de salida (o) – neurona k
Término de error	$\delta_{pj}^h$ se calcula a partir de la salida que emite el PE j y el error de salida retropropagado hacia esa capa h ( $\sum_k \delta_{pk}^o w_{kj}^o$ )	$\delta_{pk}^o$ se calcula a partir de la salida que emite el PE k ( $f_k^o(neta_{pk}^o)$ ) y el error de salida $\delta_{pk}$
	$\delta_{pj}^h = f_j^h(neta_{pj}^h) \sum_k \delta_{pk}^o w_{kj}^o$	$\delta_{pk}^o = \delta_{pk} f_k^o(neta_{pk}^o)$



# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - Comparando ambos términos de modificación de pesos:

Término de error	Capa oculta (h) – neurona j	Capa de salida (o) – neurona k
	$\delta_{pj}^h$ se calcula a partir de la salida que emite el PE j y el error de salida retropropagado hacia esa capa h ( $\sum_k \delta_{pk}^o w_{kj}^o$ ) $\delta_{pj}^h = f_j^h(neta_{pj}^h) \sum_k \delta_{pk}^o w_{kj}^o$	$\delta_{pk}^o$ se calcula a partir de la salida que emite el PE k ( $f_k^o(neta_{pk}^o)$ ) y el error de salida $\delta_{pk}$ $\delta_{pk}^o = \delta_{pk} f_k^o(neta_{pk}^o)$



# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - Comparando ambos términos de modificación de pesos:

Término de error Capa oculta (h) – neurona j	Capa de salida (o) – neurona k
$\delta_{pj}^h$ se calcula a partir de la salida que emite el PE j y el error de salida retropropagado hacia esa capa h ( $\sum_k \delta_{pk}^o w_{kj}^o$ ) $\delta_{pj}^h = f_j^h(neta_{pj}^h) \sum_k \delta_{pk}^o w_{kj}^o$	$\delta_{pk}^o$ se calcula a partir de la salida que emite el PE k ( $f_k^o(neta_{pk}^o)$ ) y el error de salida $\delta_{pk}$ $\delta_{pk}^o = \delta_{pk} f_k^o(neta_{pk}^o)$
La variación de los pesos se calcula a partir de ese $\delta_{pj}^h$ y de las entradas que recibe la neurona $w_{ji}^h(t+1) = w_{ji}^h(t) + \mu \delta_{pj}^h i_i^{h-1}$	La variación de los pesos se calcula a partir de ese $\delta_{pk}^o$ y de las entradas que recibe la neurona $w_{kj}^o(t+1) = w_{kj}^o(t) + \mu \delta_{pk}^o i_{pj}^{o-1}$



# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - Se realiza el mismo proceso, calculando el error para las capas anteriores  $h-1$  a partir de la capa  $h$ 
    - Hasta la capa de entrada
  - De esta manera, se va propagando el error hacia atrás
    - *Backpropagation* o retropropagación del error
  - Los términos de error de las unidades ocultas se calculan antes de que hayan sido modificados los pesos de las conexiones con las unidades de la capa de salida
  - $\mu$ : tasa o velocidad de aprendizaje

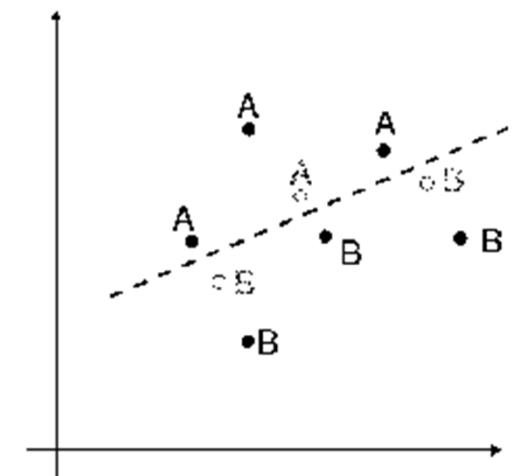


# PERCEPTRÓN MULTICAPA

- El algoritmo de *backpropagation*:
  - Algoritmo de entrenamiento: similar a la Regla Delta
    - Se inicializan los pesos de forma aleatoria
    - Se calculan los errores de todos los patrones y se modifican los pesos según ese valor de error global
    - Se repite este proceso durante un número de ciclos (*epochs*) hasta que se da una condición de parada:
      - Se ha alcanzado un error de entrenamiento aceptable
      - Se ha realizado un número de ciclos, *epochs* o épocas prefijado.
      - No se ha mejorado el menor error de validación durante una serie de ciclos seguidos
        - Para evitar el sobreentrenamiento
      - etc.

# PERCEPTRÓN MULTICAPA

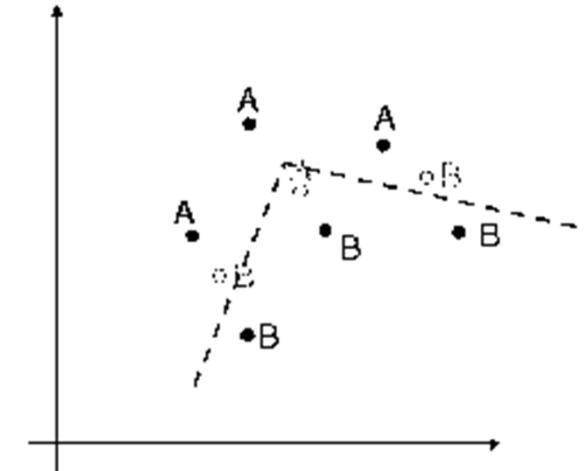
- Entrenamiento:
  - Se pueden emplear todos los datos disponibles para entrenar la red
  - Lo que se necesita es un subconjunto de datos que cubran todo el espacio de los mismos
    - **Importante que sean representativos**
    - Estos patrones que se usan para entrenar guían el proceso de entrenamiento, y la RNA aprenderá en función de ellos
  - La RNA admite la Generalización
    - Dados varios vectores de entrada (no pertenecientes al conjunto de entrenamiento), similares a patrones existentes en el conjunto de entrenamiento, la red reconocerá las similitudes entre dichos patrones
    - Patrones que no ha visto en el entrenamiento



# PERCEPTRÓN MULTICAPA

- Entrenamiento:

- Si la red se entrena mal o insuficientemente, las salidas pueden ser imprecisas.
  - También, si el conjunto de entrenamiento no fue bien escogido
- Región de incertidumbre: (entrenado con A,B).
  - Red con 2 unidades ocultas (1 capa oculta).
  - Al minimizar el error los planos que se generan se alinean tan cerca de los patrones de entrenamiento como sea posible.
  - Patrones mal escogidos
- ¿Qué ocurre si se presentan patrones en alguna región donde en el entrenamiento no hubo patrones?





# PERCEPTRÓN MULTICAPA

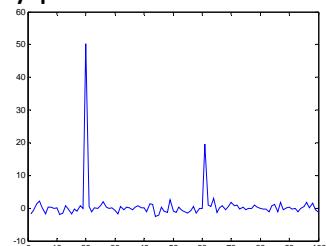
- Entrenamiento:

- La capacidad de una red para resolver un problema está ligada a los ejemplos utilizados durante el aprendizaje.
- El conjunto de entrenamiento debe:
  - Ser significativo:
    - Número suficiente de ejemplos
    - Si el conjunto de entrenamiento es reducido, la red no será capaz de resolver el problema de forma eficaz
  - Ser representativo:
    - Ejemplos diversos: todas las regiones del espacio de estados deben estar suficientemente cubiertas.
    - Si un conjunto de aprendizaje contiene muchos más ejemplos de un tipo que del resto, la red se especializará en dicho subconjunto y no será de aplicación general.

# PERCEPTRÓN MULTICAPA

- Entrenamiento:
  - Entradas: valor real arbitrariamente grande o pequeño
    - Importante normalizar las entradas para que estén entre 0 y 1 o entre -1 y 1 (depende del problema) **cada entrada por separado**
    - Dos formas de normalización:
      - Entre máximo y mínimo
        - Si todos los valores están acotados
      - Usando media y desviación típica
        - Si los valores son el resultado de una medición y algún valor puede salirse y ser muy alto o muy bajo
          - Ejemplo: temperatura, altura, etc.
          - En estos casos, si se normalizase entre máximo y mínimo, la existencia de un valor extremo fijaría ese valor como el máximo (o mínimo) y haría que, al normalizar, el resto de valores oscilaran en un rango muy pequeño
          - La RNA podría tomar el resto de valores como ruido, al tener muy poca variación

$$X = \frac{x - \mu}{\sigma}$$



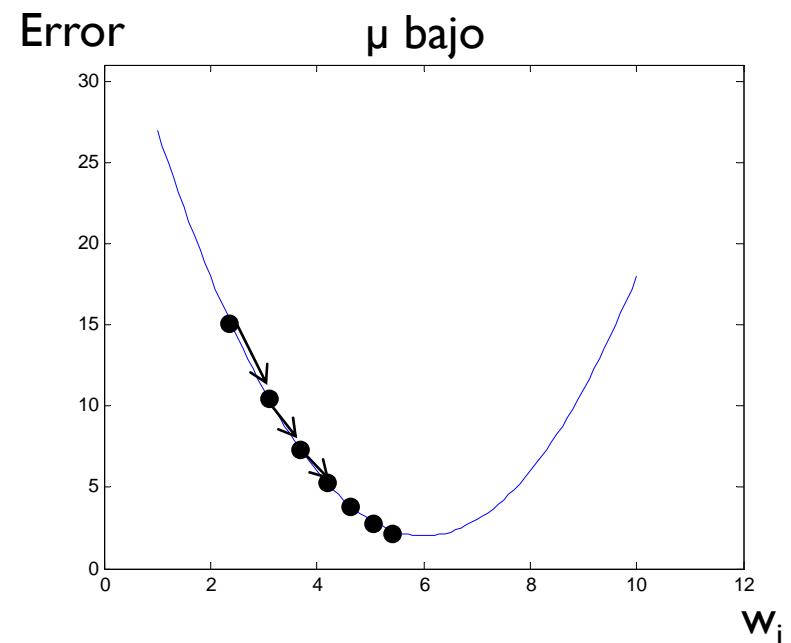
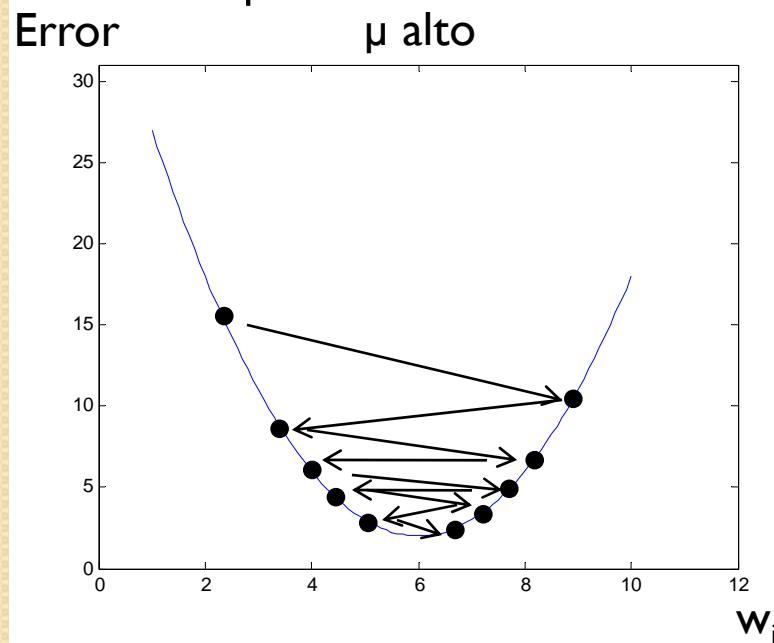


# PERCEPTRÓN MULTICAPA

- Entrenamiento:
  - Salidas:
    - Es necesario normalizar también los valores de las salidas deseadas de la BD para entrenar la red, **cada salida por separado**
    - Una vez entrenada, cuando se esté utilizando la red, para saber el valor real se desnormaliza el valor que devuelve la red
  - Funcionamiento de una red:
    - Ante un conjunto de entradas nuevo:
      - Normalizar las entradas
        - Cada entrada por separado, según los parámetros establecidos al normalizar las entradas del conjunto de entrenamiento
        - Aplicar las entradas a la red y calcular las salidas
        - Desnormalizar las salidas
          - Cada salida por separado, según los parámetros establecidos al normalizar las salidas del conjunto de entrenamiento

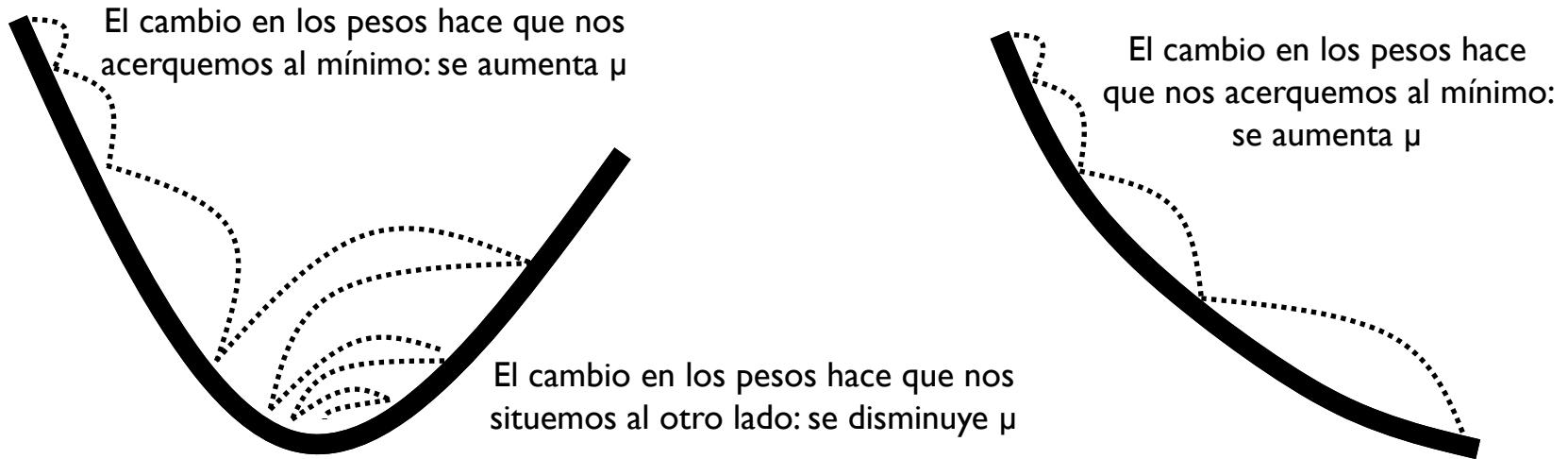
# PERCEPTRÓN MULTICAPA

- Control de convergencia:
    - En las técnicas de gradiente descendente es conveniente avanzar por la superficie de error con incrementos pequeños de los pesos
      - Incrementos grandes: se corre el riesgo de pasar por encima del punto mínimo sin conseguir estacionarse en él
      - Incrementos pequeños: aunque se tarde más en llegar, se evita que ocurra esto.



# PERCEPTRÓN MULTICAPA

- Control de convergencia:
  - El elegir un incremento adecuado influye en la velocidad con la que converge el algoritmo
    - Este control se puede realizar mediante el parámetro denominado tasa de aprendizaje
    - Normalmente se le asigna un valor pequeño (0.05-0.25) para asegurar que la red llegue a asentarse en una solución
    - Se le puede hacer decrecer a medida que avanzan los ciclos del proceso de entrenamiento
    - Proceso adaptativo en el que este parámetro aumenta y disminuye de valor





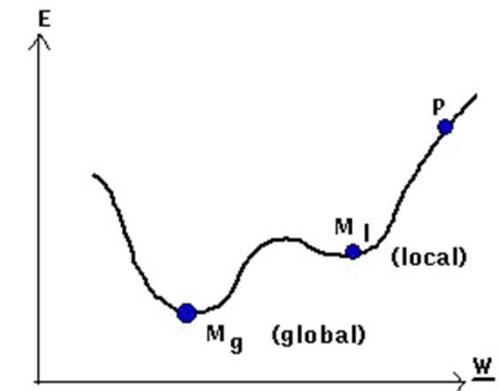
# PERCEPTRÓN MULTICAPA

- Control de convergencia:
  - Otra manera de incrementar la velocidad de aprendizaje consiste en utilizar otro parámetro llamado Momento:
    - La idea es que cambios previos en los pesos deberían influir en la dirección del movimiento en el espacio de pesos
    - Introduce cierta inercia en la actualización de los pesos
    - Este concepto se implementa cambiando la ecuación de modificación de pesos a la siguiente:

$$\vec{w}_{kj}^o(t+1) = \vec{w}_{kj}^o(t) + \mu \delta_{pk}^o \vec{i}_{pk}^{o-1} + \alpha \Delta_p \vec{w}_{kj}^o(t-1)$$

# PERCEPTRÓN MULTICAPA

- Control de convergencia:
  - Posibilidad de convergencia hacia alguno de los mínimos locales.
    - No se puede asegurar en ningún momento que el mínimo que se encuentre sea global.
    - **Proceso no determinístico**
    - Una vez que la red se asienta en un mínimo, sea local o global, cesa el aprendizaje, aunque el error siga siendo demasiado alto, si se ha alcanzado un mínimo local.
    - Algoritmo de *backpropagation* clásico
    - Algoritmos más recientes (variantes) son capaces de saltar mínimos locales
  - Cada vez que se entrena da un resultado distinto!
    - Se inicializan los pesos en un punto distinto (aleatorio)
- Si se alcanza un mínimo local y el error es satisfactorio, el entrenamiento ha sido un éxito
  - Si no sucede así, puede realizarse varias acciones para solucionar el problema:
    - Cambio de arquitectura (más capas ocultas o más PE)
    - Modificación de parámetros de aprendizaje.
    - Emplear un conjunto de pesos iniciales diferentes (es decir, entrenar otra vez)
    - Modificar el conjunto de entrenamiento o presentar los patrones en distinta secuencia.





# PERCEPTRÓN MULTICAPA

- Entrenamiento:

- Se tiene un conjunto de patrones de entrenamiento que guía el proceso
- El proceso de entrenamiento minimiza el error en esos patrones
  - El error de entrenamiento va a ser bajo, pero
  - ¿Cómo se va a comportar la red ante nuevos patrones?
  - ¿Qué error va a ofrecer ante nuevos patrones?
  - ¿Es capaz de generalizar bien?
- Después del entrenamiento, se le pasa otro conjunto de patrones
  - Conjunto de test
  - Ninguno de estos patrones está presente en el de entrenamiento
  - El valor de error en el conjunto de test es el que realmente dice cómo de bien entrenada está la red

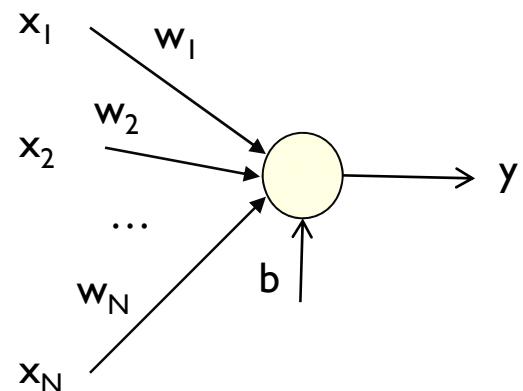


# PERCEPTRÓN MULTICAPA

- Entrenamiento: dos conjuntos de patrones:
  - Conjunto de entrenamiento
    - Guía el proceso de entrenamiento
    - Los valores de los pesos se establecen en función de estos valores
  - Conjunto de test
    - No interviene en el entrenamiento
    - Una vez entrenada, la red se evalúa con este conjunto para evaluar su capacidad de generalización
- El problema del **sobreentrenamiento**:
  - Buen error de entrenamiento
    - Pero mala generalización:
      - Si la red se evalúa con otro conjunto de patrones (test), el error es muy alto
    - La red ha aprendido los patrones de entrenamiento
      - NO las características que los definen
      - Si los patrones de entrenamiento tienen ruido, la red lo ha aprendido como si fuera una característica de estos patrones
    - Causas:
      - Topología demasiado compleja
      - Entrenamiento durante demasiado tiempo
        - No parar el entrenamiento a tiempo
      - No comprobar la generalización de la red mientras entrena

# FUNCIONAMIENTO DE UNA RED

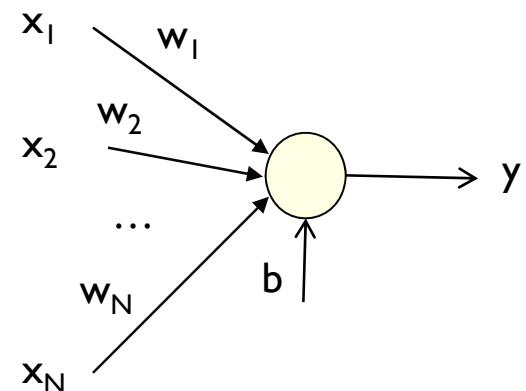
- Neurona artificial:



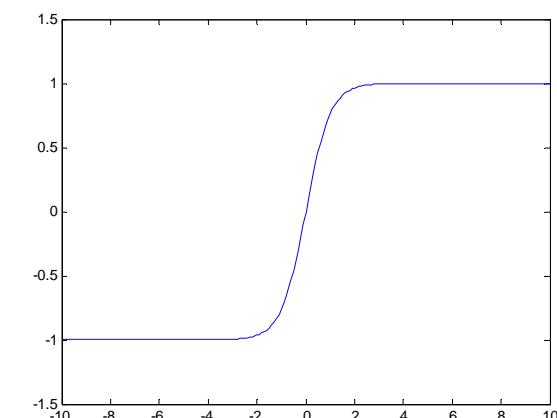
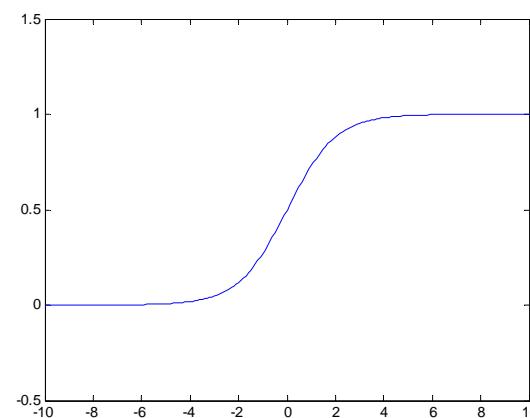
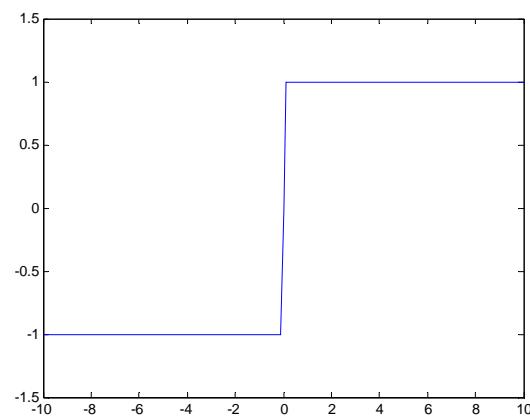
$$y = f\left(\sum_{i=1}^N (x_i * w_i) + b\right)$$

# FUNCIONAMIENTO DE UNA RED

- Neurona artificial:

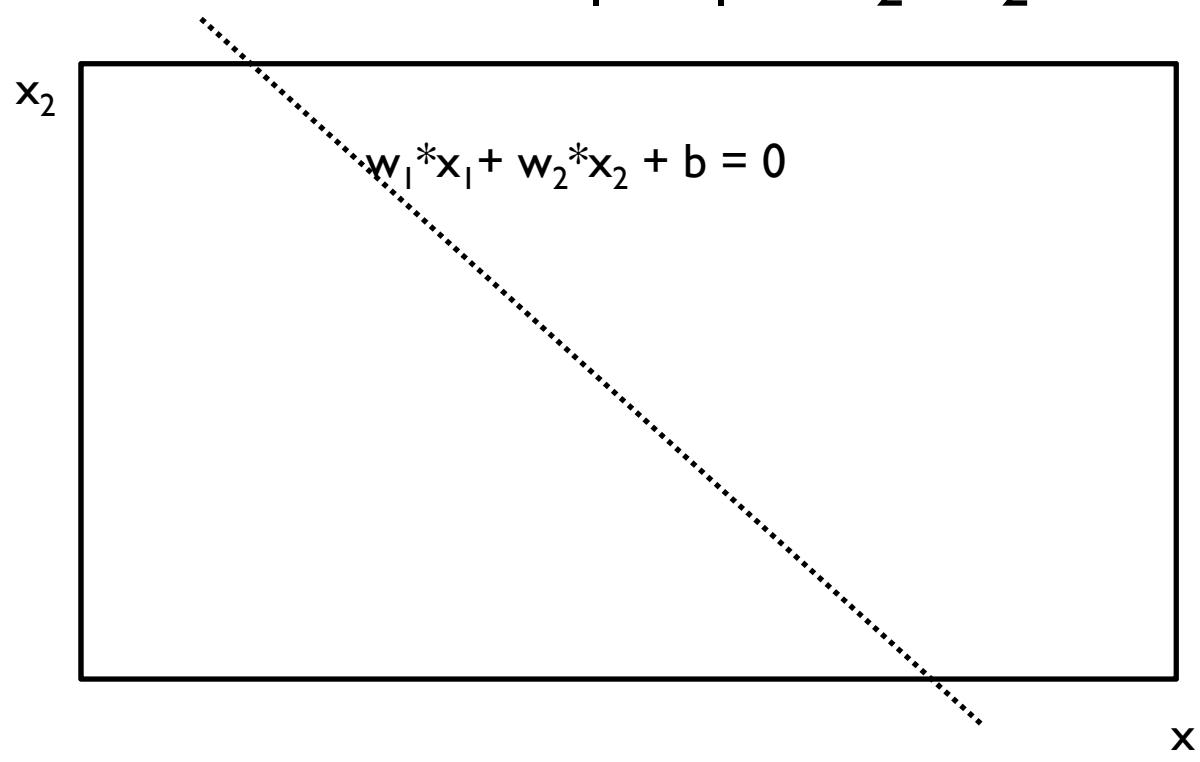


$$y = f\left(\sum_{i=1}^N (x_i * w_i) + b\right)$$



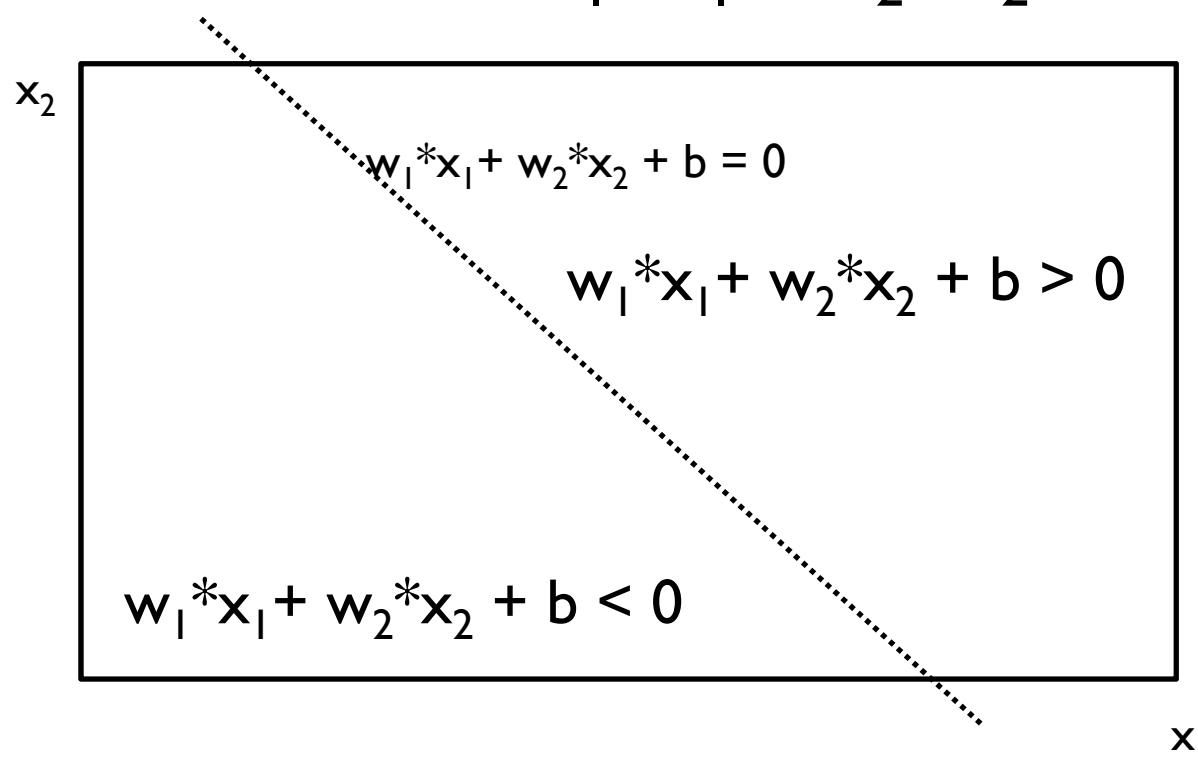
# FUNCIONAMIENTO DE UNA RED

- Si N=2 (2 entradas):
  - $y = f(x_1 * w_1 + x_2 * w_2 + b)$
  - Ec. de una recta:  $x_1 * w_1 + x_2 * w_2 + b = 0$



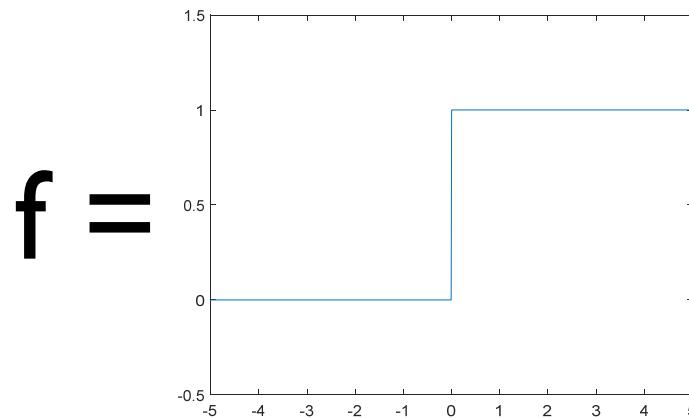
# FUNCIONAMIENTO DE UNA RED

- Si N=2 (2 entradas):
  - $y = f(x_1 * w_1 + x_2 * w_2 + b)$
  - Ec. de una recta:  $x_1 * w_1 + x_2 * w_2 + b = 0$



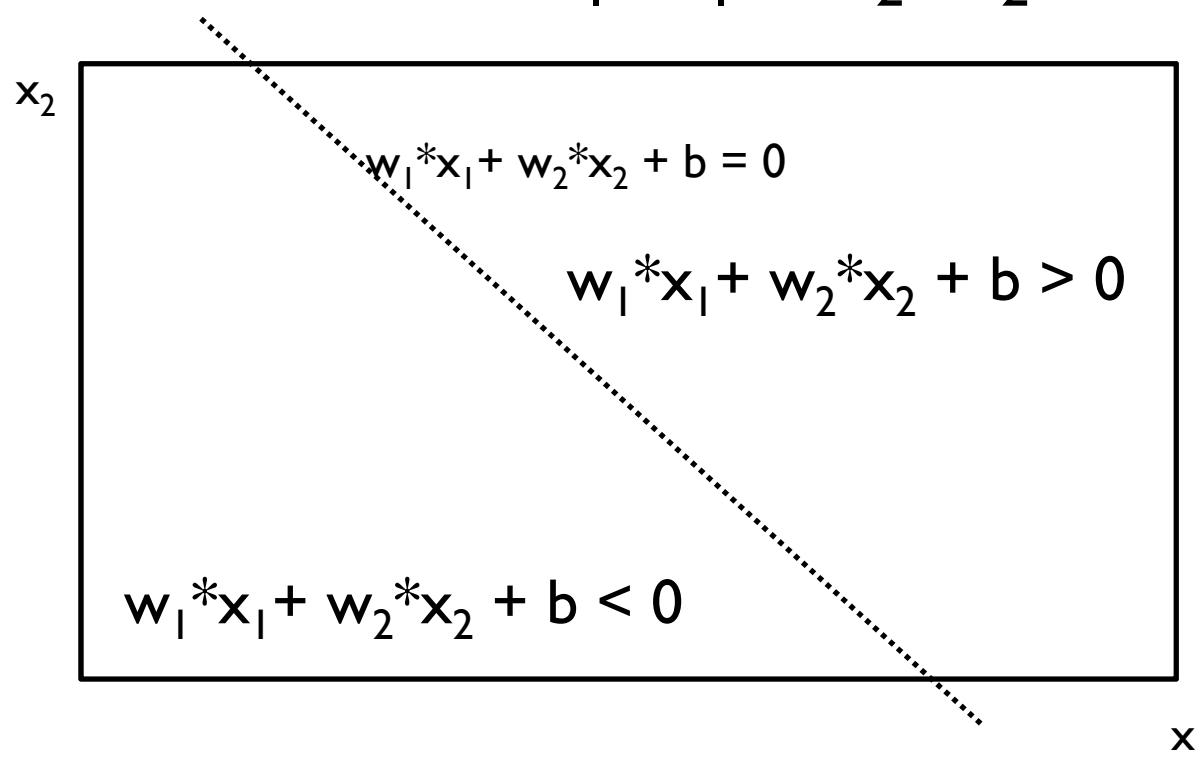
# FUNCIONAMIENTO DE UNA RED

- Si N=2 (2 entradas):
  - $y = f(x_1 * w_1 + x_2 * w_2 + b)$
  - Ec. de una recta:  $x_1 * w_1 + x_2 * w_2 + b = 0$



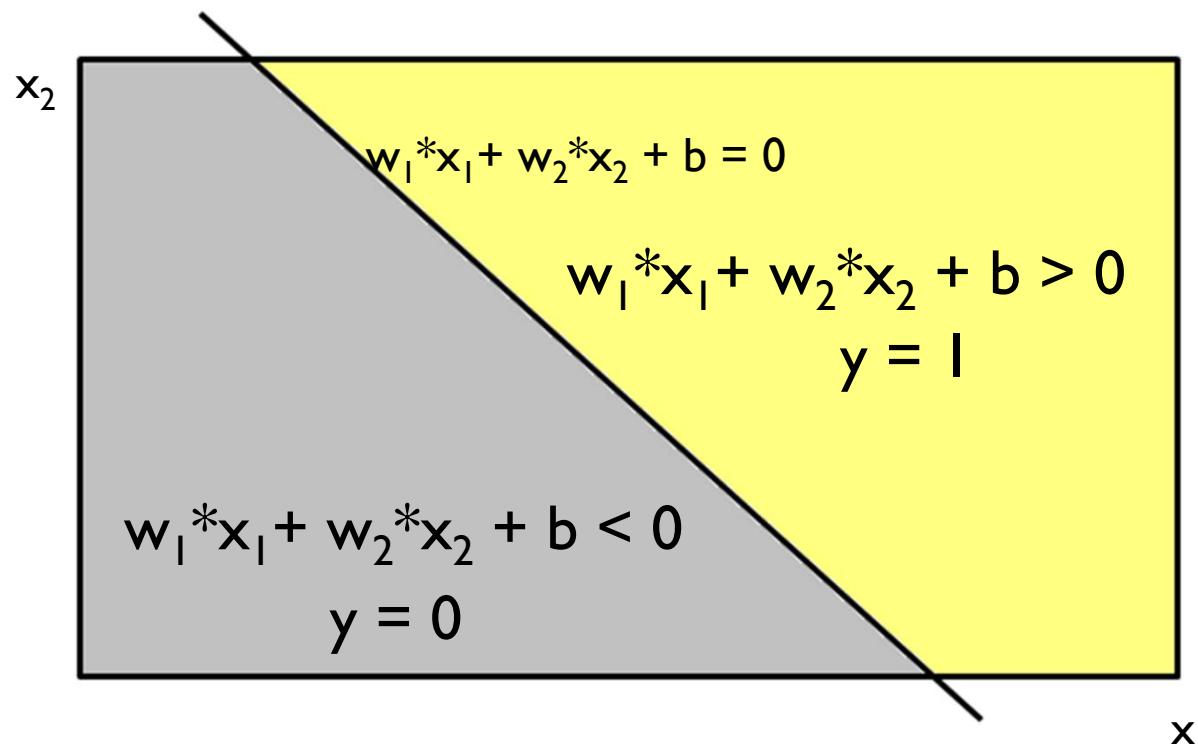
# FUNCIONAMIENTO DE UNA RED

- Si N=2 (2 entradas):
  - $y = f(x_1 * w_1 + x_2 * w_2 + b)$
  - Ec. de una recta:  $x_1 * w_1 + x_2 * w_2 + b = 0$



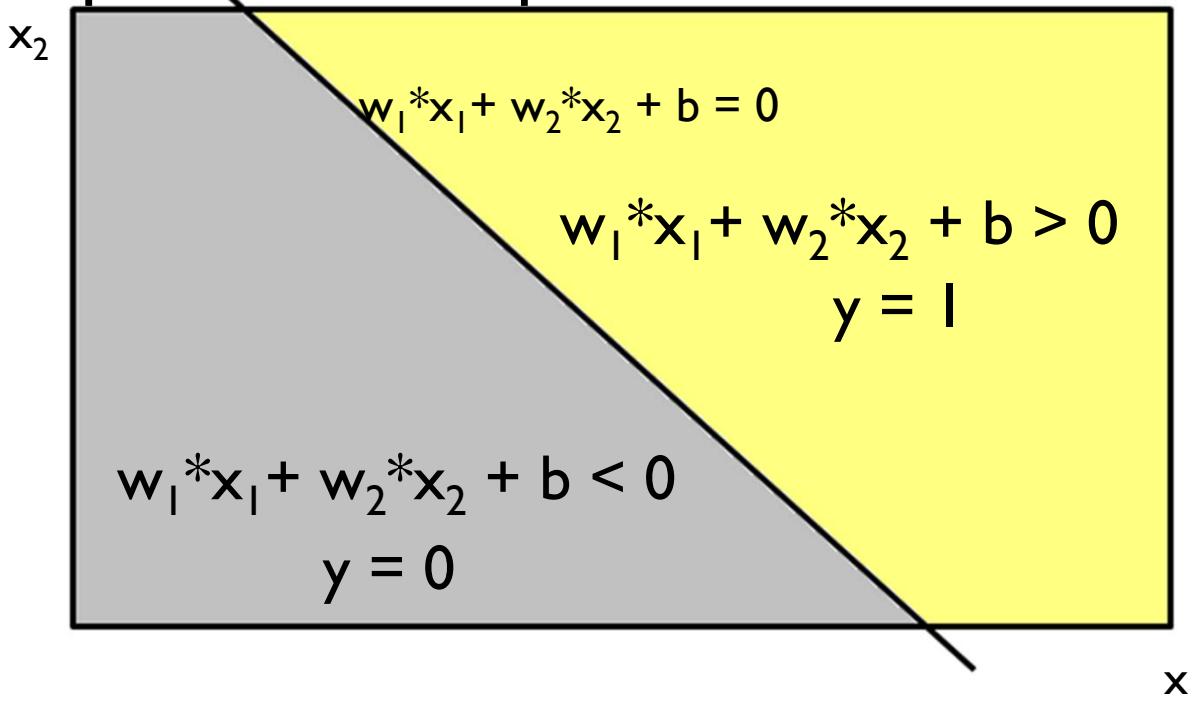
# FUNCIONAMIENTO DE UNA RED

- Si N=2 (2 entradas):
  - $y = f(x_1 * w_1 + x_2 * w_2 + b)$
  - Ec. de una recta:  $x_1 * w_1 + x_2 * w_2 + b = 0$



# FUNCIONAMIENTO DE UNA RED

- Si N=2 (2 entradas):
  - La neurona divide el espacio de entrada en dos semiplanos
  - Los patrones se separan linealmente



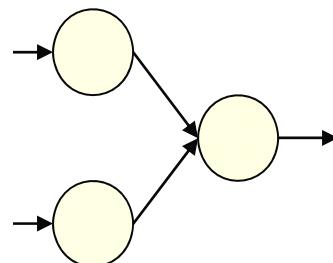


# FUNCIONAMIENTO DE UNA RED

- Si N=3 (3 entradas):
  - $y = f(x_1 * w_1 + x_2 * w_2 + x_3 * w_3 + b)$
  - Ecuación de un plano en un entorno de 3 dimensiones:
    - $x_1 * w_1 + x_2 * w_2 + x_3 * w_3 + b = 0$
    - El plano divide el espacio de 3 dimensiones en 2 subespacios
- En general, para N entradas:
  - $y = f(\sum_{i=1}^N (x_i * w_i) + b)$
  - Ecuación de un *hiperplano* en un espacio de N dimensiones:
$$\sum_{i=1}^N (x_i * w_i) + b = 0$$
    - El hiperplano divide el espacio en dos subespacios de N dimensiones

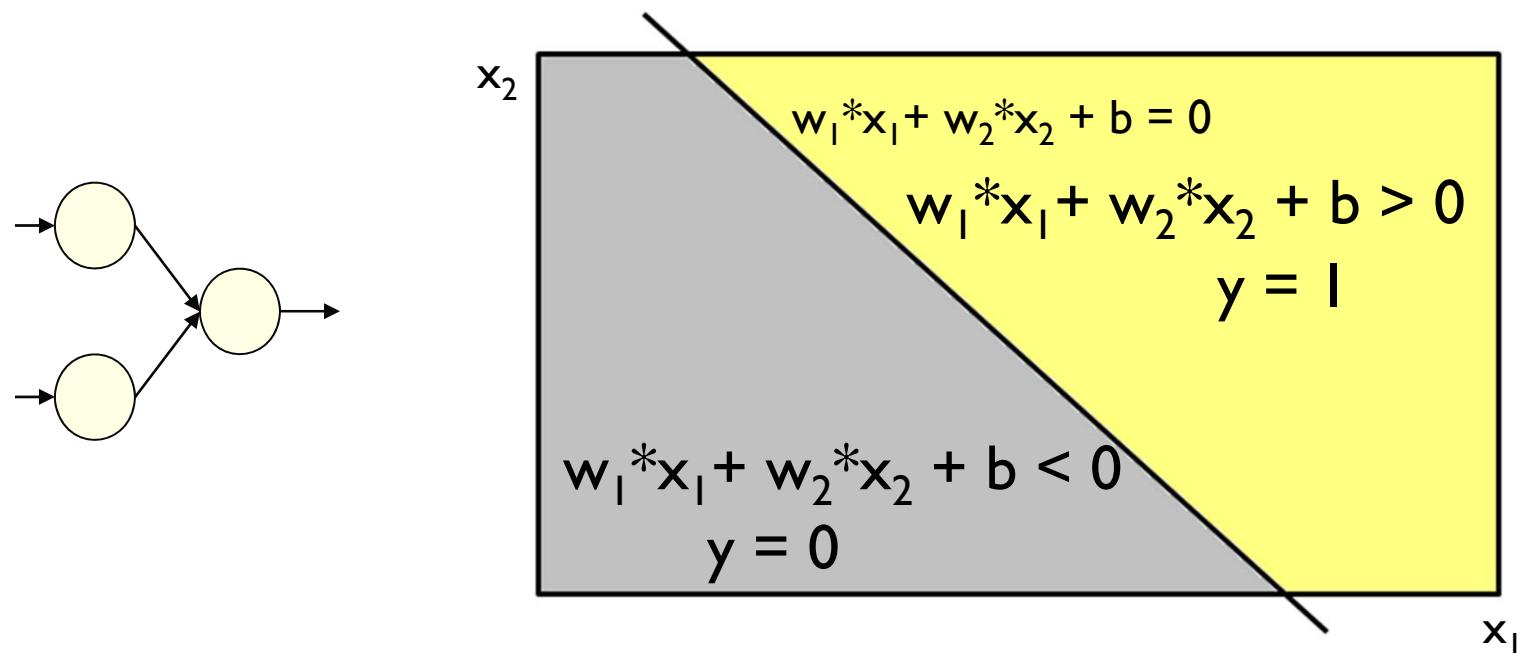
# FUNCIONAMIENTO DE UNA RED

- Caso N=2 (2 entradas):
  - Más fácil de representar
  - Generalizable a N dimensiones
  - Una RNA con capa de entrada y una salida



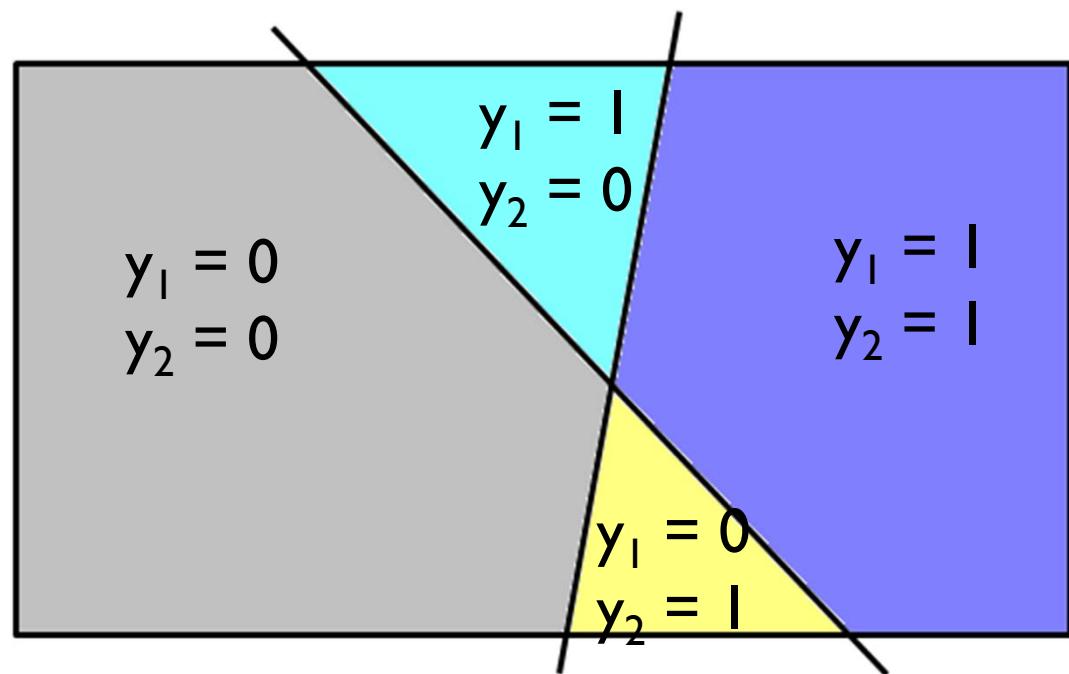
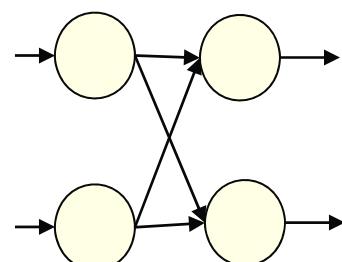
# FUNCIONAMIENTO DE UNA RED

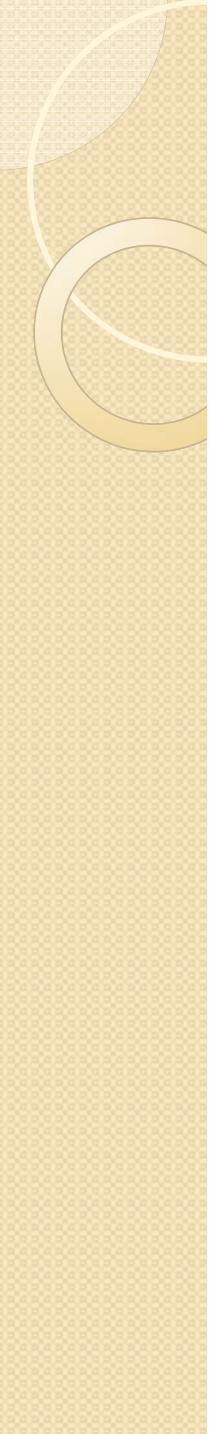
- Caso N=2 (2 entradas):
  - Más fácil de representar
  - Generalizable a N dimensiones
  - Una RNA con capa de entrada y una salida



# FUNCIONAMIENTO DE UNA RED

- Caso N=2 (2 entradas):
  - Más fácil de representar
  - Generalizable a N dimensiones
  - Una RNA con capa de entrada y dos salidas



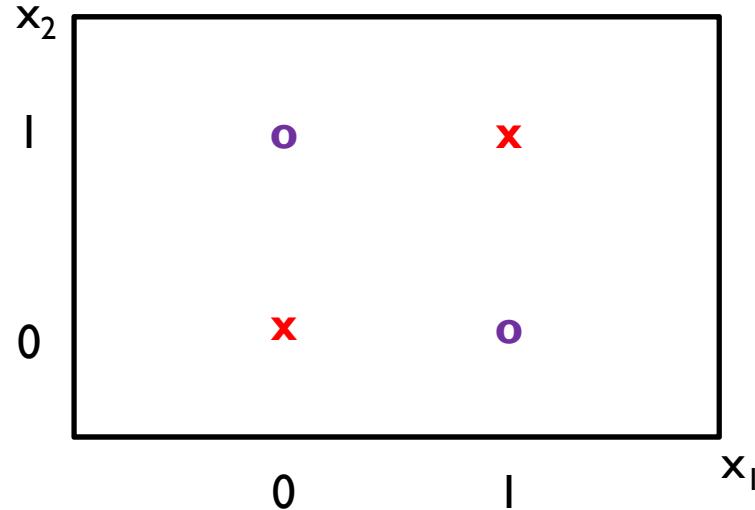
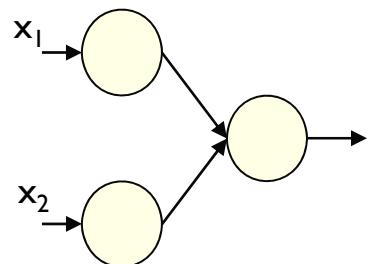


# FUNCIONAMIENTO DE UNA RED

- Caso  $N=2$  (2 entradas):
  - Más fácil de representar
  - Generalizable a  $N$  dimensiones
  - Una RNA con capa de entrada y dos salidas
    - Cada neurona de salida divide el espacio en dos partes
- En general, para  $M$  salidas:
  - Cada neurona divide el espacio en 2 partes: crea un *hiperplano*

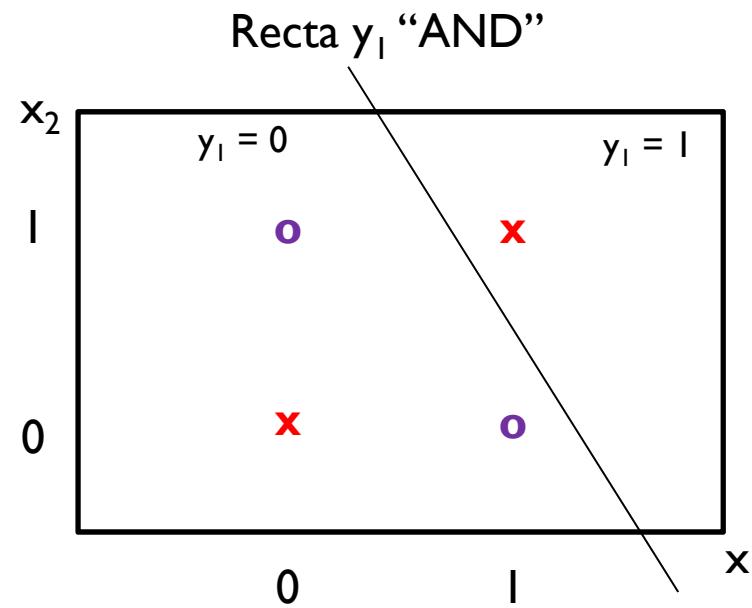
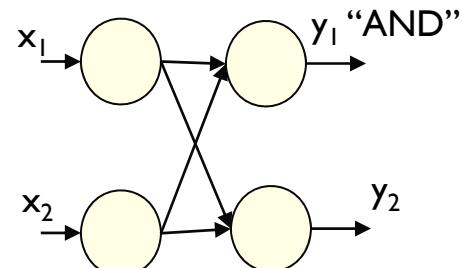
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - Los patrones no se pueden separar linealmente
    - No se puede trazar una línea que divida el espacio en dos partes y clasifique los patrones
  - Una RNN monocapa no puede solucionarlo



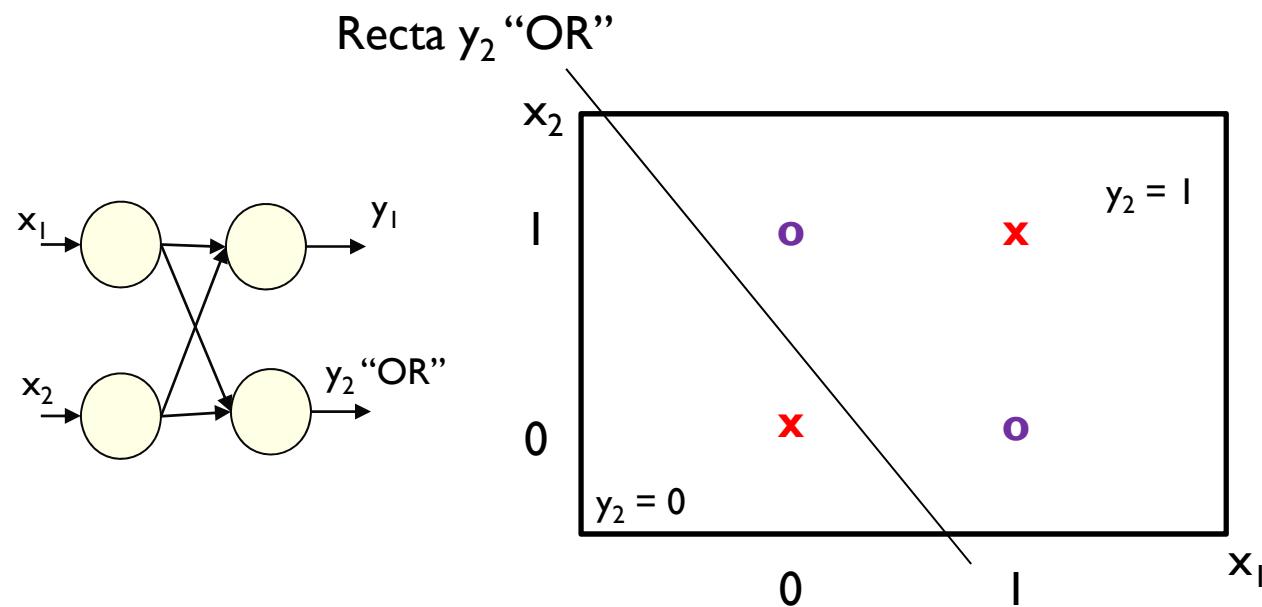
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - Solución: usar una red con dos neuronas
  - Cada neurona separa el espacio en dos partes:



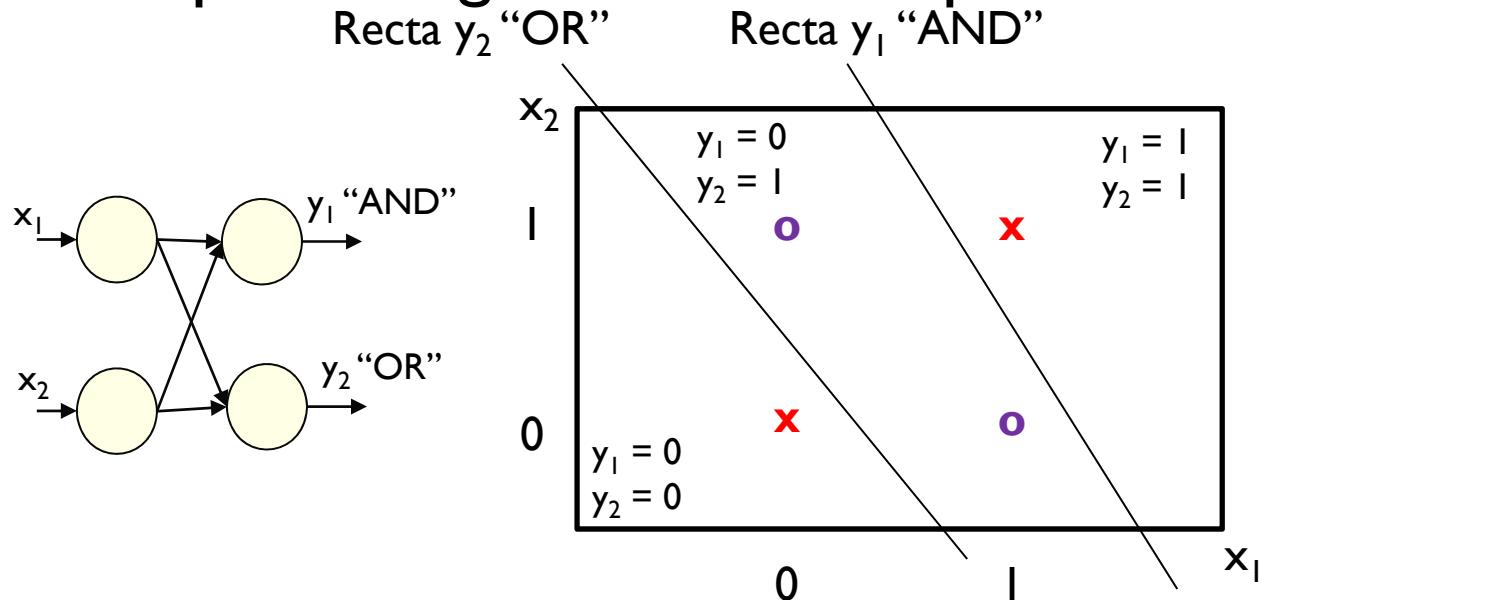
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - Solución: usar una red con dos neuronas
  - Cada neurona separa el espacio en dos partes:



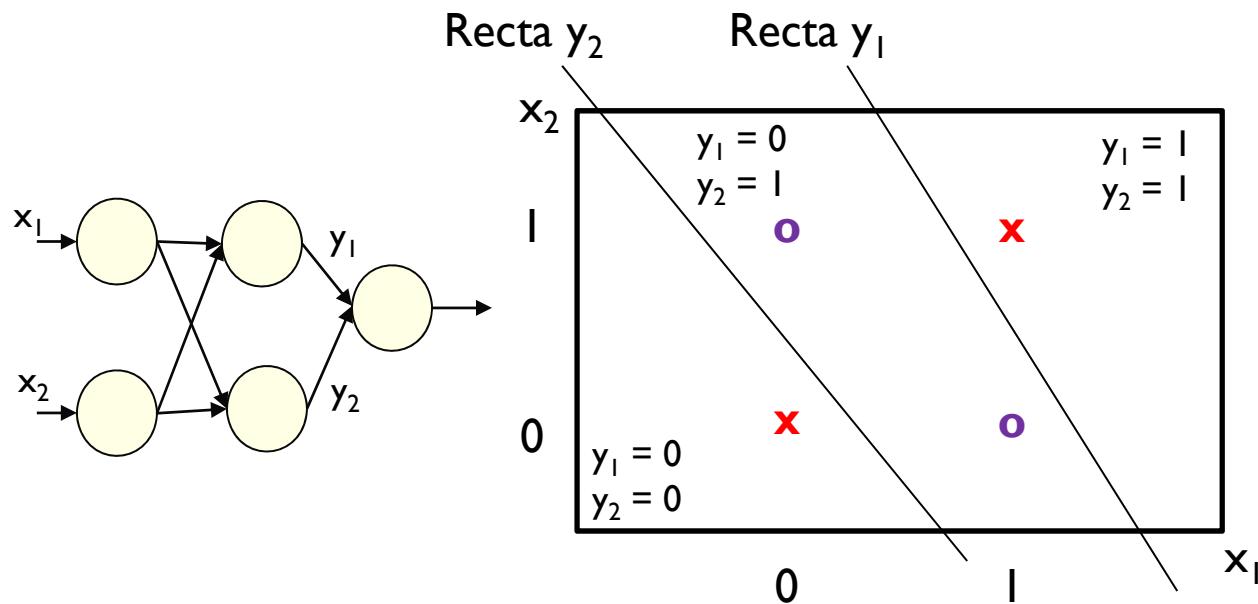
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - Solución: usar una red con dos neuronas
  - Cada neurona separa el espacio en dos partes
  - Cada combinación de  $(y_1, y_2)$  designa una zona del espacio original delimitada por las rectas:



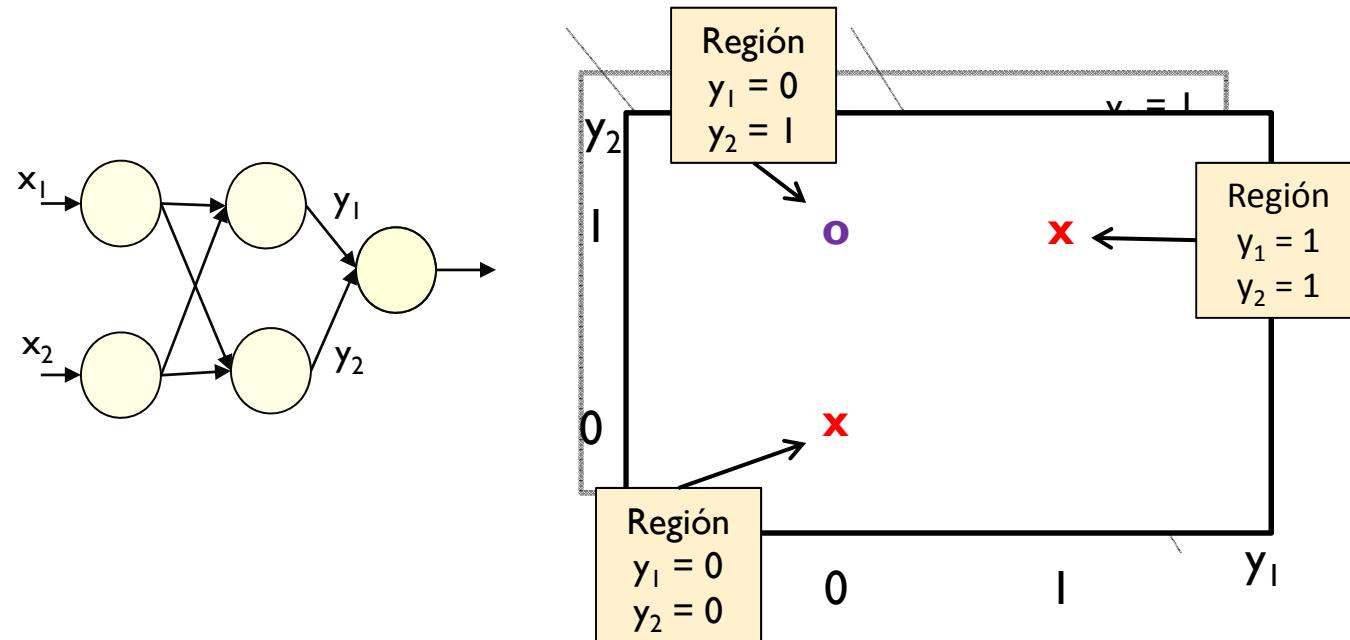
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - Para computar la salida, se añade una nueva capa, con una neurona de salida:
    - La neurona de salida recibe como entradas  $y_1, y_2$
    - Cada entrada ( $y_1, y_2$ ) es una región del espacio original



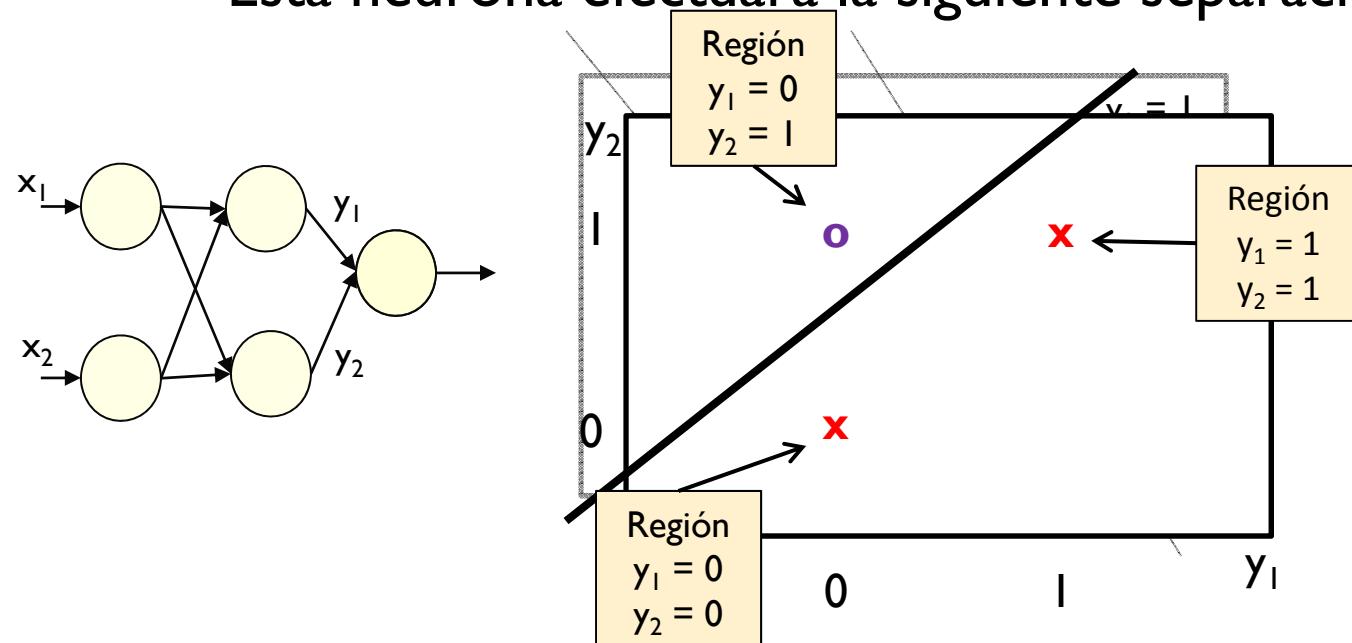
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - Para computar la salida, se añade una nueva capa, con una neurona de salida:
  - La neurona de salida recibe como entradas  $y_1, y_2$
  - Opera en el espacio formado por  $y_1, y_2$



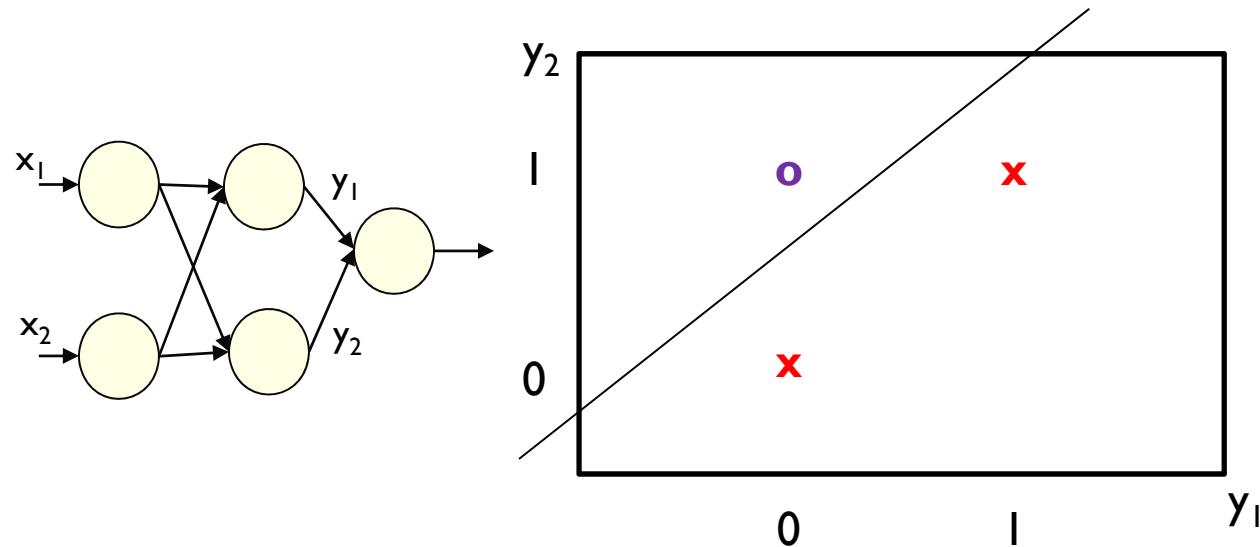
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - Para computar la salida, se añade una nueva capa, con una neurona de salida:
    - La neurona de salida recibe como entradas  $y_1, y_2$
    - Opera en el espacio formado por  $y_1, y_2$
    - Esta neurona efectuará la siguiente separación (lineal):



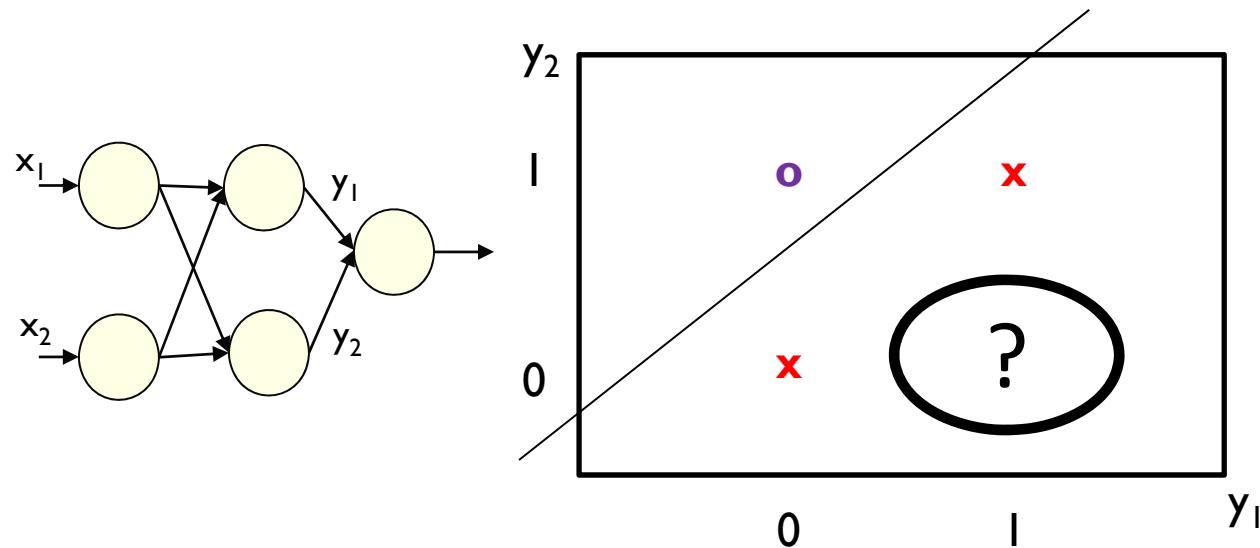
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - Para realizar esa operación:
    - Salida = 1 si ( $w_1 * y_1 + w_2 * y_2 + b > 0$ )
    - Salida = 0 si ( $w_1 * y_1 + w_2 * y_2 + b < 0$ )
    - Separación lineal



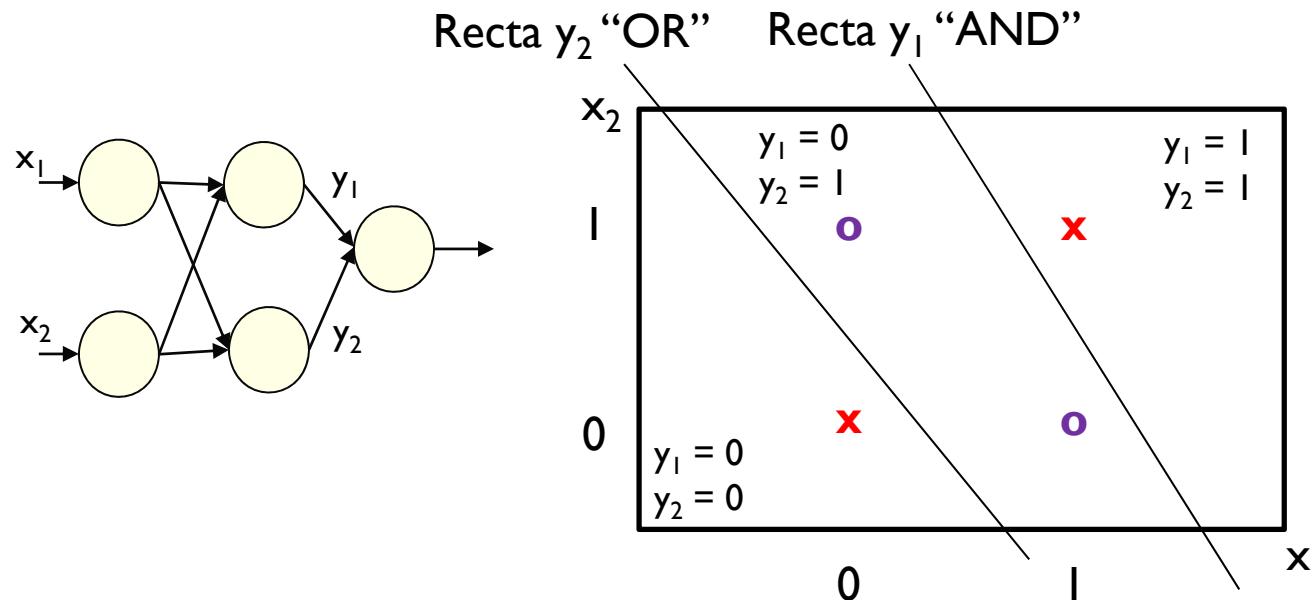
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - ¿Caso  $y_1=1, y_2=0$ ?
    - Caso no posible
    - No se puede crear el subespacio  $y_1=1, y_2=0$ 
      - Sería el espacio que cumpliese a la vez  
 $(x_1 \text{ AND } x_2)$  y NOT  $(x_1 \text{ OR } x_2)$



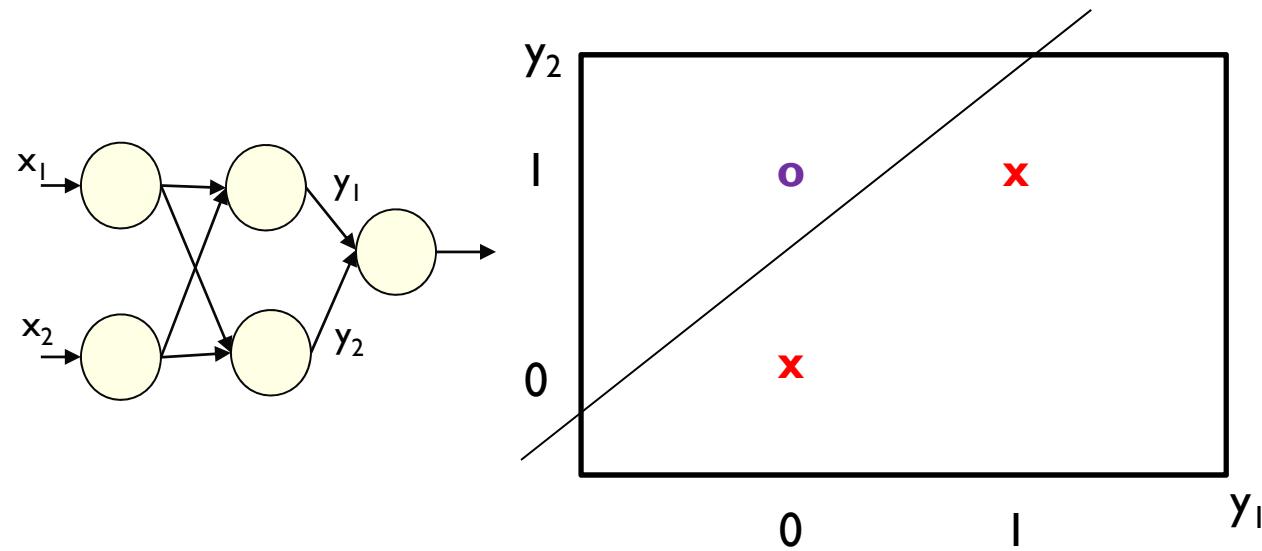
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - ¿Caso  $y_1=1, y_2=0$ ?
    - Caso no posible
    - No se puede crear el subespacio  $y_1=1, y_2=0$ 
      - Sería el espacio que cumpliese a la vez  
 $(x_1 \text{ AND } x_2)$  y NOT  $(x_1 \text{ OR } x_2)$



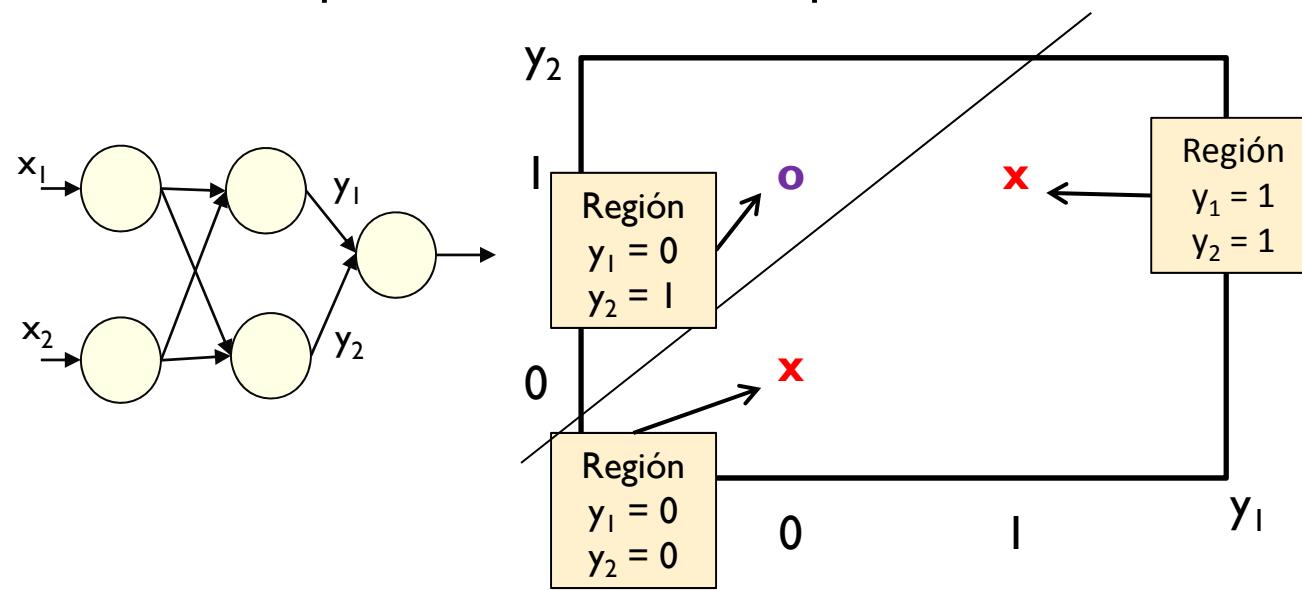
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - $x_1, x_2$  formaban un problema que no es linealmente separable
    - Pero  $y_1, y_2$  sí que lo son



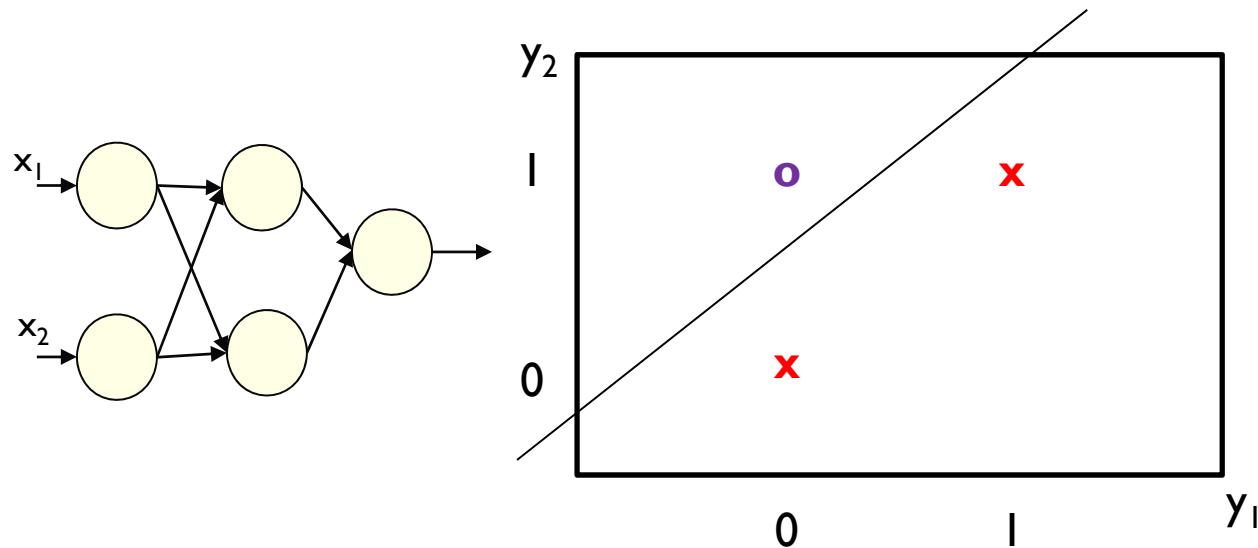
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - La neurona de salida recibe dos entradas
    - Recibe como entrada un espacio de 2 dimensiones
    - Cada entrada es cada uno de los semiespacios determinados por los hiperplanos de la primera capa oculta
    - Crea otro hiperplano (recta) que divide este nuevo espacio en dos partes, uniendo semiespacios



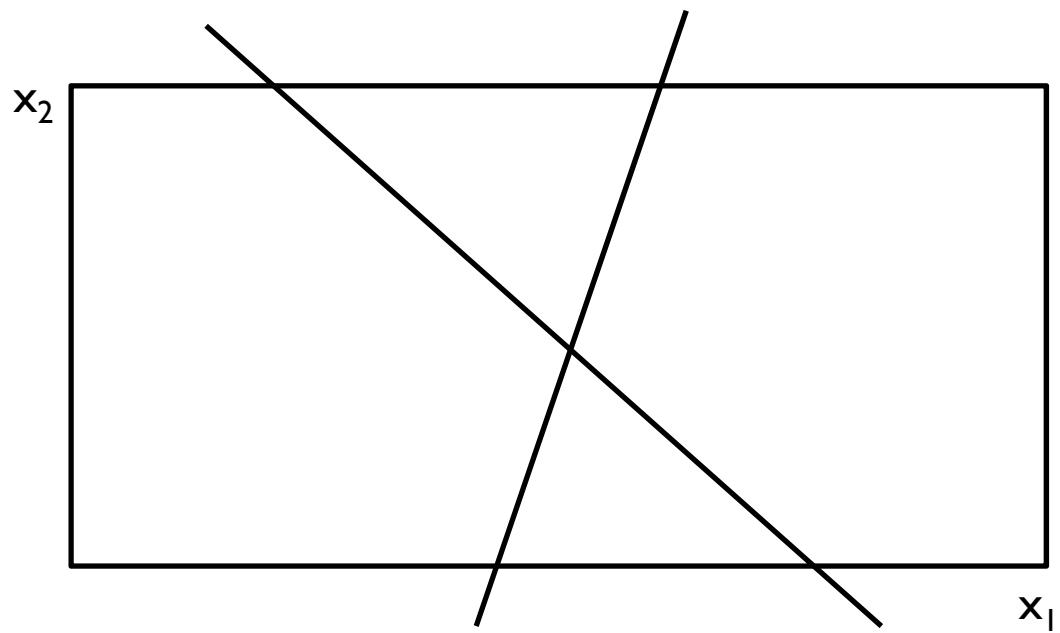
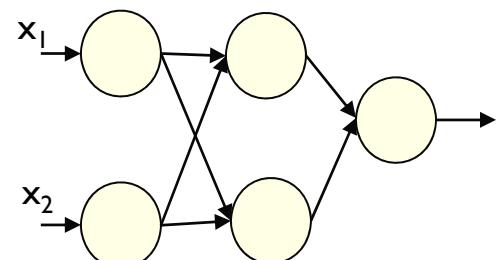
# FUNCIONAMIENTO DE UNA RED

- El problema del EXOR:
  - La neurona de salida combina los semiespacios especificados por las neuronas anteriores
    - Los combina asignándoles la misma salida
      - Asigna la misma salida a  $(y_1=1, y_2=1)$  y a  $(y_1=0, y_2=0)$



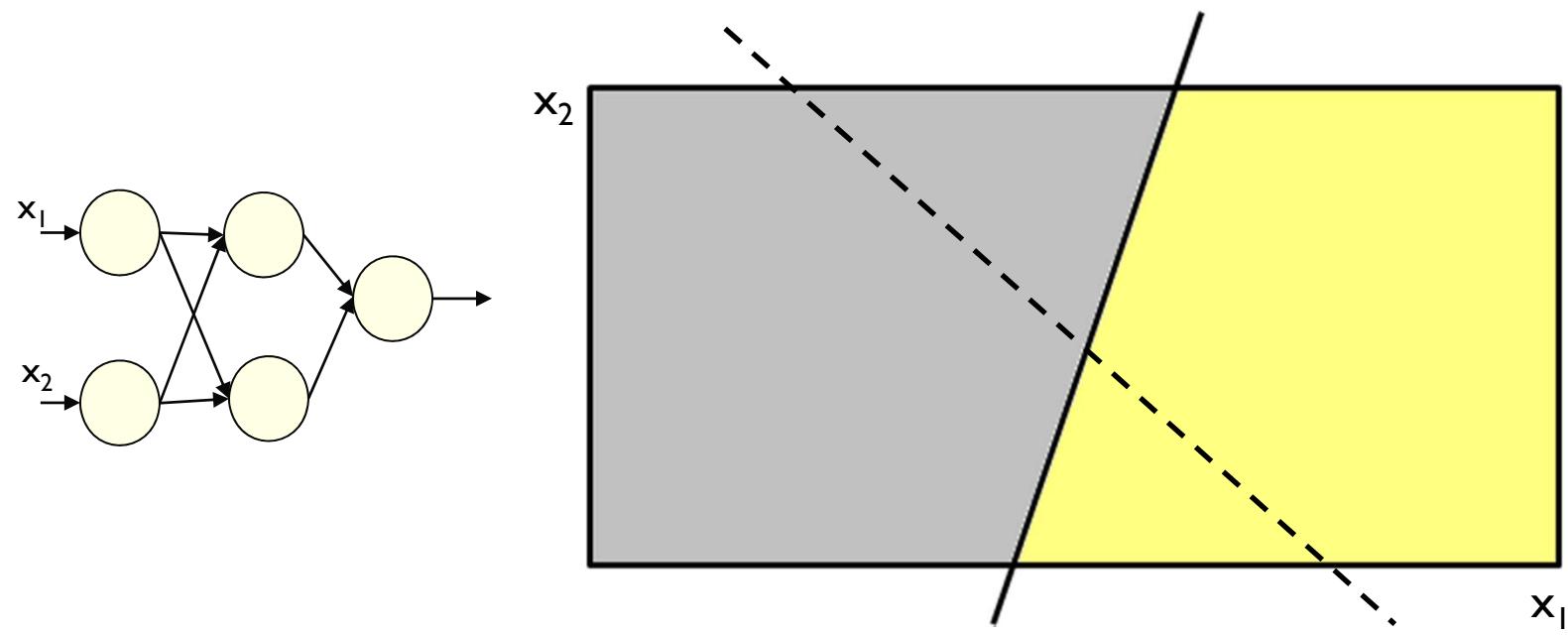
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta:



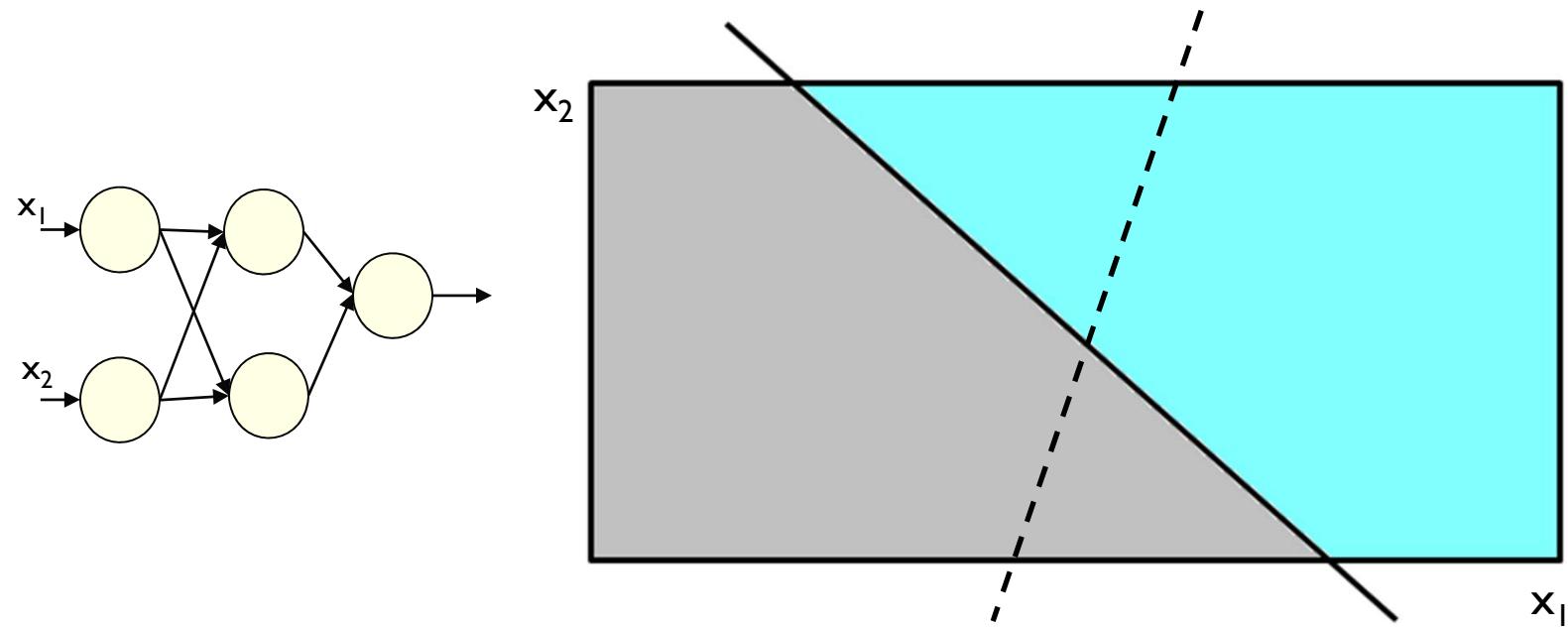
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta:
  - Las neuronas de la primera capa **crean** semiespacios



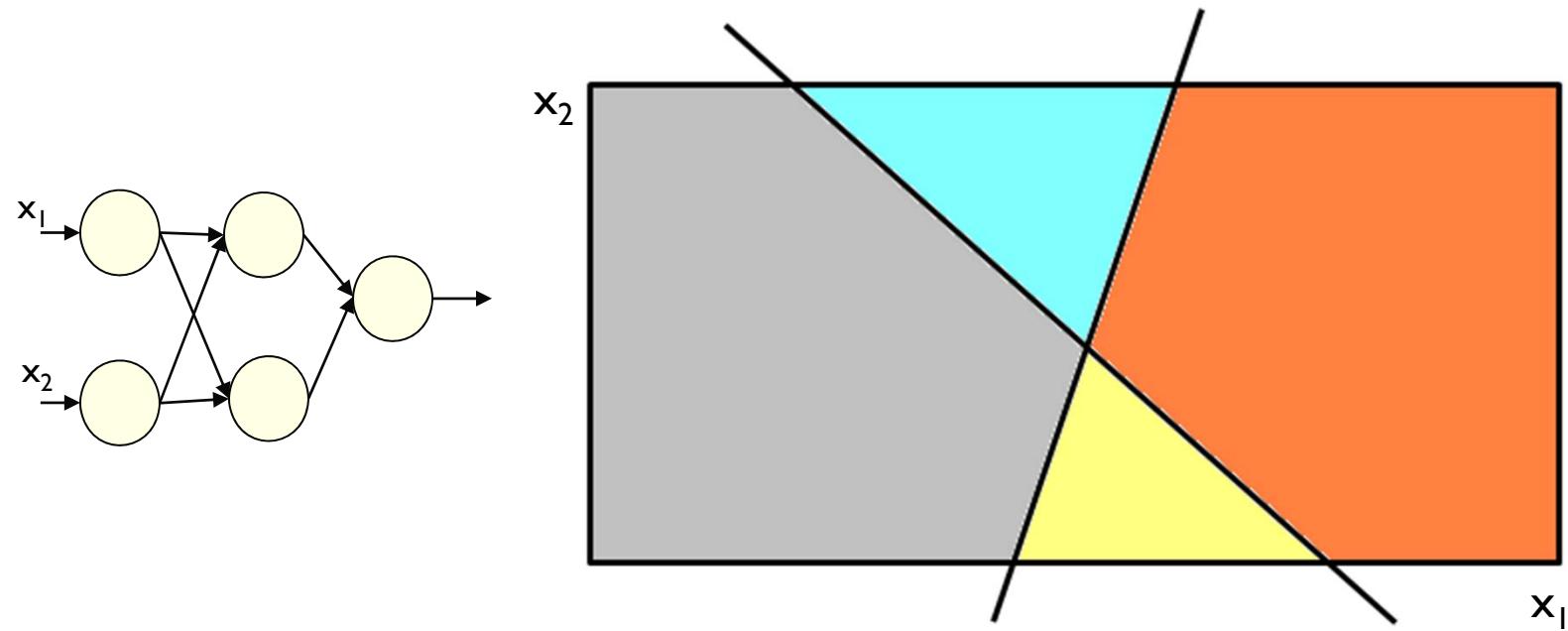
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta:
  - Las neuronas de la primera capa **crean** semiespacios



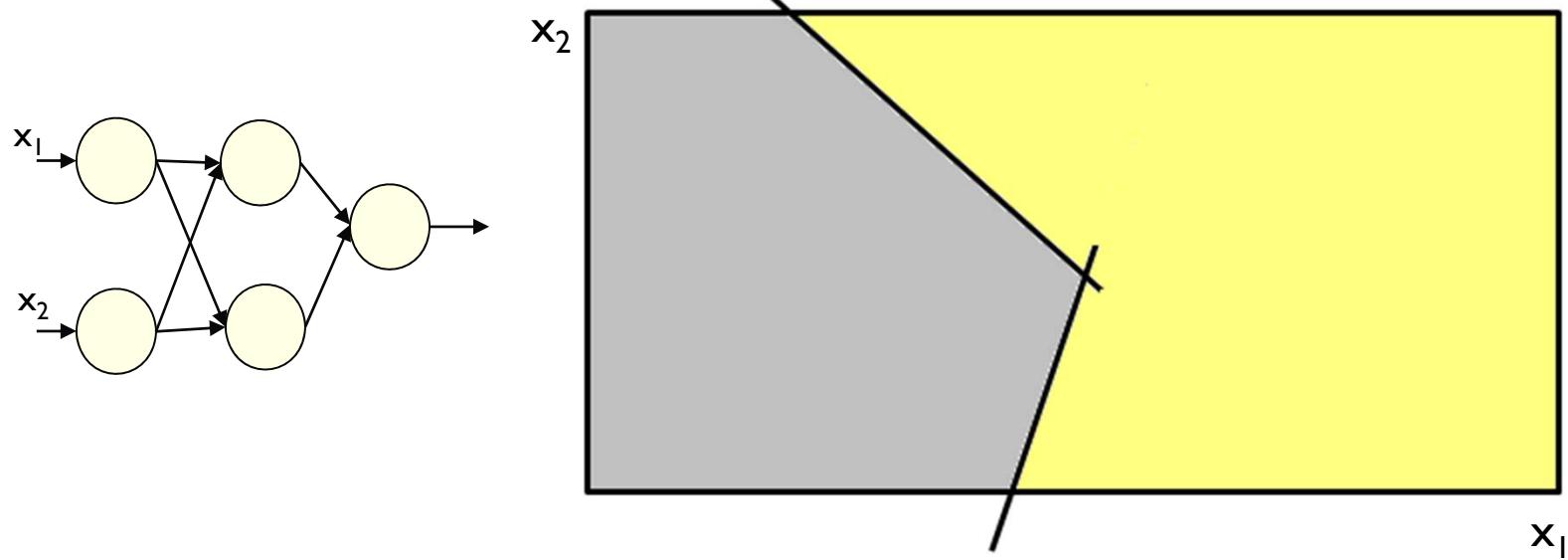
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta:
  - Las neuronas de la primera capa **crean** semiespacios



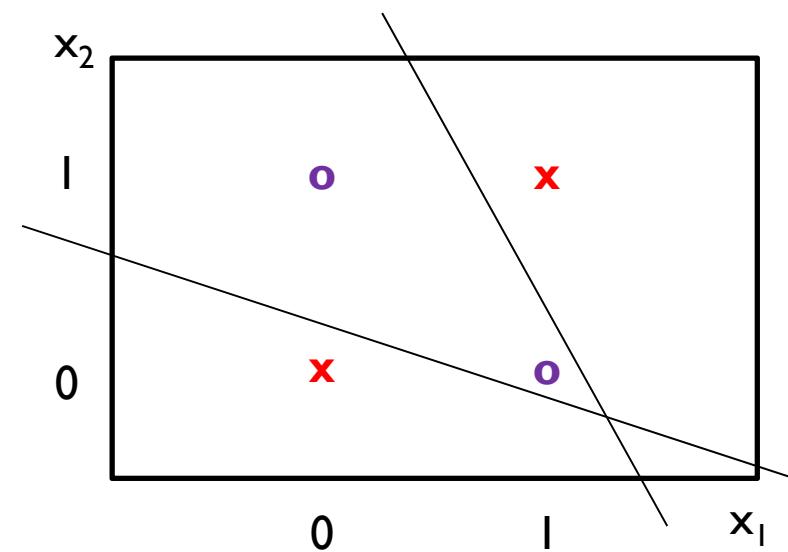
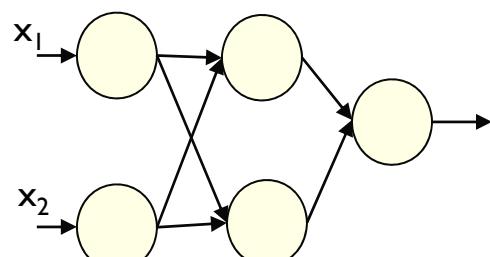
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta:
  - Las neuronas de la primera capa **crean** semiespacios
  - Las neuronas de la capa de salida **unen** semiespacios creando regiones
    - Asigna la misma salida



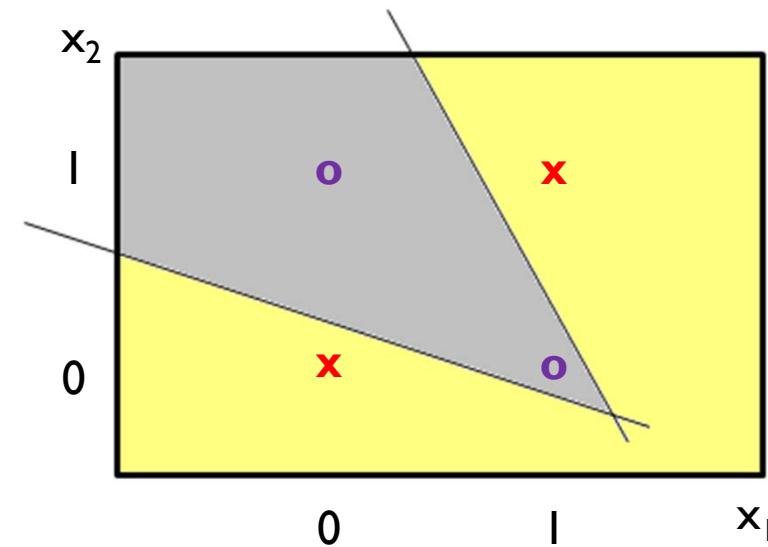
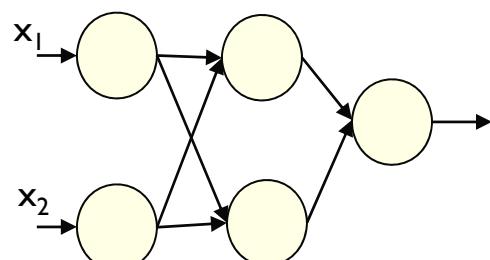
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta: El problema EXOR:
  - Las neuronas de la primera capa **crean** semiespacios
  - Las neuronas de la capa de salida **unen** semiespacios creando regiones



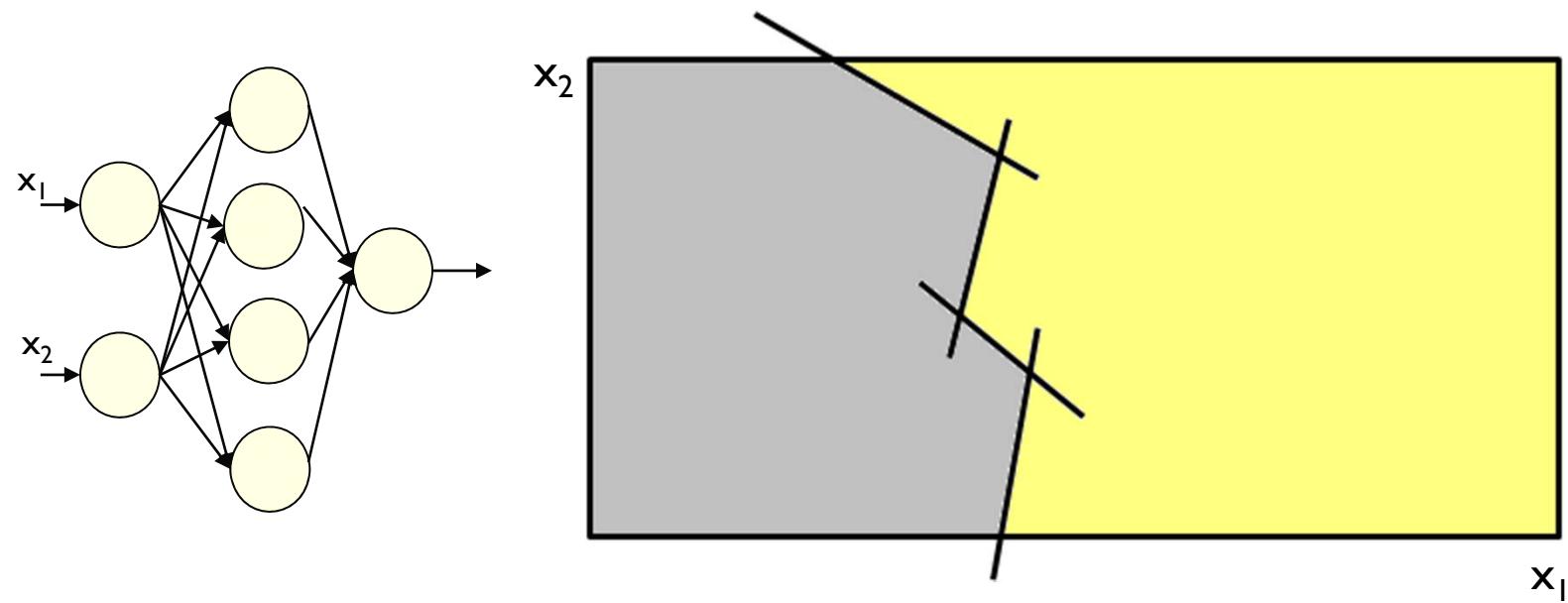
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta: El problema EXOR:
  - Las neuronas de la primera capa **crean** semiespacios
  - Las neuronas de la capa de salida **unen** semiespacios creando regiones



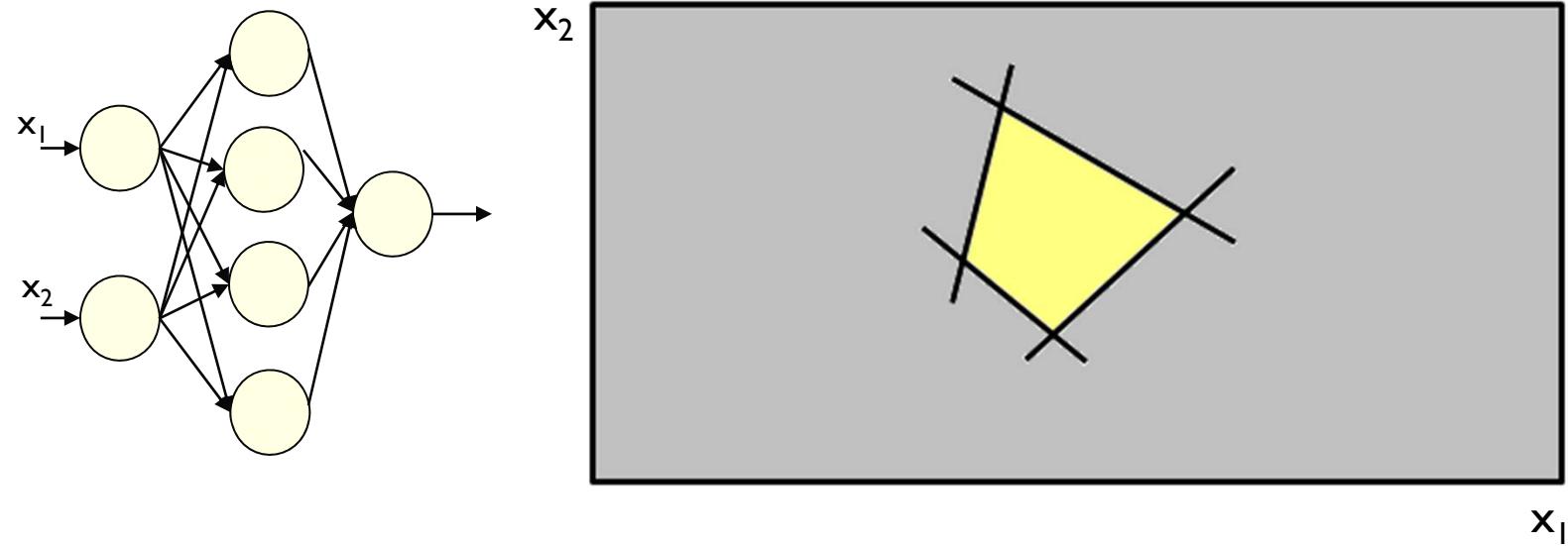
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta:
  - Se puede dividir el espacio en dos partes arbitrariamente complejas
    - Pero cada parte forma un solo bloque



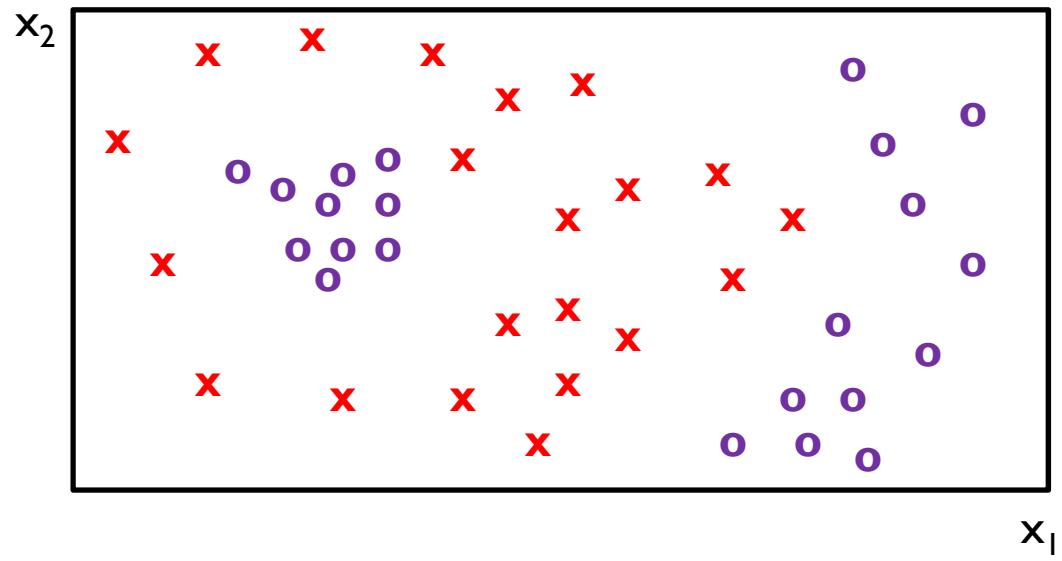
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta:
  - Se puede dividir el espacio en dos partes arbitrariamente complejas
    - Pero cada parte forma un solo bloque



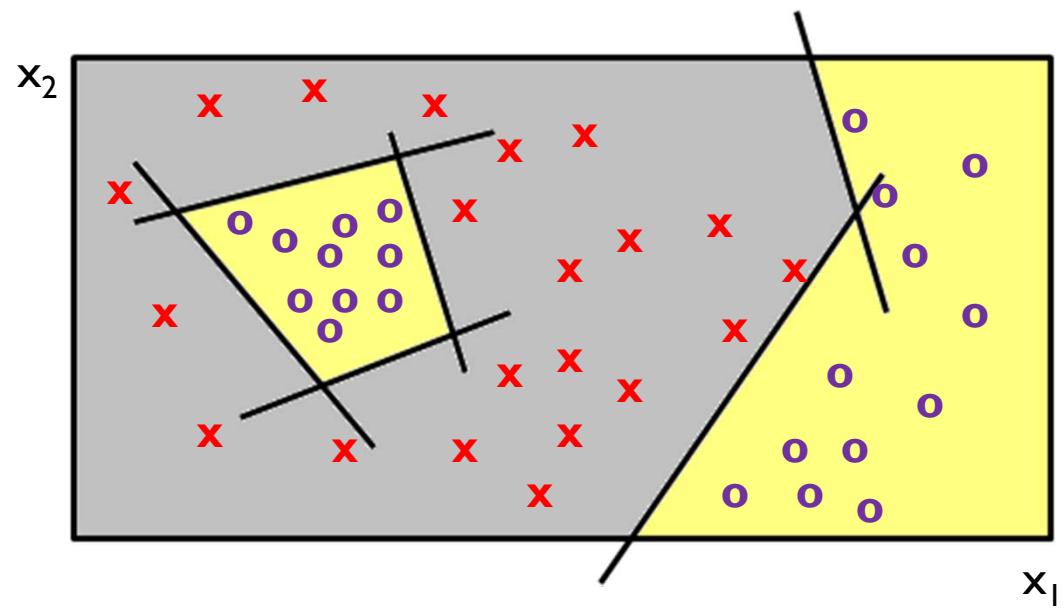
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta:
  - ¿Cómo dividir el espacio de forma más compleja?



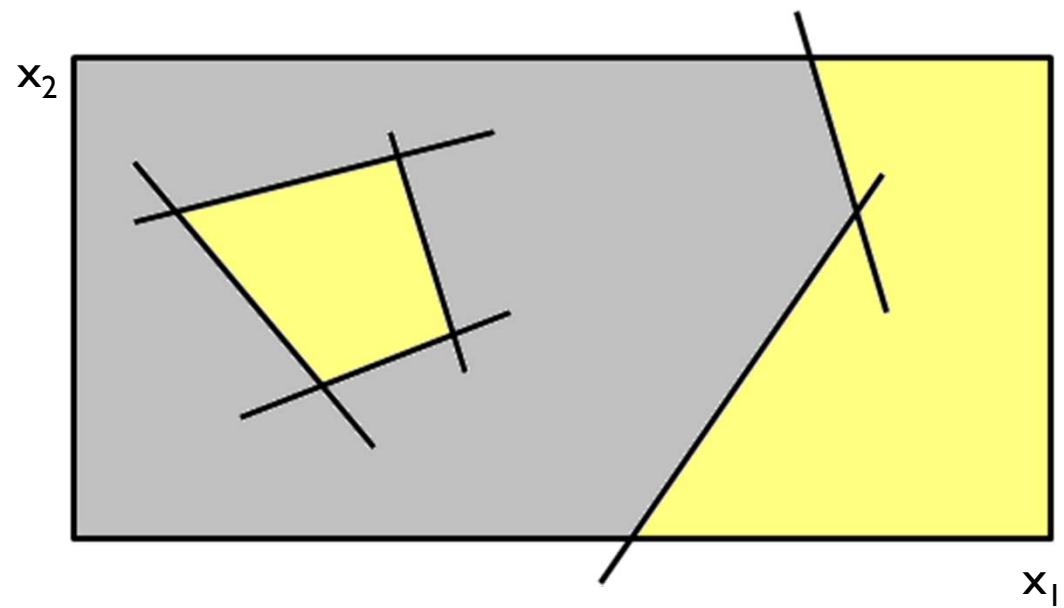
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta:
  - ¿Cómo dividir el espacio de forma más compleja?



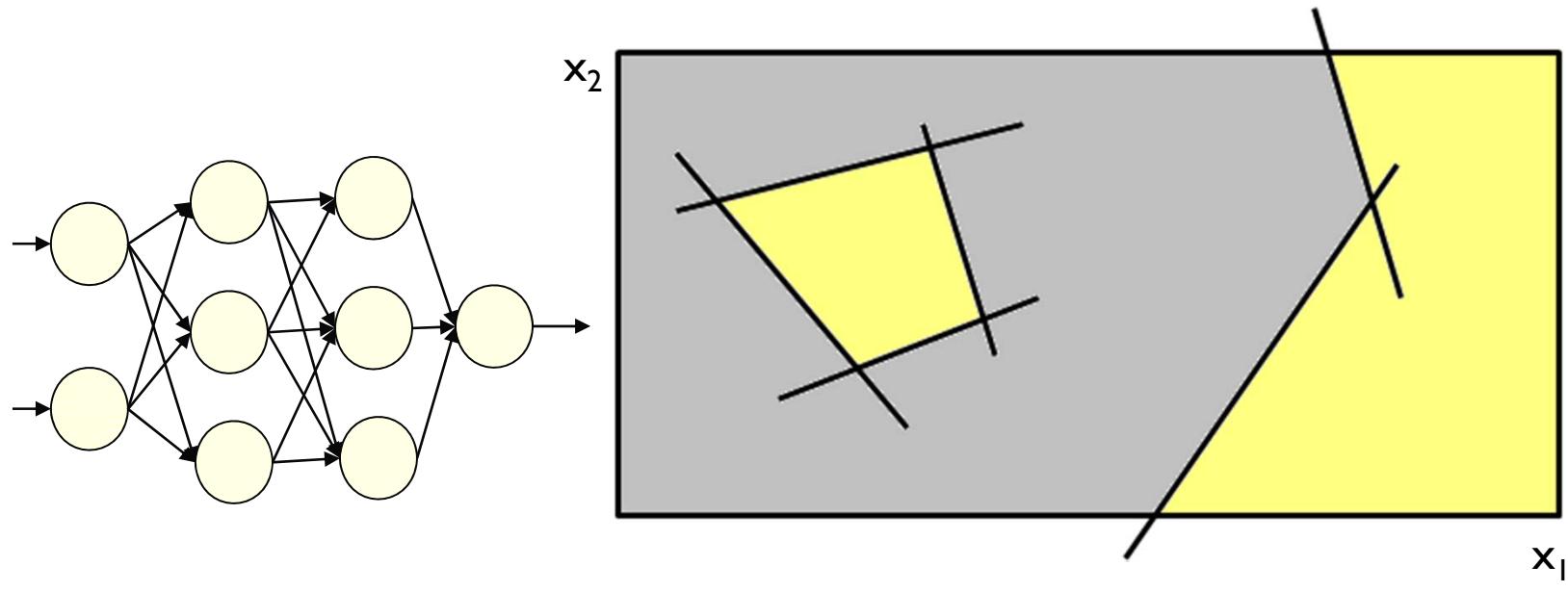
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta:
  - ¿Cómo dividir el espacio de forma más compleja?



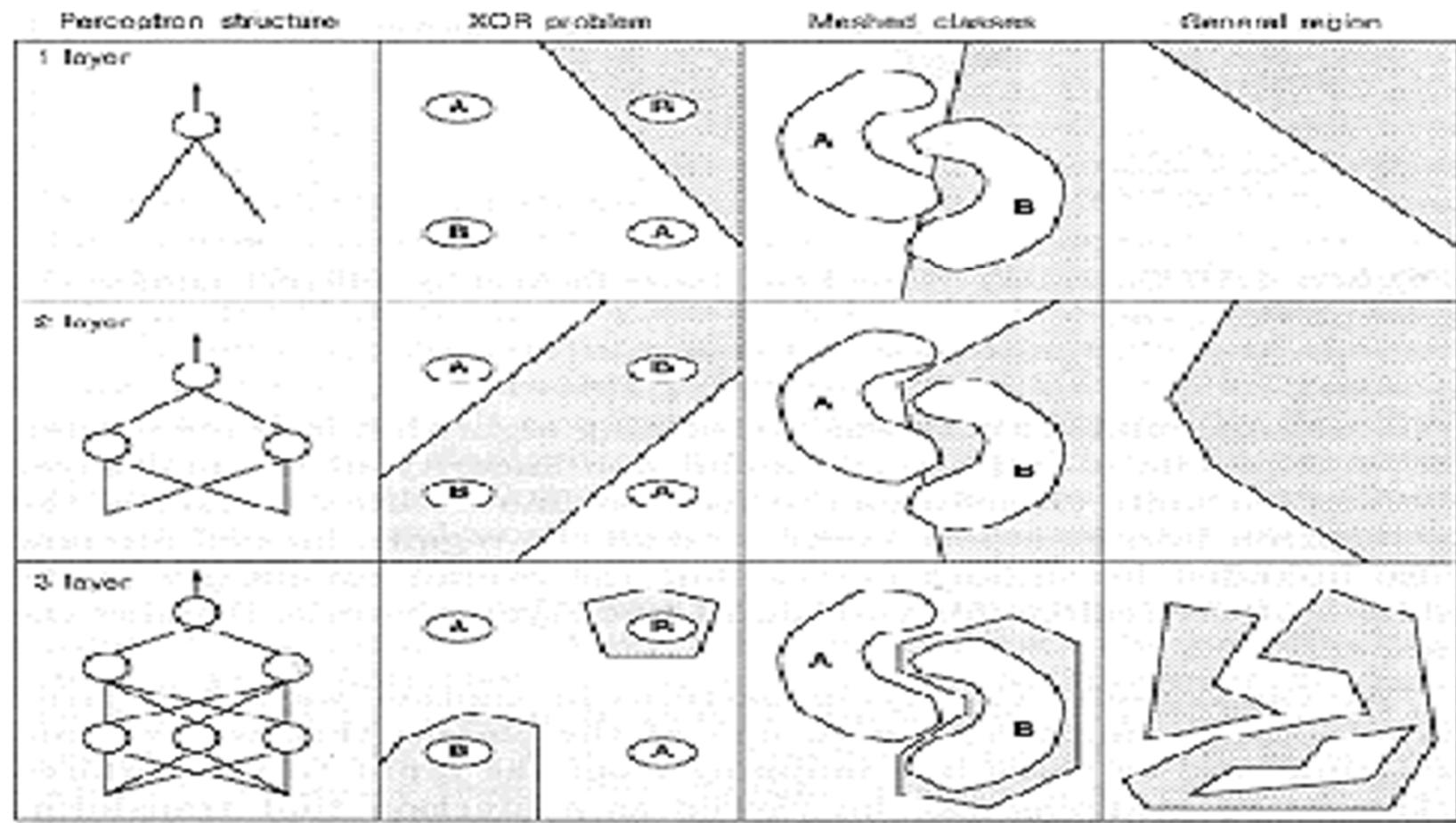
# FUNCIONAMIENTO DE UNA RED

- RNA con una capa oculta:
  - Solución: añadir una segunda capa oculta
    - Primera capa oculta: divide el espacio en semiespacios
    - Segunda capa oculta: une semiespacios creando regiones
    - Tercera capa (salida): une regiones que pertenecen a la misma clase



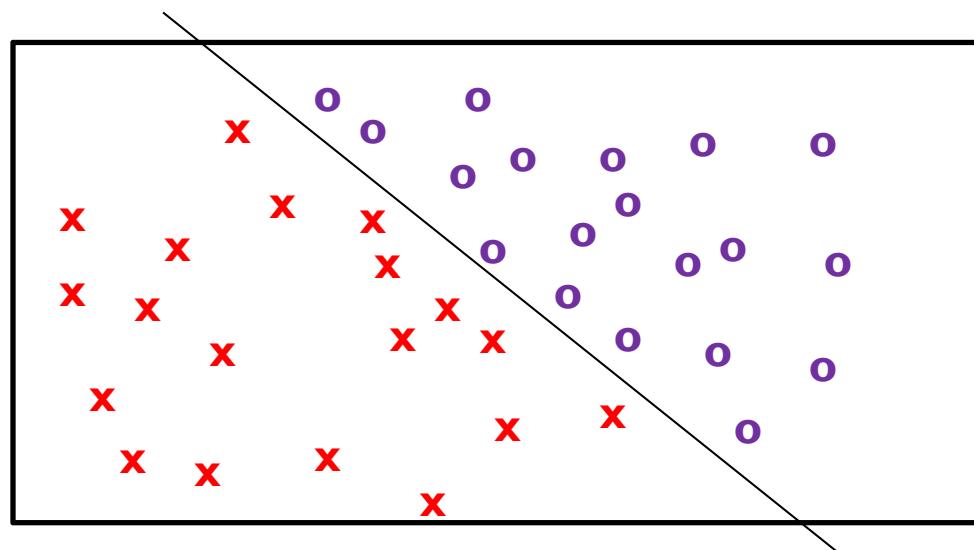
# FUNCIONAMIENTO DE UNA RED

- RNA con varias capas ocultas:



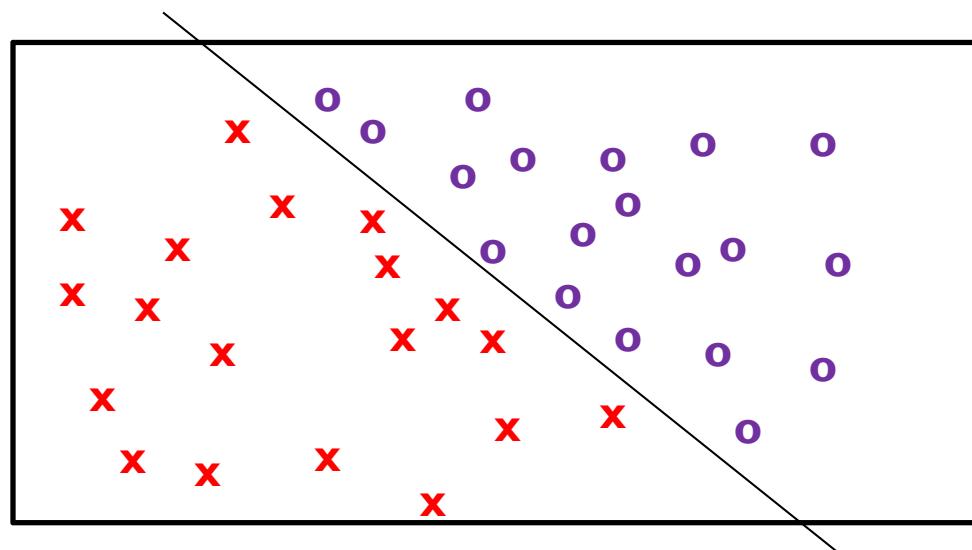
# SOBREENTRENAMIENTO

- Los patrones se distribuyen en el espacio
  - Situación ideal: poder visualizarlos y saber cuántas neuronas se necesitan para separarlos
    - En general, tienen N dimensiones: imposible visualizar
    - Ejemplo: linealmente separables: con una RNA con sólo una neurona de salida se pueden separar



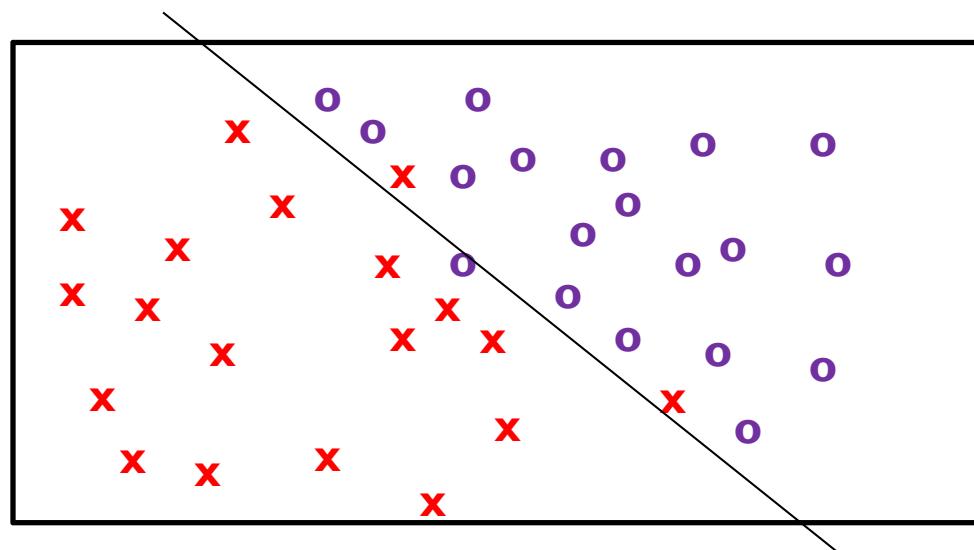
# SOBREENTRENAMIENTO

- El mundo real:
  - No se pueden visualizar los patrones
  - Estos tienen ruido
    - Los patrones se han movido de la situación que deberían tener
    - Ejemplo: Patrones sin ruido:



# SOBREENTRENAMIENTO

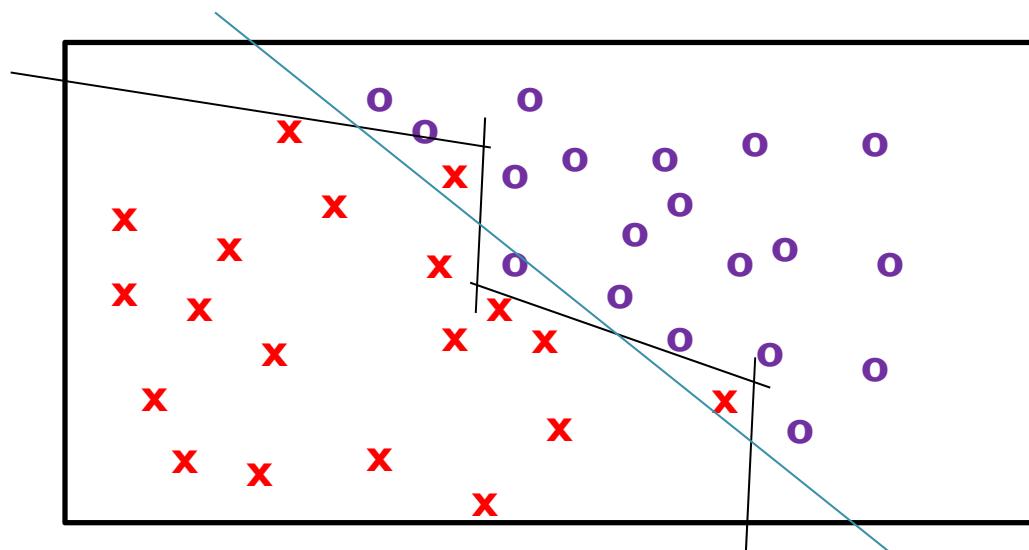
- El mundo real:
  - No se pueden visualizar los patrones
  - Estos tienen ruido
    - Los patrones se han movido de la situación que deberían tener
    - Ejemplo: Patrones con ruido:



# SOBREENTRENAMIENTO

- El mundo real:

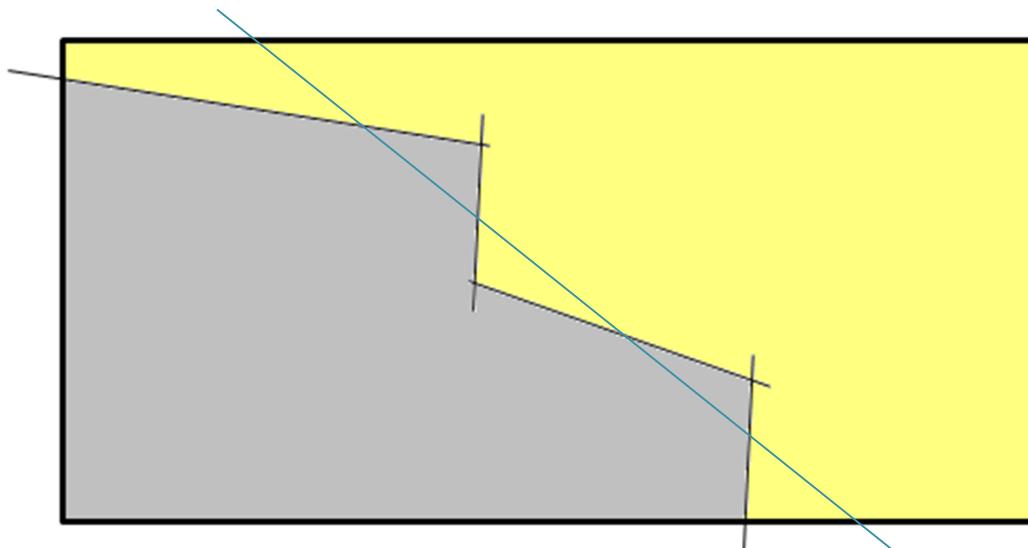
- Si se usa una RNA con 4 neuronas en la primera capa y una capa de salida, se podría generar esto:
  - 100% de precisión en el conjunto de **entrenamiento**
  - Sin embargo, el sistema no generaliza bien: ha aprendido el ruido
  - Con nuevos patrones, el sistema se comportará mal



# SOBREENTRENAMIENTO

- El mundo real:

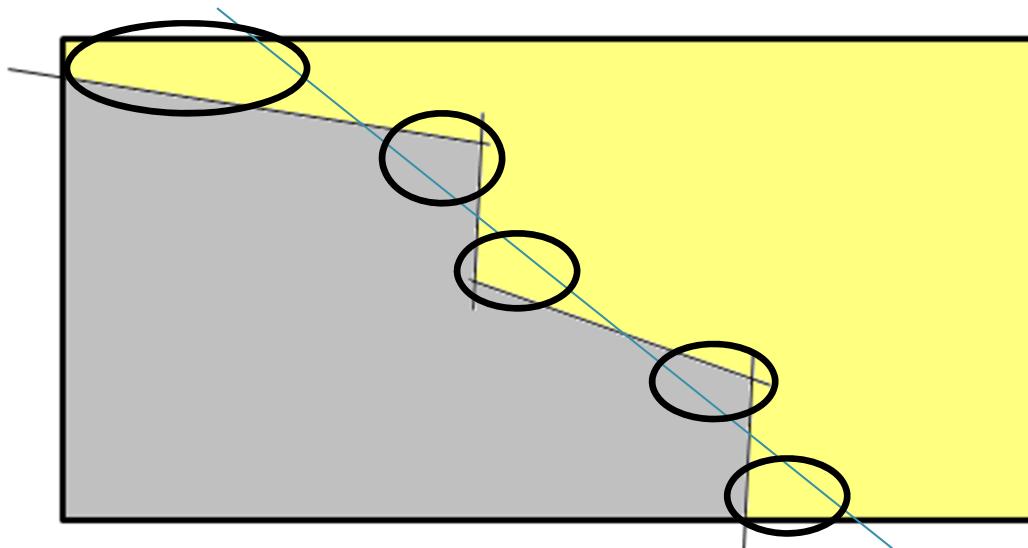
- Si se usa una RNA con 4 neuronas en la primera capa y una capa de salida, se podría generar esto:
  - 100% de precisión en el conjunto de **entrenamiento**
  - Sin embargo, el sistema no generaliza bien: ha aprendido el ruido
  - Con nuevos patrones, el sistema se comportará mal



# SOBREENTRENAMIENTO

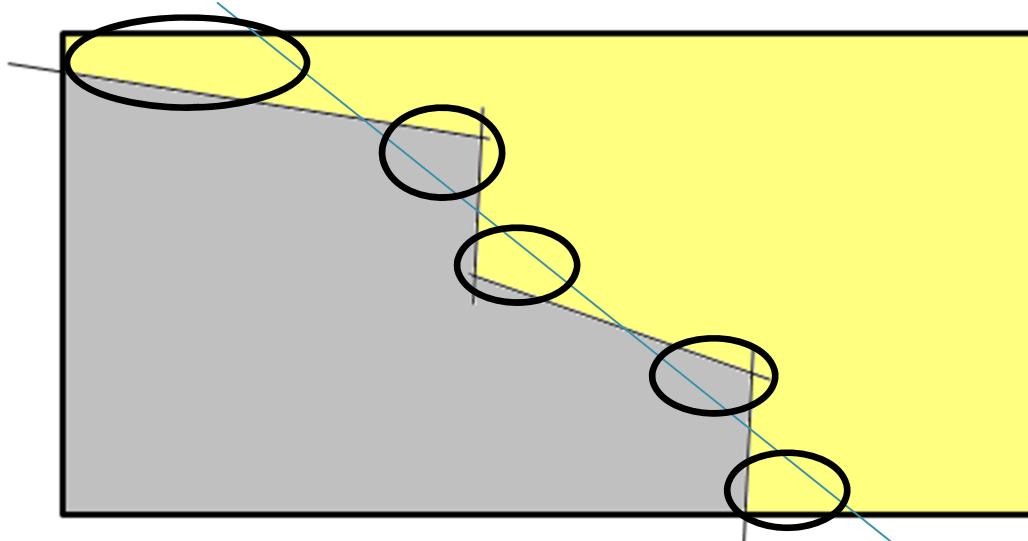
- El mundo real:

- Si se usa una RNA con 4 neuronas en la primera capa y una capa de salida, se podría generar esto:
  - 100% de precisión en el conjunto de **entrenamiento**
  - Sin embargo, el sistema no generaliza bien: ha aprendido el ruido
  - Con nuevos patrones, el sistema se comportará mal, si están en estas zonas:



# SOBREENTRENAMIENTO

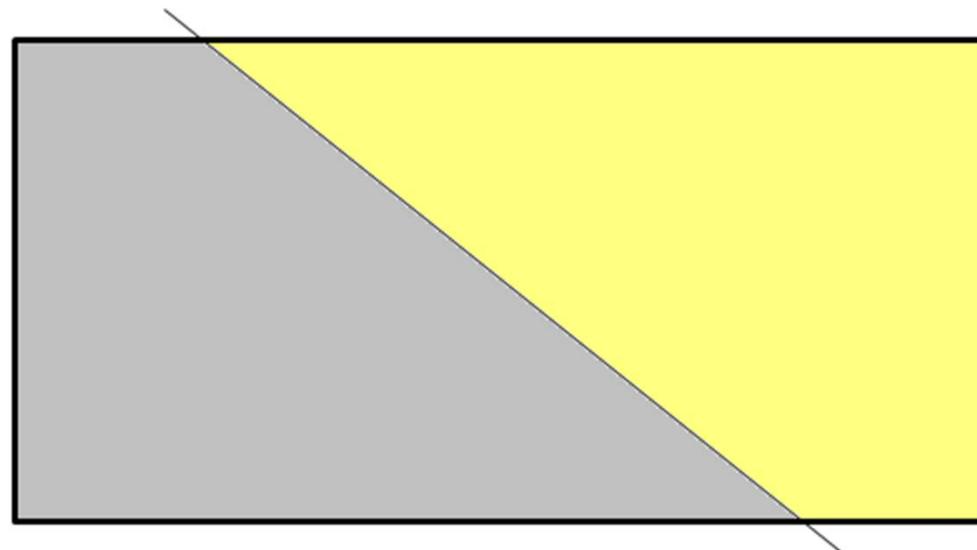
- Primera causa del sobreentrenamiento:
  - Red demasiado compleja para el problema a resolver
    - Pero a priori no se sabe cuál es la complejidad idónea





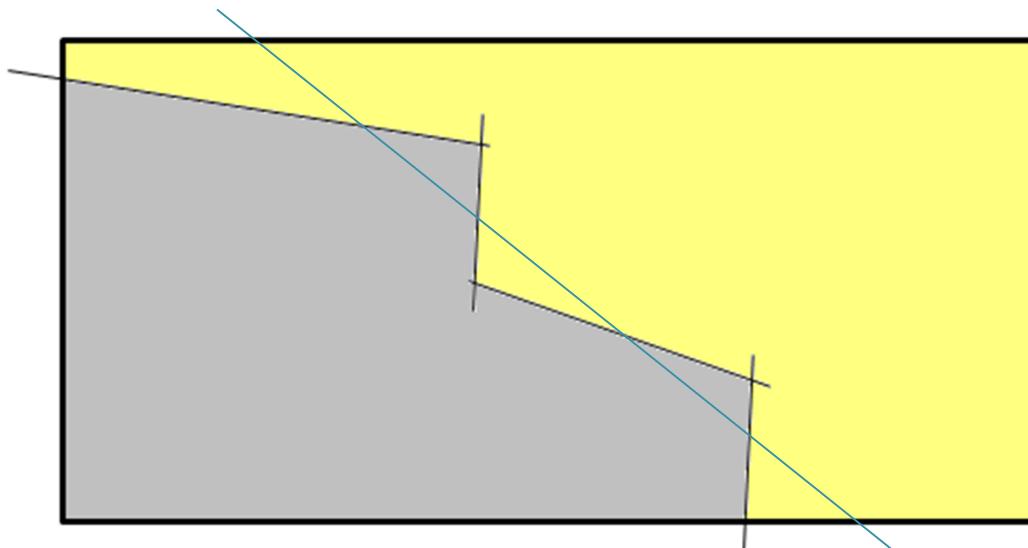
# SOBREENTRENAMIENTO

- El proceso de entrenamiento de esta red compleja será así:
  - Primeros ciclos:
    - Redundancia en la red: todavía no ha aprendido formas complejas



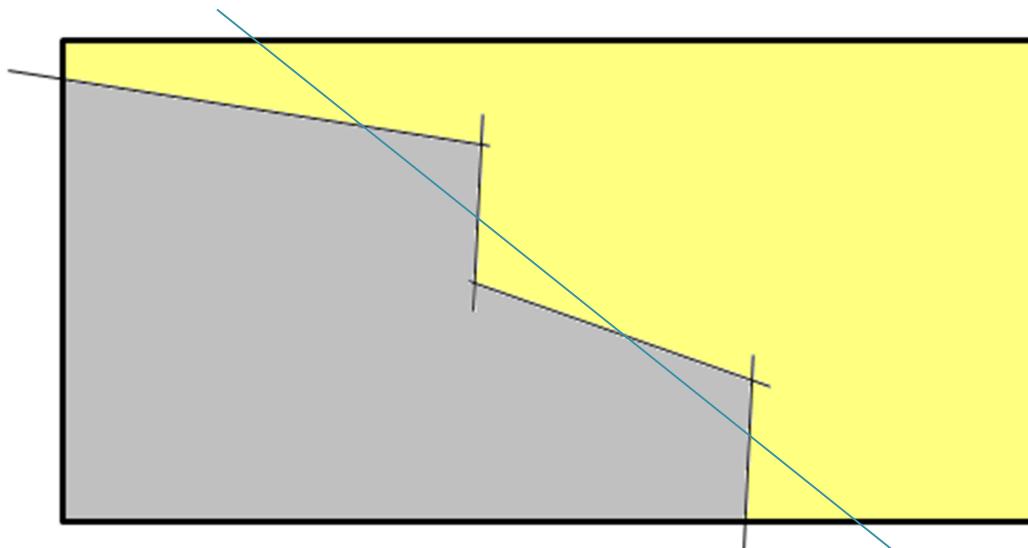
# SOBREENTRENAMIENTO

- El proceso de entrenamiento de esta red compleja será así:
  - Ciclos posteriores:
    - Se ha forzado a la red a aprender las formas complejas



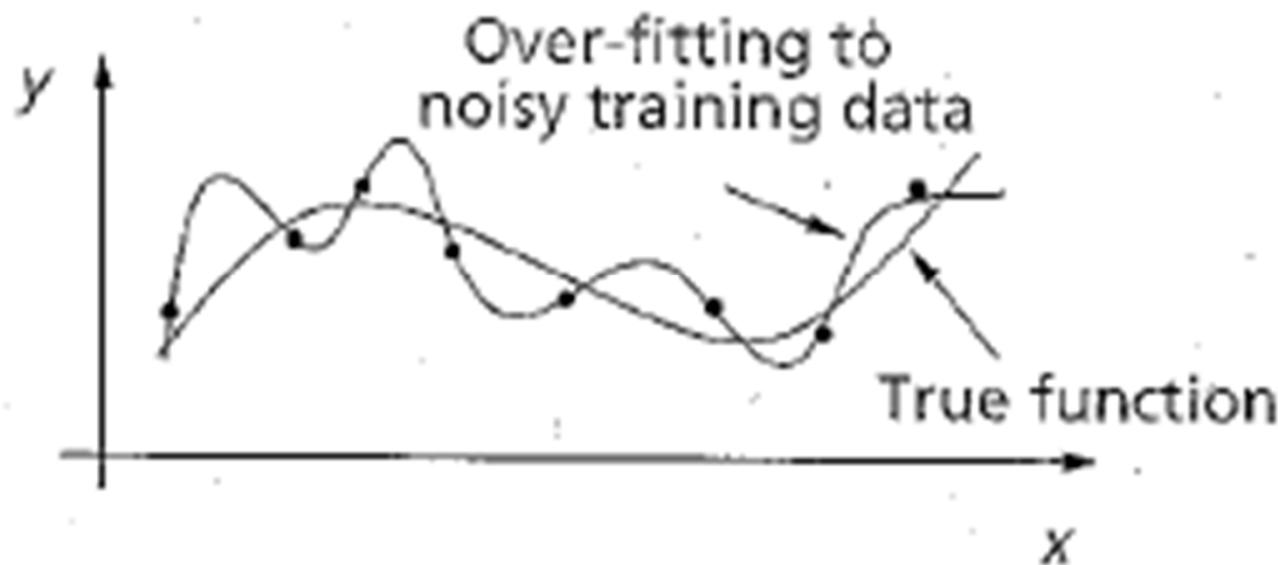
# SOBREENTRENAMIENTO

- Segunda causa del sobreentrenamiento:
  - Forzar la red a aprender durante demasiados ciclos
    - Habitualmente, el sobreentrenamiento se da por ambas causas a la vez



# SOBREENTRENAMIENTO

- Ejemplo:
  - RNA para modelizar una función desconocida  $y=f(x)$
  - Conjunto de entrenamiento formado por pares  $(x, y)$ , resultado de mediciones
    - Ante una determinada entrada  $x_i$ , se ha registrado una salida  $y_i$
    - La relación entre  $x_i$  e  $y_i$  es desconocida
    - Estos datos contienen cierto ruido





# SOBREENTRENAMIENTO

- Ejemplo:
  - Cáncer de mama – Wisconsin:
  - 683 pacientes
  - 31 atributos
  - 1 salida deseada: cáncer benigno/maligno
  - Se sabe que esta base de datos tiene ruido, y si se consigue una precisión en la clasificación superior a determinado porcentaje, el sistema está sobreentrenado



# SOBREENTRENAMIENTO

- Cómo evitar el sobreentrenamiento:
  - **Regularización**
  - Varias técnicas disponibles:
    - L1: añadir a la función de coste la suma de los valores absolutos de los pesos
    - L2: añadir a la función de coste la suma de los cuadrados valores de los pesos
    - Decaimiento de pesos
    - *Dropout*
    - Normalización por lotes (*batch normalization*)
    - Aumento de datos (*Data Augmentation*)
    - **Parada temprana** (*Early Stopping*)



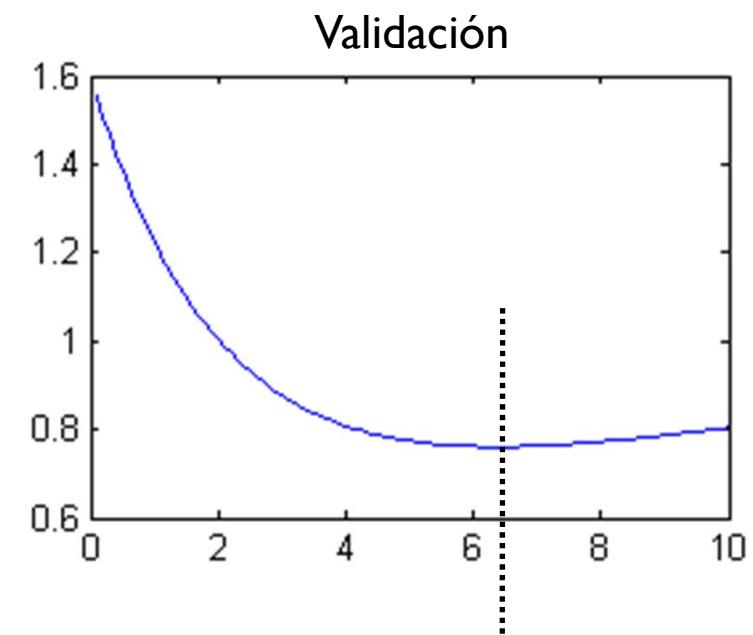
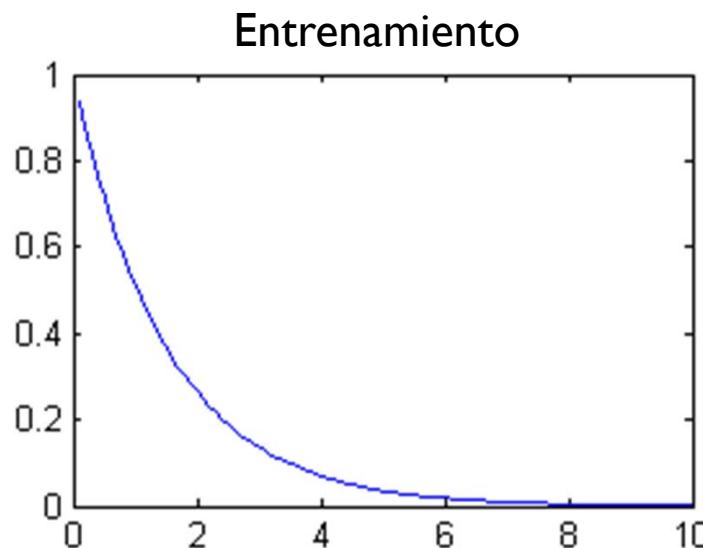
# SOBREENTRENAMIENTO

- Cómo evitar el sobreentrenamiento:

- Utilizar un conjunto de validación
  - No interfiere en el entrenamiento pero lo supervisa y lo para si es necesario
  - Habitualmente, este conjunto se obtiene dividiendo el conjunto de entrenamiento en entrenamiento y validación
    - Disjuntos!
- A la vez que se entrena la red, se va evaluando con el conjunto de validación
  - Se realiza una “estimación” de cómo generaliza la red, pero no interviene en el entrenamiento
    - Se espera que el conjunto de test tendrá un error similar al del conjunto de validación
  - Si el error en el conjunto de validación aumenta, la red se está sobreentrenando
    - Porque aumenta el error con patrones que no intervienen en el entrenamiento
    - Parar el entrenamiento!

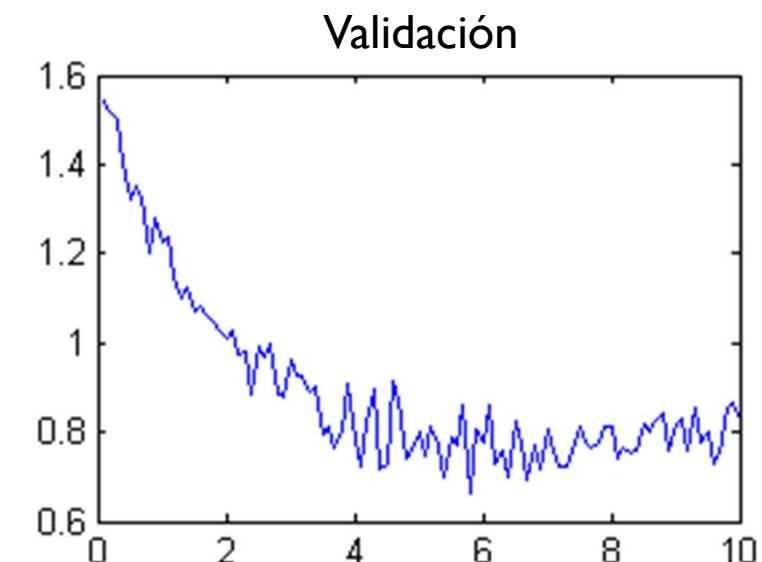
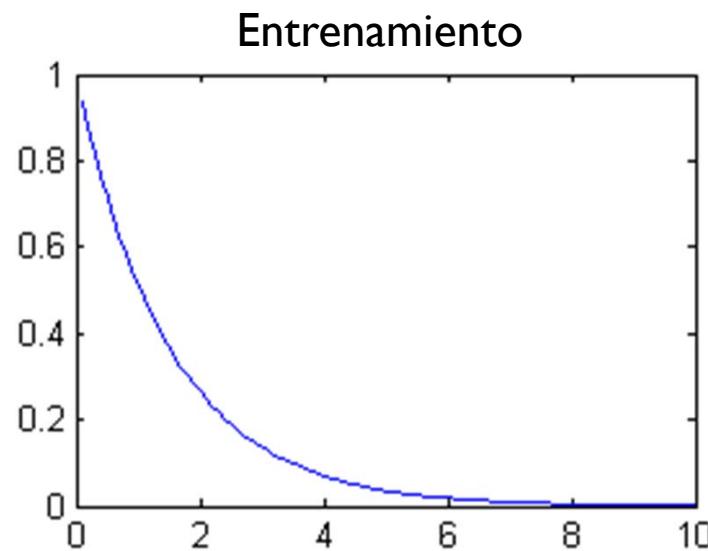
# SOBREENTRENAMIENTO

- Cómo evitar el sobreentrenamiento:
  - A la vez que se entrena la red, se va evaluando con el conjunto de validación
  - Resultados en validación ideal:



# SOBREENTRENAMIENTO

- Cómo evitar el sobreentrenamiento:
  - A la vez que se entrena la red, se va evaluando con el conjunto de validación
  - Resultados en validación real:



- ¿Cuándo parar el entrenamiento?



# SOBREENTRENAMIENTO

- Parar el entrenamiento:
  - Prechelt L. Early Stopping-But When? *Neural Networks: Tricks of the Trade*, 1996, 55-69
  - Varias estrategias:
    - Establecer un nuevo parámetro: número de ciclos sin mejorar la validación
      - Si transcurre este número de ciclos sin mejorar el error de validación, se para el entrenamiento
    - Entrenar durante un número de ciclos muy alto y, durante el entrenamiento, memorizar la red con mejor resultado de validación
      - Cuando finalice el entrenamiento, se devuelve esa red y no la red final entrenada (posiblemente sobreentrenada)
    - etc.



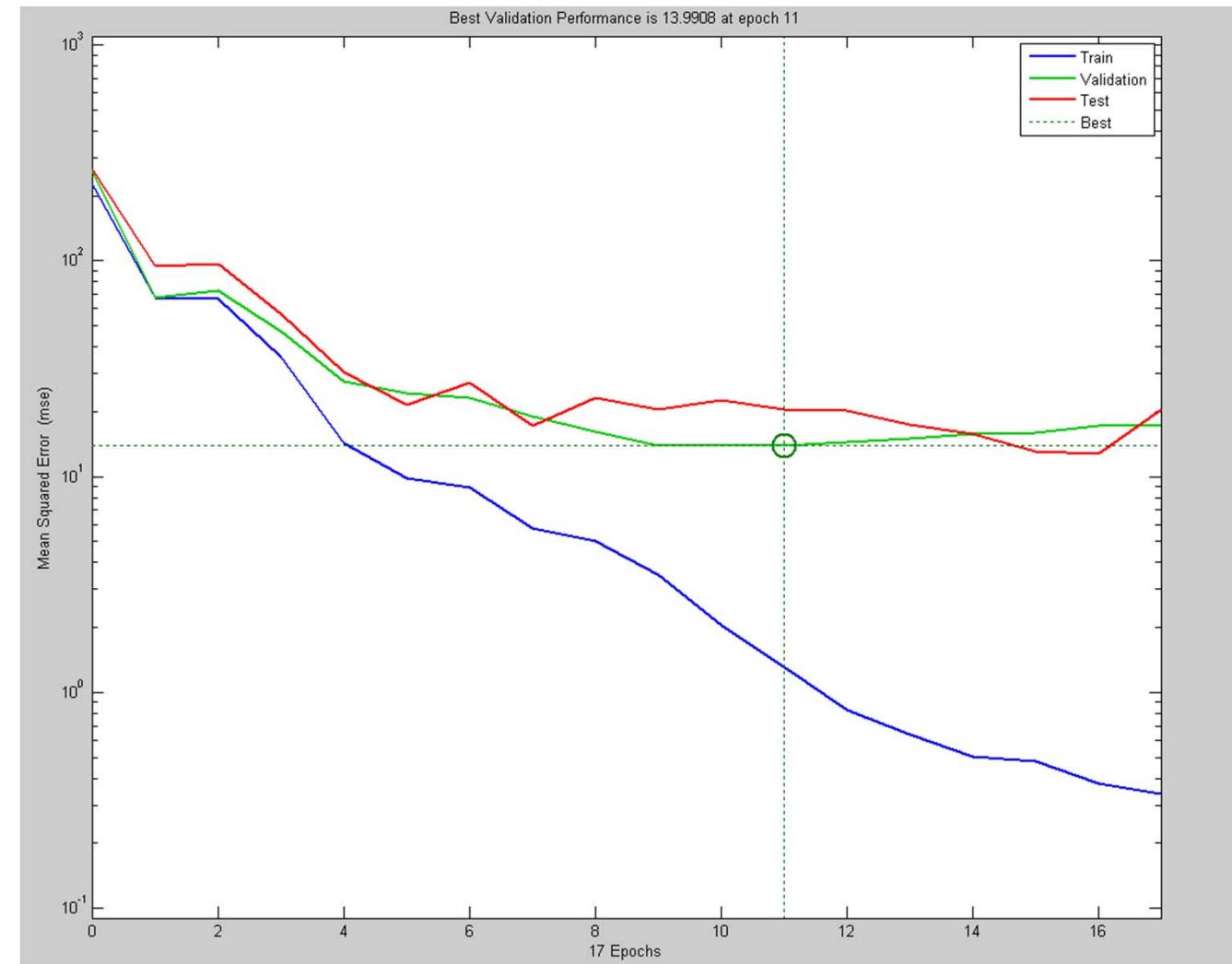
# SOBREENTRENAMIENTO

- Para entrenar correctamente hacen falta 3 conjuntos de patrones:
  - Entrenamiento
    - Guía el proceso de entrenamiento
  - Validación
    - No participa directamente en el entrenamiento
    - Supervisa el entrenamiento, y lo para si es necesario
    - Dictamina cuál es la red a devolver cuando finaliza el entrenamiento
  - Test
    - No interviene en nada en el entrenamiento
    - Se evalúa una vez haya finalizado el entrenamiento, con la red resultante (posiblemente la de mejor validación)
    - Analiza la capacidad de generalización de la red
    - Es la única manera de saber si la red está bien entrenada
      - El conjunto de entrenamiento guía el entrenamiento y el de validación, de alguna manera, también participa
      - Es necesario este conjunto puesto que no interviene en nada
    - Generalmente este conjunto se tiene antes de entrenar, así que, a medida que se entrena, se va calculando ese error de test
      - Pero este valor de error NO se utiliza para tomar ninguna decisión

# SOBREENTRENAMIENTO

- Para entrenar correctamente hacen falta 3 conjuntos de patrones:

Red devuelta: la de menor error en el conjunto de validación, NO en el conjunto de test



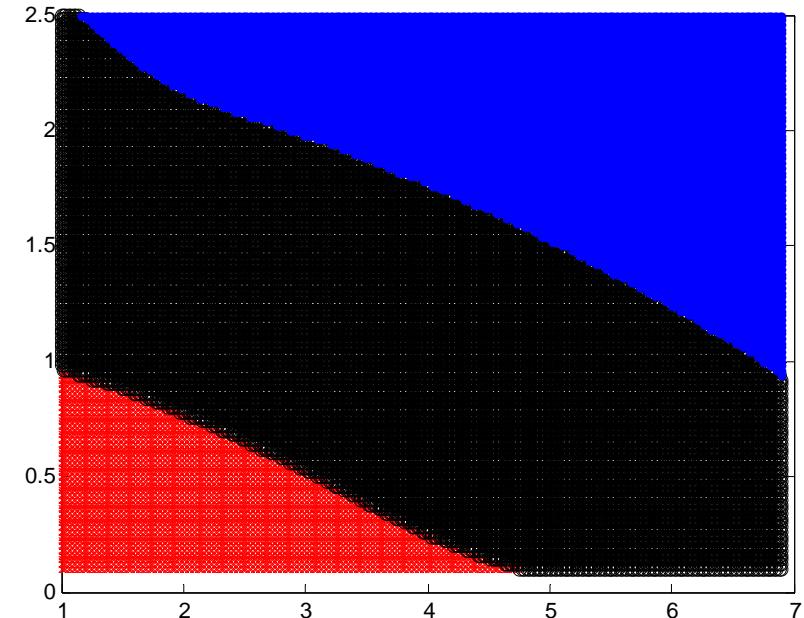
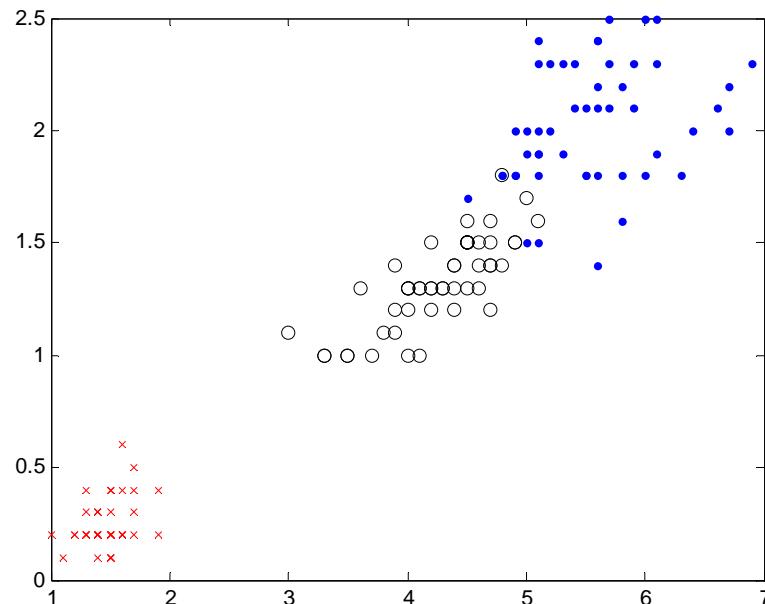


# SOBREENTRENAMIENTO

- Ejemplo: Clasificación de flores iris:
  - Conocida base de datos de clasificación
  - Contiene datos de 150 flores iris, cada una con 4 atributos, clasificables en 3 posibles clases: setosa, virginica y versicolor
    - Los patrones se pueden representar mediante los 2 atributos que contienen más información
    - Se entrena solamente con esos dos atributos

# SOBREENTRENAMIENTO

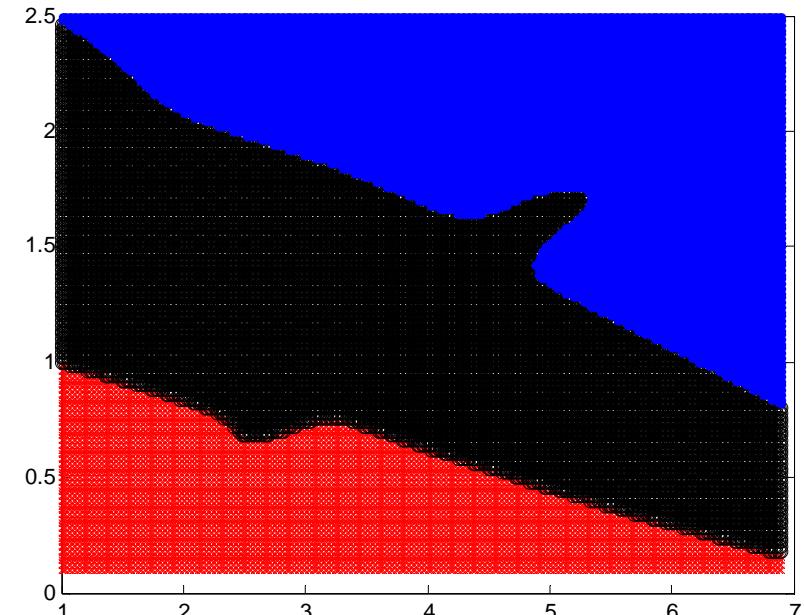
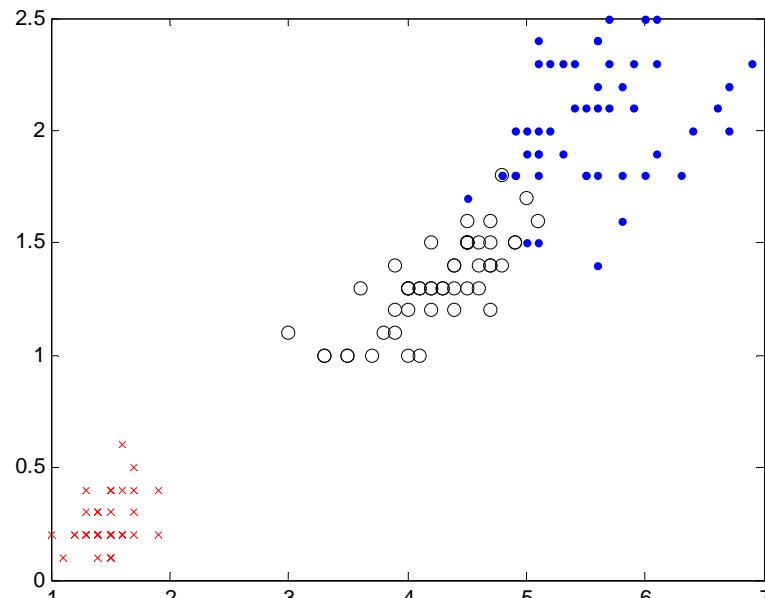
- Ejemplo: Clasificación de flores iris:
  - RNA no sobreentrenada:



97.3% de precisión:  
4 errores en la clasificación

# SOBREENTRENAMIENTO

- Ejemplo: Clasificación de flores iris:
  - RNA sobreentrenada:



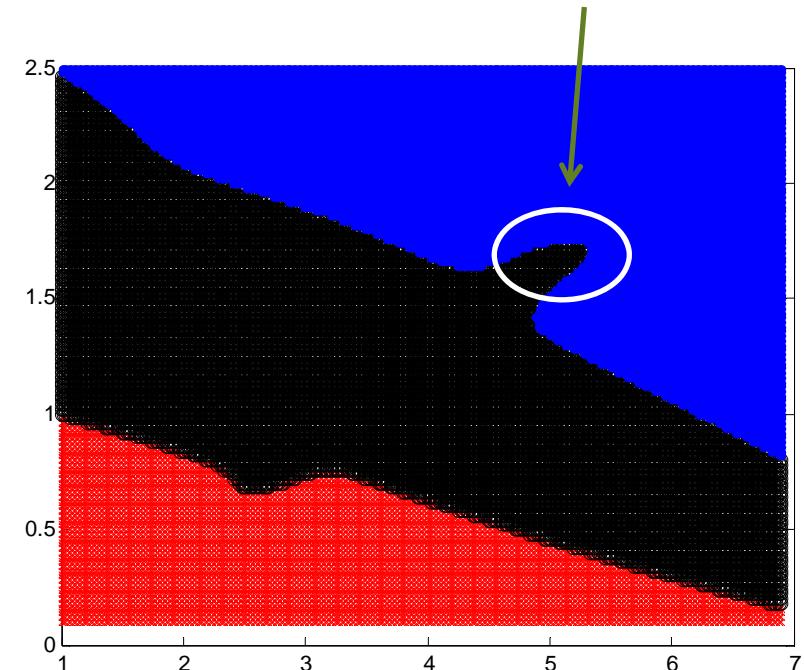
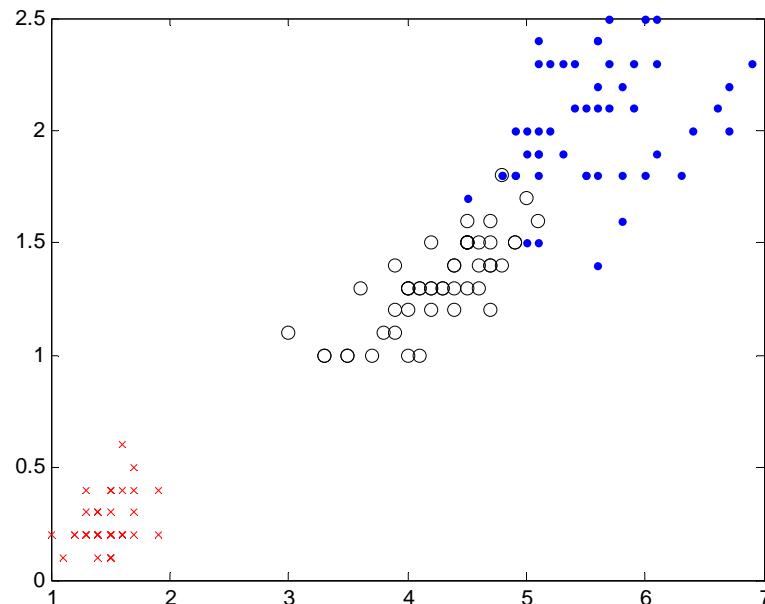
99.3% de precisión:  
1 error en la clasificación

# SOBREENTRENAMIENTO

- Ejemplo: Clasificación de flores iris:

- RNA sobreentrenada:

¿Qué se debería de devolver ante un nuevo patrón en esta zona?



99.3% de precisión:  
1 error en la clasificación



# APLICACIONES

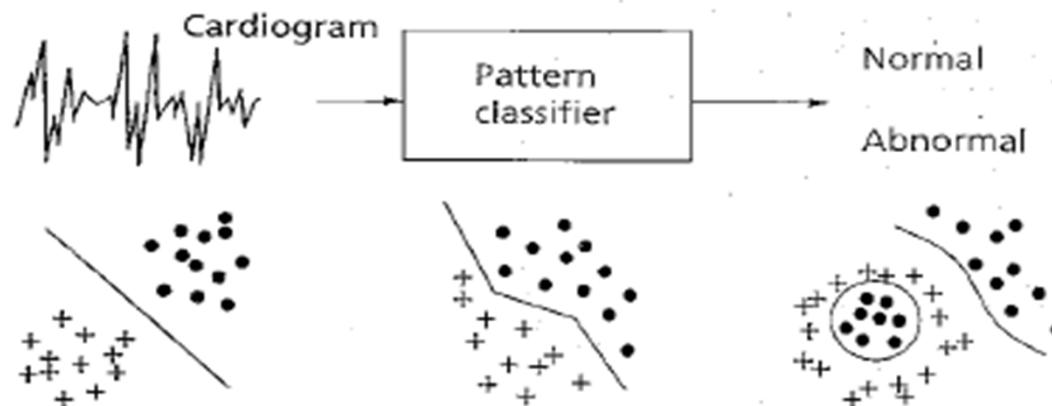
- Se usan para el reconocimiento de patrones
- Problemas donde más importante es el patrón que los datos exactos
- ¿Cuándo usar una RNA o aplicar programación convencional?
  - RNA:
    - No se sabe cómo resolver el problema
      - No se puede escribir un algoritmo que resuelva el problema
      - No se puede explicitar el conocimiento
    - El sistema hallado no requiere explicación
- Aplicaciones:
  - Clasificación
  - Predicción
  - *Clustering* (Agrupamiento)
  - Aproximación de curvas
  - Regresión

# APLICACIONES

- Aplicaciones:

- Clasificación

- Clasifica objetos en un número finito de clases, dadas sus propiedades (entradas)
    - Busca una función de mapeo que permita separar la clase 1 de la clase 2 y esta de la clase 3...
    - El número de clases es finito y conocido
    - Se conoce la pertenencia a la clase de cada patrón

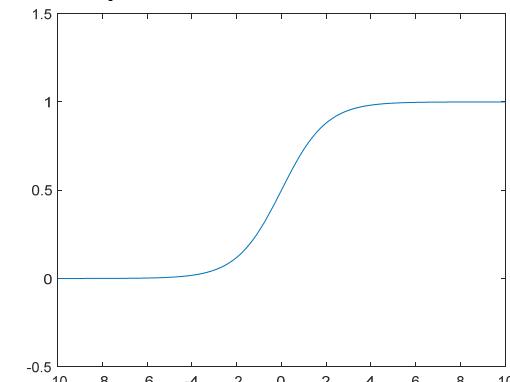


# APLICACIONES

- Aplicaciones:

- Clasificación

- Número de salidas de la RNA:
    - Si solo hay dos clases: 1 única neurona de salida
      - Ejemplo: A/B o A/-A
      - Salida deseada: 0/1 (negativo/positivo)
      - Función de transferencia habitual en la capa de salida: Logarítmica sigmoidal
        - Asigna un valor entre 0 y 1
        - Interpretación: certidumbre, seguridad o probabilidad de clasificar un caso como 1 (positivo)
      - Umbral
        - Generalmente en 0.5



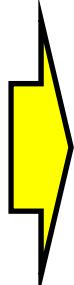
# APLICACIONES

- Aplicaciones:

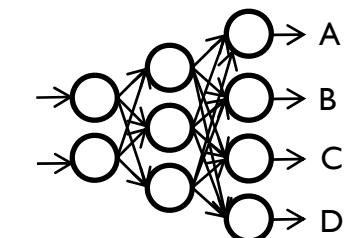
- Clasificación

- Número de salidas de la RNA:
    - Si hay más de dos clases: 1 neurona de salida por clase
      - Ejemplo: A/B/C/D
      - Para cada patrón, la salida deseada es 0 para cada neurona de salida, excepto para aquella que corresponda a esa clase

Entradas				Salida deseada
$x_1$	$x_2$	$\dots$	$x_n$	
				A
				C
				A
				D
...	...	...	...	...



Entradas				Salidas deseadas			
$x_1$	$x_2$	$\dots$	$x_n$	$y_1$	$y_2$	$y_3$	$y_4$
				1	0	0	0
				0	0	1	0
				1	0	0	0
				0	0	0	1
...	...	...	...	...	...	...	...



# APLICACIONES

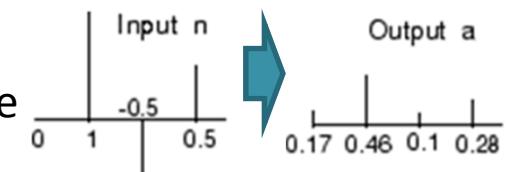
- Aplicaciones:

- Clasificación

- Número de salidas de la RNA:
    - Si hay más de dos clases: 1 neurona de salida por clase
      - Función de transferencia habitual en la capa de salida: Lineal
        - Devuelve un valor continuo que puede no estar acotado
        - Las salidas de cada neurona pueden ser positivas o negativas
      - Habitualmente a las salidas de las neuronas de salida se aplica una función *softmax*:

$$\hat{y}_i = \frac{e^{y_i}}{\sum_{j=1}^C e^{y_j}}$$

- Pasa valores reales a valores comprendidos entre 0 y 1
        - La suma de todos es igual a 1
        - Cada valor representa la certidumbre, seguridad o probabilidad de pertenencia a esa clase
        - Salida final: clase con mayor certidumbre





# APLICACIONES

- Aplicaciones:

- Clasificación

- Observación:
      - Si, además de las clases de salida, existe la posibilidad de no pertenencia a ninguna de esas clases, se añade una nueva clase para reflejar esta posibilidad
        - Ejemplo: A/B/C/"ninguna" -> A/B/C/D
        - Ejemplo: A/B/"ninguna" -> A/B/C
          - Pasaría del caso de 2 clases de salida a 3, y se necesitarían 3 neuronas de salida en lugar de una

# APLICACIONES

- Aplicaciones:

- Clasificación

- Otra manera de representar las salidas:
      - Una única neurona de salida
      - Se divide la salida de la neurona en tantos intervalos como clases existan
      - Ejemplo: clases A/B/C
      - Si la neurona de salida emite salidas en [0,1], las salidas deseadas serán 0, 0.5, 1

Entradas				Salida deseada
$x_1$	$x_2$	...	$x_n$	
				A
				C
				B
...	...	...	...	...

→

Entradas				Salida deseada
$x_1$	$x_2$	...	$x_n$	
				0
				1
				0.5
...	...	...	...	...

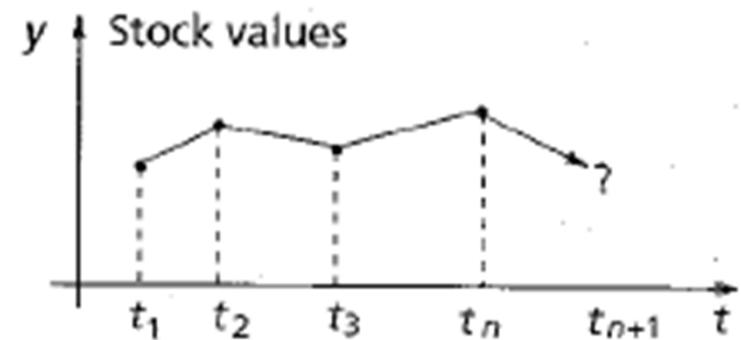
- La salida que emite la red se clasifica según los intervalos:
      - 0-0.33 → A
      - 0.33-0.66 → B
      - 0.66-1 → C
    - Clases A/B/C/D:
      - Salidas deseadas: 0, 0.33, 0.66, 1
      - Intervalos: 0-0.25 0.25-0.5 0.5-0.75 0.75-1
    - Método indicado cuando las clases de salida tienen una graduación determinada
      - Por ejemplo: Clasificación de señales de bolsa
        - Clases de salida: Fuerte venta / venta / mantener / compra / fuerte compra
    - En general, se aplica el método anterior

# APLICACIONES

- Aplicaciones:

- Predicción

- Intenta determinar la función que mapea un conjunto de variables de entrada en una (o más) variables de salida.
    - Es básicamente numérica
    - Está basada en supuestos estadísticos
    - Ejemplos:
      - Monitoreo la reserva de plazas en empresas de aviación.
      - Predicciones financieras a corto plazo

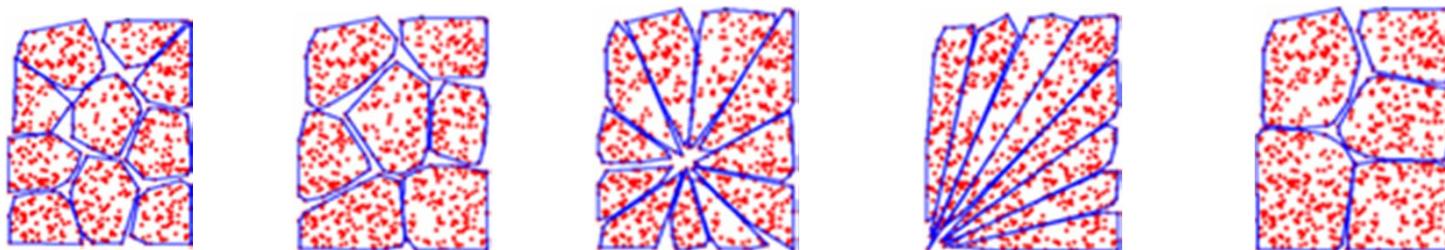


# APLICACIONES

- Aplicaciones:

- *Clustering* (Clasificación no supervisada):

- Intenta agrupar una serie de objetos en grupos
    - Cada objeto es representado por un vector de atributos n-dimensional
    - Los objetos que forman cada grupo deben ser similares
    - La similaridad es medida del grado de proximidad
    - Luego cada grupo es etiquetado
    - No se conoce a priori el número de clases o *clusters*
    - No se conoce a qué *cluster* pertenece cada patrón
      - Aprendizaje no supervisado
      - **No para perceptrones multicapa**

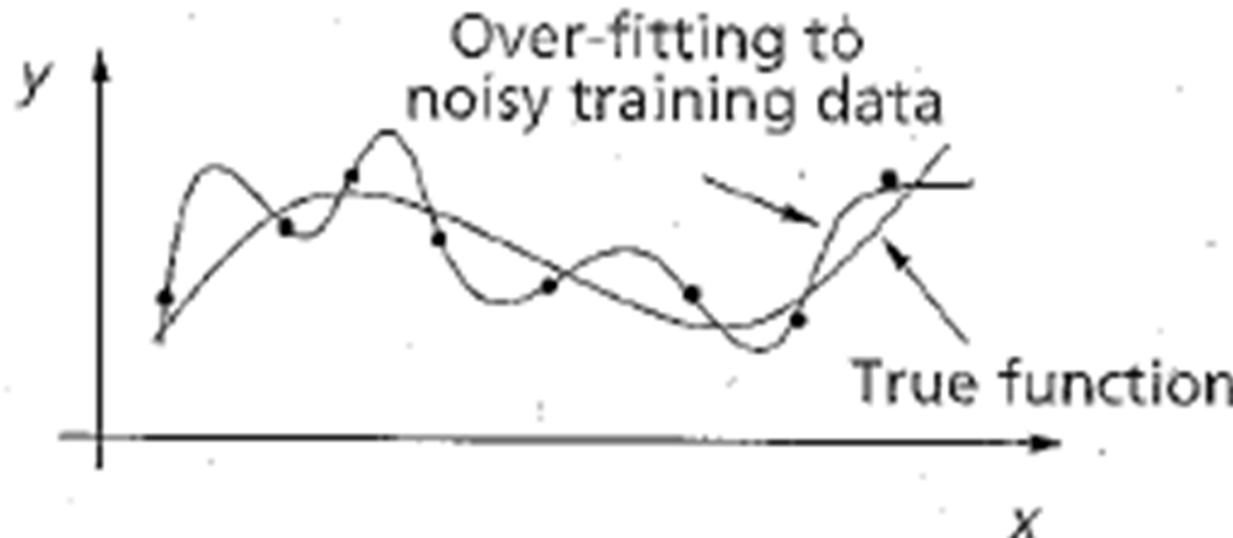


# APLICACIONES

- Aplicaciones:

- Aproximación de curvas (*fitting*):

- Reconocer la curva que subyace tras un conjunto de patrones  $(x_i, y)$





# APLICACIONES

- Aplicaciones:

- Regresión:

- Genérico
    - A partir de una BD o conjunto de patrones, intentar encontrar una RNA que modelice la relación entre dos conjuntos de ellos
      - Patrones: (entradas, salidas deseadas)
      - Relación desconocida
      - Modelo debe de ser generalizable, es decir, que ante nuevos patrones desconocidos devuelva unas salidas coherentes
    - Función de transferencia habitual en la capa de salida: Lineal



# APLICACIONES

- Aplicaciones:

- Regresión:

- Ejemplo: Base de datos de precios de casas en Boston:
    - Base de datos con 506 patrones
    - 13 entradas:
      - 1. per capita crime rate by town
      - 2. proportion of residential land zoned for lots over 25,000 sq.ft.
      - 3. proportion of non-retail business acres per town
      - 4. 1 if tract bounds Charles river, 0 otherwise
      - 5. nitric oxides concentration (parts per 10 million)
      - 6. average number of rooms per dwelling
      - 7. proportion of owner-occupied units built prior to 1940
      - 8. weighted distances to five Boston employment centres
      - 9. index of accessibility to radial highways
      - 10. full-value property-tax rate per \$10,000
      - 11. pupil-teacher ratio by town
      - 12.  $1000(Bk - 0.63)^2$ , where Bk is the proportion of blacks by town
      - 13. % lower status of the population
    - Salida deseada:
      - Median values of owner-occupied homes in \$1000's.
    - This data is available from the UCI Machine Learning Repository:  
<http://mlearn.ics.uci.edu/MLRepository.html>
    - Murphy, P.M., Aha, D.W. (1994). UCI Repository of machine learning databases [<http://www.ics.uci.edu/~mlearn/MLRepository.html>]. Irvine, CA: University of California, Department of Information and Computer Science.



# APLICACIONES

- Aplicaciones:

- Regresión:

- Ejemplo: Uso de energía en edificios
      - Predecir el uso de energía de una casa, basado en la hora del día y varios factores atmosféricos
      - Base de datos con 4209 patrones
      - 14 entradas:
        - 1-10: Coded day of the week, time of day
        - 11: Temperature
        - 12: Humidity
        - 13: Solar strength
        - 14: Wind
      - Salidas deseadas: uso de energía



# APLICACIONES

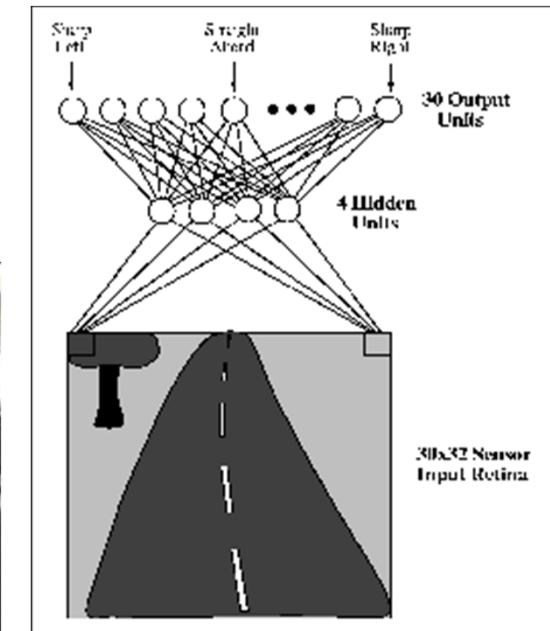
- Aplicaciones:

- Regresión:

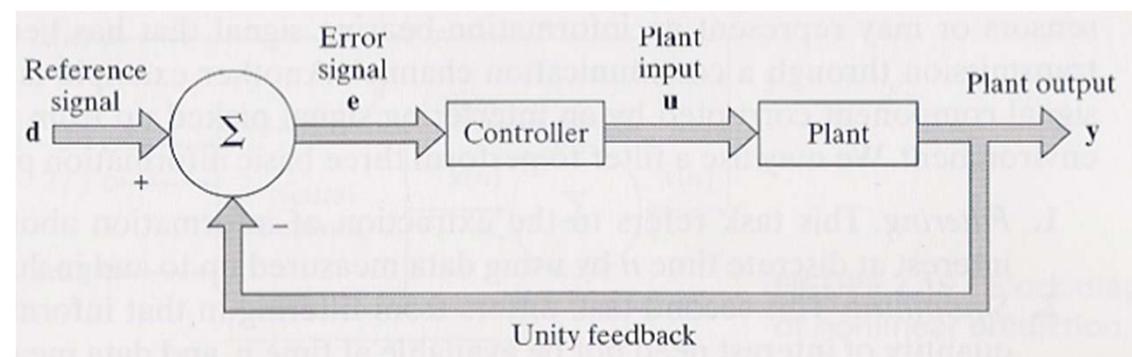
- Ejemplo: Estimar dos aspectos de un motor a partir de dos mediciones
    - La base de datos contiene 1199 patrones
    - Entradas:
      - 1: Fuel rate
      - 2: Speed
    - Salidas deseadas:
      - 1: Torque
      - 2: Nitrous oxide emissions
    - Prof. Martin T. Hagan, Colorado State University
    - <http://hagan.ecen.ceat.okstate.edu/>

# APLICACIONES

- Aplicaciones:
  - Control: Ejemplos:
    - Conducción de un vehículo



- Planta industrial





# APLICACIONES

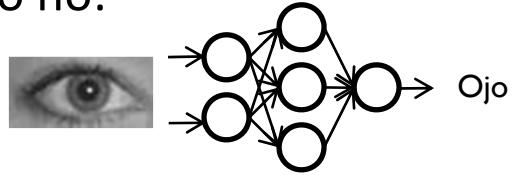
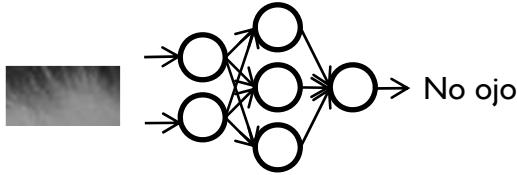
- Algunas posibles aplicaciones:
  - Espacio aéreo
    - Pilotos automáticos de alto desempeño, simulaciones y predicciones de trayectoria de vuelo, sistemas de control de vuelo, detección de fallas en componentes de la nave.
  - Automotriz
    - Sistemas automáticos de navegación, comando por voz
  - Bancos
    - Lectores de documentos, evaluadores de asignación de crédito, identificador de firmas.
  - Electrónica
    - Predicción de secuencias de códigos, control de procesos, análisis de fallas de circuitos, visión de máquina, síntesis de voz, modelado no lineal.
  - Robótica
    - Control de trayectorias, control de manipuladores, sistemas de visión.
  - Voz
    - Reconocimiento de voz, compresión de voz, sintetizadores de texto a voz.



# APLICACIONES

- Algunas posibles aplicaciones:
  - **Telecomunicaciones**
    - Compresión de datos e imágenes, servicios automáticos de información, traducción de lenguaje hablado en tiempo real.
  - **Transporte**
    - Sistemas enrutadores, diagnóstico de motores, tiempos y movimientos.
  - **Seguridad**
    - Reconocimiento de rostros, identificación y acceso de personas
  - **Financieros**
    - Evaluación de bienes raíces, consultor de préstamos, valuación de bonos corporativos, análisis del uso de la línea de crédito, predicción de tipo de cambio.
  - **Manufactura**
    - Control de procesos de manufactura, análisis y diseño de productos, diagnóstico de máquinas y procesos, identificación de partes en tiempo real, sistemas de inspección de calidad, predicción de fin de proceso, análisis de mantenimiento de máquinas, modelado de sistemas dinámicos.
  - **Medicina**
    - Detección de cáncer mamario o en la piel, análisis de EEG y ECG, diseño de prótesis, optimización de tiempos de trasplante, reducción de gastos en hospitales.
  - **Oficinas postales, Verificación remota, etc.**
  - **etc.**

# APLICACIONES

- Ejemplo de aplicación: RNA 2008/2009:
  - Detección de ojos en imágenes de caras:
    - Problema de clasificación:
      - A partir de una imagen pequeña que puede contener totalmente un ojo, una RNA es capaz de distinguir si tiene un ojo o no:
    - 
    - 
  - De esa imagen (de dimensiones variables):
    - Se toman ciertas características numéricas
    - Las características se usan como entrada a una RNA
    - La RNA emite como salida una clase

# APLICACIONES

- Ejemplo de aplicación: RNA 2008/2009:
  - Detección de ojos en imágenes de caras:
    - Cómo crear esa RNA:
      1. A partir de una BD de caras de personas, se toman distintas porciones:
        - Distintos tamaños





# APLICACIONES

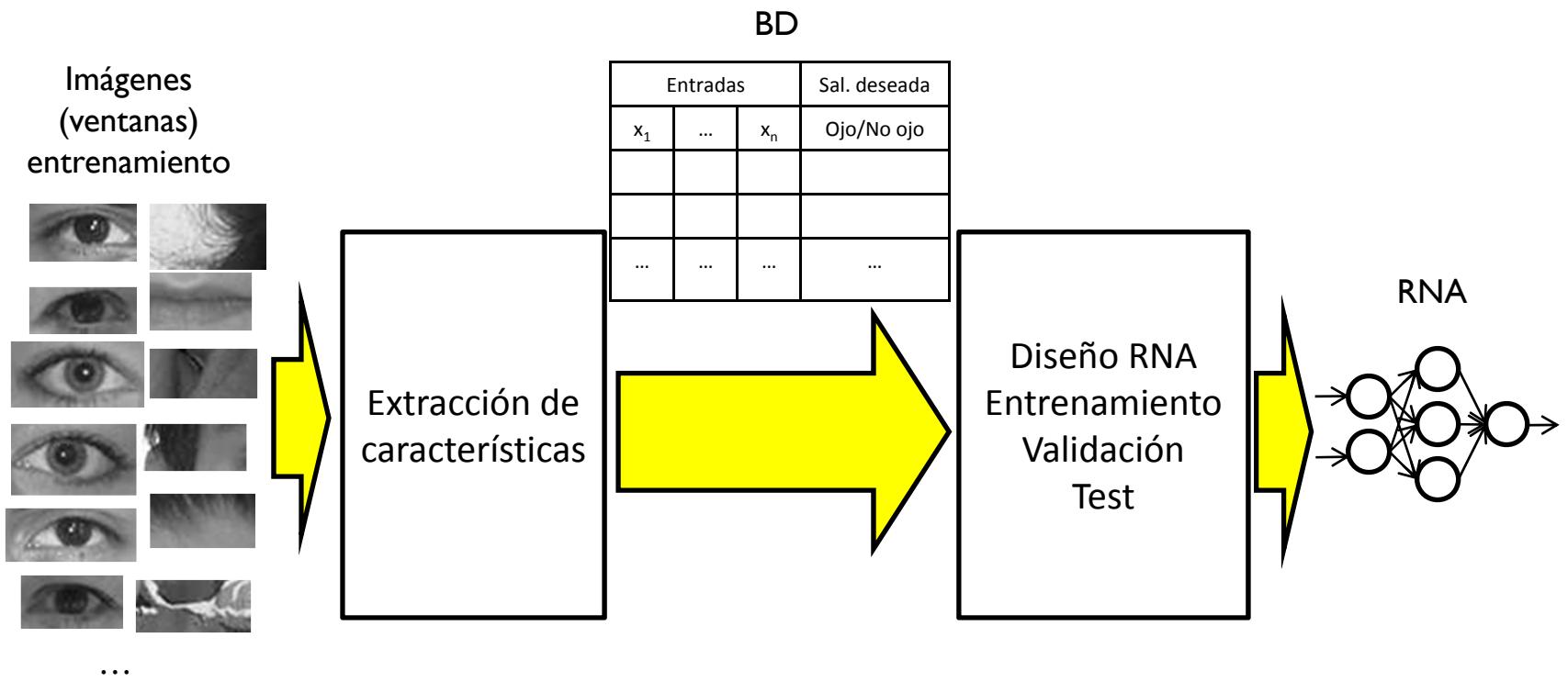
- Ejemplo de aplicación: RNA 2008/2009:
  - Detección de ojos en imágenes de caras:
    - Cómo crear esa RNA:
      2. De esas porciones, se extraen las características y se crea la BD:

Entradas				Salida deseada
$x_1$	$x_2$	...	$x_n$	Ojo/No ojo
				0
				0
				0
				1
				1
...	...	...	...	...

3. Se diseña (topología) la RNA y se entrena y valida con esa BD

# APLICACIONES

- Ejemplo de aplicación: RNA 2008/2009:
  - Detección de ojos en imágenes de caras:



# APLICACIONES

- Ejemplo de aplicación: RNA 2008/2009:
  - Detección de ojos en imágenes de caras:
    - Una vez se tiene esa RNA, para detectar ojos:



# APLICACIONES

- Ejemplo de aplicación: RNA 2008/2009:
  - Detección de ojos en imágenes de caras:
    - Una vez se tiene esa RNA, para detectar ojos:



# APLICACIONES

- Ejemplo de aplicación: RNA 2008/2009:
  - Detección de ojos en imágenes de caras:
    - Una vez se tiene esa RNA, para detectar ojos:



# APLICACIONES

- Ejemplo de aplicación: RNA 2008/2009:
  - Detección de ojos en imágenes de caras:
    - Una vez se tiene esa RNA, para detectar ojos:



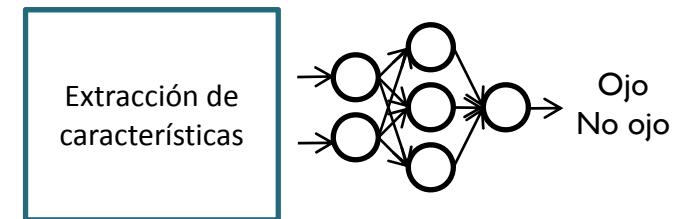
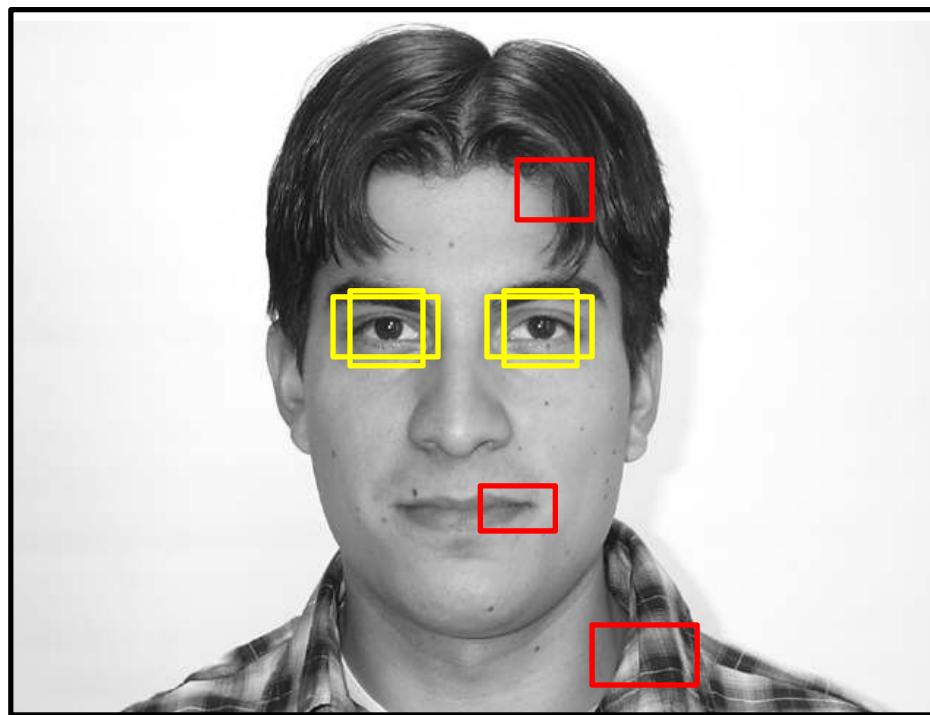
# APLICACIONES

- Ejemplo de aplicación: RNA 2008/2009:
  - Detección de ojos en imágenes de caras:
    - Una vez se tiene esa RNA, para detectar ojos:



# APLICACIONES

- Ejemplo de aplicación: RNA 2008/2009:
  - Detección de ojos en imágenes de caras:
    - Una vez se tiene esa RNA, para detectar ojos:



Verdadero positivo

Falso positivo

# APLICACIONES

- Ejemplo de aplicación:
  - Detección de ojos en imágenes de caras:
    - Problemas similares:
      - Detección y clasificación de señales de tráfico en fotografías (2008/2009)
      - Detección de matrículas en fotografías (2008/2009)
      - Detección de semáforos en fotografías (2008/2009, 2009/2010)



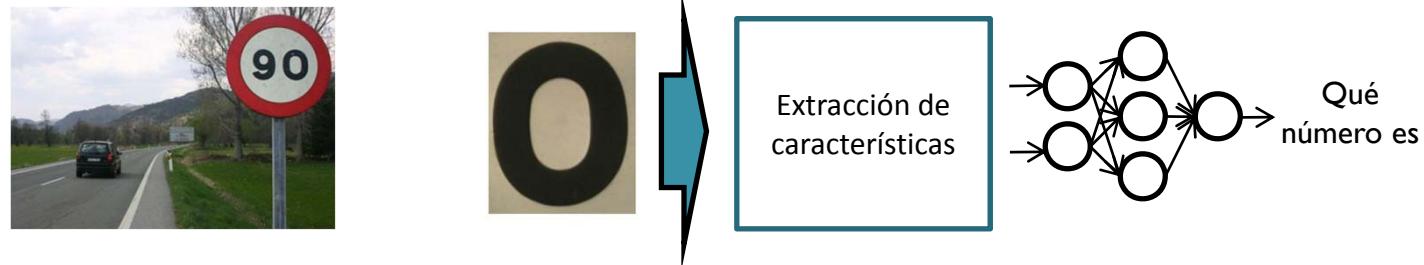
# APLICACIONES

- Ejemplo de aplicación:
  - Detección de ojos en imágenes de caras:
  - Problemas similares:
    - ¿Dónde está Wally?



# APLICACIONES

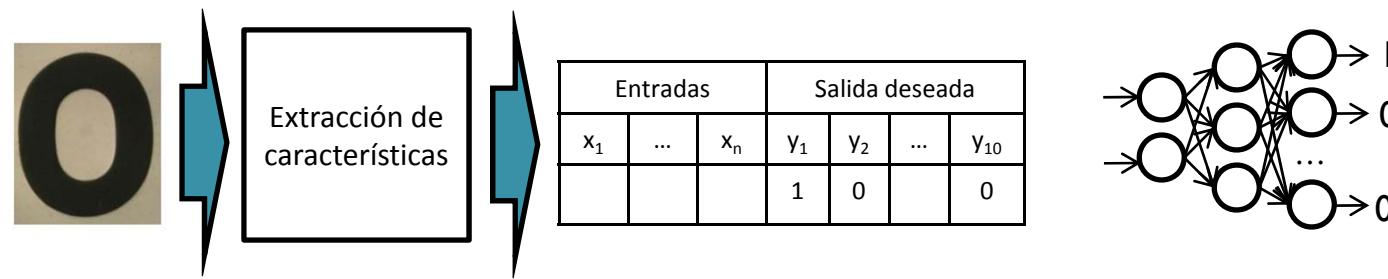
- Ejemplo de aplicación:
  - Reconocimiento de números
    - Señales de tráfico a partir de fotografías (RNA 2009/2010, 2010/2011)



- Números impresos en envíos postales (reconocimiento del código postal) (RNA 2011/2012)

# APLICACIONES

- Ejemplo de aplicación:
  - Reconocimiento de números
    - Salida RNA:
    - Indica qué número representa la imagen de entrada, caracterizada por una serie de características
    - Problema de **clasificación**, no de regresión
    - Se necesitan 10 neuronas de salida, una por cada dígito
      - La salida deseada será 1 para la neurona de salida que corresponda con ese número



# APLICACIONES

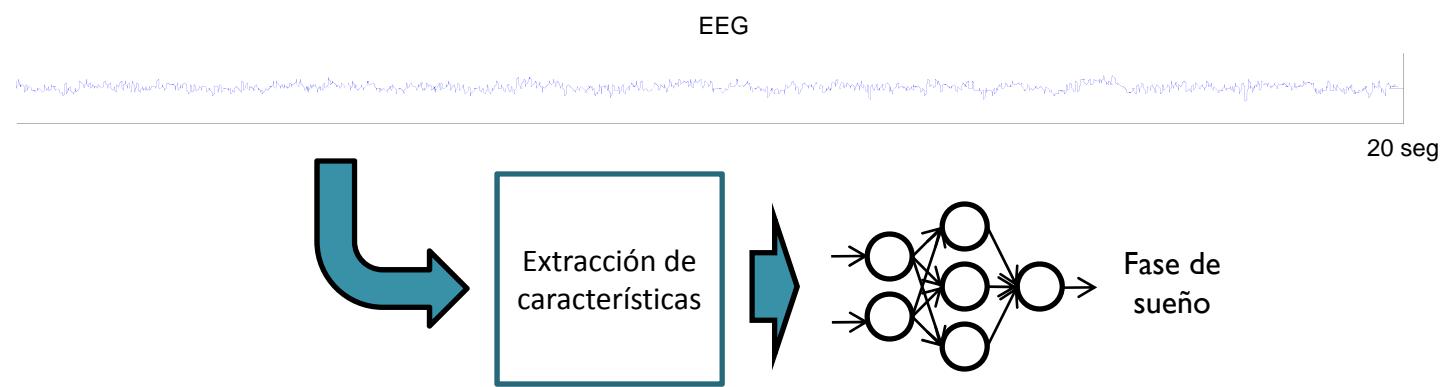
- Ejemplo de aplicación:
  - Identificación de cartas de la baraja a partir de imágenes (RNA 2009/2010):
    - Problema de clasificación



- Varias RR.NN.AA.:
  - Clasificación números (aplicación anterior)
  - Clasificación palos

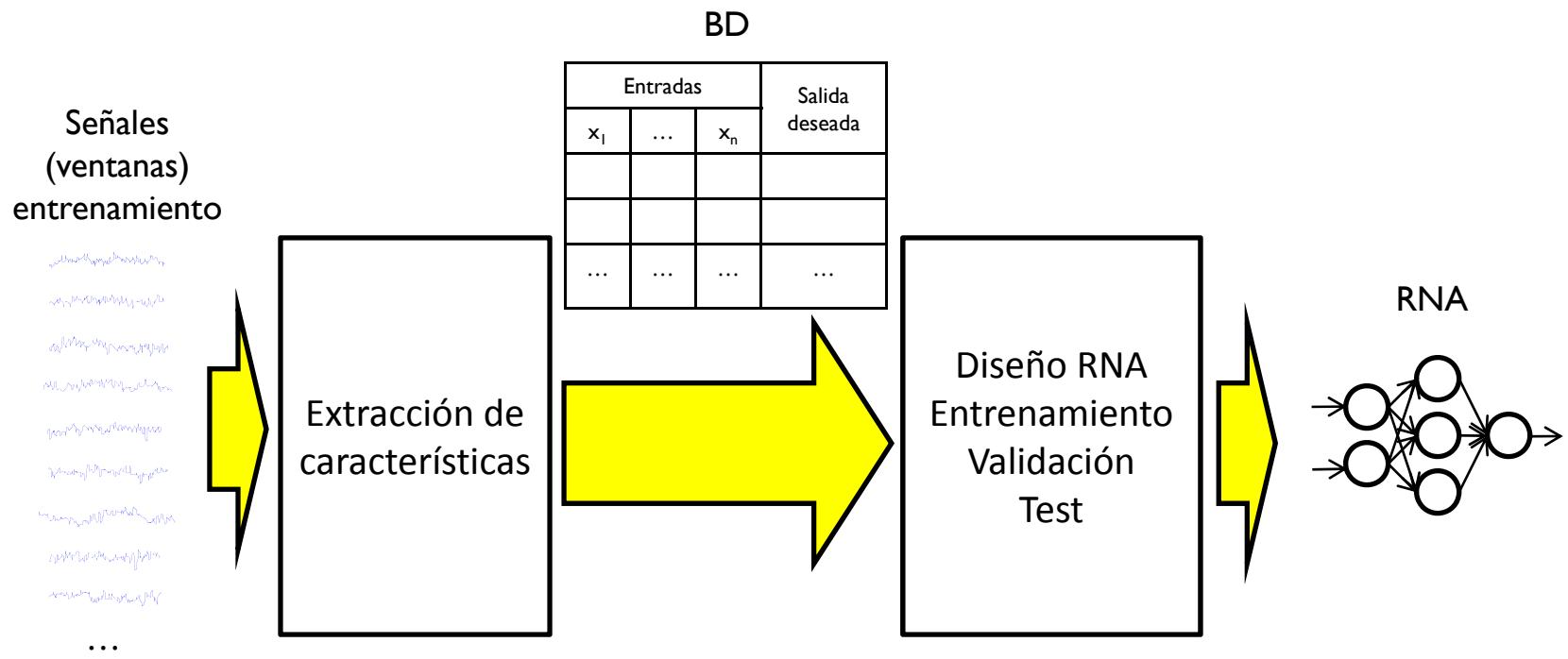
# APLICACIONES

- Ejemplo de aplicación:
  - Clasificación de señales:
    - Clasificación de estados de sueño a partir de señales de EEG (RNA 2011/2012)
    - Fases de sueño:
      - Etapa 1 (adormecimiento)
      - Etapa 2 (sueño ligero)
      - Etapa 3 (transición hacia el sueño profundo)
      - Etapa 4 (sueño delta)
      - Fase REM



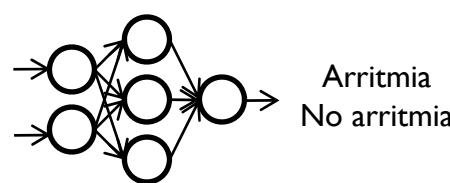
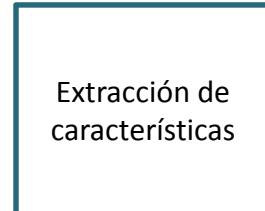
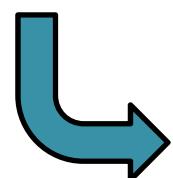
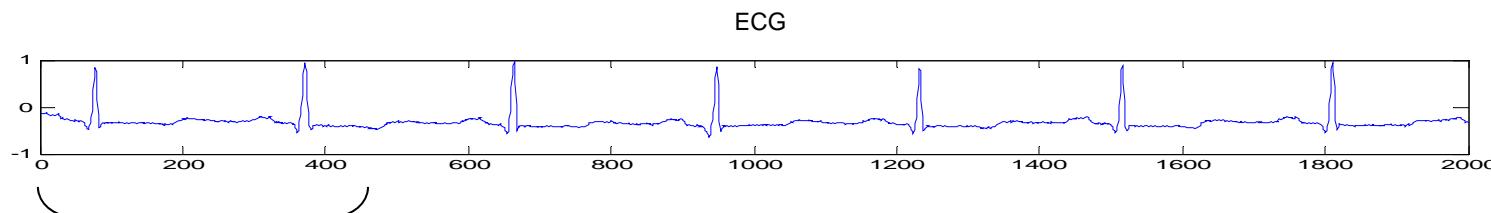
# APLICACIONES

- Ejemplo de aplicación:
  - Clasificación de señales:
    - Clasificación de estados de sueño a partir de señales de EEG (RNA 2011/2012)
    - Construcción de la RNA: similar al caso de clasificación de imágenes



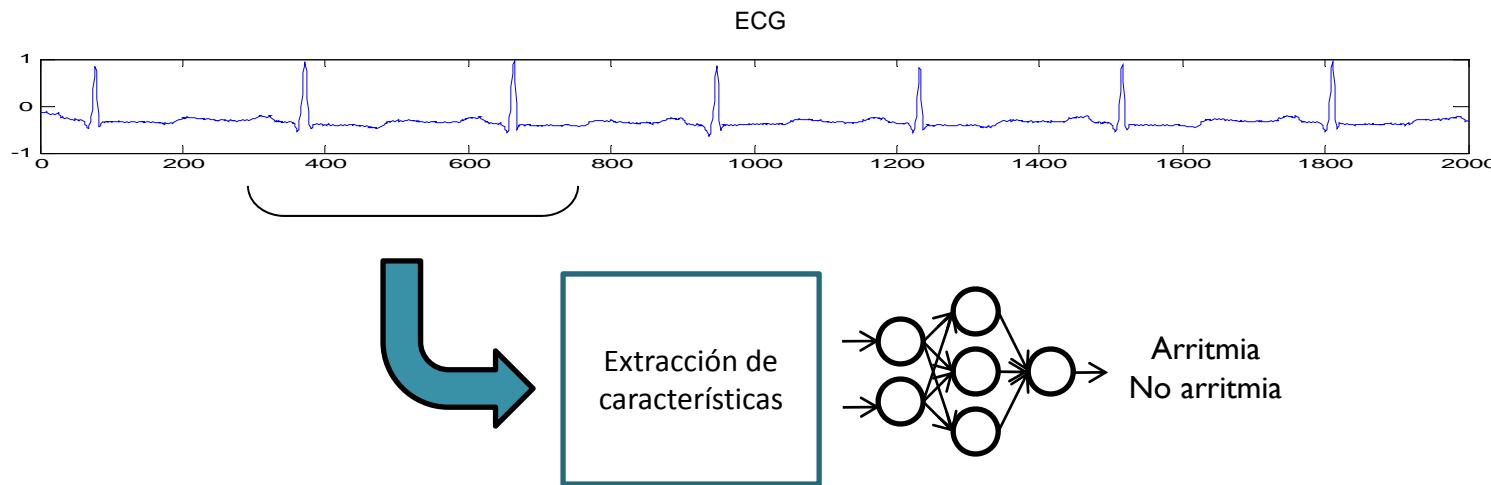
# APLICACIONES

- Ejemplo de aplicación:
  - Detección de arritmias en señales de ECG:
    - Problema de clasificación
    - Similar a la detección de objetos en imágenes
      - Usar una ventana con solapamiento, analizar las distintas ventanas



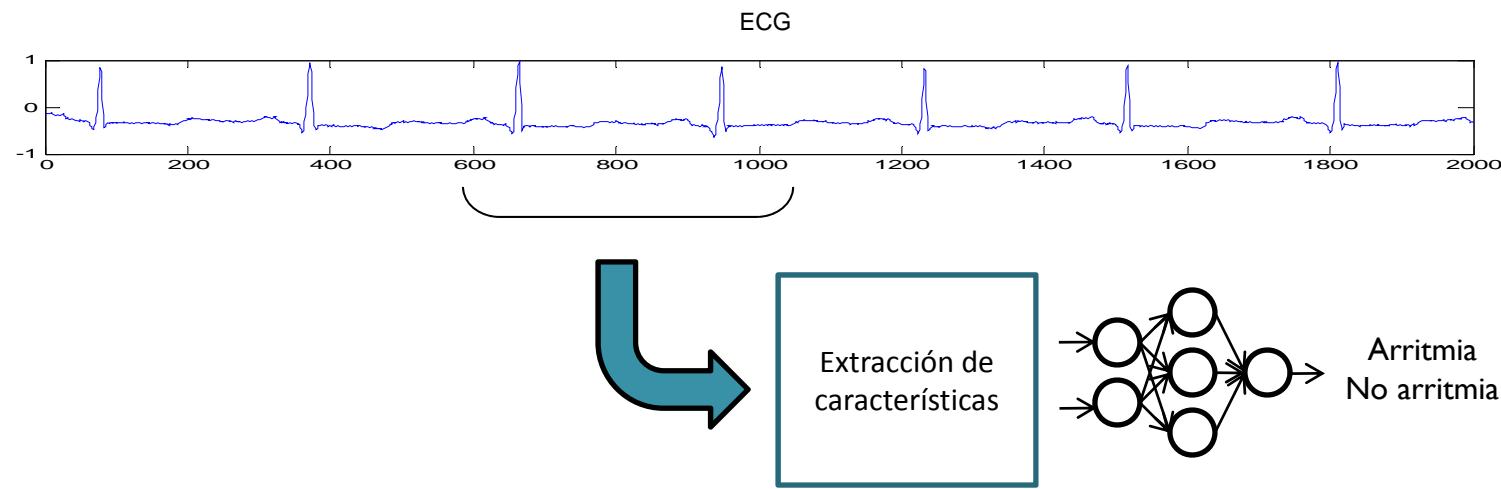
# APLICACIONES

- Ejemplo de aplicación:
  - Detección de arritmias en señales de ECG:
    - Problema de clasificación
    - Similar a la detección de objetos en imágenes
      - Usar una ventana con solapamiento, analizar las distintas ventanas



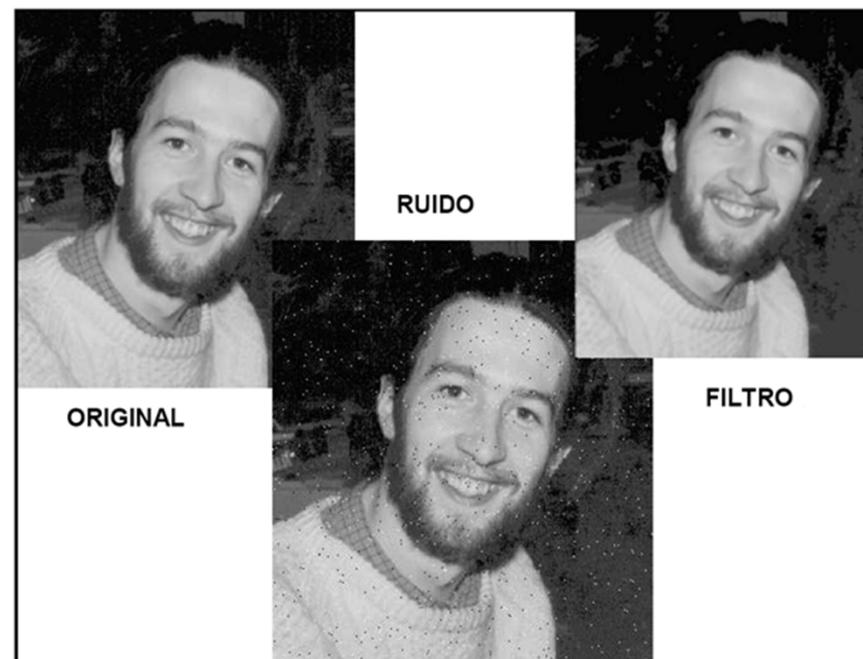
# APLICACIONES

- Ejemplo de aplicación:
  - Detección de arritmias en señales de ECG:
    - Problema de clasificación
    - Similar a la detección de objetos en imágenes
      - Usar una ventana con solapamiento, analizar las distintas ventanas



# APLICACIONES

- Ejemplo de aplicación:
  - Eliminación de ruido:
    - Es usada para filtrar el ruido de patrones contaminados, por ejemplo, limpieza de señales en líneas telefónicas y limpieza de imágenes.





# CONCLUSIONES

- Ventajas del uso de RR.NN.AA.:
  - Debido a su constitución y a sus fundamentos, las RR.NN.AA. presentan un gran número de características semejantes a las del cerebro, por ejemplo:
    - aprender de la experiencia
    - generalizar de casos anteriores a nuevos casos
    - abstraer características esenciales a partir de entradas con información irrelevante
  - Esto hace que se apliquen en múltiples áreas, por las ventajas que ofrecen:
    1. Aprendizaje Adaptativo
      - Capacidad de aprender a realizar tareas basadas en un entrenamiento o una experiencia inicial.
    2. Autoorganización
      - Las redes neuronales usan su capacidad de aprendizaje adaptativo para organizar la información que reciben durante el aprendizaje y/o la operación.
      - Una RNA puede crear su propia organización o representación de la información que recibe mediante una etapa de aprendizaje.
      - Esta autoorganización permite a las redes neuronales de responder apropiadamente cuando se les presentan datos o situaciones a los que no habían sido expuestas anteriormente.



# CONCLUSIONES

- Ventajas del uso de RR.NN.AA.:
  - 3. Tolerancia a fallos: Dos aspectos distintos:
    - Pueden aprender a reconocer patrones con ruido, distorsionados, o incompleta.
    - Pueden seguir realizando su función (con cierta degradación) aunque se destruya parte de la red.
    - Comparados con los sistemas computacionales tradicionales, los cuales pierden su funcionalidad en cuanto sufren un pequeño error de memoria, en las redes neuronales, si se produce un fallo en un pequeño número de neuronas, aunque el comportamiento del sistema se ve influenciado, sin embargo no sufre una caída repentina.
      - La razón por la que las redes neuronales son tolerantes a fallos es que tienen su información distribuida en las conexiones entre neuronas, con cierto grado de redundancia.
      - En cambio, en la mayoría de los ordenadores algorítmicos y sistemas de recuperación de datos se almacena cada pieza de información en un estado único, localizado y direccionable.



# CONCLUSIONES

- Ventajas del uso de RR.NN.AA.:
  - 4. Procesamiento en paralelo
  - 5. Operación en tiempo real
    - Los computadores neuronales pueden ser realizados en paralelo, y se diseñan y fabrican máquinas con hardware especial para obtener esta capacidad.
  - 6. Fácil inserción dentro de la tecnología existente.
    - Debido a que una red puede ser rápidamente entrenada, comprobada, verificada y trasladada a una implementación hardware de bajo costo, es fácil insertar RR.NN.AA. para aplicaciones específicas dentro de sistemas existentes (chips, por ejemplo).
    - De esta manera, las redes neuronales se pueden utilizar para mejorar sistemas de forma incremental, y cada paso puede ser evaluado antes de acometer un desarrollo más amplio.



# CONCLUSIONES

- Desventajas del uso de RR.NN.AA.:
  1. Implementación deficiente como sistemas paralelos
    - Si se implementan en máquinas en serie, solo se ejecuta una instrucción a la vez.
    - Por ello, modelar procesos paralelos en máquinas serie puede ser un proceso que consuma mucho tiempo.
  2. No existencia de una regla para construir una RNA para un problema dado
    - Hay muchos factores a tener en cuenta: el algoritmo de aprendizaje, la arquitectura, el número de neuronas por capa, el número de capas, la representación de los datos, etc.
  3. Imposibilidad de explicar el funcionamiento de una RNA
    - Una RNA funciona como una “caja negra”: ante unas entradas, genera unas salidas, pero no es posible comprender la relación entre ellas.
    - Cuando se modela estadísticamente se puede ver qué variables forman parte del modelo o cuáles de las que finalmente se utilizaron para modelar fueron seleccionadas y las relaciones entre las mismas.
    - Esto provoca que en algunos entornos en los que es necesario explicar el funcionamiento, no se puedan utilizar, por ejemplo, en medicina.
  4. Cuanto más flexible se requiera que sea una RNA, más información habrá que enseñarle
    - Más patrones de entrenamiento.