



**DECSAI**

**Departamento de Ciencias de la Computación e I.A.**

Universidad de Granada



# Backpropagation

Fernando Berzal, [berzal@acm.org](mailto:berzal@acm.org)

# Backpropagation



- Introducción
- Modelo de neurona artificial
- Perceptrones
- Backpropagation
- En la práctica...
  - Parámetros
  - Optimización
  - Generalización
  - Invarianza
- Apéndice: Softmax



# Introducción



## **Redes Neuronales Artificiales [RNA]** **Artificial Neural Networks [ANN]**

Red de elementos o unidades de procesamiento simples (EP/UP en español o PE/PU en inglés), interconectados entre sí mediante conexiones sinápticas, en las que cada conexión tiene un peso (fuerza) que se ajusta a partir de la experiencia (datos).



# Introducción



## **Redes Neuronales Artificiales [RNA]** **Artificial Neural Networks [ANN]**

- **Modelo de Neurona Artificial:**  
Modelo de cómputo simple. Cada neurona recibe estímulos de otras neuronas, los agrega y transmite una respuesta de acuerdo a su función de activación.
  
- **Modelo de Red Neuronal [Topología]:**  
Estructura de la red neuronal.  
Organización y número de neuronas y conexiones.



# Introducción



## **Redes Neuronales Artificiales [RNA]** **Artificial Neural Networks [ANN]**

Las redes neuronales artificiales proporcionan un modelo de cómputo paralelo y distribuido capaz de aprender a partir de ejemplos (datos)

Los **algoritmos de aprendizaje** (asociados a modelos concretos de redes) permiten ir modificando los pesos de las conexiones sinápticas de forma que la red aprenda a partir de los ejemplos que se le presentan.



# Introducción



## **Redes Neuronales Artificiales [RNA]** **Artificial Neural Networks [ANN]**

- No se programan, se entrenan.
- Necesitan disponer de ejemplos, en un número suficiente y una distribución representativa para ser capaces de generalizar correctamente.
- Requieren un proceso de validación para evaluar la “calidad” del aprendizaje conseguido.



# Introducción



## Redes Neuronales Artificiales [RNA] Artificial Neural Networks [ANN]

Veremos cómo...

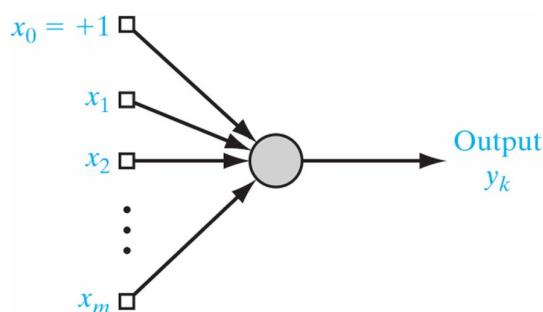
- Entrenar redes neuronales artificiales (para distintos modelos de red).
- Preparar (preprocesar) los ejemplos necesarios para su entrenamiento.
- Evaluar la calidad del proceso de aprendizaje.



## Introducción: Modelos de redes

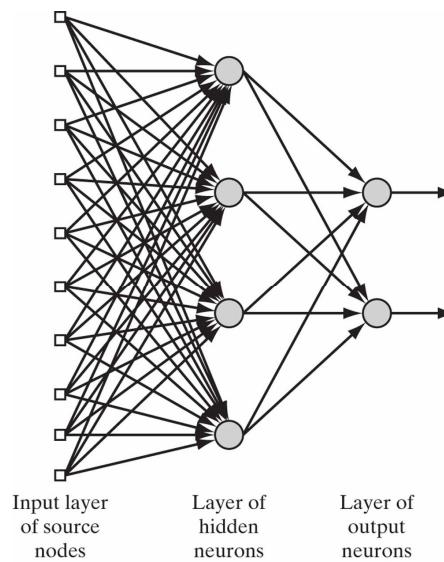


### Perceptrón



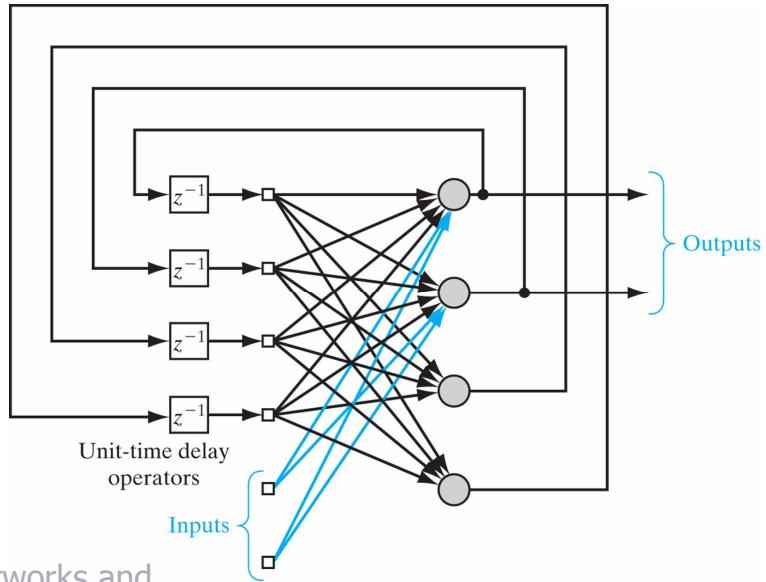
[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]

### Redes feed-forward (topología por capas)



# Introducción: Modelos de redes

## Redes recurrentes

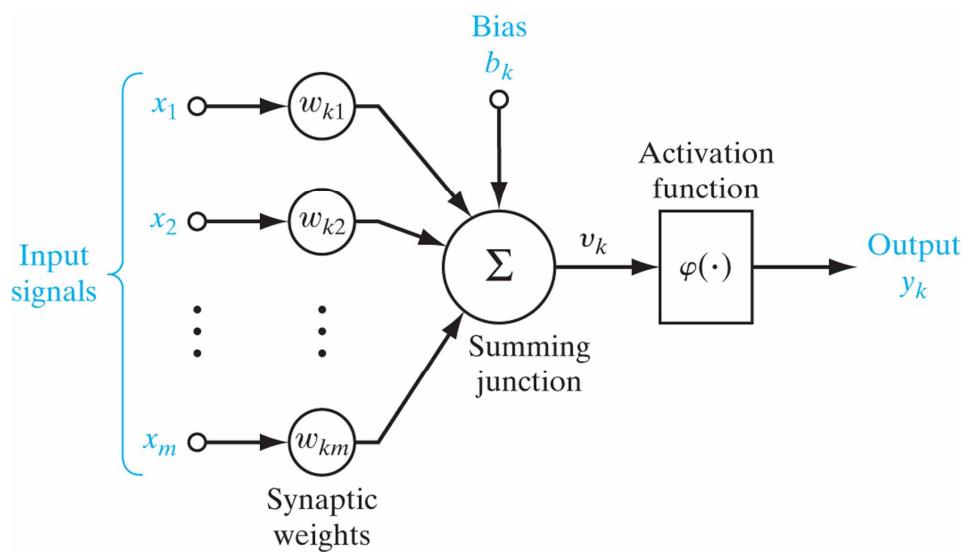


[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# Modelo de neuronal artificial

## Modelo no lineal de una neurona artificial



[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]

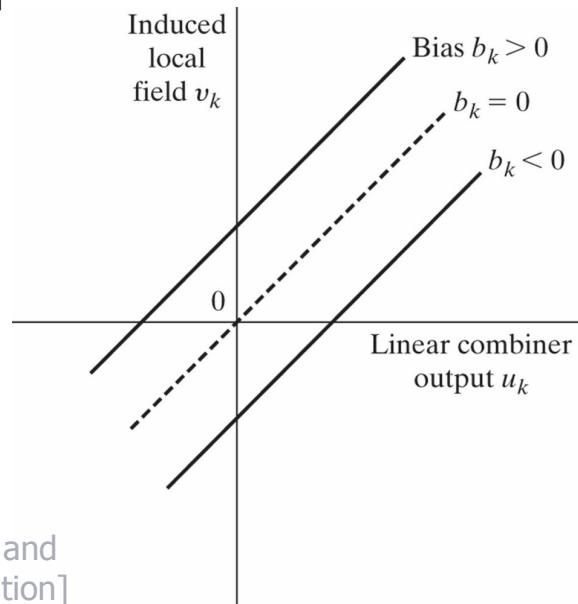


# Modelo de neuronal artificial



Transformación afín producida por la presencia del **sesgo  $b_k$  [bias]**

$$v_k = b_k \text{ cuando } u_k = 0$$



[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]

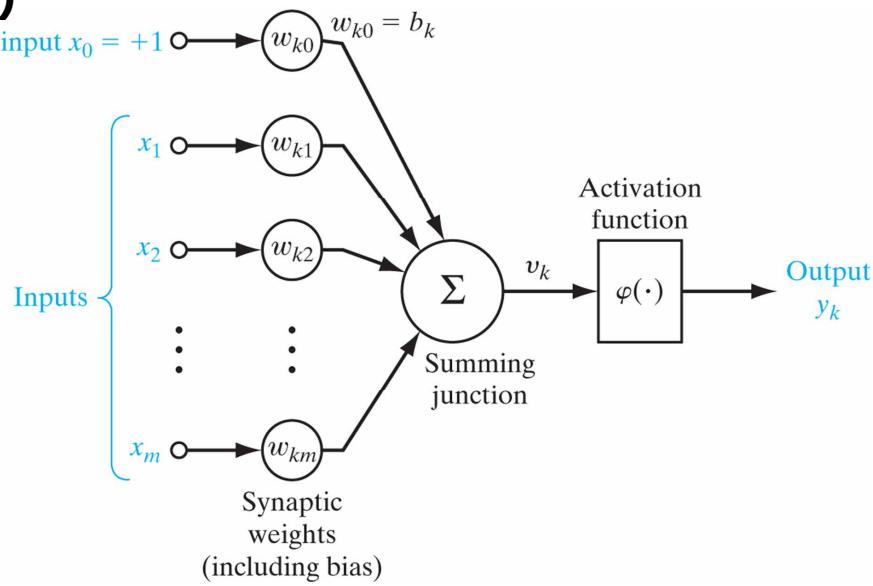


# Modelo de neuronal artificial



**Modelo no lineal de una neurona artificial  
( $w_{k0} = b_k$ )**

Fixed input  $x_0 = +1$



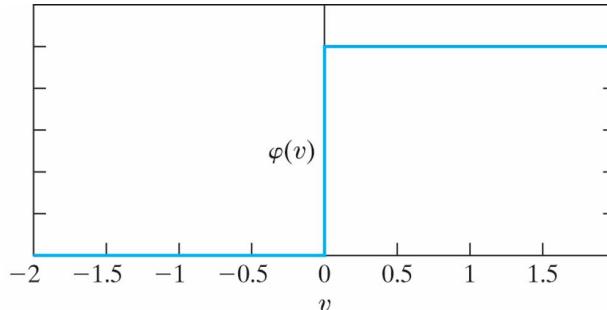
[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



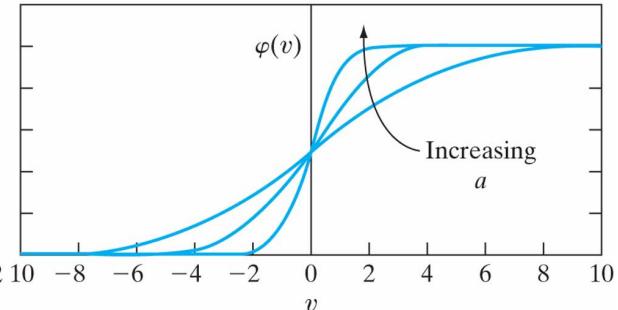
# Modelo de neuronal artificial



## Funciones de activación $\varphi(v)$



Neuronas binarias



Neuronas sigmoidales

[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# Modelo de neurona artificial



## Funciones de activación sigmoidales

- Función logística [0,1]

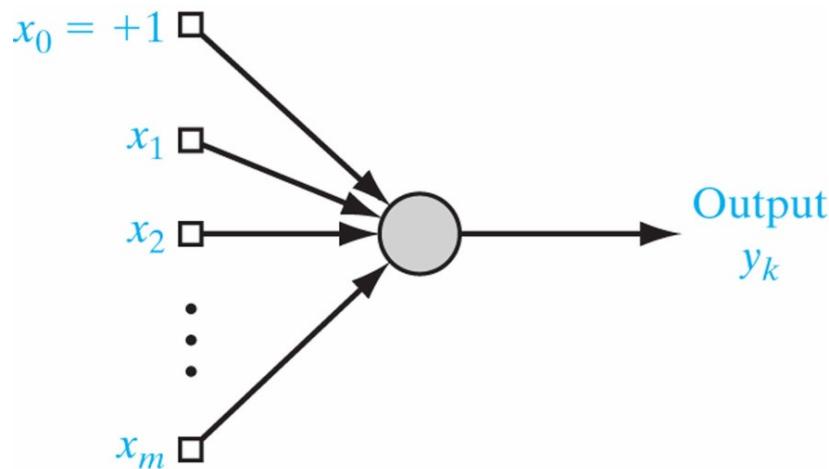
$$\varphi(v) = \frac{1}{1 + e^{-av}}$$

- Tangente hiperbólica [-1,1]

$$\varphi(v) = \tanh v = \frac{e^v - e^{-v}}{e^v + e^{-v}}$$



# Perceptrones



# Perceptrones



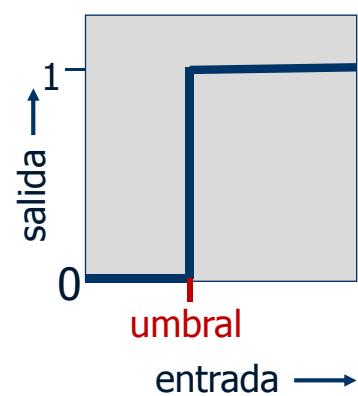
## Modelo de neurona

Neuronas binarias con umbral

[McCulloch & Pitts, 1943]

$$z = \sum_i x_i w_i$$

$$y = \begin{cases} 1 & \text{si } z \geq 0 \\ 0 & \text{en otro caso} \end{cases}$$



Asumiendo  $x_0=1$  y  $w_0=b$  (umbral  $\theta=-b$ )



# Perceptrones



## La primera generación de redes neuronales

- Popularizados por Frank Rosenblatt en los años 60.
- Minsky y Papert analizaron lo que podían hacer y mostraron sus limitaciones en su libro de 1969.

Muchos pensaron que esas limitaciones se extendían a todos los modelos de redes neuronales, aunque no es así.

Su algoritmo de aprendizaje todavía se usa para tareas en las que los vectores de características contienen millones de elementos.



# Perceptrones



## Reconocimiento de patrones

- Se convierten los datos de entrada en un vector de características  $x_j$ .
- Se aprenden los pesos asociados a cada una de esas características para obtener un valor escalar a partir de cada vector de entrada.
- Si este valor escalar se halla por encima de un umbral, se decide que el vector de entrada corresponde a un ejemplo de la clase objetivo ( $y_k=1$ ).

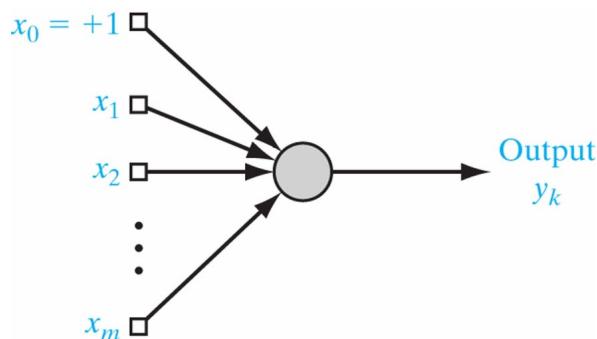


# Perceptrones



## Algoritmo de aprendizaje

Un umbral (positivo) es equivalente a un sesgo/bias (negativo), por lo que podemos evitar tratar de forma separada el umbral añadiendo una entrada fija  $x_0 = +1$ . De esta forma, aprendemos el umbral como si fuese un peso más.



# Perceptrones



## Algoritmo de aprendizaje

Se seleccionan ejemplos del conjunto de entrenamiento utilizando cualquier política que garantice que todos los ejemplos de entrenamiento se acabarán escogiendo:

- Si la salida es correcta, se dejan los pesos tal cual.
- Si la unidad de salida incorrectamente da un cero, se añade el vector de entrada al vector de pesos.
- Si la unidad de salida incorrectamente da un uno, se resta el vector de entrada del vector de pesos.



# Perceptrones



TABLE 1.1 Summary of the Perceptron Convergence Algorithm

Variables and Parameters:

- $\mathbf{x}(n)$  =  $(m + 1)$ -by-1 input vector  
=  $[+1, x_1(n), x_2(n), \dots, x_m(n)]^T$   
 $\mathbf{w}(n)$  =  $(m + 1)$ -by-1 weight vector  
=  $[b, w_1(n), w_2(n), \dots, w_m(n)]^T$   
 $b$  = bias  
 $y(n)$  = actual response (quantized)  
 $d(n)$  = desired response  
 $\eta$  = learning-rate parameter, a positive constant less than unity

1. Initialization. Set  $\mathbf{w}(0) = \mathbf{0}$ . Then perform the following computations for time-step  $n = 1, 2, \dots$
2. Activation. At time-step  $n$ , activate the perceptron by applying continuous-valued input vector  $\mathbf{x}(n)$  and desired response  $d(n)$ .

3. Computation of Actual Response. Compute the actual response of the perceptron as

$$y(n) = \text{sgn}[\mathbf{w}^T(n)\mathbf{x}(n)]$$

where sgn( $\cdot$ ) is the signum function.

4. Adaptation of Weight Vector. Update the weight vector of the perceptron to obtain

$$\mathbf{w}(n + 1) = \mathbf{w}(n) + \eta[d(n) - y(n)]\mathbf{x}(n)$$

where

$$d(n) = \begin{cases} +1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathcal{C}_1 \\ -1 & \text{if } \mathbf{x}(n) \text{ belongs to class } \mathcal{C}_2 \end{cases}$$

5. Continuation. Increment time step  $n$  by one and go back to step 2.

[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# Perceptrones

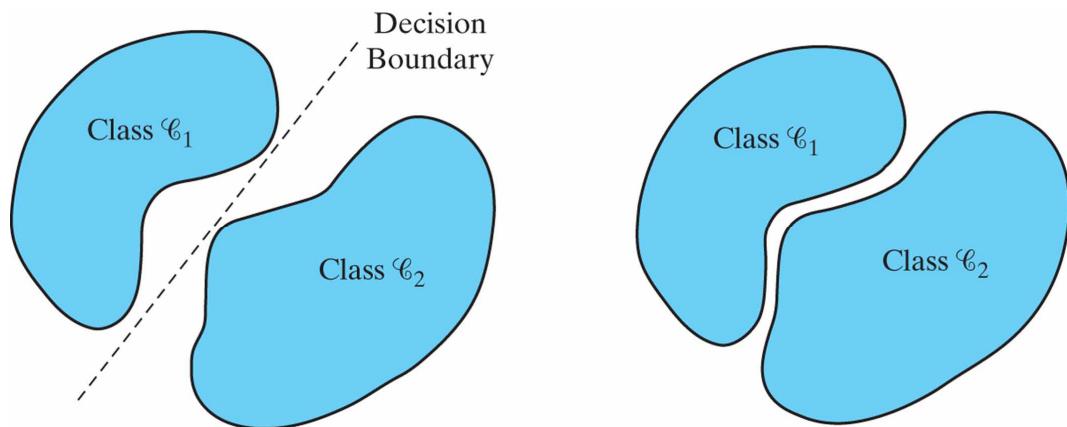


## Algoritmo de aprendizaje

- El algoritmo de aprendizaje del perceptrón garantiza encontrar un conjunto de pesos que proporcione la respuesta correcta **si tal conjunto existe**.
- El perceptrón es un modelo de clasificación lineal, por lo cual será capaz de clasificar correctamente los ejemplos de entrada siempre que las clases sean linealmente separables.



# Perceptrones



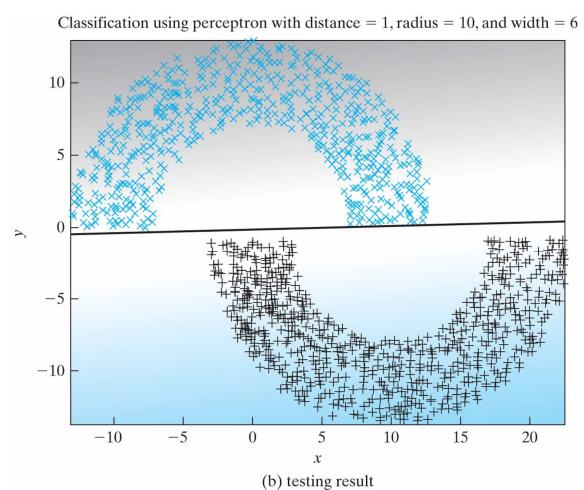
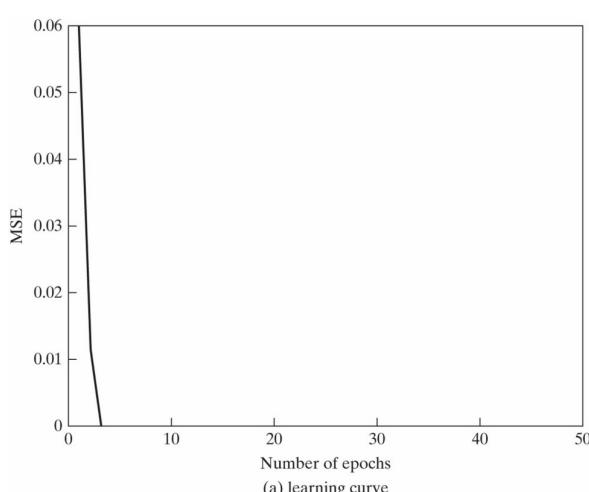
Clases  
linealmente separables

Clases  
linealmente no separables

[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# Perceptrones

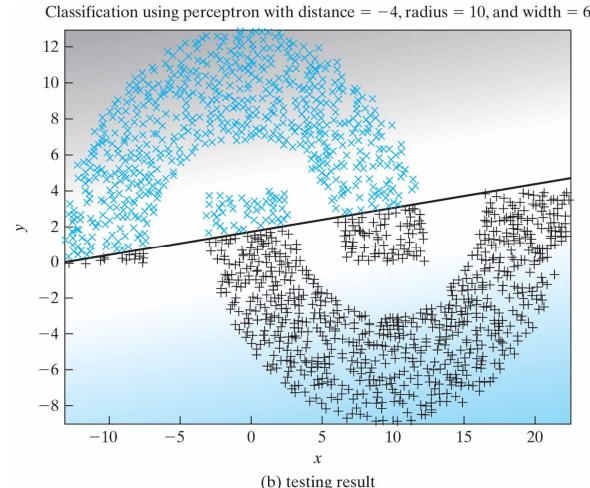
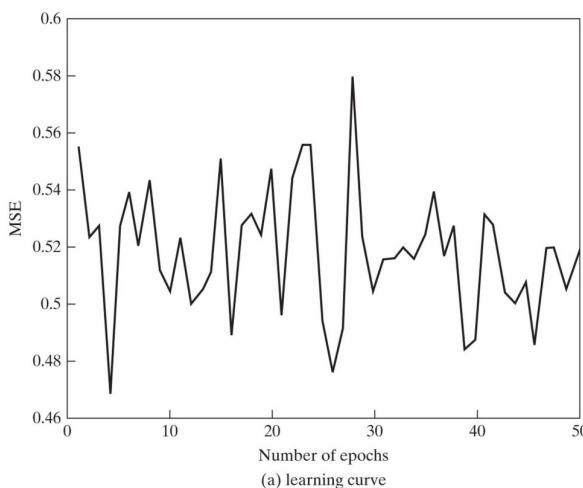


Clases linealmente separables

[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# Perceptrones



## Clases linealmente no separables

[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# Perceptrones



## Algoritmo de aprendizaje: Interpretación geométrica

Espacio de pesos:

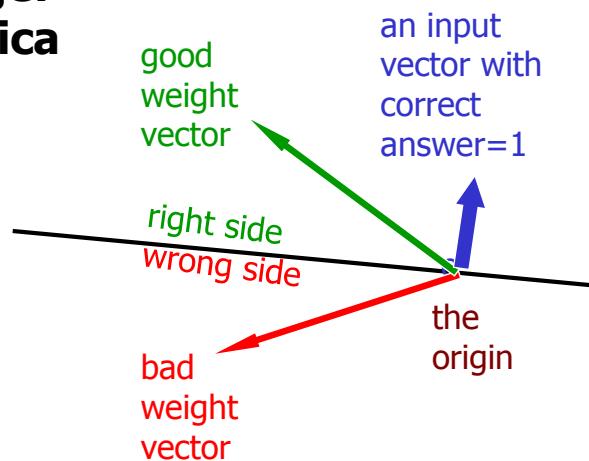
- Una dimensión por cada peso.
- Cada punto del espacio representa un valor particular para el conjunto de pesos.
- Cada caso de entrenamiento corresponde a un hiperplano que pasa por el origen (tras eliminar el umbral e incluirlo como un peso más)



# Perceptrones



## Algoritmo de aprendizaje: Interpretación geométrica



Cada caso define un hiperplano:

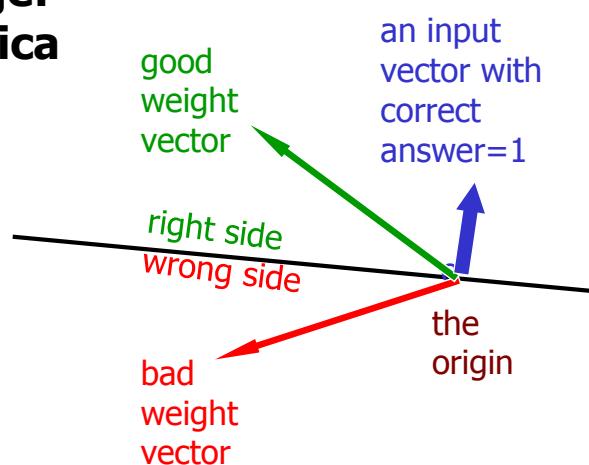
- Hiperplano perpendicular al vector de entrada.
- Los pesos deben quedar a un lado del hiperplano.



# Perceptrones



## Algoritmo de aprendizaje: Interpretación geométrica



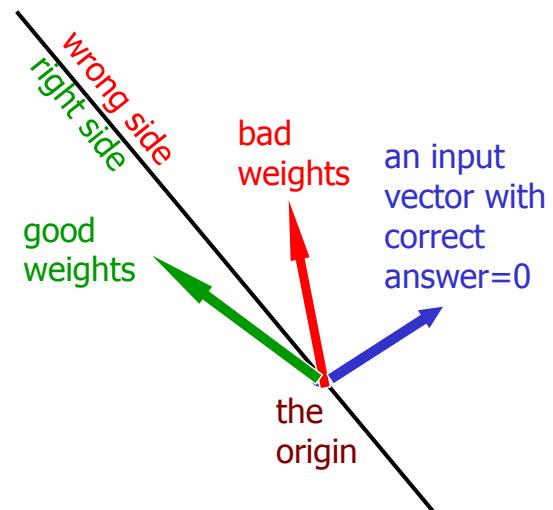
Si el producto escalar del vector de entrada con el vector de pesos es negativo, la salida será la equivocada.



# Perceptrones



## Algoritmo de aprendizaje: Interpretación geométrica



Si el producto escalar del vector de entrada con el vector de pesos es negativo, la salida será la equivocada.

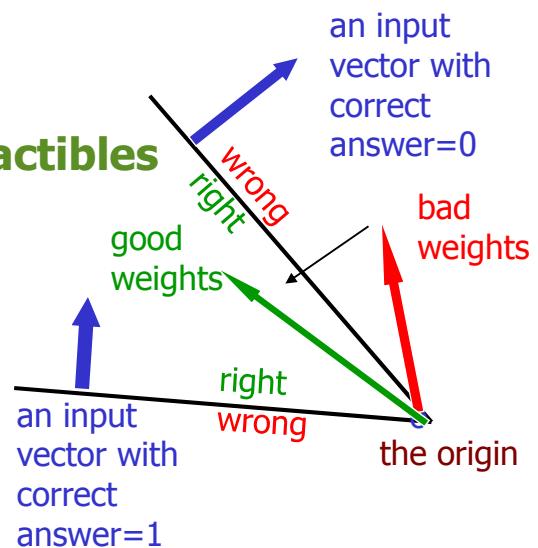


# Perceptrones



## Algoritmo de aprendizaje: Interpretación geométrica

### El hipercono de soluciones factibles



Para que el aprendizaje sea correcto, debemos encontrar un punto que esté en el lado correcto de todos los hiperplanos (que puede que no exista!!!).

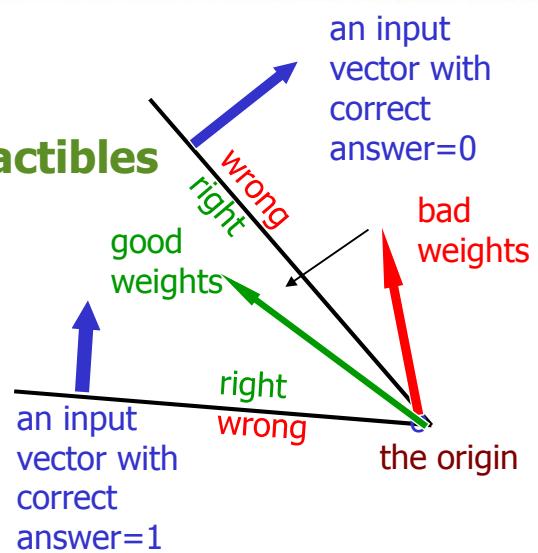


# Perceptrones



**Algoritmo de aprendizaje:  
Interpretación geométrica**

**El hipercono de soluciones factibles**



Si existe un conjunto de pesos que proporcionen la respuesta adecuada para todos los casos, estará en un hipercono con su ápice en el origen.

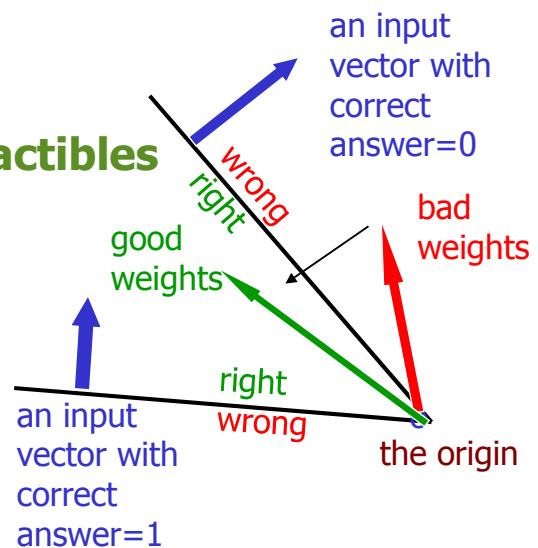


# Perceptrones



**Algoritmo de aprendizaje:  
Interpretación geométrica**

**El hipercono de soluciones factibles**



Al definir el hipercono de soluciones factibles una región convexa, la media de dos buenos vectores de pesos será también un buen vector de pesos...



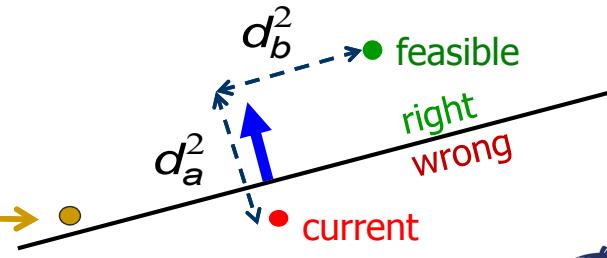
# Perceptrones



## Algoritmo de aprendizaje: Corrección del algoritmo

Hipótesis errónea: Cada vez que el perceptrón comete un error, el algoritmo de aprendizaje acerca el vector de pesos actual hacia todas las soluciones factibles.

Problem case: The weight vector may not get closer to this feasible vector!

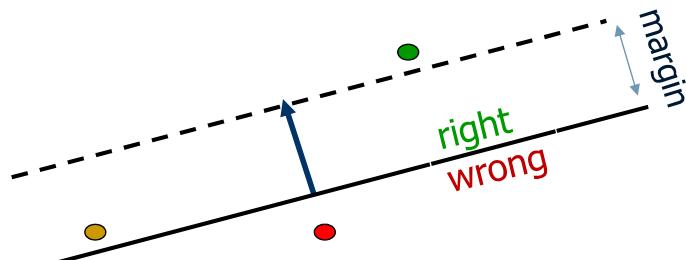


# Perceptrones



## Algoritmo de aprendizaje: Corrección del algoritmo

Consideramos los vectores de pesos “generosamente factibles” que quedan, dentro de la región factible, con un margen al menos tan grande como la longitud del vector de entrada.

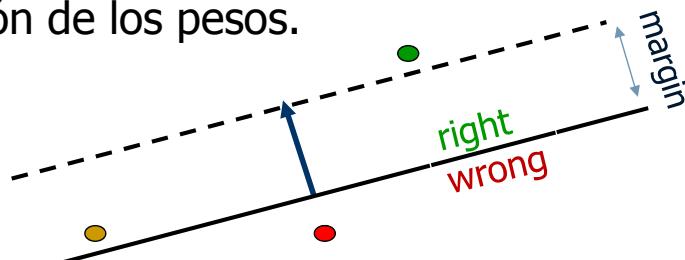


# Perceptrones



## Algoritmo de aprendizaje: Corrección del algoritmo

Cada vez que el perceptrón se equivoca, el cuadrado de la distancia a todos esos vectores de pesos “generosamente factibles” siempre se decrementa en, al menos, el cuadrado de la longitud del vector de actualización de los pesos.

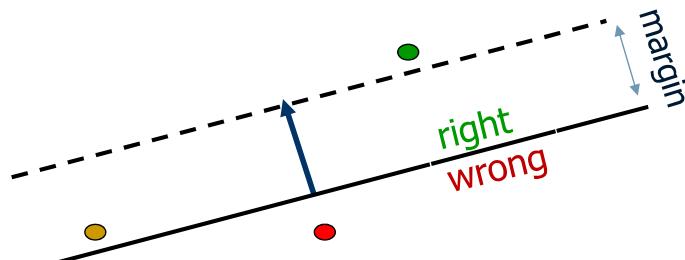


# Perceptrones



## Algoritmo de aprendizaje: Corrección del algoritmo

Por tanto, tras un número finito de errores, el vector de pesos deberá estar en la región factible, si ésta existe.



# Perceptrones



## Limitaciones

- Si se pueden elegir todas las características que deseemos, se puede hacer cualquier cosa.

p.ej. Con una entrada para cada posible vector ( $2^n$ ), se puede discriminar cualquier función booleana, si bien el perceptrón no generalizará bien.

- Si las entradas vienen determinadas, existen severas limitaciones sobre lo que un perceptrón puede aprender (p.ej. XOR y EQ).



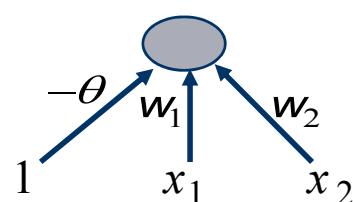
# Perceptrones



## Limitaciones

Un perceptrón no puede decidir si dos bits son iguales:

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



4 casos definen 4 desigualdades imposibles de satisfacer:

$$w_1 + w_2 \geq \theta, \quad 0 \geq \theta$$

$$w_1 < \theta, \quad w_2 < \theta$$



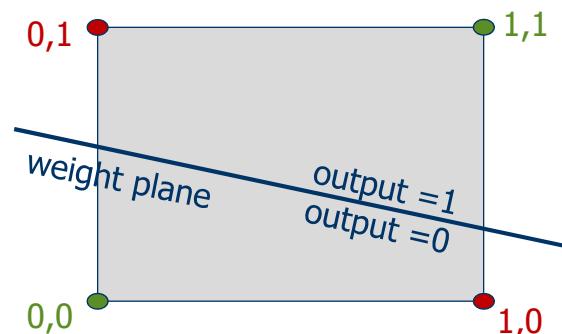
# Perceptrones



## Limitaciones:

### Interpretación geométrica

Casos positivos y negativos NO pueden separarse por un plano



### Espacio de datos:

- Una dimensión por cada característica.
- Un vector de entrada es un punto.
- Un vector de pesos define un hiperplano.
- El hiperplano es perpendicular al vector de pesos y está a una distancia del origen dada por el umbral  $\theta$ .



# Perceptrones



## Limitaciones

Diferenciar entre diferentes patrones que tengan el mismo número de píxeles tampoco se puede si se admiten traslaciones:

	pattern A
	pattern A
	pattern A
	pattern B
	pattern B
	pattern B



# Perceptrones



## Limitaciones

- El **teorema de invarianza de grupos** de Minsky y Papert establece que la parte del perceptrón que aprende no es capaz de reconocer patrones si las transformaciones a las que pueden estar sometidos dichos patrones forman un grupo.
- La parte interesante del reconocimiento de patrones debe resolverse manualmente (añadiendo nuevas características), pero no puede aprenderse usando un perceptrón...



# Backpropagation



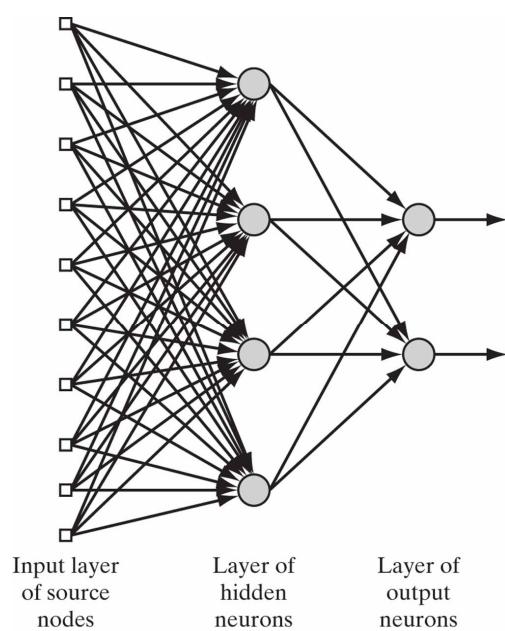
## Feed-forward neural networks

Topología de red más habitual en la práctica

Sin realimentación,  
organizadas por capas:

- Capa de entrada
- Capa oculta
- Capa de salida

[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# Backpropagation

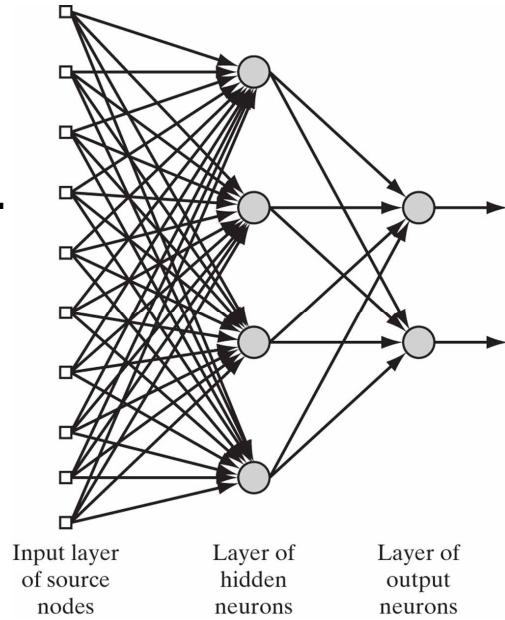


## Feed-forward neural networks

Topología de red más habitual en la práctica

Si hay más de una capa oculta, se denominan  
**“deep” neural networks.**

[Haykin: “Neural Networks and Learning Machines”, 3<sup>rd</sup> edition]



# Backpropagation



- Las redes neuronales sin neuronas ocultas (p.ej. perceptrones) están muy limitadas con respecto a lo que son capaces de aprender.
- El uso de capas de adicionales de neuronas lineales no ayuda (el resultado seguiría siendo lineal), por lo que necesitamos múltiples capas de unidades no lineales.



# Backpropagation



## Problema

¿Cómo entrenamos estas redes multicapa?

- Necesitamos un algoritmo eficiente que nos permita adaptar todos los pesos de una red multicapa, no sólo los de la capa de salida.
- Aprender los pesos correspondientes a las neuronas de las capas ocultas equivale a aprender nuevas características (no presentes en el conjunto de entrenamiento), lo que resulta especialmente difícil porque nadie nos dice directamente qué es lo que deberíamos aprender en esas unidades ocultas.



# Backpropagation



## Problema

¿Cómo entrenamos estas redes multicapa?

- El algoritmo utilizado por el perceptrón no puede extenderse para redes multicapa (garantiza su convergencia en una región convexa, pero no en un problema no convexo, en el que la media de dos buenas soluciones puede no ser buena).
- Las redes multicapa no usan el algoritmo de aprendizaje del perceptrón, por lo que no es correcto llamarlas perceptrones multicapa (aunque sí usual).



# Backpropagation



## Problema

¿Cómo entrenamos estas redes multicapa?

- En vez de fijarnos en los cambios de los pesos, nos fijaremos en los cambios de las salidas, que intentaremos acercar a las salidas deseadas (estrategia válida para problemas no convexos).

Empezaremos por un ejemplo de juguete...

una neurona lineal



# Backpropagation



## Neurona lineal

a.k.a. filtro lineal

$$y = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$$

OBJETIVO:

Minimizar la suma de los errores sobre todos los ejemplos del conjunto de entrenamiento.

Podríamos resolver el problema analíticamente, pero la solución sería difícil de generalizar, por lo que diseñaremos un algoritmo iterativo (menos eficiente) que luego podamos generalizar...



# Backpropagation



## Neurona lineal

Todos los días tomamos café, zumo y tostadas en la cafetería, pero sólo al final de la semana nos pasan el total de la cuenta, sin darnos el precio individual de cada producto. Tras varias semanas, deberíamos ser capaces de adivinar esos precios individuales...

### Algoritmo iterativo

- Empezar con una estimación (aleatoria) de los precios.
- Ajustar los precios estimados para que encajen con los importes facturados.



# Backpropagation



## Neurona lineal

Cada semana, el total nos impone una restricción lineal sobre los precios de las cantidades consumidas:

$$\text{total} = w_{\text{café}}x_{\text{café}} + w_{\text{zumo}}x_{\text{zumo}} + w_{\text{tostada}}x_{\text{tostada}}$$

- Los precios son los pesos  $w = (w_{\text{café}}, w_{\text{zumo}}, w_{\text{tostada}})$ .
- Las entradas corresponden a las cantidades consumidas de cada producto.

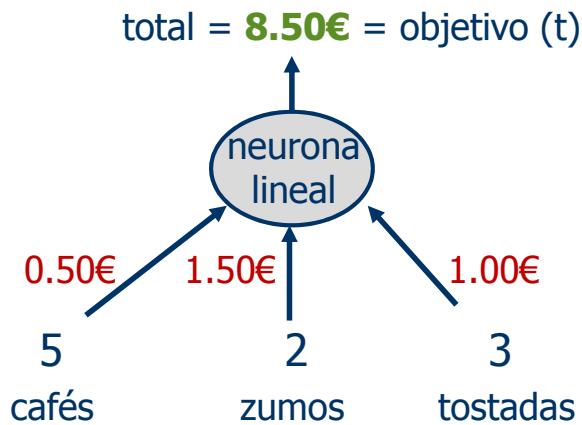


# Backpropagation



## Neurona lineal

Los precios reales, utilizados para pasarnos la factura:



# Backpropagation



## Neurona lineal

Precios estimados inicialmente (0.50€):

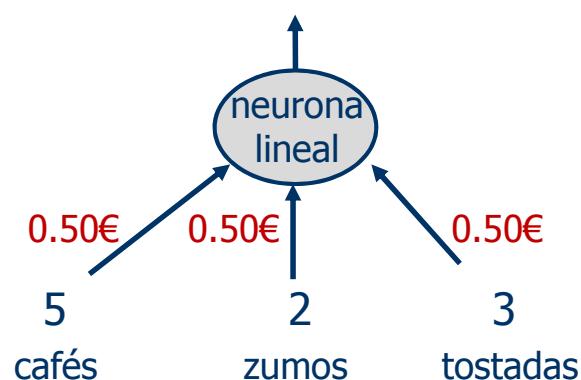
Error residual = 3.50€

Corrección de los pesos:

### Regla delta

$$\Delta w_i = \eta x_i(t - y)$$

total = **5.00€** = estimación (y)



$\eta$  [eta] es la tasa de aprendizaje utilizada



# Backpropagation



## Neurona lineal

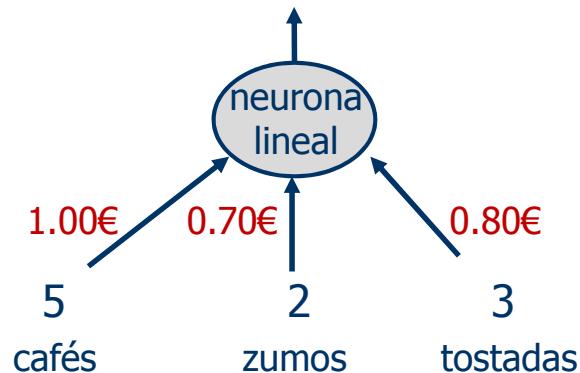
Precios ajustados  
usando  $\eta = 1/35$ :

Error residual = 3.50€

$\Delta w = (+50, +20, +30)$

Nuevo error = -0.30€

total = **8.80€** = estimación (y)



NOTA:

Nuestra estimación del precio del café ha empeorado!!



# Backpropagation



## Neurona lineal

Derivación de la regla delta

- Error  
(residuos al cuadrado) 
$$E = \frac{1}{2} \sum_{n \in training} (t^n - y^n)^2$$
- Derivada del error 
$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_n \frac{\partial y^n}{\partial w_i} \frac{dE^n}{dy^n} = -\sum_n x_i^n (t^n - y^n)$$
- Ajuste de los pesos  
en proporción al error 
$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} = \eta \sum_n x_i^n (t^n - y^n)$$



# Backpropagation



## Neurona lineal

Algoritmo de aprendizaje iterativo

- Con una tasa de aprendizaje lo suficientemente pequeña, nos iremos acercando a la mejor solución posible.
- El algoritmo puede ser muy lento si existe correlación entre las variables de entrada (si siempre pedimos zumo y tostadas, será difícil determinar cómo repartir el importe y determinar el precio de esos productos).
- Tenemos que elegir un valor adecuado para un parámetro del algoritmo: la tasa de aprendizaje ( $\eta$ ).

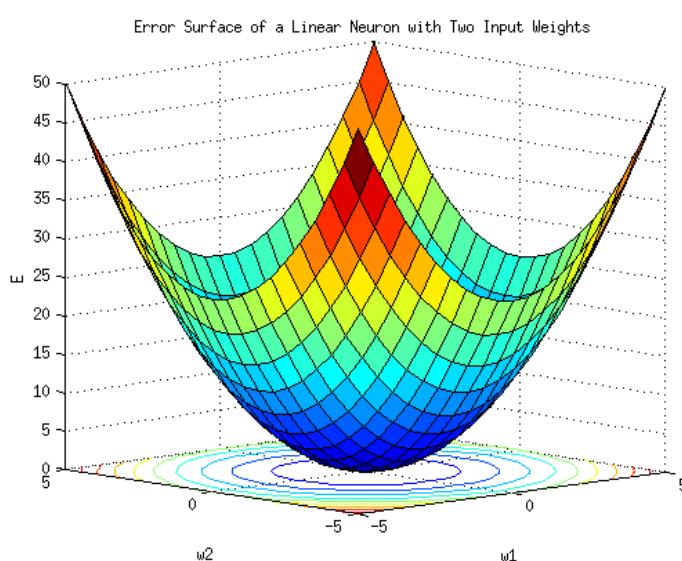


# Backpropagation



## Neurona lineal

Superficie de error

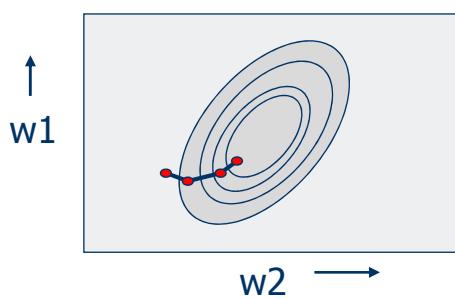


# Backpropagation

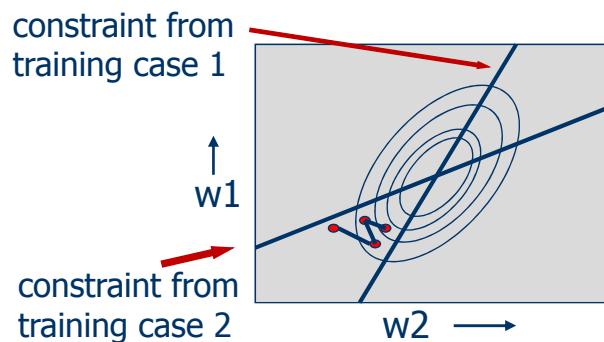


## Neurona lineal

Aprendizaje por lotes [batch learning] vs. online [stochastic]



Batch learning



Online learning



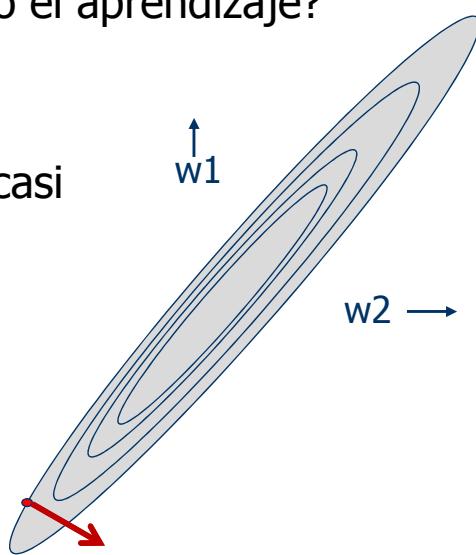
# Backpropagation



## Neurona lineal

¿Por qué puede ser muy lento el aprendizaje?

Si la elipse es muy alargada, la dirección del gradiente es casi perpendicular a la dirección hacia el mínimo...



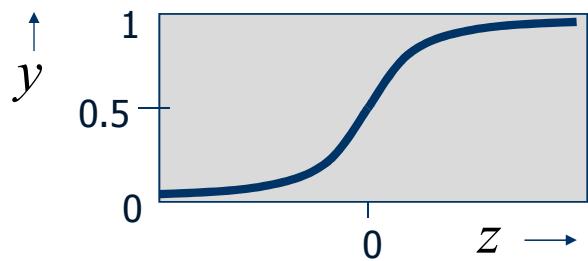
# Backpropagation



## Neurona sigmoidal Función logística

$$z = \sum_i x_i w_i$$

$$y = \frac{1}{1 + e^{-z}}$$



# Backpropagation



## Neurona sigmoidal Función logística

$$z = \sum_i x_i w_i \quad y = \frac{1}{1 + e^{-z}}$$

- Derivadas del logit ( $z$ ) con respecto a pesos y entradas

$$\frac{\partial z}{\partial w_i} = x_i \quad \frac{\partial z}{\partial x_i} = w_i$$

- Derivada de la salida ( $y$ ) con respecto al logit ( $z$ )

$$\frac{dy}{dz} = y(1 - y)$$





# Backpropagation

## Neurona sigmoidal

Función logística

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

$$\frac{dy}{dz} = \frac{-1(-e^{-z})}{(1 + e^{-z})^2} = \left( \frac{1}{1 + e^{-z}} \right) \left( \frac{e^{-z}}{1 + e^{-z}} \right) = y(1 - y)$$

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



# Backpropagation



## Neurona sigmoidal

Función logística

$$\frac{\partial y}{\partial w_i} = \frac{\partial z}{\partial w_i} \frac{dy}{dz} = x_i y(1 - y)$$

$$\frac{\partial E}{\partial w_i} = \sum_n \frac{\partial y^n}{\partial w_i} \frac{\partial E}{\partial y^n} = - \sum_n [x_i^n | y^n (1 - y^n) | (t^n - y^n)]$$

↑  
Término extra  
(pendiente de la función logística)

Regla delta



# Backpropagation



Recapitulemos:

- Las redes neuronales sin unidades ocultas tienen muchas limitaciones, pero añadir características manualmente (como en el perceptrón) es tedioso.
- Nos gustaría ser capaces de encontrar características automáticamente, sin necesidad de conocer cada problema al detalle ni recurrir a un proceso de prueba y error...



# Backpropagation



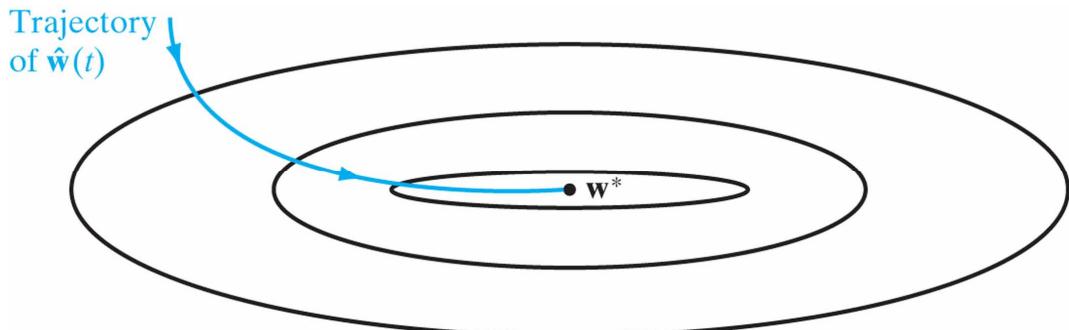
## IDEA

No sabemos qué deben hacer las neuronas ocultas, pero podemos calcular cómo cambia el error cuando cambia su actividad.

- En vez de utilizar la salida deseada para entrenar las neuronas ocultas, usamos la derivada del error con respecto a sus actividades ( $\delta E/\delta y$ ).
- La actividad de cada neurona oculta puede tener efectos en muchas neuronas de salida, por lo que debemos combinarlos.
- Una vez que tenemos las derivadas del error para todas las unidades ocultas, podemos calcular las derivadas del error para sus pesos de entrada.



# Backpropagation



[Haykin: "Neural Networks and Learning Machines", 3rd edition]



# Backpropagation

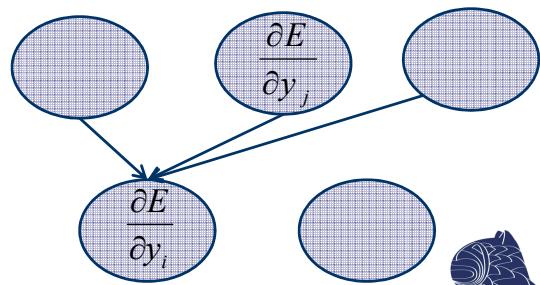


## Propagación de errores

- Convertimos la discrepancia entre la salida de la red y su salida deseada en una derivada del error ( $\delta E / \delta y$ ).
- Calculamos las derivadas del error de cada capa oculta a partir de las derivadas del error de la capa superior.
- Usamos  $\delta E / \delta y$  para obtener  $\delta E / \delta w$ .

$$E = \frac{1}{2} \sum_{j \in \text{output}} (t_j - y_j)^2$$

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$



# Backpropagation

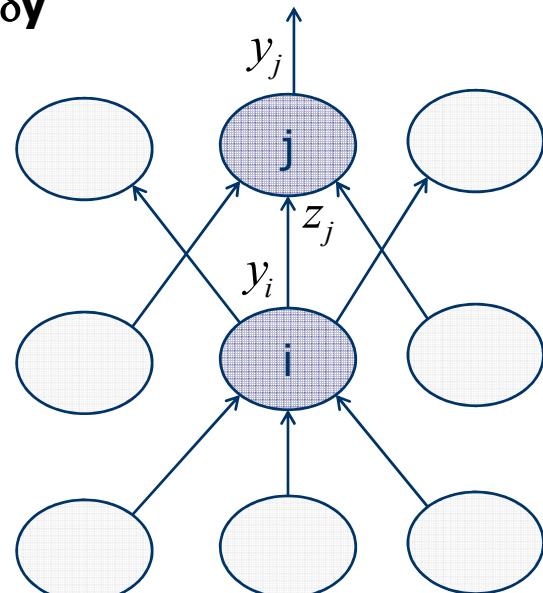


## Propagación de errores $\delta E/\delta y$

$$\frac{\partial E}{\partial z_j} = \frac{dy_j}{dz_j} \frac{\partial E}{\partial y_j} = y_j(1-y_j) \frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dz_j}{dy_i} \frac{\partial E}{\partial z_j} = \sum_j w_{ij} \frac{\partial E}{\partial z_j}$$

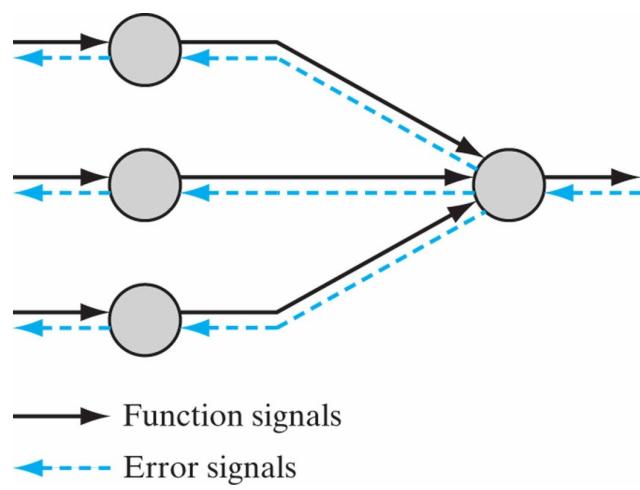
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}} \frac{\partial E}{\partial z_j} = y_i \frac{\partial E}{\partial z_j}$$



# Backpropagation



El algoritmo de propagación de errores nos proporciona un método eficiente para calcular las derivadas del error  $\delta E/\delta w$  para cada peso.



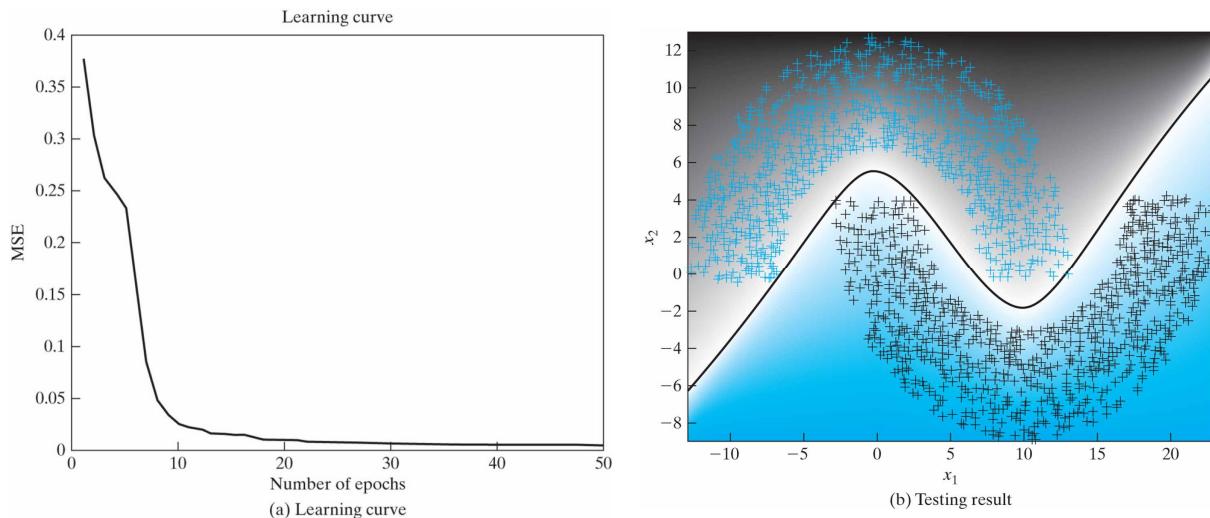
[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# Backpropagation



## Ejemplo



[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



## En la práctica



### Algoritmo de aprendizaje de redes multicapa

Aspectos que debemos considerar en su diseño:

- **Parámetros:** ¿Qué topología de red utilizamos?
- **Optimización:** ¿Cómo obtenemos los pesos?
- **Generalización:** ¿Cómo conseguimos que la red funcione bien con datos distintos a los del conjunto de entrenamiento?
- **Invarianza:** ¿Cómo conseguimos que la red sea robusta frente a transformaciones comunes en los datos?



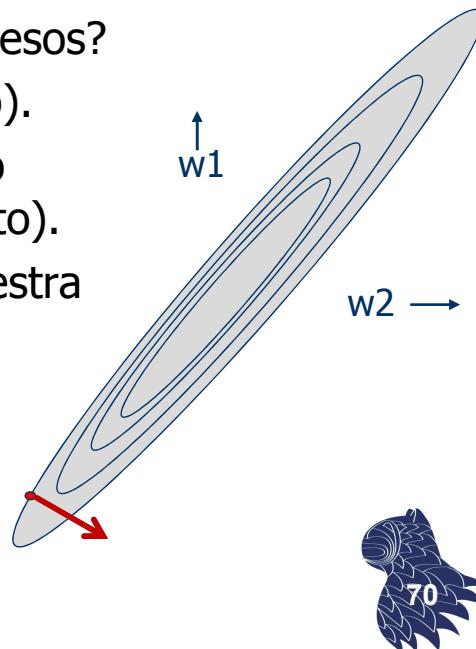
# En la práctica



## Modo de entrenamiento

¿Con qué frecuencia se ajustan los pesos?

- **Online** (después de cada ejemplo).
- **Batch** (después de cada recorrido sobre el conjunto de entrenamiento).
- **Mini-batch** (después de una muestra del conjunto de entrenamiento)



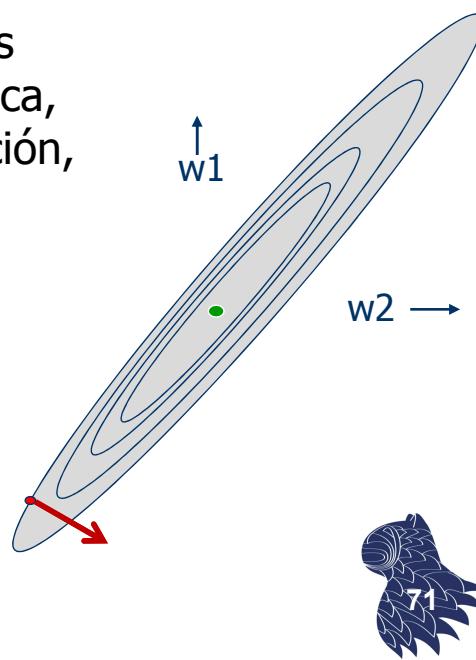
# En la práctica



## "Batch learning", a.k.a. Full-batch gradient descent

Aunque para las neuronas no lineales la superficie de error no sea cuadrática, localmente nos vale como aproximación, por lo que el aprendizaje "full batch" sigue presentando problemas:

La dirección de máxima pendiente del gradiente no apunta al mínimo salvo que la elipse sea un círculo.



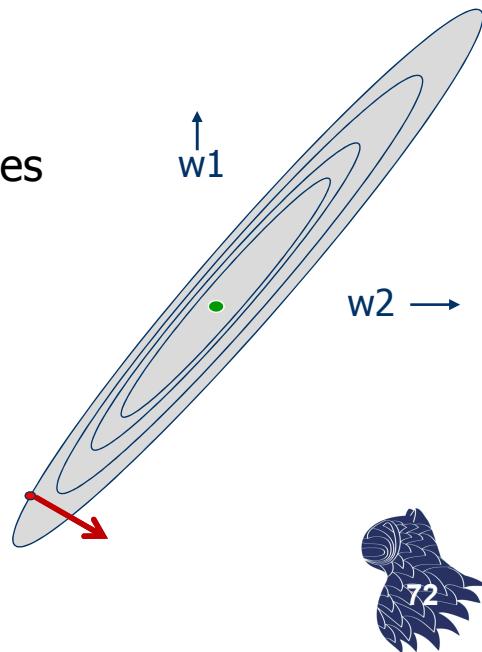
# En la práctica



## “Batch learning”, a.k.a. Full-batch gradient descent

Lo que nos gustaría conseguir:

- Movernos más rápido en direcciones con gradientes pequeños pero consistentes.
- Movernos más despacio en direcciones con gradientes grandes pero inconsistentes.



# En la práctica



## Online learning

Estimación del gradiente a partir del error observado para cada ejemplo de entrenamiento.

Ventajas del aprendizaje online:

- Mucho más rápido que el aprendizaje por lotes.
- Suele obtener mejores soluciones.
- Facilita adaptarse a cambios.



# En la práctica



## Online learning vs. Batch learning

Pese a las ventajas del aprendizaje “online”, existen motivos que justifican utilizar el aprendizaje por lotes.

Ventajas del aprendizaje por lotes:

- Condiciones de convergencia bien conocidas.
- Muchas técnicas de optimización sólo funcionan con batch learning (p.ej. gradientes conjugados).
- Análisis teórico (dinámica y convergencia).



# En la práctica



## Mini-batch gradient descent

Si existe redundancia en el conjunto de entrenamiento, el gradiente de su primera mitad será similar al de su segunda mitad, por lo que podemos actualizar los pesos usando una mitad y volver a obtener un gradiente para esos pesos actualizados usando la segunda mitad.

La versión extrema de esta estrategia es el “online learning” (actualizar los pesos después de cada ejemplo).

Ambas son formas de **gradiente descendente estocástico** [stochastic gradient descent].



# En la práctica



## Mini-batch gradient descent

El uso de mini-lotes (balanceados para las distintas clases en problemas de aprendizaje supervisado) suele ser mejor que el aprendizaje “online”.

- Se requieren menos cálculos para actualizar los pesos.
- El cálculo simultáneo del gradiente para muchos casos puede realizarse utilizando operaciones con matrices que se pueden implementar de forma muy eficiente utilizando GPUs.



# En la práctica



## Efficient BackProp [LeCun et al.]

Consejos para la implementación de backpropagation

### RECOMENDACIÓN: ENTRENAMIENTO

Elegir los ejemplos que proporcionan mayor información para el entrenamiento de la red.

- Barajar los ejemplos para que ejemplos consecutivos (casi) nunca pertenezcan a la misma clase.
- Presentar los ejemplos que producen mayores errores con mayor frecuencia que los ejemplos que producen menos errores



# En la práctica



## Efficient BackProp [LeCun et al.]

Consejos para la implementación de backpropagation

RECOMENDACIÓN: PREPROCESAMIENTO

Normalización de las entradas (p.ej. z-scores).

- La media de cada variable de entrada en el conjunto de entrenamiento debería ser cercana a cero.
- La escala de las variables de entrada debería ajustarse para que sus covarianzas sean similares.
- Si es posible, las variables de entrada deberían estar decorreladas (sin correlación).



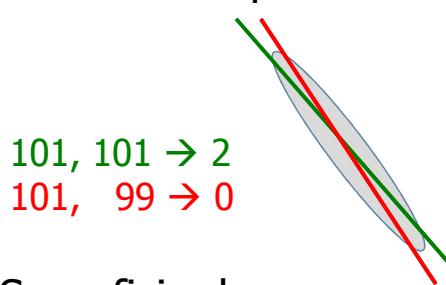
# En la práctica



## Efficient BackProp

Consejos para la implementación de backpropagation

EJEMPLO: Desplazamiento (media 0)



Superficie de error  
(datos originales)



Superficie de error  
(datos con media 0)



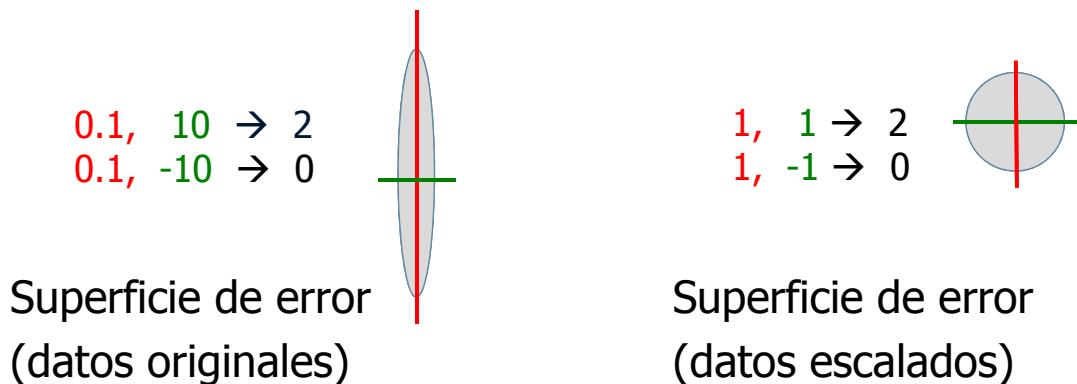
# En la práctica



## Efficient BackProp

Consejos para la implementación de backpropagation

EJEMPLO: Escalado



# En la práctica



## Efficient BackProp

Consejos para la implementación de backpropagation

Un método para decorrelar las variables de entrada:

### Análisis de componentes principales [PCA]

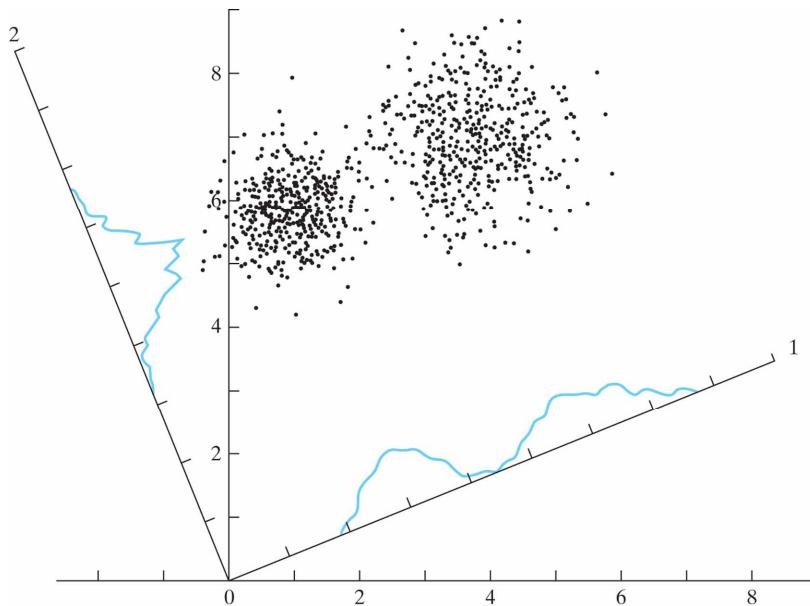
- Permite reducir la dimensionalidad de los datos (eliminando componentes con menores eigenvalues).
- Permite convertir una superficie de error elíptica en una circular, en la que el gradiente apunta directamente al mínimo (dividiendo los componentes principales por las raíces cuadradas de sus respectivos eigenvalues).



# En la práctica



## Análisis de componentes principales [PCA]



[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# En la práctica



## Análisis de componentes principales [PCA]



Compresión de imágenes usando componentes principales

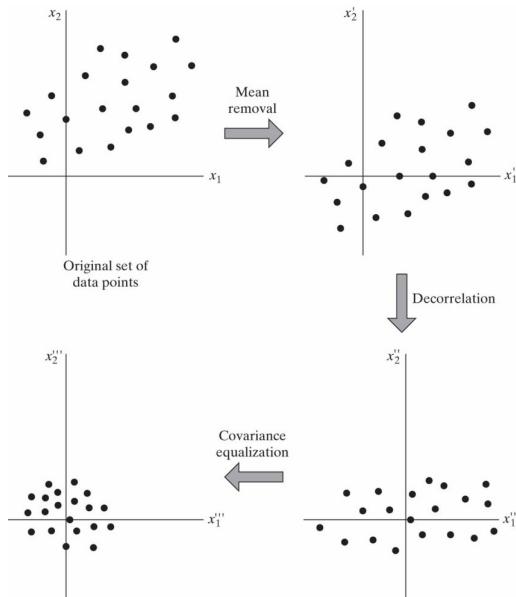
[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# En la práctica



## Análisis de componentes principales [PCA]



[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# En la práctica



## Funciones de activación

RECOMENDACIONES [LeCun et al.]

- Las sigmoides simétricas, como la tangente hiperbólica, suelen converger más rápidamente que la función logística:  $\tanh(x) = 2 * \text{logistic}(2x) - 1$ .
- Función recomendada:  $f(x) = 1.7159 \tanh(2x/3)$ .
- En ocasiones, resulta útil añadir un pequeño término lineal para evitar zonas planas (de gradiente 0), p.ej.  $f(x) = \tanh(x) + ax$



# En la práctica



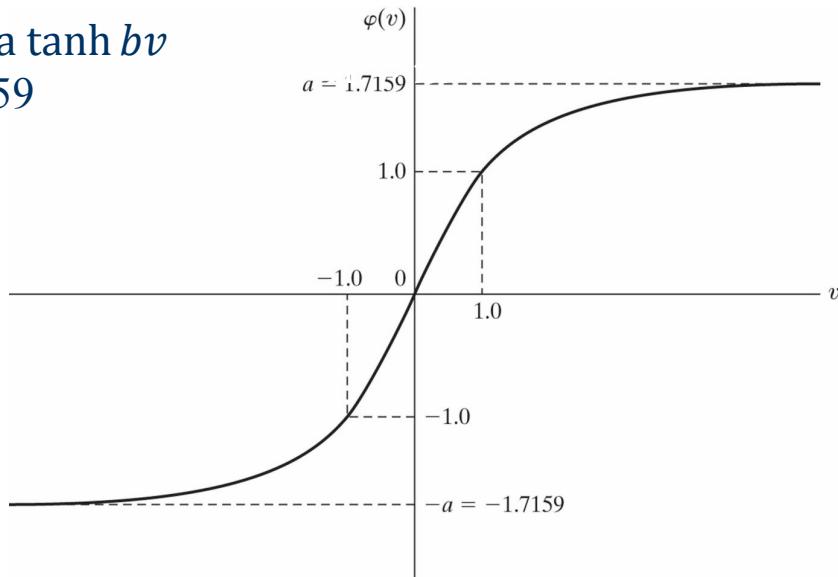
## Funciones de activación

Función de activación recomendada [LeCun et al.]

$$\varphi(v) = a \tanh bv$$

$$a = 1.7159$$

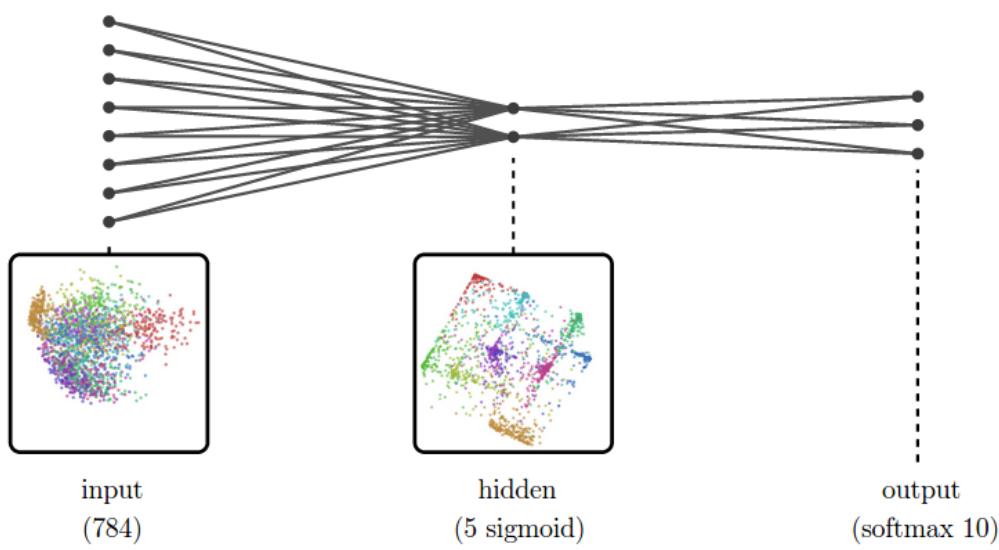
$$b = 2/3$$



# En la práctica



## Funciones de activación sigmoidales



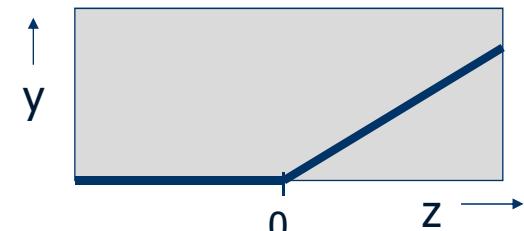
# En la práctica



## Funciones de activación

En ocasiones, especialmente en “deep learning”, se utilizan unidades lineales rectificadas (ReLU) porque su entrenamiento suele ser mucho más rápido:

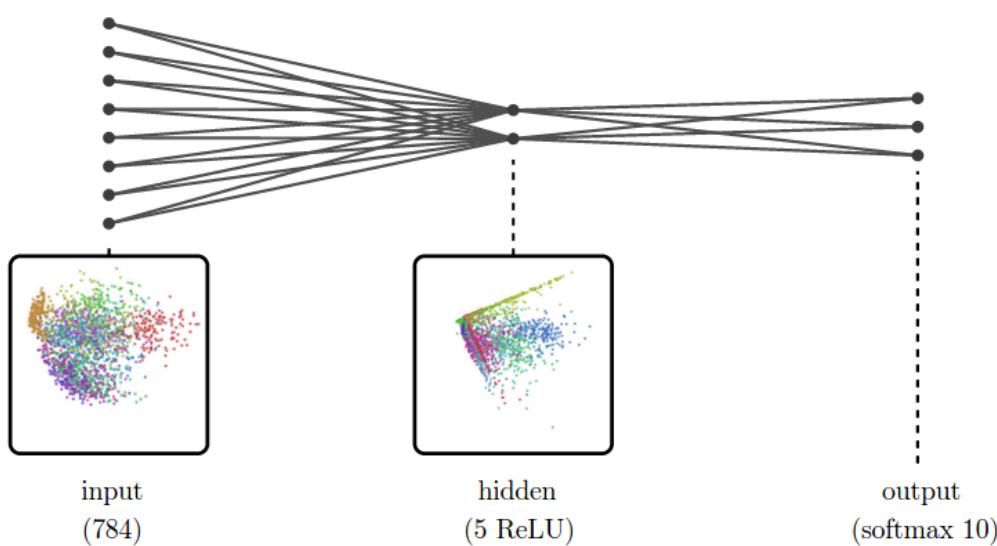
$$z = \sum_i x_i w_i$$
$$y = \begin{cases} z & \text{si } z \geq 0 \\ 0 & \text{en otro caso} \end{cases}$$



# En la práctica



## Funciones de activación: ReLU



# En la práctica



## Efficient BackProp [LeCun et al.]

Consejos para la implementación de backpropagation

### RECOMENDACIÓN: VALORES DE SALIDA

- En problemas de clasificación,  $\{+1, -1\}$ .
- Para evitar problemas de saturación en las unidades de salida, se pueden elegir puntos en el rango de la sigmoide que maximicen su segunda derivada.



# En la práctica



## Inicialización de los pesos

- Si dos neuronas tienen exactamente los mismos pesos, siempre tendrán el mismo gradiente, por lo que no serán capaces de aprender características diferentes.
- Se rompe la simetría inicializando los pesos con valores aleatorios (pequeños).



# En la práctica



## Inicialización de los pesos

- Si una neurona tiene muchas conexiones de entrada ("fan-in" elevado), pequeños cambios en muchos de sus pesos de entrada pueden hacer que nos pasemos.
- Normalmente, queremos pesos más pequeños cuando el "fan-in" es alto, por lo que se suelen inicializar los pesos aleatorios proporcionalmente a  $1/\sqrt{\text{fan-in}}$ .

NOTA:

La misma regla puede aplicarse a la tasa de aprendizaje.



# En la práctica



## Inicialización de los pesos

Consejos para la implementación de backpropagation

RECOMENDACIÓN [LeCun et al.]:

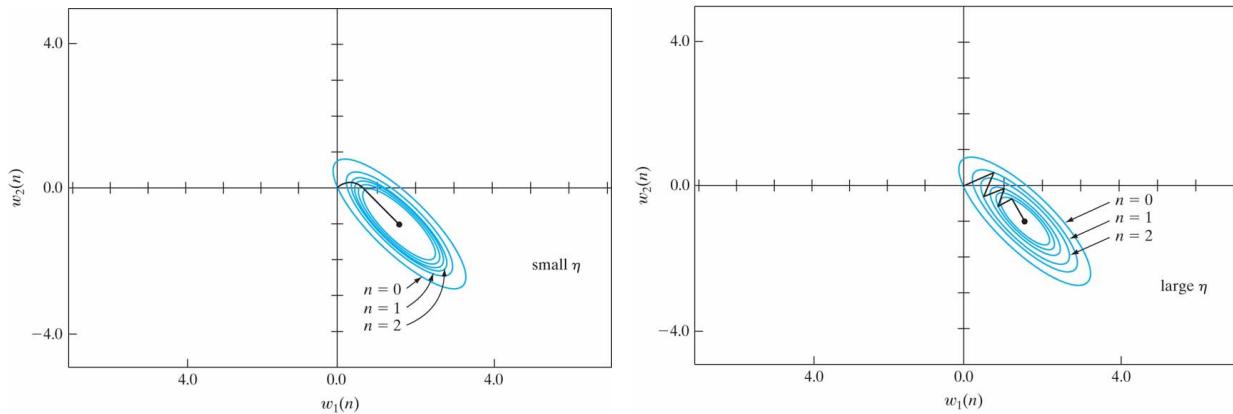
Asumiendo que el conjunto de entrenamiento se ha normalizado y que se usa la función de activación  $\phi(v) = 1.7151 \tanh(2v/3)$ , los pesos se deberían inicializar aleatoriamente utilizando una distribución (p.ej. uniforme) con media **0** y desviación estándar  $\sigma_w = m^{-1/2}$ , donde  $m$  es el fan-in del nodo (número de conexiones que llegan al nodo).



# En la práctica



## Tasa de aprendizaje



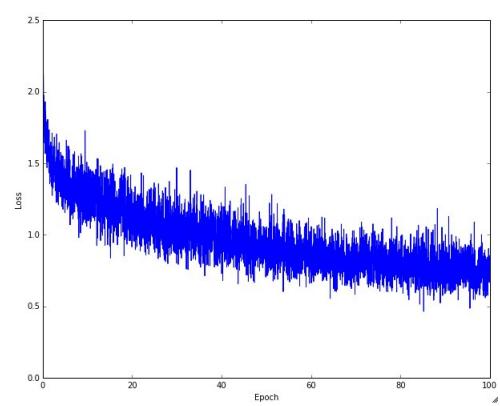
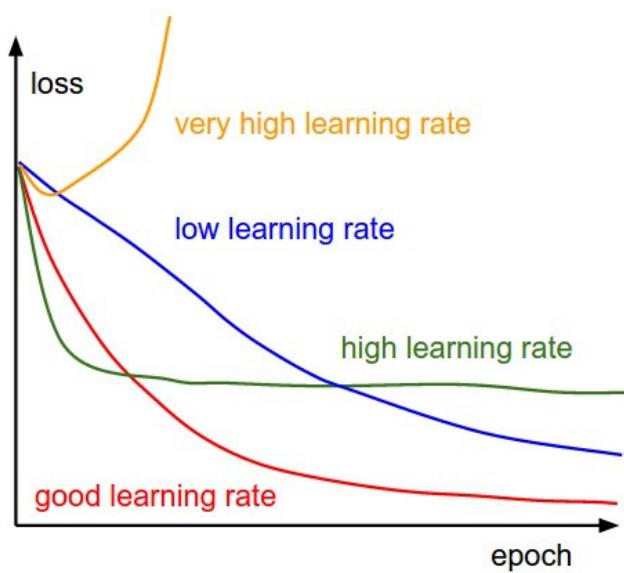
¿Cuánto se ajustan los pesos? Tasa de aprendizaje  
[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# En la práctica



## Tasa de aprendizaje



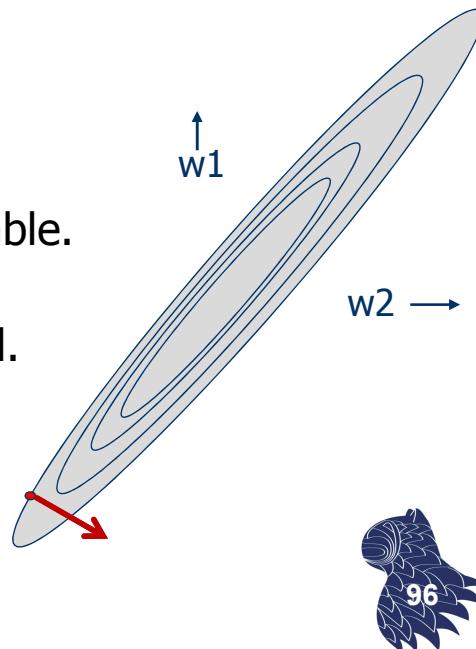
# En la práctica



## Tasa de aprendizaje

¿Cuánto se ajustan los pesos?

- Tasa global de aprendizaje fija.
- Tasa global de aprendizaje adaptable.
- Tasas de aprendizaje ajustadas para cada conexión/peso de la red.



# En la práctica



## Tasa de aprendizaje [LeCun et al.]

Consejos para la implementación de backpropagation

RECOMENDACIÓN: TASAS DE APRENDIZAJE / LEARNING RATES

Para igualar la velocidad de aprendizaje:

- Se le puede dar a cada peso su tasa de aprendizaje.
- Las tasas de aprendizaje deberían ser proporcionales a la raíz cuadrada del número de entradas de la neurona.
- Los pesos de capas inferiores deberían ser mayores que los de capas superiores.



# En la práctica



## Tasa de aprendizaje [Hinton et al.] usando mini-batch learning

Se estima una tasa de aprendizaje inicial:

- Si el error crece u oscila,  
se reduce la tasa de aprendizaje automáticamente.
- Si el error se va reduciendo de forma consistente pero  
lenta, se aumenta la tasa de aprendizaje.



# En la práctica



## Tasa de aprendizaje [Hinton et al.] usando mini-batch learning

Al final del entrenamiento, casi siempre ayuda reducir la tasa de aprendizaje (se eliminan fluctuaciones en los pesos finales debidas a variaciones entre mini-lotes).

La tasa de aprendizaje se reduce si la estimación del error cometido deja de disminuir (estimación obtenida de un conjunto de validación distinto al de entrenamiento).

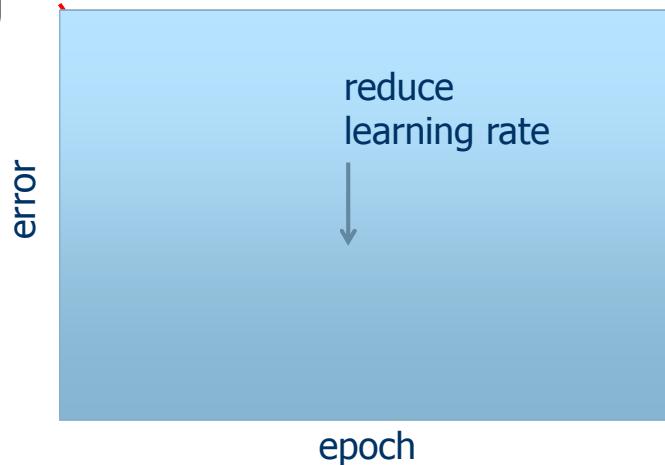


# En la práctica



**Tasa de aprendizaje** [Hinton et al.]  
usando mini-batch learning

¡Cuidado al reducir  
la tasa de aprendizaje!



Reducir la tasa de aprendizaje reduce fluctuaciones aleatorias debidas a los distintos gradientes de los distintos mini-lotes, pero hace el aprendizaje más lento.



# En la práctica



**Tasa de aprendizaje** [Hinton et al.]

PROBLEMA FRECUENTE

Si comenzamos con una tasa de aprendizaje demasiado elevada, los pesos de las neuronas ocultas adquirirán valores positivos muy altos o negativos muy bajos.

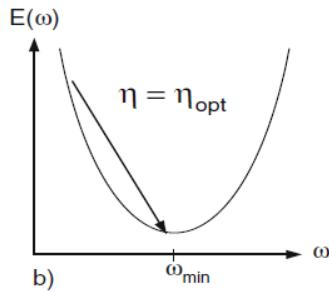
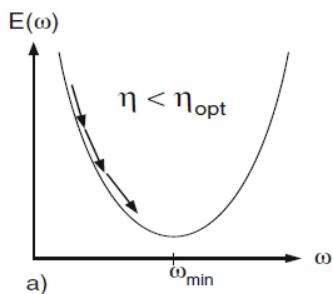
- Las derivadas del error para las neuronas ocultas serán minúsculas y el error no disminuirá.
- La saturación de las neuronas ocultas hace que nos quedemos estancados en una meseta [plateau], que se suele confundir con un mínimo local.



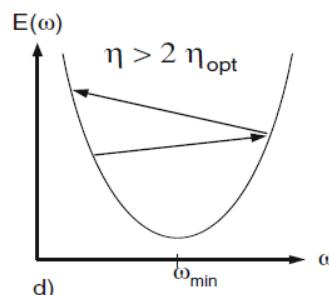
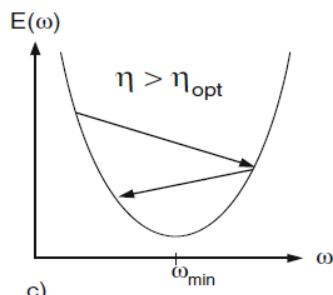
# En la práctica: Optimización



## TEORÍA: CONVERGENCIA DEL GRADIENTE DESCENDIENTE



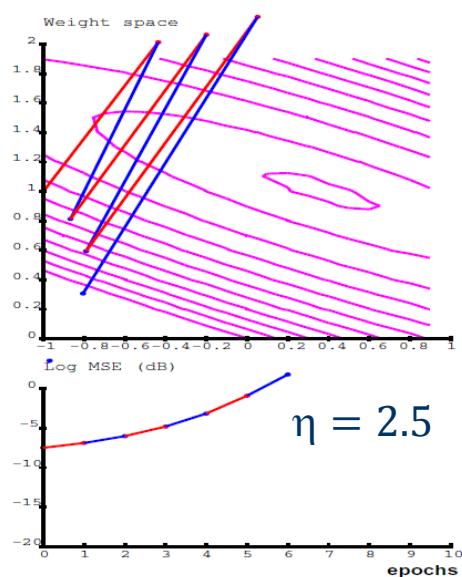
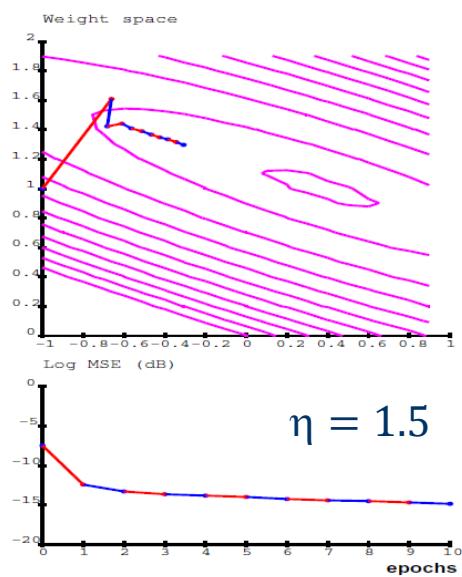
$$W(t+1) = W(t) - \eta \frac{dE(W)}{dW}$$



# En la práctica: Optimización



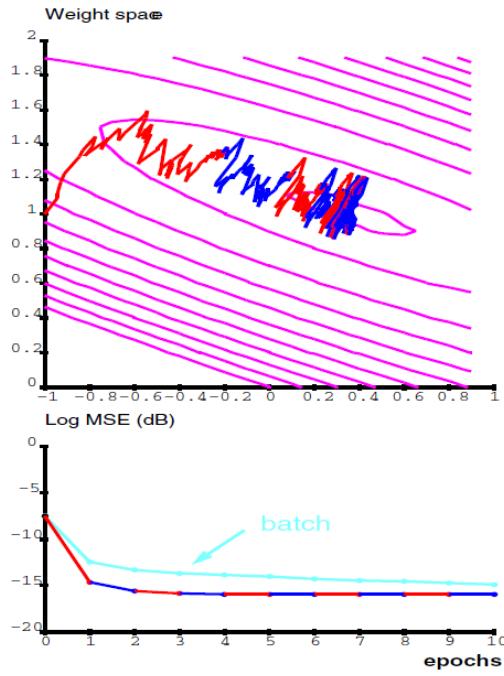
## EJEMPLO: BATCH LEARNING



# En la práctica: Optimización



## STOCHASTIC/ONLINE VS. BATCH LEARNING



# En la práctica: Optimización



## IDEA: Aprendizaje por perturbación de los pesos (p.ej. algoritmos evolutivos)

Modificamos aleatoriamente los pesos  
y comprobamos si la perturbación mejora la red:  
**Demasiado ineficiente.**

Una alternativa menos mala:  
Modificamos las actividades de las neuronas ocultas  
(una vez que sabemos cómo queremos modificar la  
actividad de una neurona, podemos calcular cómo  
cambiar los pesos).

Hay menos actividades que pesos, pero  
backpropagation sigue siendo más eficiente.



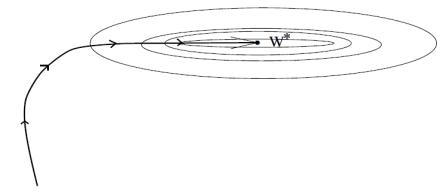
# En la práctica: Optimización



## Implementación de backpropagation

Para acelerar la convergencia, se pueden usar:

- Momentos:  $\Delta w(t + 1) = \eta \frac{\partial E_{t+1}}{\partial w} + \mu \Delta w(t)$
- Tasas de aprendizaje adaptativas.
- rprop & rmsprop
- Técnicas de optimización que tengan en cuenta la curvatura del error (técnicas de segundo orden)



# En la práctica: Optimización



## Momentos

$$\Delta w(t + 1) = \eta \frac{\partial E_{t+1}}{\partial w} + \mu \Delta w(t)$$

- En vez de utilizar el gradiente para modificar la “posición” del vector de pesos, lo utilizamos para modificar su “velocidad”.
- Su inercia/momento  $\mu$  hace que tienda a seguir moviéndose en la misma dirección.



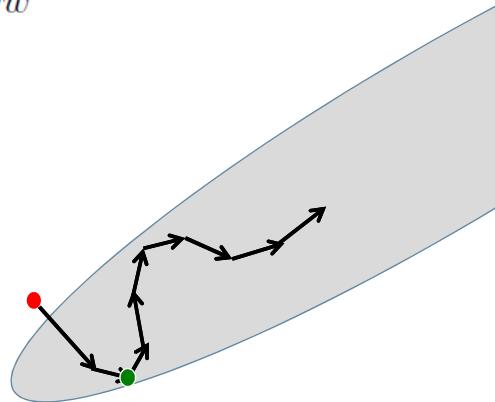
# En la práctica: Optimización



## Momentos

$$\Delta w(t+1) = \eta \frac{\partial E_{t+1}}{\partial w} + \mu \Delta w(t)$$

- Se amortiguan oscilaciones en direcciones de alta curvatura combinando gradientes de signo contrario.
- Se aumenta la velocidad en direcciones con un gradiente pequeño pero consistente.



# En la práctica: Optimización



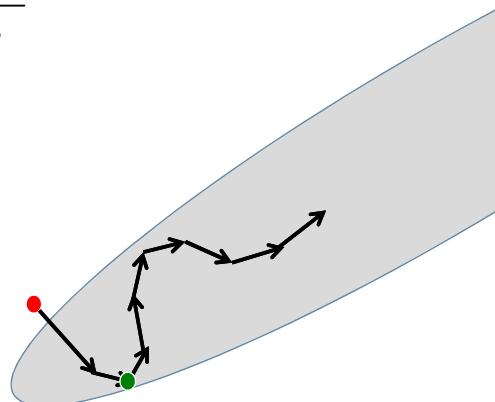
## Momentos

Velocidad:  $v(t) = \mu v(t-1) - \eta \frac{\partial E(t)}{\partial w}$

- Momento  $\mu \leq 1$

Cambio de los pesos = Velocidad:

$$\begin{aligned}\Delta w(t) &= v(t) \\ &= \mu v(t-1) - \eta \frac{\partial E(t)}{\partial w} \\ &= \mu \Delta w(t-1) - \eta \frac{\partial E(t)}{\partial w}\end{aligned}$$



# En la práctica: Optimización



## Momentos

- Si la superficie de error es un plano inclinado, se alcanza una velocidad terminal (mucho más rápido que el gradiente descendente):

$$\mathbf{v}(\infty) = \frac{1}{1-\mu} \left( -\eta \frac{\partial E}{\partial \mathbf{w}} \right)$$

- Al comienzo, los gradientes pueden ser elevados, por lo que se empieza con un momento pequeño (0.5).
- Cuando desaparecen los grandes gradientes (y los pesos se estancan), podemos ir aumentando el momento hasta su valor final (0.9 o incluso 0.99).



# En la práctica: Optimización



## Momentos

- El uso de momentos nos permite aprender con tasas que causarían oscilaciones divergentes en el gradiente descendente.
- El método estándar primero calcula el gradiente en la posición actual y luego da un salto en la dirección del gradiente acumulado.



# En la práctica: Optimización



## Momentos: Método de Nesterov

Versión mejorada basada en el método de optimización de funciones convexas de Nesterov:

- Primero se da un salto en la dirección del gradiente acumulado previamente.
- Luego se mide el gradiente en la posición a la que se llega tras el salto y se hace una corrección:

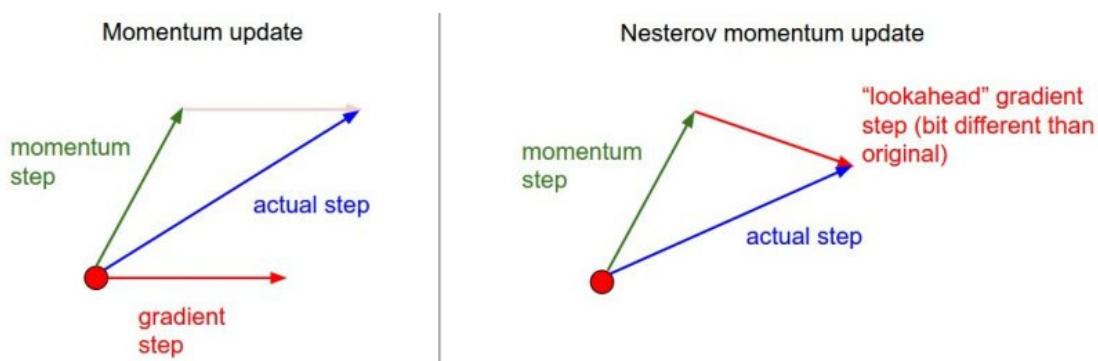
¡Mejor corregir el error después de cometerlo!



# En la práctica: Optimización



## Momentos: Método de Nesterov



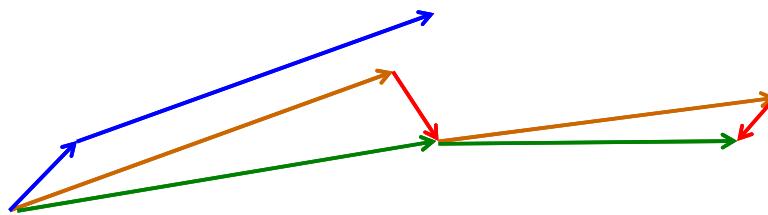
Momento estándar vs. Método de Nesterov



# En la práctica: Optimización



## Momentos: Método de Nesterov



Momento estándar vs. Método de Nesterov

Vector marrón = salto

Vector rojo = corrección

Vector verde = gradiente acumulado

Vector azul = Momento estándar



# En la práctica: Optimización



## Tasas de aprendizaje adaptativas

- Uso de tasas de aprendizaje distintas para cada uno de los parámetros de la red.
- Se puede ajustar la tasa de aprendizaje de cada parámetro de la red (peso) en función de la consistencia del gradiente para ese parámetro.



# En la práctica: Optimización



## Tasas de aprendizaje adaptativas

En una red multicapa,  
la tasa de aprendizaje más adecuada puede variar:

- Las magnitudes de los gradientes son muy diferentes para las distintas capas de la red.
- El “fan-in” de cada nodo determina el efecto causado por el cambio simultáneo de los pesos de entrada (para corregir un mismo error).

Solución: Uso de una tasa de aprendizaje global (fijada manualmente) multiplicada por una ganancia local que se determina empíricamente para cada peso.



# En la práctica: Optimización



## Tasas de aprendizaje adaptativas

Una forma de hacerlo [Hinton et al.]

- Inicialmente, la ganancia local es 1 para todos los pesos.

$$\Delta w_{ij} = -\eta g_{ij} \frac{\partial E}{\partial w_{ij}}$$

- Se incrementa la ganancia local si el gradiente para ese peso no cambia de signo, se disminuye si lo hace.

$$if \left( \frac{\partial E}{\partial w_{ij}}(t) \frac{\partial E}{\partial w_{ij}}(t-1) \right) > 0 \\ then g_{ij}(t) = g_{ij}(t-1) + 0.05 \\ else g_{ij}(t) = g_{ij}(t-1) * 0.95$$



# En la práctica: Optimización



## Tasas de aprendizaje adaptativas

Una forma de hacerlo [Hinton et al.]

Aumentos aditivos,  
descensos multiplicativos.

$$\Delta w_{ij} = -\eta g_{ij} \frac{\partial E}{\partial w_{ij}}$$

- Las ganancias elevadas caen rápidamente cuando si se producen oscilaciones.
- Si el gradiente es totalmente aleatorio, la ganancia se mantendrá en torno a 1 si sumamos  $+\delta$  la mitad de las veces y multiplicamos por  $(1-\delta)$  la otra mitad.

$$if \left( \frac{\partial E}{\partial w_{ij}}(t) \frac{\partial E}{\partial w_{ij}}(t-1) \right) > 0$$

$$then g_{ij}(t) = g_{ij}(t-1) + 0.05$$

$$else g_{ij}(t) = g_{ij}(t-1) * 0.95$$



# En la práctica: Optimización



## Tasas de aprendizaje adaptativas

Otros trucos para mejorar su funcionamiento:

- Limitar las ganancias para que siempre se mantengan en un rango razonable (p.ej.  $[0.1, 10]$  ó  $[0.01, 100]$ ).
- Utilizar aprendizaje por lotes o mini-lotes (se reducen los cambios en el signo del gradiente debidos al error de muestreo propio del aprendizaje “online”).
- Incorporar momentos a las tasas de aprendizaje adaptativas (p.ej. concordancia de signo entre el gradiente de un peso y su “velocidad”).



# En la práctica: Optimización



## rprop [resilient backpropagation]

Utiliza sólo el signo del gradiente.

¿Por qué? La magnitud del gradiente puede ser muy diferente para distintos pesos e ir variando a lo largo del aprendizaje, lo que hace difícil escoger una tasa de aprendizaje global.

En "full-batch learning", podemos eliminar esa variabilidad usando sólo el signo del gradiente:

- Todas las actualizaciones de pesos serán de la misma magnitud.
- Facilita escapar de mesetas con pequeños gradientes.



# En la práctica: Optimización



## rprop [resilient backpropagation]

Utiliza sólo el signo del gradiente...

... y la idea de las tasas de aprendizaje adaptativas:

- Se multiplica por  $\eta^+ > 1$  si no cambia el signo de los dos últimos gradientes (p.ej. 1.2).
- Se multiplica por  $\eta^- < 1$  si cambia el signo de los dos últimos gradientes (p.ej. 0.5).

RECOMENDACIÓN [Mike Shuster]:

Limitar el tamaño de los cambios,  $10^{-6} < \Delta w < 50$ .



# En la práctica: Optimización



**rprop [resilient backpropagation]**  
no funciona con mini-lotes [mini-batches]

Con el gradiente descendente estocástico,  
se promedian los gradientes entre distintos mini-lotes  
(cuando la tasa de aprendizaje es pequeña):

- Si, para un peso, el gradiente es +0.1 en 9 mini-lotes y -0.9 en uno, queremos que el peso no varíe mucho.
- rprop aumentaría el peso nueve veces y lo disminuiría sólo una, de forma que el peso tendería a crecer :-(



# En la práctica: Optimización



## **rmsprop**

Versión “mini-batch” de rprop

- Se divide la tasa de aprendizaje de cada peso por una media móvil de las magnitudes de los gradientes recientes para ese peso.
- Combina la robustez de rprop con la eficiencia de los mini-lotes [mini-batches] y promedia los gradientes de los distintos mini-lotes.



# En la práctica: Optimización



## rmsprop

Versión “mini-batch” de rprop

- rprop es equivalente a usar el gradiente dividiendo ese gradiente por su magnitud.
- rmsprop mantiene una media móvil del gradiente al cuadrado para cada peso de la red neuronal:

$$\text{MeanSquare}(w, t) = 0.9\text{MeanSquare}(w, t-1) + 0.1 \left( \frac{\partial E}{\partial w}(t) \right)^2$$

- rmsprop divide el gradiente por  $\sqrt{\text{MeanSquare}(w, t)}$



# En la práctica: Optimización



## vSGD

**[variance-based Stochastic Gradient Descent]**

Yann LeCun et al.:

“No More Pesky Learning Rates”

ICML’2013

Algoritmo alternativo para ir ajustando las tasas de aprendizaje, un factor crítico en el rendimiento del gradiente descendente estocástico [SGD].

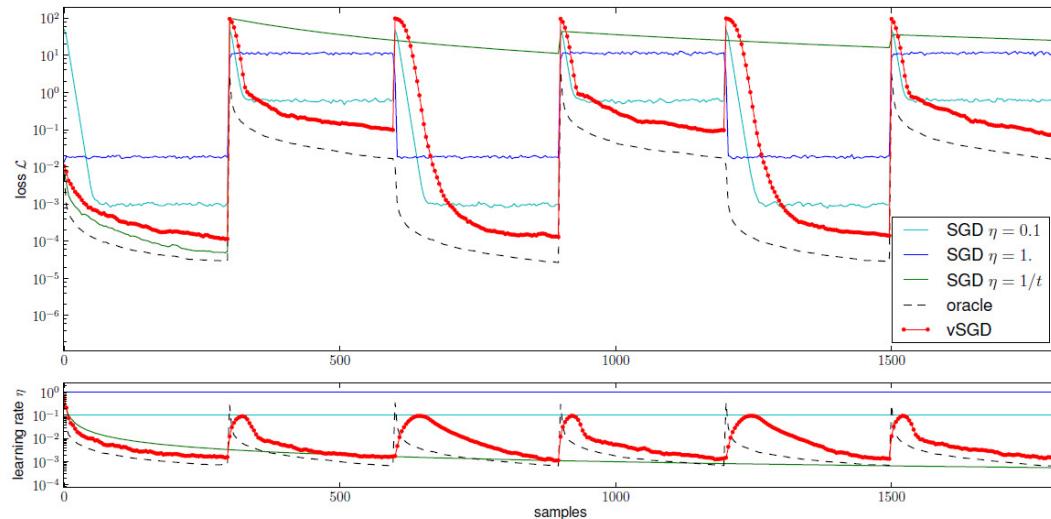


# En la práctica: Optimización



## "No More Pesky Learning Rates"

[LeCun et al., ICML'2013]

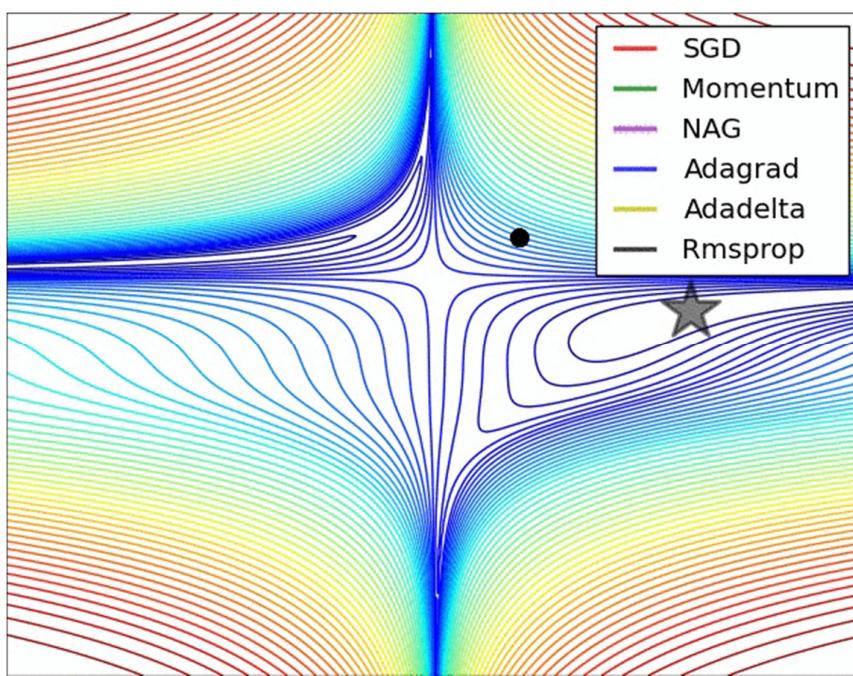


126

# En la práctica: Optimización



## SGD [Stochastic Gradient Descent]



Alec Radford  
<https://twitter.com/alecrad>

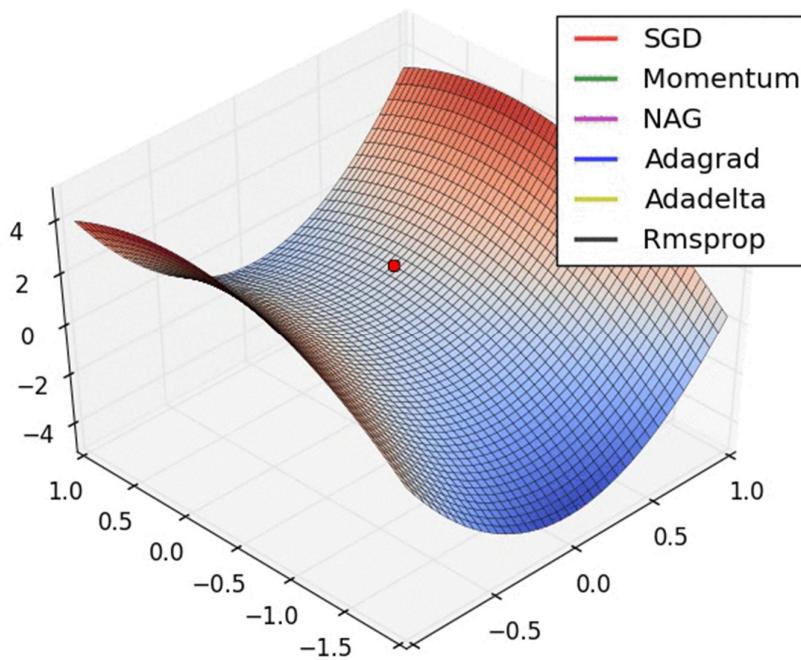


127

# En la práctica: Optimización



SGD [Stochastic Gradient Descent] @ saddle point



Alec Radford  
<https://twitter.com/alecrad>

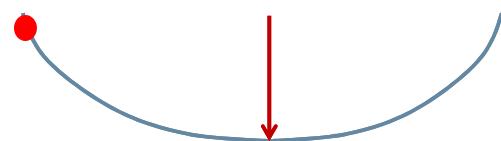


# En la práctica: Optimización

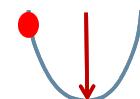


## Técnicas de optimización de segundo orden

¿Cuánto podemos reducir el error moviéndonos en una dirección?



Asumiendo una curvatura constante en la superficie de error (i.e. superficie cuadrática), la reducción máxima del error dependerá de la relación entre el gradiente y la curvatura.

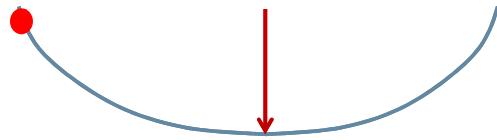


# En la práctica: Optimización

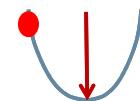


## Técnicas de optimización de segundo orden

¿En qué dirección deberíamos movernos?



Una buena dirección puede ser aquélla en la que la relación gradiente/curvatura sea elevada, aunque el gradiente en sí sea pequeño.



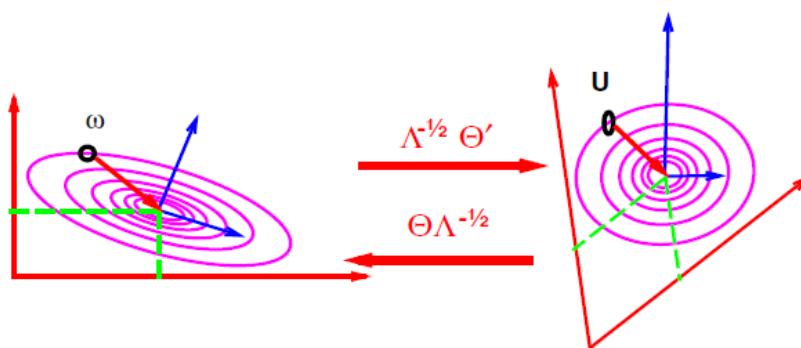
# En la práctica: Optimización



## Técnicas de optimización de segundo orden

Método de Newton

$$\Delta w = \eta \left( \frac{\partial^2 E}{\partial w^2} \right)^{-1} \frac{\partial E}{\partial w} = \eta H(w)^{-1} \frac{\partial E}{\partial w}$$



Newton Algorithm here .....

....is like Gradient Descent there



# En la práctica: Optimización



## Técnicas de optimización de segundo orden

Método de Newton

$$\Delta w = \eta \left( \frac{\partial^2 E}{\partial w^2} \right)^{-1} \frac{\partial E}{\partial w} = \eta H(w)^{-1} \frac{\partial E}{\partial w}$$

El método de Newton multiplica el gradiente por la inversa de la matriz de curvatura / matriz Hessiana  $H^{-1}$

Esta operación transforma elipsoides en esferas, por lo que si la superficie fuese realmente cuadrática, llegaríamos al mínimo en un solo paso.



# En la práctica: Optimización



## Técnicas de optimización de segundo orden

Método de Newton

$$\Delta w = \eta \left( \frac{\partial^2 E}{\partial w^2} \right)^{-1} \frac{\partial E}{\partial w} = \eta H(w)^{-1} \frac{\partial E}{\partial w}$$

**Inconveniente:** Hace falta almacenar e invertir una matriz Hessiana de tamaño NxN, lo que requiere  $O(N^3)$  por iteración, lo que no resulta práctico.

Además, si la función de error no es cuadrática, ni siquiera tenemos garantías de convergencia :-)



# En la práctica: Optimización



## Técnicas de optimización de segundo orden

“Hessian-free optimization”

- Cada elemento de la matriz de curvatura especifica cómo cambia el gradiente en una dirección conforme nos movemos en otra dirección.
- La matriz de curvatura tiene demasiados términos para que la usemos en la práctica, pero podemos aproximarla de distintas formas,
  - p.ej. Gradientes conjugados
  - L-BFGS



# En la práctica: Optimización



## Técnicas de optimización de segundo orden

“Hessian-free optimization”

- Aproximamos la matriz de curvatura y, asumiendo que la aproximación es correcta, minimizamos el error utilizando una técnica eficiente (gradientes conjugados).
- Realizamos otra aproximación y volvemos a minimizar...



# En la práctica: Optimización

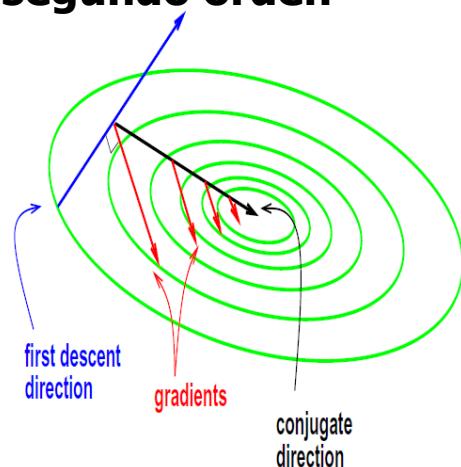


## Técnicas de optimización de segundo orden

### Gradientes conjugados

#### IDEA

Utilizamos una secuencia de pasos, cada uno de los cuales encuentra el mínimo en una dirección.



Para no deshacer la minimización que ya hayamos conseguido, nos aseguramos de que cada dirección sea “conjugada” con respecto a las direcciones previas.

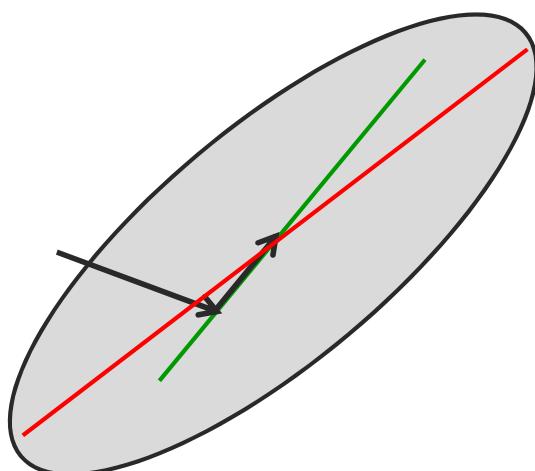


# En la práctica: Optimización



## Técnicas de optimización de segundo orden

### Gradientes conjugados



El gradiente en la dirección del primer paso es 0 en todos los puntos de la línea verde, por lo que podemos movernos a lo largo de la línea verde sin afectar la minimización realizada en la primera dirección.



# En la práctica: Optimización



## Técnicas de optimización de segundo orden

### Gradientes conjugados

- Después de N pasos, el gradiente conjugado garantiza encontrar el óptimo en una superficie cuadrática N-dimensional.

NOTA: En muchos menos de N pasos, estaremos muy cerca del óptimo

- Los optimizadores “Hessian-free” realizan una aproximación cuadrática de la superficie de error y usan gradientes conjugados para minimizar.

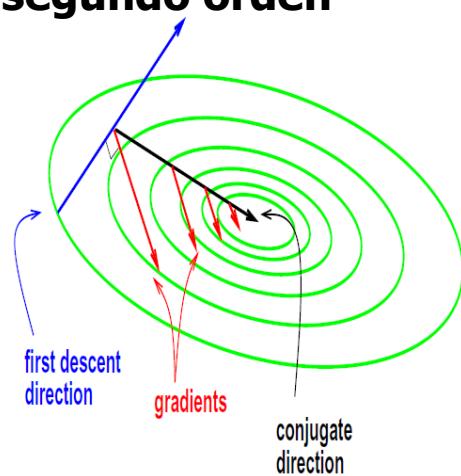


# En la práctica: Optimización



## Técnicas de optimización de segundo orden

### Gradientes conjugados



- Algoritmo lineal,  $O(N)$ .
- No usa la matriz Hessiana de forma explícita.

NOTA:

Sólo funciona con aprendizaje por lotes [batch learning].



# En la práctica: Optimización



## Técnicas de optimización de segundo orden

### Gradientes conjugados

TABLE 4.3 Summary of the Nonlinear Conjugate-Gradient Algorithm for the Supervised Training of a Multilayer Perceptron

#### Initialization

Unless prior knowledge on the weight vector  $\mathbf{w}$  is available, choose the initial value  $\mathbf{w}(0)$  by using a procedure similar to that described for the back-propagation algorithm.

#### Computation

1. For  $\mathbf{w}(0)$ , use back propagation to compute the gradient vector  $\mathbf{g}(0)$ .
2. Set  $\mathbf{s}(0) = \mathbf{r}(0) = -\mathbf{g}(0)$ .
3. At time-step  $n$ , use a line search to find  $\eta(n)$  that minimizes  $\mathcal{L}_{\text{av}}(\eta)$  sufficiently, representing the cost function  $\mathcal{L}_{\text{av}}$ , expressed as a function of  $\eta$  for fixed values of  $\mathbf{w}$  and  $\mathbf{x}$ .
4. Test to determine whether the Euclidean norm of the residual  $\mathbf{r}(n)$  has fallen below a specified value, that is, a small fraction of the initial value  $\|\mathbf{r}(0)\|$ .
5. Update the weight vector:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\mathbf{s}(n)$$

6. For  $\mathbf{w}(n+1)$ , use back propagation to compute the updated gradient vector  $\mathbf{g}(n+1)$ .

7. Set  $\mathbf{r}(n+1) = -\mathbf{g}(n+1)$ .

8. Use the Polak–Ribièvre method to calculate:

$$\beta(n+1) = \max\left\{\frac{\mathbf{r}^T(n+1)(\mathbf{r}(n+1) - \mathbf{r}(n))}{\mathbf{r}^T(n)\mathbf{r}(n)}, 0\right\}$$

9. Update the direction vector:

$$\mathbf{s}(n+1) = \mathbf{r}(n+1) + \beta(n+1)\mathbf{s}(n)$$

10. Set  $n = n + 1$ , and go back to step 3.

*Stopping criterion.* Terminate the algorithm when the condition

$$\|\mathbf{r}(n)\| \leq \epsilon \|\mathbf{r}(0)\|$$

is satisfied, where  $\epsilon$  is a prescribed small number.

[Haykin: "Neural Networks and Learning Machines", 3rd edition]



# En la práctica: Optimización



## Técnicas de optimización de segundo orden

Métodos quasi-Newton, p.ej. BFGS

[Broyden–Fletcher–Goldfarb–Shanno algorithm]

- Algoritmo cuadrático,  $O(N^2)$ .
- Estima iterativamente la inversa de la matriz Hessiana.

**Limited-Memory BFGS** [L-BFGS] aproxima el algoritmo BFGS utilizando una cantidad de memoria lineal, por lo que se utiliza a menudo (p.ej. MATLAB)  
[https://en.wikipedia.org/wiki/Limited-memory\\_BFGS](https://en.wikipedia.org/wiki/Limited-memory_BFGS)

NOTA: Como los métodos anteriores, sólo se puede utilizar en aprendizaje por lotes [batch learning].



# En la práctica: Optimización



## Resumen

No existe una receta simple, pero sí recomendaciones:

Para conjuntos de datos pequeños (~10,000 ejemplos):

- Aprendizaje por lotes [full-batch learning].
- Gradientes conjugados o L-BFGS.
- Tasas de aprendizaje adaptativas o rprop.

Para conjuntos de datos grandes:

- Aprendizaje por mini-lotes [mini-batch learning].
- rmsprop [Hinton] o vSGD [LeCun]



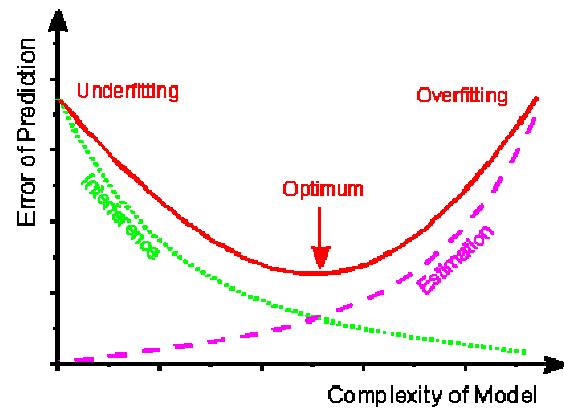
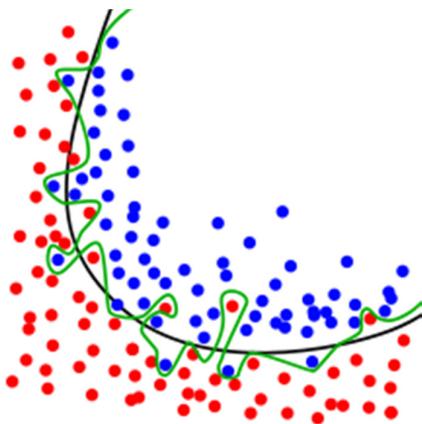
# En la práctica: Generalización



- El conjunto de entrenamiento contiene patrones útiles, pero también ruido: regularidades accidentales debidas al conjunto de datos particular utilizado (error de muestreo).
- Cuando ajustamos la red a los datos del conjunto de entrenamiento, no podemos diferenciar las regularidades reales de las debidas al conjunto de entrenamiento utilizado, por lo que corremos el riesgo de “sobreaprender” [overfitting].



# En la práctica: Generalización



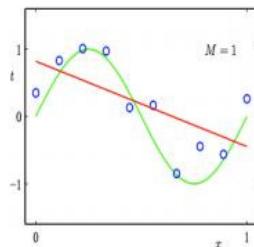
## Sobreaprendizaje [overfitting]



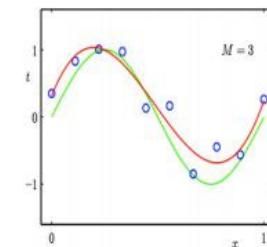
# En la práctica: Generalización



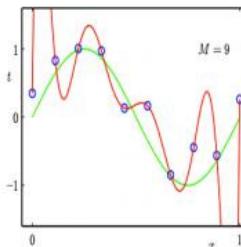
Regression:



predictor too inflexible:  
cannot capture pattern



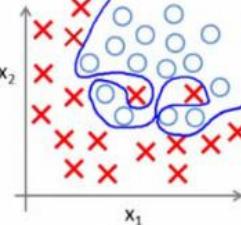
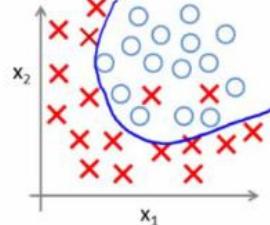
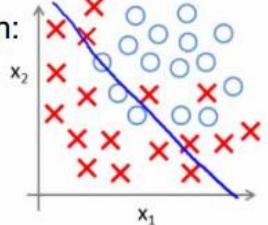
M = 3



M = 9

predictor too flexible:  
fits noise in the data

Classification:



## Underfitting & overfitting



# En la práctica: Generalización



Estrategias para evitar el sobreaprendizaje:

- Obtener más datos (la mejor opción si tenemos capacidad para entrenar la red usando más datos).
- Ajustar los parámetros de la red para que tenga la capacidad adecuada (suficiente para identificar las regularidades en los datos, pero no demasiada para ajustarse a las espúreas, suponiendo que sean más débiles que las auténticas).



# En la práctica: Generalización



Una tercera estrategia: Combinar modelos

- **"Model averaging"** (a.k.a. "ensembles"):  
Muchos modelos diferentes con distintos parámetros o el mismo tipo de modelo utilizando distintos subconjuntos del conjunto de entrenamiento [bagging].
- **"Bayesian fitting"** (enfoque bayesiano):  
Utilizando una única arquitectura de red, se combinan las predicciones realizadas por muchos vectores de pesos diferentes.



# En la práctica: Generalización



## Capacidad de la red: Topología

Algunas formas de limitar la capacidad de la red actuando sobre su topología:

- **Arquitectura de la red:**

Se limita el número de capas ocultas y/o el número de unidades por capa.

- **Weight sharing:**

Se reduce el número de parámetros de la red haciendo que distintas neuronas comparten los mismos pesos (p.ej. redes convolutivas).



# En la práctica: Generalización



## Capacidad de la red: Entrenamiento

Algunas formas de limitar la capacidad de la red actuando sobre su algoritmo de entrenamiento:

- **Early stopping:** Se comienza a entrenar la red con pesos pequeños y se para el entrenamiento antes de que sobreaprenda.

- **Weight decay:** Se penalizan los pesos grandes en función de sus valores al cuadrado (penalización L2) o absolutos (penalización L1).

- **Ruido:** Se añade ruido a los pesos o actividades de las neuronas de la red que se está entrenando.



# En la práctica: Generalización



En la práctica, se pueden combinar varias técnicas concretas para prevenir el sobreaprendizaje:

- Weight sharing
- Weight decay
- Ruido (sobre pesos, entradas o actividades)
- Early stopping
- Dropout
- Generative pre-training → Deep Learning

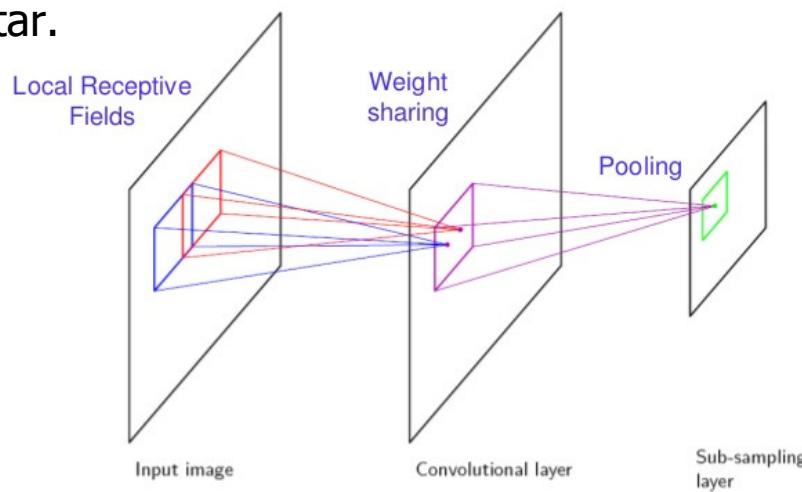


# En la práctica: Generalización



## Weight sharing

Se usan los mismos pesos para diferentes neuronas, reduciendo así el número de parámetros que hay que ajustar.



Ejemplo: Redes convolutivas



# En la práctica: Generalización



## Weight decay

Incorporación de una penalización en la función de coste (como la regularización en las técnicas de regresión)

$$\begin{aligned} E &= \frac{1}{2} \sum_i \|t_i - y_i\|^2 = \frac{1}{2} \sum_i \sum_j (t_{ij} - y_{ij})^2 \\ \downarrow \\ J &= \frac{1}{2m} \sum_i \|t_i - y_i\|^2 + \frac{\lambda}{2} \sum_l \sum_i \sum_j (w_{ij}^{(l)})^2 \end{aligned}$$

donde  $m$  es el número de ejemplos del conjunto de entrenamiento y  $\lambda$  un factor de regularización.



# En la práctica: Generalización



## Weight decay

La penalización mantiene los pesos pequeños, evitando que la red utilice pesos que no necesite realmente.

$$J = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial J}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

$$\text{cuando } \frac{\partial J}{\partial w_i} = 0, \quad w_i = -\frac{1}{\lambda} \frac{\partial E}{\partial w_i}$$

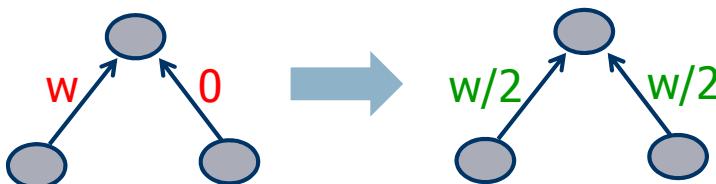


# En la práctica: Generalización



## Weight decay

- Se reduce el sobreaprendizaje evitando que la red se ajuste demasiado al conjunto de entrenamiento.
- Se consigue un modelo más “suave” en el que las salidas cambian más lentamente cuando cambian las entradas.



Si la red tiene dos entradas similares, preferirá poner la mitad del peso en cada una, en vez de todo en una sola.

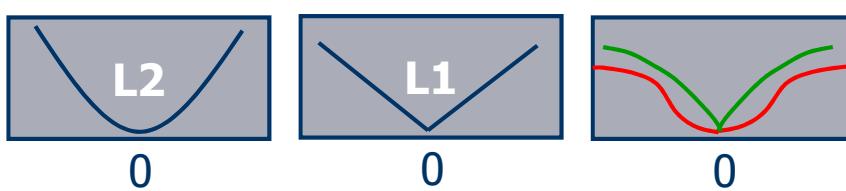


# En la práctica: Generalización



## Weight decay: Tipos de penalización

- La penalización estándar añade un término que incluye los pesos al cuadrado (regularización L2).
- En ocasiones, funciona mejor penalizar usando el valor absoluto de los pesos (regularización L1), con lo que muchos pesos acaban siendo exactamente 0.
- En otras ocasiones, puede ser recomendable utilizar una penalización que no tenga apenas efecto sobre pesos grandes (para permitir algunos pesos grandes en la red).



# En la práctica: Generalización



## Restricciones sobre los pesos

En vez de incluir una penalización sobre los pesos de la red en la función de coste, podemos establecer directamente restricciones sobre los pesos.

p.ej. Establecemos un máximo sobre la magnitud del vector de pesos de entrada de cada neurona (si una actualización viola esta restricción, el vector de pesos se reescala).



# En la práctica: Generalización



## Restricciones sobre los pesos

Ventajas de las restricciones sobre las penalizaciones:

- Obtienen valores razonables para los pesos.
- Evitan que la actividad de las neuronas ocultas se quede atascada en torno a 0.
- Evitan que los pesos “exploten”.

Suelen ser más efectivas que una penalización fija a la hora de conseguir que los pesos irrelevantes tiendan hacia cero.



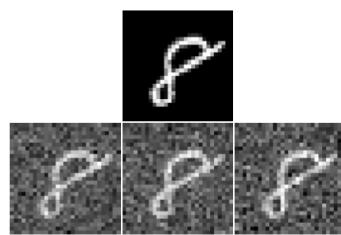
# En la práctica: Generalización



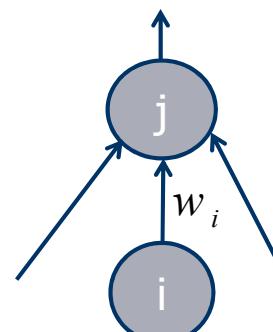
## Ruido

Regularización L2 añadiendo ruido a las entradas

Si añadimos ruido gaussiano a las entradas de una neurona lineal, la varianza del ruido se amplifica antes de llegar a la siguiente capa:



$$y_j + N(0, w_i^2 \sigma_i^2)$$



$$x_i + N(0, \sigma_i^2)$$



# En la práctica: Generalización



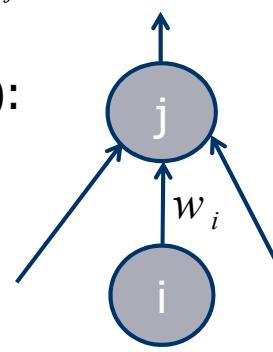
## Ruido

Regularización L2 añadiendo ruido a las entradas

El ruido amplificado se añade a la salida (realiza una contribución al error):

Minimizar el error cuando se introduce ruido en las entradas regulariza los pesos.

$$y_j + N(0, w_i^2 \sigma_i^2)$$



$$x_i + N(0, \sigma_i^2)$$



# En la práctica: Generalización



## Ruido

Regularización L2 añadiendo ruido a las entradas

- Salida:  $y^{noisy} = \sum_i w_i x_i + \sum_i w_i \varepsilon_i$  donde  $\varepsilon_i$  se muestrea de  $N(0, \sigma_i^2)$

- Error: 
$$\begin{aligned} E[(y^{noisy} - t)^2] &= E\left[\left(y + \sum_i w_i \varepsilon_i - t\right)^2\right] = E\left[\left((y - t) + \sum_i w_i \varepsilon_i\right)^2\right] \\ &= (y - t)^2 + E\left[2(y - t)\sum_i w_i \varepsilon_i\right] + E\left[\left(\sum_i w_i \varepsilon_i\right)^2\right] \\ &= (y - t)^2 + E\left[\sum_i w_i^2 \varepsilon_i^2\right] \\ &= (y - t)^2 + \sum_i w_i^2 \sigma_i^2 \end{aligned}$$

El error gaussiano  
en las entradas  
equivale a una  
penalización L2



# En la práctica: Generalización



## Ruido

Ruido sobre los pesos

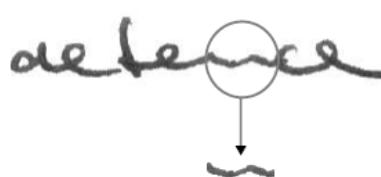
Añadir ruido a los pesos de una red neuronal multicapa no lineal también puede ayudar (aunque no sea exactamente equivalente a una penalización L2 sobre los pesos).

### EJEMPLO

Redes recurrentes  
para el reconocimiento  
de textos manuscritos

Alex Graves:

Supervised Sequence Labelling with Recurrent Neural Networks  
Springer, 2012. ISBN 978-3-642-24797-2



# En la práctica: Generalización



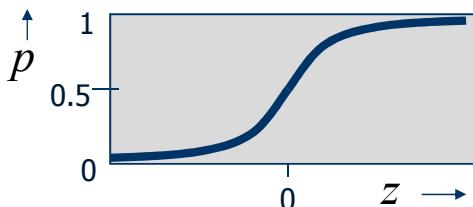
## Ruido

Ruido sobre las actividades

$$p = \frac{1}{1 + e^{-z}}$$

Otra alternativa: Se usa backpropagation para entrenar una red multicapa con unidades logísticas,

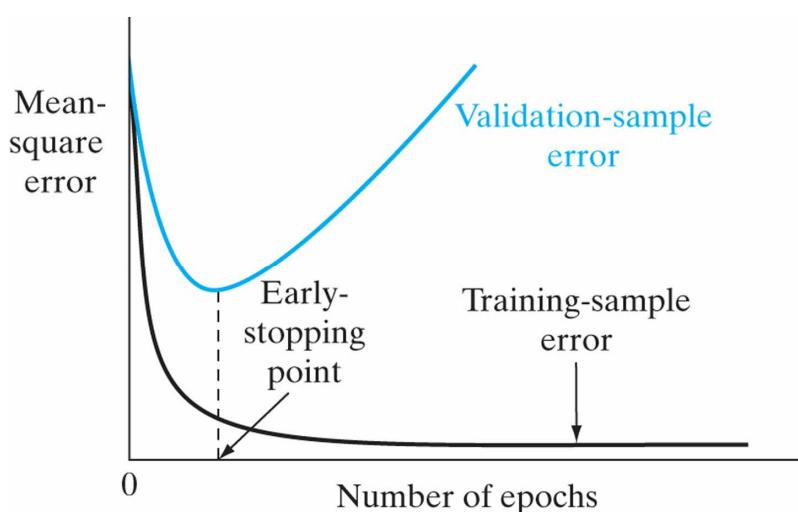
- como si las neuronas fuesen unidades binarias estocásticas en el paso hacia adelante,
- pero se comportan de la forma normal en la propagación de errores hacia atrás



Entrenamiento más lento, con peores resultados sobre el conjunto de entrenamiento pero mejores resultados sobre el conjunto de prueba :-)



# En la práctica: Generalización



## Early stopping

[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# En la práctica: Generalización



## Early stopping

- Puede resultar demasiado costoso probar con diferentes penalizaciones sobre los pesos de la red, por lo que puede ser aconsejable empezar el entrenamiento con pesos muy pequeños y dejar que crezcan hasta que el rendimiento sobre el conjunto de validación comience a empeorar.
- Se limita la capacidad de la red al impedir que los pesos crezcan demasiado.



# En la práctica: Generalización



## Early stopping

¿Por qué funciona?

- Cuando los pesos son muy pequeños, las neuronas de las capas ocultas se mantienen en su rango lineal: Una red con capas ocultas de neuronas lineales no tiene más capacidad que una red lineal sin capas ocultas.
- Conforme crecen los pesos, las neuronas ocultas comienzan a comportarse de forma no lineal (y la capacidad de la red aumenta).



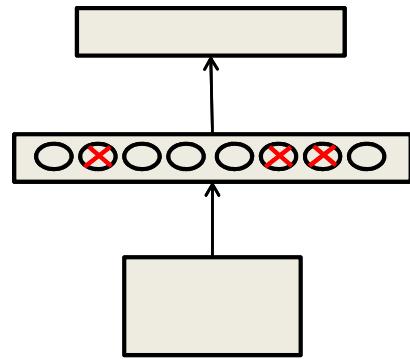
# En la práctica: Generalización



## Dropout

Técnica de regularización

- Para cada caso de entrenamiento, se omite aleatoriamente cada neurona oculta con probabilidad 0.5 (la mitad de las neuronas de las capas ocultas no se utilizan para cada caso del conjunto de entrenamiento).
- Se evita que las neuronas ocultas dependan/confíen demasiado en el trabajo de otras neuronas ocultas de su misma capa.



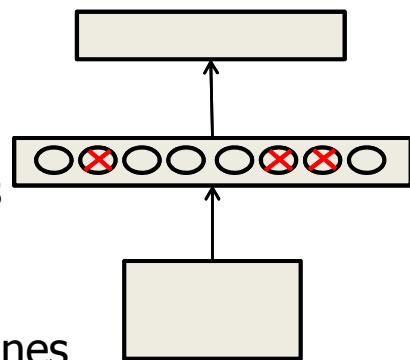
# En la práctica: Generalización



## Dropout

Técnica de regularización

- Si una neurona oculta sabe qué otras neuronas ocultas existen, pueden co-adaptarse sobre el conjunto de entrenamiento, pero las co-adaptaciones no funcionarán bien sobre el conjunto de prueba:  
**las conspiraciones complejas no son robustas!** [Hinton].
- Si tiene que trabajar correctamente con muchos conjuntos de neuronas, es más probable que haga algo que sea individualmente útil.



# En la práctica: Generalización



## Dropout & model averaging

Una forma de combinar múltiples modelos

- Para cada caso de entrenamiento, estamos muestreando de una familia de  $2^H$  arquitecturas diferentes (redes que comparten los mismos pesos).
- Sólo se entrena algunos de los posibles modelos y cada modelo sólo recibe un caso de entrenamiento (versión extrema de bagging).
- Al compartir pesos, se regularizan todos los modelos (mucho mejor que las penalizaciones L2 o L1).



# En la práctica: Generalización



## Dropout

- A la hora de utilizar la red, podríamos muestrear múltiples modelos y calcular la media geométrica de sus salidas...

$$\begin{array}{r} \text{Modelo A: } .3 \quad .2 \quad .5 \\ \text{Modelo B: } .1 \quad .8 \quad .1 \\ \hline \text{Combinado } \sqrt{.03} \sqrt{.16} \sqrt{.05} / \text{sum} \end{array}$$

- ... pero es mejor utilizar todas las neuronas ocultas, dejando en la mitad sus pesos de salida (w/2 equivale a calcular la media geométrica de las predicciones de los  $2^H$  modelos posibles).



# En la práctica: Generalización



## Dropout

¿Y si tenemos varias capas ocultas?

- Usamos la red “promedio” con todos los pesos de salida de las neuronas ocultas divididos por 2. Matemáticamente, no es equivalente a promediar todos los modelos posibles, pero es una buena aproximación (y muy rápida).
- La alternativa es aplicar de forma estocástica el modelo varias veces con la misma entrada (lo que nos permite hacernos una idea de la incertidumbre de la respuesta proporcionada por la red neuronal).



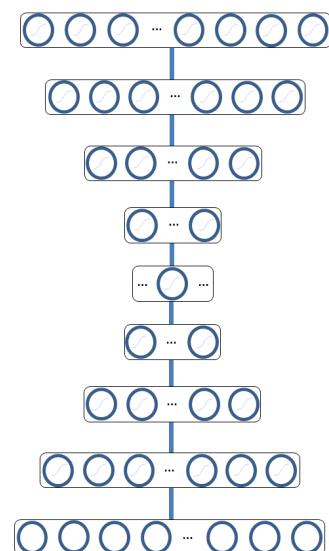
# En la práctica: Generalización



## Dropout

¿Algo que decir sobre la capa de entrada?

Se puede utilizar dropout sobre la capa de entrada, siempre que usemos una probabilidad mayor para mantener las unidades de entrada.



NOTA:

Los “denoising autoencoders” de Bengio et al. utilizan este truco → Deep Learning



# En la práctica: Generalización

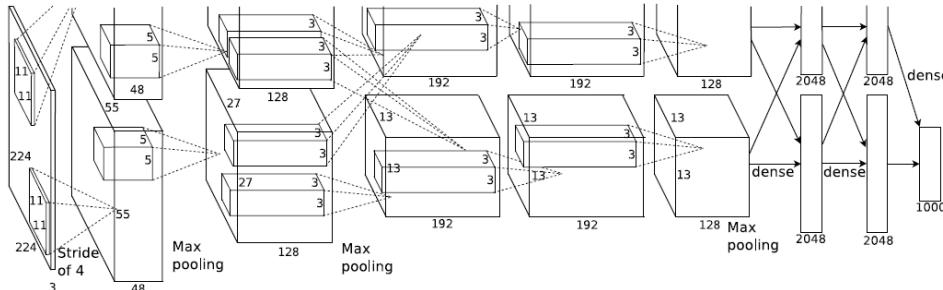


## Dropout

En deep learning, dropout reduce significativamente el error, evitando el sobreaprendizaje.

IMAGENET

Alex Krizhevsky (NIPS 2012)



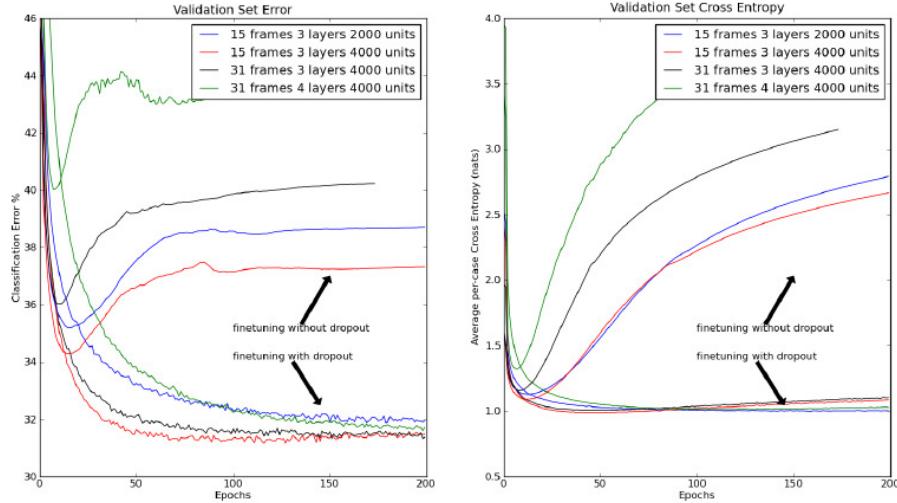
[Hinton] En deep learning, si tu red no sobreaprende, entonces es que deberías usar una red más grande.



## En la práctica: Generalización



## Dropout



Cualquier red que se entrene con “early stopping” puede hacerlo mejor si se utiliza dropout, aunque costará más tiempo entrenarla.



# En la práctica: Generalización



## Hiper-parámetros (o meta-parámetros)

Una de las principales dificultades prácticas del uso de redes neuronales es la destreza que requiere establecer todos sus parámetros ("arte" más que ciencia)

- p.ej. Número de capas
- Número de neuronas por capa
- Tipo de neuronas
- Penalización de los pesos
- Tasas de aprendizaje
- ...



# En la práctica: Generalización



## Hiper-parámetros (o meta-parámetros)

¿Cómo elegir los hiper-parámetros de una red neuronal?

**Método incorrecto:** Se prueban montones de alternativas para ver cuál funciona mejor en el conjunto de test.

- Fácil de hacer, pero nos da una impresión engañosa de lo bien que funcionará la red en la práctica:
- La configuración que funcione mejor sobre el conjunto de prueba puede que no sea la que funcione mejor en otros conjuntos de prueba (o los nuevos casos sobre los que queramos aplicar la red neuronal).



# En la práctica: Generalización



## Hiper-parámetros (o meta-parámetros)

¿Cómo elegir los hiper-parámetros de una red neuronal?

Un método mejor: **Conjunto de validación.**

Se divide el conjunto de datos disponible en tres partes:

- Conjunto de **entrenamiento** (para aprender los parámetros del modelo, i.e. los pesos de la red).
- Conjunto de **validación** (no se utiliza en el entrenamiento, sino para decidir qué hiper-parámetros resultan más adecuados)
- Conjunto de **prueba** (para obtener una estimación no sesgada de lo bien que funciona la red).



# En la práctica: Generalización



## Hiper-parámetros (o meta-parámetros)

¿Cómo elegir los hiper-parámetros de una red neuronal?

## Validación cruzada

- Dividimos el conjunto de datos en N subconjuntos.
- Utilizamos N-1 subconjuntos de conjunto de entrenamiento y el subconjunto restante de conjunto de prueba para obtener N estimaciones del error.



# En la práctica: Generalización



## Hiper-parámetros (o meta-parámetros)

¿Cómo elegir los hiper-parámetros de una red neuronal?

### Aprendizaje automático [Machine Learning]

En vez de probar todas las combinaciones posibles de parámetros, podemos muestrear el espacio de posibles combinaciones.

p.ej. Metaheurísticas (algoritmos genéticos)

Optimización bayesiana (procesos gaussianos)



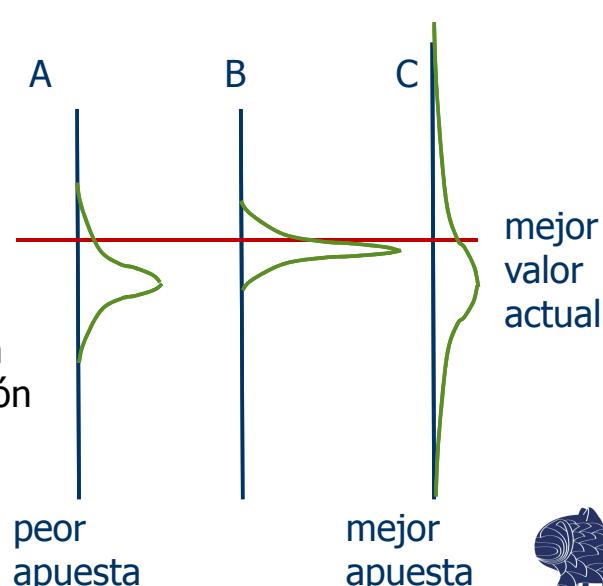
# En la práctica: Generalización



## Hiper-parámetros (o meta-parámetros)

¿Cómo elegir los hiper-parámetros de una red neuronal?

Modelo de procesos gaussianos:  
A partir de la mejor configuración conocida, se elige una combinación de hiperparámetros tal que la mejora esperada sea grande (sin preocuparse por la posibilidad de empeorar).



[Snoek, Larochelle & Adams, NIPS 2012]



# En la práctica: Generalización



## Hiper-parámetros (o meta-parámetros)

¿Cómo elegir los hiper-parámetros de una red neuronal?

## Aprendizaje automático [Machine Learning]

- Mucho mejor que ir haciendo pruebas manualmente (no es el tipo de tarea que los humanos hacemos bien).
- Evita sesgos psicológicos no deseados: método menos propenso a funcionar mejor con el método que nos gusta y peor con el que no (las personas no podemos evitarlo ;-)



# En la práctica: Invarianza



Técnicas [ad hoc] que nos permiten incorporar conocimiento previo en el diseño de una red neuronal:

- Restringir la arquitectura de la red (uso de conexiones locales, a.k.a. "campos receptivos").
- Restringir la selección de pesos de la red (uso de pesos compartidos por varias neuronas, i.e. "weight-sharing").



# En la práctica: Invarianza



## Motivación

El reconocimiento de objetos es difícil por varios motivos:

- Segmentación (varios objetos en la misma imagen).
- Ocultación (partes ocultas detrás de otros objetos).
- Iluminación (valores de los píxeles determinados tanto por la iluminación ambiental como por el objeto al que corresponden).
- Punto de vista (cambios en el punto de vista causan cambios en la imagen que se capta).
- Deformación (los objetos pueden aparecer de varias deformadas de muchas maneras [no afines]),  
p.ej. textos manuscritos.



# En la práctica: Invarianza



## Justificación

Nuestros sistemas visual y auditivo parecen haberse especializado para tratar sin problemas determinadas transformaciones (sin que seamos conscientes de su complejidad real).

0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	9	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9

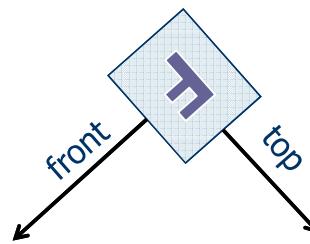


# En la práctica: Invarianza



## Segmentación: Preprocesamiento de los datos

Identificamos una región alrededor de un objeto y la utilizamos como marco de coordenadas de referencia para un conjunto de píxeles normalizado (p.ej. MNIST).



No siempre es fácil:

¡Hay que reconocer el objeto para delimitar su región!



# En la práctica: Invarianza



## Por fuerza bruta

- En la fase de entrenamiento, utilizamos imágenes bien segmentadas y ya normalizadas.
- En la fase de prueba, probamos con distintas regiones (variando escala y orientación).

The photos you uploaded were grouped automatically so you can quickly label and notify friends in these pictures.  
(Friends can always untag themselves.)

Who is this?	Who is this?	Who is this?
Who is this?	Who is this?	Who is this?

## EJEMPLO

Reconocimiento de caras

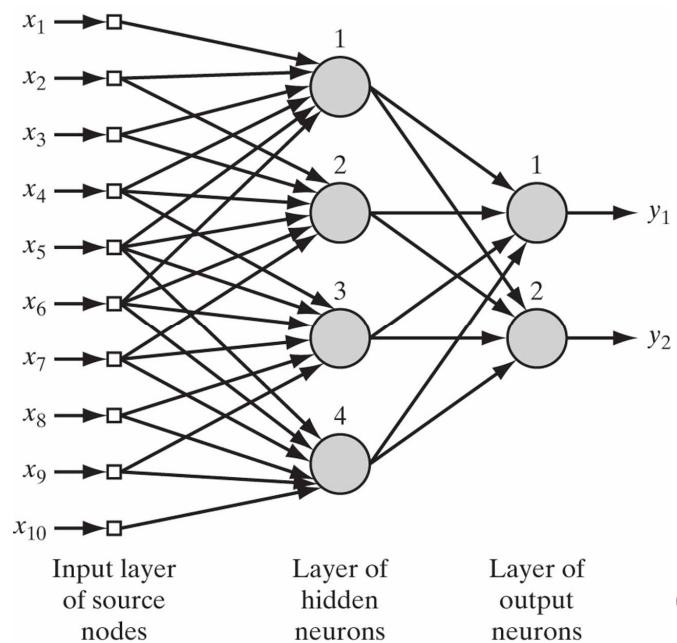


# En la práctica: Invarianza



## Redes convolutivas

Campos receptivos  
y pesos compartidos



[Haykin: "Neural Networks and Learning Machines", 3<sup>rd</sup> edition]



# En la práctica: Invarianza



## Redes convolutivas

- Se usan múltiples copias de los mismos detectores de características en distintas posiciones.
- La replicación reduce el número de parámetros (pesos) que deben ajustarse.
- Se pueden utilizar distintos tipos de detectores (cada uno con su “mapa” de detectores replicados): cada fragmento de la imagen se representa de distintas formas.



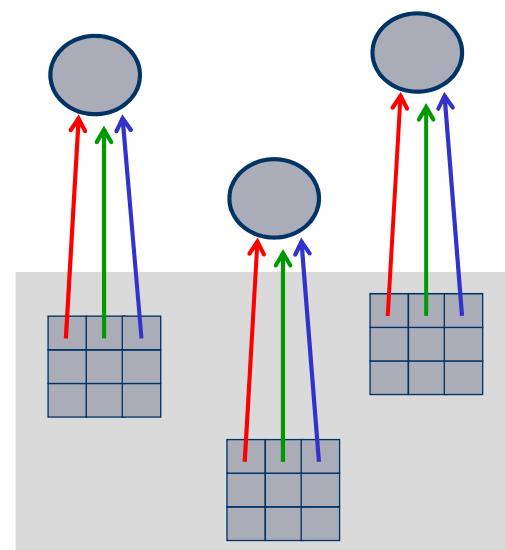
# En la práctica: Invarianza



## Redes convolutivas

Cada unidad detecta una característica en una región diferente de la imagen.

Todas comparten los mismos pesos.



# En la práctica: Invarianza



## Redes convolutivas

1 <small>x1</small>	1 <small>x0</small>	1 <small>x1</small>	0	0
0 <small>x0</small>	1 <small>x1</small>	1 <small>x0</small>	1	0
0 <small>x1</small>	0 <small>x0</small>	1 <small>x1</small>	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved  
Feature

## Convolución

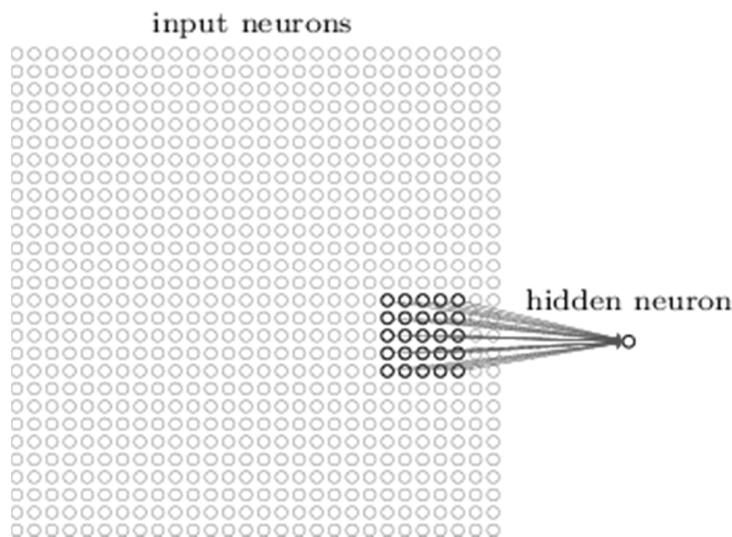
<http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution/>



# En la práctica: Invarianza



## Redes convolutivas



Convolución: "Local receptive fields"

<http://neuralnetworksanddeeplearning.com/>



# En la práctica: Invarianza



## Redes convolutivas

Backpropagation con restricciones sobre los pesos:

*Para obligar  $w_1 = w_2$   
tenemos que garantizar  $\Delta w_1 = \Delta w_2$*

*Calculamos  $\frac{\partial E}{\partial w_1}$  y  $\frac{\partial E}{\partial w_2}$*

*Usamos  $\frac{\partial E}{\partial w_1} + \frac{\partial E}{\partial w_2}$  para  $w_1$  y  $w_2$*

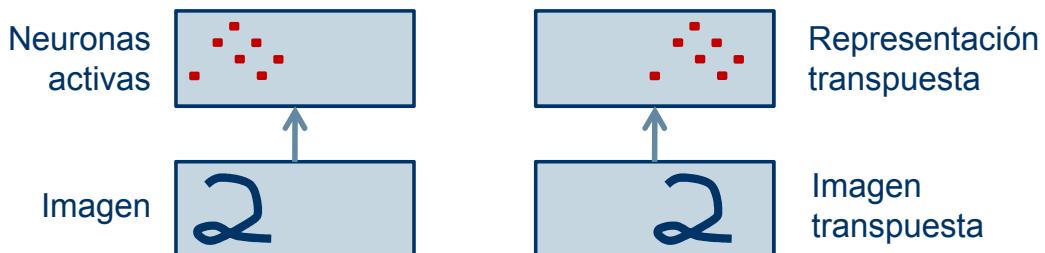


# En la práctica: Invarianza



## Redes convolutivas

¿Qué consiguen los detectores replicados?



Los detectores replicados no hacen que la actividad neuronal sea invariante frente a traslaciones:  
su actividad es “equivariante”.

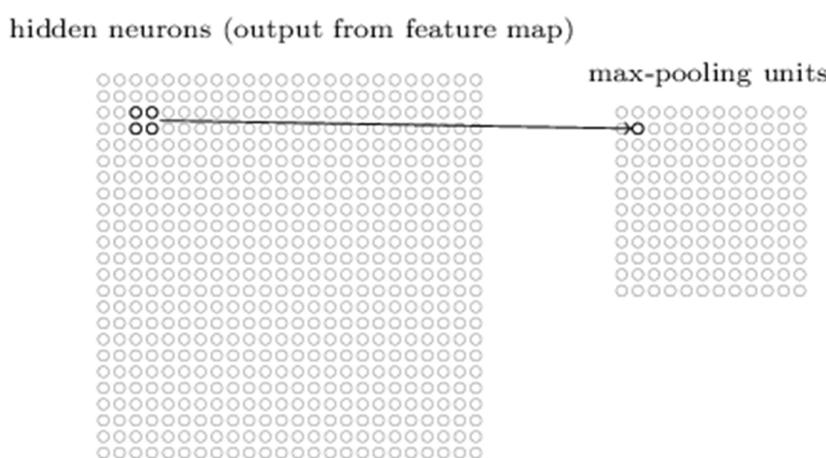
Si la característica resulta útil en algunas posiciones, la red incorpora detectores para esa característica en todas las posiciones



# En la práctica: Invarianza



## Redes convolutivas: Pooling



Pooling (reducción de la dimensionalidad)

<http://neuralnetworksanddeeplearning.com/>



# En la práctica: Invarianza



## Redes convolutivas: Pooling

¿Qué se consigue agregando las salidas de receptores de características replicados?

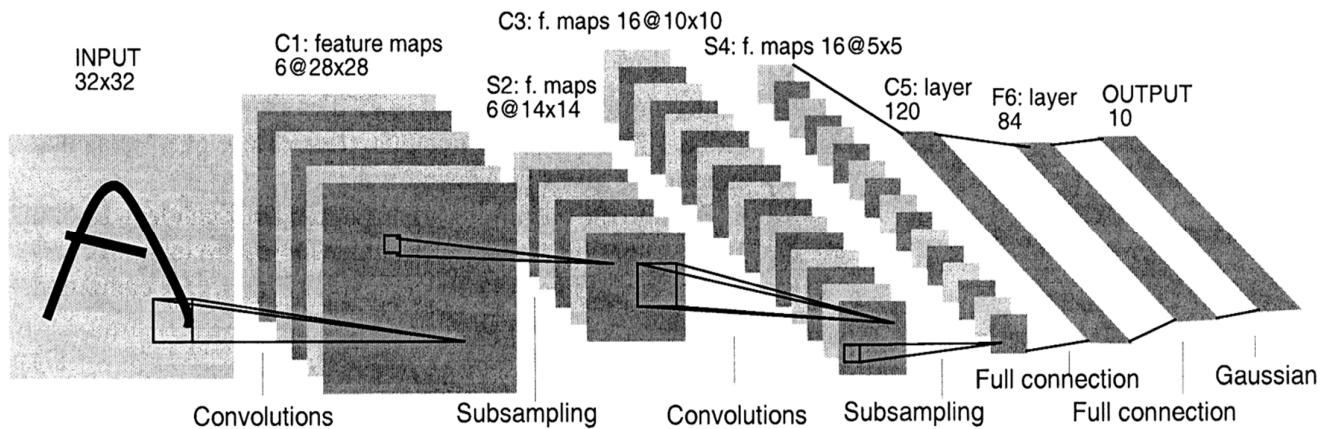
- Una reducción del número de entradas de las siguientes capas de la red neuronal, lo que nos permite tener más características distintas en paralelo.
- Una pequeña cantidad de invarianza frente a traslaciones en cada nivel (p.ej. 2x2 max-pooling).
- Problema: Tras varios niveles de “pooling”, se pierde información acerca de la posición exacta de las cosas.



# En la práctica: Invarianza



## Redes convolutivas



## EJEMPLO: LeNet5

<http://yann.lecun.com/exdb/lenet/>



# En la práctica: Invarianza



## Redes convolutivas

4	3	8	1	5	6	2	3	6	1
4	8	7	5	7	6	7	2	3	4
9	4	8	7	5	0	3	2	8	3
8	5	8	3	0	9	9	1	4	1
9	2	1	3	3	9	0	5	6	8
4	7	9	4	2	7	4	9	9	9
7	4	8	5	6	6	8	3	3	9
1	9	6	0	5	7	0	6	4	1
2	5	4	7	2	5	9	1	6	5
4	8								

EJEMPLO: Los 82 errores de **LeNet5**

<http://yann.lecun.com/exdb/lenet/>



# En la práctica: Invarianza



Técnicas que nos permiten diseñar redes robustas, invariantes frente a determinadas transformaciones (rotaciones o cambios de escala en imágenes; cambios de volumen, velocidad o tono en reconocimiento de voz):

- Invarianza por **extracción de características**.
- Invarianza por **estructura**.
- Invarianza por **entrenamiento**.

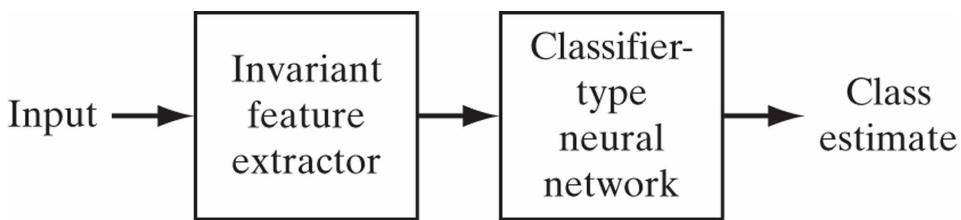


# En la práctica: Invarianza



## Invarianza por extracción de características

Preprocesamiento del conjunto de entrenamiento, del que se extraen características “esenciales” que sean invariantes con respecto a las transformaciones deseadas).



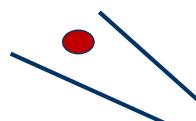
[Haykin: “Neural Networks and Learning Machines”, 3<sup>rd</sup> edition]



# En la práctica: Invarianza



## Invarianza por extracción de características



### EJEMPLO

Par de líneas, más o menos paralelas, con algo en medio.

### IDEA

Con un número suficiente de características de este tipo, podríamos esperar ser capaces de identificar un objeto...



# En la práctica: Invarianza



## Invarianza por estructura, p.ej. redes convolutivas

Red diseñada de forma que sus conexiones sinápticas hagan que versiones transformadas de la entrada produzcan salidas similares.

- Conectividad de la red ("local receptive fields")
- Restricciones sobre los pesos ("weight sharing")
- ...

Menos tedioso que diseñar/extrair a mano características, aunque introduce prejuicios acerca de la forma particular de resolver el problema que tengamos en mente.



# En la práctica: Invarianza



## Invarianza por entrenamiento

Red entrenada a partir de un conjunto de entrenamiento ampliado, en el que se incluyen versiones transformadas de los ejemplos originales del conjunto de entrenamiento.

- El entrenamiento puede ser mucho más costoso.
- La optimización de los pesos de la red puede descubrir formas novedosas de utilizar la red multicapa
  - ... que a nosotros no se nos hayan ocurrido :-)
  - ... que nosotros nunca sepamos interpretar :-(



# En la práctica: Invarianza



## Invarianza por entrenamiento

LeNet utiliza conocimiento del problema para diseñar la estructura de la red (invarianza por estructura):

82 errores (0.82%)

Ciresan et al. (2010) crearon **cuidadosamente** nuevos ejemplos de entrenamiento aplicando distintos tipos de transformaciones, tras lo que entrenaron una “deep, dumb net” usando una GPU:

35 errores (0.35%)



# En la práctica: Invarianza



## Invarianza por entrenamiento

1 2 1 7	1 1 7 1	9 8 9 8	9 9 5 9	9 9 7 9	5 5 3 5	3 8 2 3
4 9 4 9	5 5 3 5	9 4 9 7	4 9 4 9	4 4 9 4	2 2 0 2	5 5 3 5
1 6 1 6	9 4 9 4	0 0 6 0	6 6 0 6	6 6 8 6	1 1 7 9	1 1 7 1
9 9 4 9	0 0 5 0	3 5 3 5	8 8 9 8	7 9 7 9	7 7 1 7	1 1 6 1
2 7 2 7	8 8 5 8	2 2 7 8	1 6 1 6	6 5 6 5	4 4 9 4	0 0 6 0

EJEMPLO: Los 35 errores de Ciresan et al. (2010)  
<http://arxiv.org/abs/1003.0358>



# Softmax



El uso del error cuadrático como medida de error tiene algunos inconvenientes:

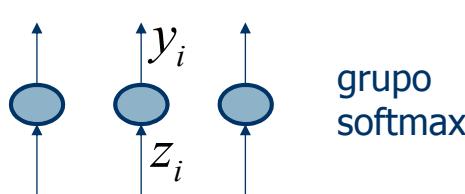
- Si la salida deseada es 1 y la salida actual es 0.000001 el gradiente es prácticamente 0, por lo que una unidad logística difícilmente conseguirá corregir el error.
- Si estamos ante un problema de clasificación y queremos estimar la probabilidad de cada clase, sabemos que la suma de las salidas debería ser 1, pero no estamos usando esa información para entrenar la red neuronal.



# Softmax



¿Existe alguna función de coste alternativa que funcione mejor? Sí, una que fuerza que las salidas de la red representen una distribución de probabilidad.



$$y_i = \frac{e^{z_i}}{\sum_{j \in \text{group}} e^{z_j}}$$

$$\frac{\partial y_i}{\partial z_i} = y_i(1 - y_i)$$



# Softmax



## Entropía cruzada [cross-entropy]

La función de coste asociada a softmax:  $C = -\sum_j t_j \log y_j$

- El gradiente de C es muy grande si el valor deseado ( $t$ ) es 1 pero la salida obtenida ( $y$ ) está cercana a 0.

$$\frac{\partial C}{\partial z_i} = \sum_j \frac{\partial C}{\partial y_j} \frac{\partial y_j}{\partial z_i} = y_i - t_i$$

La pendiente de  $\delta C/\delta y$   
compensa el valor bajo de  $\delta y/\delta z$



# Softmax



## Una interpretación alternativa

UFLDL Tutorial, <http://ufldl.stanford.edu/tutorial/>

La regresión lineal consiste en encontrar una función  $\mathbf{h}_\theta(\mathbf{x}) = \theta^T \mathbf{x}$  en la que los parámetros  $\theta$  se eligen de forma que se minimiza una función de coste  $J(\theta)$ :

$$J(\theta) = \frac{1}{2} \sum_i (h_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{2} \sum_i (\theta^T x^{(i)} - y^{(i)})^2$$

Para minimizar dicha función usando el gradiente descendente, calculamos  $\nabla_\theta J(\theta)$ :

$$\frac{\partial J(\theta)}{\partial \theta_j} = \sum_i x_j^{(i)} (h_\theta(x^{(i)}) - y^{(i)}) = \sum_i x_j^{(i)} (\theta^T x^{(i)} - y^{(i)})$$



# Softmax



## Una interpretación alternativa

UFLDL Tutorial, <http://ufldl.stanford.edu/tutorial/>

De la misma forma, podemos predecir una variable discreta utilizando regresión logística:

$$P(y = 1 | x) = h_{\theta}(x) = \sigma(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

$$P(y = 0 | x) = 1 - P(y = 1 | x) = 1 - h_{\theta}(x)$$

donde  $\sigma(z)$  es la función logística.



# Softmax



## Una interpretación alternativa

UFLDL Tutorial, <http://ufldl.stanford.edu/tutorial/>

Nuestro objetivo en regresión logística es buscar valores para  $\theta$  de forma que  $h_{\theta}(\mathbf{x})$  sea grande cuando  $\mathbf{x}$  pertenece a la clase 1 y pequeño si pertenece a la clase 0.

La siguiente función de coste nos sirve:

$$J(\theta) = - \sum_i \left( y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_{\theta}(x^{(i)})) \right)$$

NOTA: Sólo uno de los dos términos es distinto de cero para cada ejemplo (según sea de una clase u otra).



# Softmax



## Una interpretación alternativa

UFLDL Tutorial, <http://ufldl.stanford.edu/tutorial/>

¿De dónde sale esa función de coste?

Si asumimos que los ejemplos del conjunto de entrenamiento se generaron de forma independiente, la función de verosimilitud [likelihood] de los parámetros es

$$\begin{aligned} L(\theta) &= p(\vec{y} \mid X; \theta) \\ &= \prod_{i=1}^m p(y^{(i)} \mid x^{(i)}; \theta) \\ &= \prod_{i=1}^m (h_\theta(x^{(i)}))^{y^{(i)}} (1 - h_\theta(x^{(i)}))^{1-y^{(i)}} \end{aligned}$$



# Softmax



## Una interpretación alternativa

UFLDL Tutorial, <http://ufldl.stanford.edu/tutorial/>

¿De dónde sale esa función de coste?

Dicha función de verosimilitud [likelihood] resulta más fácil de maximizar tomando logaritmos [log-likelihood]:

$$\begin{aligned} \ell(\theta) &= \log L(\theta) \\ &= \sum_{i=1}^m y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h(x^{(i)})) \end{aligned}$$



# Softmax



## Una interpretación alternativa

UFLDL Tutorial, <http://ufldl.stanford.edu/tutorial/>

Para maximizar el log-likelihood, minimizamos la función de coste  **$J(\theta) = -\log L(\theta)$** , para lo que calculamos  $\nabla_{\theta} J(\theta)$ :

$$\frac{\partial J(\theta)}{\partial \theta_j} = \sum_i x_j^{(i)} (h_{\theta}(x^{(i)}) - y^{(i)})$$

O, en forma vectorial:

$$\nabla_{\theta} J(\theta) = \sum_i x^{(i)} (h_{\theta}(x^{(i)}) - y^{(i)})$$



# Softmax



La regresión softmax (o regresión logística multinomial) no es más que una generalización de la regresión logística cuando tenemos más de dos clases distintas.

Si tenemos K clases, tendremos K vectores de parámetros  $\theta_k$ :

$$h_{\theta}(x) = \begin{bmatrix} P(y=1|x) \\ P(y=2|x) \\ \vdots \\ P(y=K|x) \end{bmatrix} = \frac{1}{\sum_{j=1}^K e^{\theta_j^T x}} \begin{bmatrix} e^{\theta_1^T x} \\ e^{\theta_2^T x} \\ \vdots \\ e^{\theta_K^T x} \end{bmatrix}$$



# Softmax



La función de coste asociada a la regresión logística la podríamos reescribir como:

$$\begin{aligned} J(\theta) &= -\sum_i (y_i \log h_\theta(x_i) + (1 - y_i)(1 - \log h_\theta(x_i))) \\ &= -\sum_i t_i \log P(t_i | x_i) \end{aligned}$$

Extendiendo esta función de coste a la regresión softmax, la función de coste resultante es, como esperamos, la entropía cruzada:

$$C = J(\theta) = -\sum_i t_i \log P(t_i | x_i) = -\sum_j t_j \log y_j$$



## Referencias

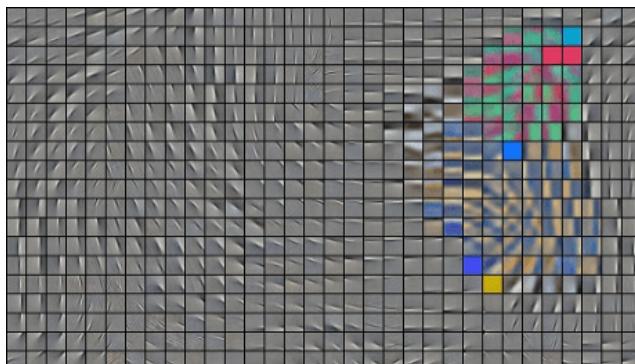


### **Neural Networks for Machine Learning**

by Geoffrey Hinton

(University of Toronto & Google)

<https://www.coursera.org/course/neuralnets>

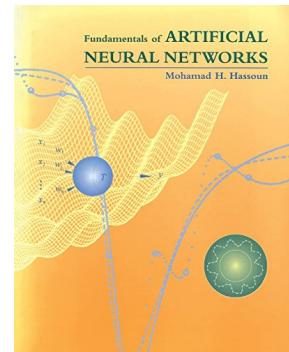
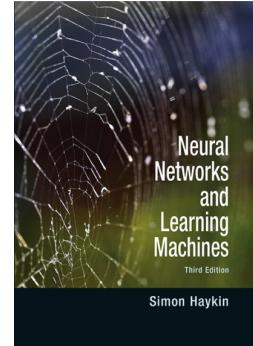


# Bibliografía



## Lecturas recomendadas

- Simon Haykin:  
**Neural Networks  
and Learning Machines**  
Prentice Hall, 3rd edition, 2008  
ISBN 0131471392
- Mohamad Hassoun:  
**Fundamentals of  
Artificial Neural Networks**  
MIT Press, 2003  
ISBN 0262514672

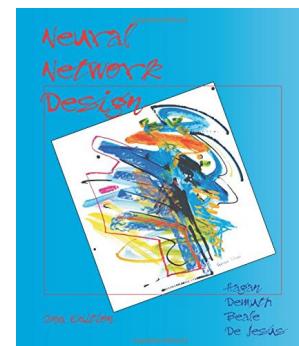
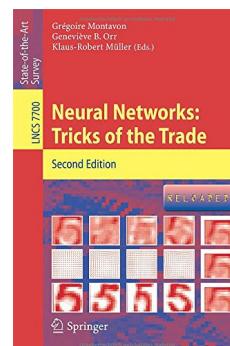


# Bibliografía



## Consejos “prácticos”

- Montavon, Orr & Müller (eds.):  
**Neural Networks:  
Tricks of the Trade**  
Springer, 2nd edition, 2012  
ISBN 364235288X
- Martin T. Hagan, Howard B. Demuth,  
Mark H. Beale & Orlando de Jesús:  
**Neural Network Design**, 2nd edition, 2014  
ISBN 0971732116  
<http://hagan.okstate.edu/NNDesign.pdf>

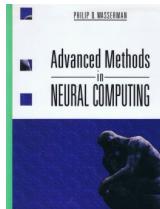
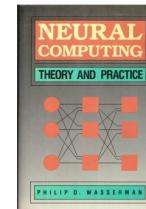
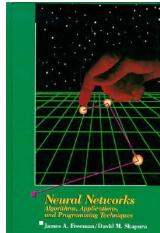
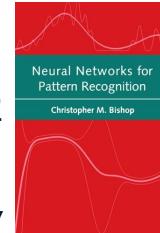


# Bibliografía complementaria



## Redes neuronales artificiales

- Christopher M. Bishop:  
**Neural Networks for Pattern Recognition**  
Oxford University Press, 1996. ISBN 0198538642
- James A. Freeman & David M. Skapura:  
**Neural Networks: Algorithms, Applications, and Programming Techniques**  
Addison-Wesley, 1991. ISBN 0201513765
- Philip D. Wasserman:  
**Neural Computing: Theory and Practice**,  
Van Nostrand Reinhold, 1989. ISBN 0442207433
- Philip D. Wasserman:  
**Advanced Methods in Neural Computing**  
Van Nostrand Reinhold, 1993. ISBN 0442004613



# Bibliografía



## Bibliografía en castellano

- James A. Freeman & David M. Skapura:  
**Redes Neuronales: Algoritmos, aplicaciones y técnicas de programación**  
Addison-Wesley / Díaz de Santos, 1993.  
ISBN 020160115X

