



UNIVERSIDAD DE CORDOBA



Aprendizaje profundo

Algoritmo de retropropagación (*BackPropagation*, BP)

Pedro Antonio Gutiérrez (pagutierrez@uco.es) y Javier Sánchez Mone-
dero (jsanchezm@uco.es)

21 de febrero de 2024

Departamento de Informática y Análisis Numérico
Universidad de Córdoba.



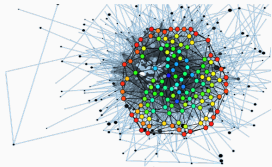
- Introducción
- Neurona artificial
- Redes neuronales
- Aprendizaje
- Clasificación
- RNAs en Weka
- Conclusiones

Resumen

Objetivos del tema

Introducción a las redes neuronales artificiales (RNAs):

- Motivación del uso de este tipo de modelos.
- Modelo básico: neurona artificial o perceptrón simple.
- Modelo más avanzado: RNA o perceptrón multicapa.
- Algoritmo de entrenamiento: algoritmo de retropropagación o descenso por gradiente.
- Limitaciones del algoritmo básico y mecanismos para evitarlas.
- Adaptación del algoritmo y del modelo para problemas de clasificación.
- Uso de RNAs en Weka.
- Ventajas/desventajas de las RNAs.



Introducción

Neurona artificial

Redes neuronales

Aprendizaje

Clasificación

RNAs en Weka

Conclusiones

El cerebro como un dispositivo computacional

- Los animales son capaces de reaccionar de forma adaptativa a los cambios en su entorno externo e interno, y utilizan su sistema nervioso para implementar estas conductas.
- Un modelo adecuado de simulación del sistema nervioso debe ser capaz de producir respuestas y comportamientos similares en los sistemas artificiales.
- El sistema nervioso está construido por unidades relativamente simples, las neuronas, por lo que copiar su comportamiento y funcionalidad puede ser la solución.

El cerebro como un dispositivo computacional

¿Cómo es capaz de trabajar tan bien nuestro cerebro?

- **Paralelismo masivo:** el cerebro es un sistema de procesamiento de señales o de información que está compuesto por un gran número de elementos simples de procesamiento, llamados neuronas.
- **Conexionismo:** el cerebro es un sistema neuronal altamente interconectado, de tal forma que el estado de una neurona afecta al potencial de un gran número del resto de neuronas conectadas de acuerdo a sus pesos o enlaces.
- **Memoria asociativa distribuida:** el almacenamiento de la información en el cerebro esta concentrado en las conexiones sinápticas de la red neuronal, o con más precisión, en los patrones de esas conexiones y en su fuerza (pesos).

El cerebro como un dispositivo computacional

- **Motivación:** Los algoritmos de aprendizaje desarrollados durante siglos no pueden abordar la complejidad de los problemas reales.
- Sin embargo, el **cerebro humano** es la más sofisticada computadora para resolver problemas extremadamente complejos.
 - **Tamaño razonable:** 10^{11} neuronas (células neuronales), donde solo una pequeña porción son utilizadas.
 - **Nodos de procesamiento simples:** las células no contienen demasiada información.
 - **Masivamente paralelo:** cada región del cerebro realiza tareas específicas.
 - **Tolerante a fallos y robusto:** la información se salva, principalmente, en las conexiones entre neuronas, que pueden llegar a regenerarse.

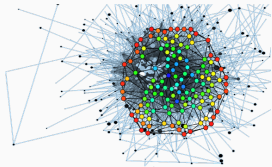
Comparando el cerebro contra un ordenador

	Ordenador	Cerebro humano
Unidades de computación	1 CPU, 10^5 puertas	10^{11} neuronas
Unidades de almacenamiento	10^9 bits RAM	10^{14} sinapsis
Tiempo por ciclo	10^{-8} segundos	10^{-3} segundos
Ancho de banda	10^{22} bits / segundo	10^{28} bits / segundo

Incluso siendo un millón de veces más rápido el ordenador que el cerebro en velocidad de computación, un cerebro termina siendo un millón de veces más rápido que el ordenador (gracias a la gran cantidad de elementos de computación).

- Reconocer una cara:
 - Cerebro < 1s.
 - Computadora: billones de ciclos.

- En la mayoría de problemas de clasificación/regresión, la formalización de una regla de decisión es muy compleja (o imposible).
- Una red neuronal artificial es un **modelo no lineal** de clasificación o regresión, capaz de acumular conocimiento (aprender) acerca de sus coeficientes y estructura, a través de un algoritmo (algoritmo de retropropagación).
- Después del proceso de aprendizaje, una red (modelo no lineal) es capaz de aproximar una función continua, la cual se supone que será nuestra regla de decisión.



Introducción

Neurona artificial

Redes neuronales

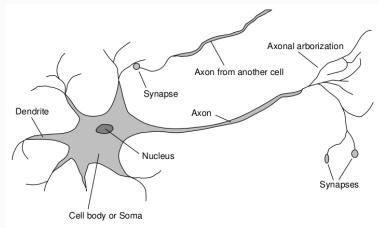
Aprendizaje

Clasificación

RNAs en Weka

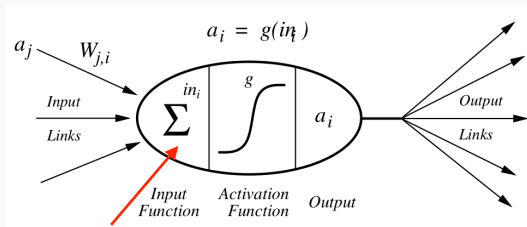
Conclusiones

Neurona biológica



- Una neurona no hace nada hasta que la influencia de sus entradas alcanza un determinado nivel.
- La neurona produce una salida en la forma de pulso que parte del núcleo, baja por el axón y termina en sus ramificaciones.
- O se dispara o no hace nada (dispositivo “**todo-o-nada**”).
- La salida causa la excitación o inhibición de otras neuronas.

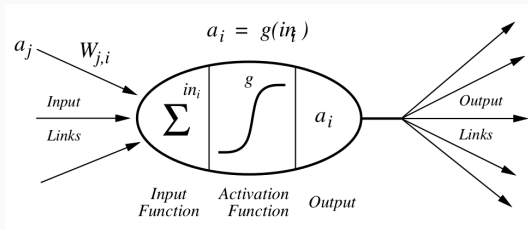
Neurona artificial (perceptrón)



Suma ponderada de las entradas

- Las neuronas artificiales son nodos conectados a otros nodos mediante enlaces.
- A cada enlace se le asocia un **peso**. ($w_{j,i}$)
- El peso determina la naturaleza (**excitatoria** $+$ o **inhibitoria** $-$) y la fuerza (**valor absoluto**) de la influencia entre nodos.
- Si la influencia de todos los enlaces de entrada es suficientemente alta, el nodo se activa.

Neurona artificial (perceptrón)



- Cada nodo i tiene varias conexiones de entrada y de salida, cada una con sus pesos.
- La salida es una función de la suma ponderada de las entradas.

Neurona artificial (perceptrón)

- Cada enlace de entrada a la neurona i le proporciona un valor de activación a_j que proviene de otra neurona.
- A menudo, la **función de entrada** es la suma ponderada de dichos valores de activación:

$$in_i(a_1, \dots, a_{n_i}) = \sum_{j=1}^{n_i} W_{j,i} a_j$$

- La salida de la neurona es el resultado de aplicar la **función de activación** sobre la función de entrada.

$$out_i = g(in_i) = g \left(\sum_{j=1}^{n_i} W_{j,i} a_j \right)$$

Neurona artificial (perceptrón)

En forma vectorial:

$$out_i = f(\mathbf{x}, \mathbf{w}),$$

donde \mathbf{x} es el vector de entradas a la neurona, \mathbf{w} es el vector de pesos sinápticos y la función f es la composición de la función de entrada y la función de activación:

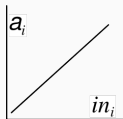
$$f(\mathbf{x}, \mathbf{w}) = g(in(\mathbf{x}, \mathbf{w})).$$

Para neuronas aditivas, podemos usar el producto escalar:

$$f(\mathbf{x}, \mathbf{w}) = g(\mathbf{x} \cdot \mathbf{w}).$$

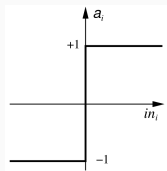
Funciones de activación $g(\cdot)$

- Función lineal: $\text{linear}(x) = x$



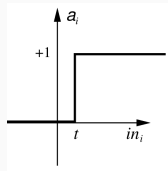
- Función signo:

$$\text{sign}(x) = \begin{cases} +1, & \text{si } x \geq 0 \\ -1, & \text{si } x < 0 \end{cases}$$



- Función de salto o de umbral:

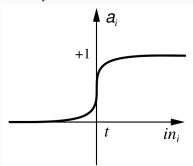
$$\text{step}_t(x) = \frac{\text{sign}(x-t)+1}{2} = \begin{cases} 1, & \text{si } x \geq t \\ 0, & \text{si } x < t \end{cases}$$



Funciones de activación $g(\cdot)$

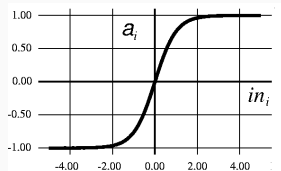
- Función sigmoide (logística):

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



- Función tangente hiperbólica:

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



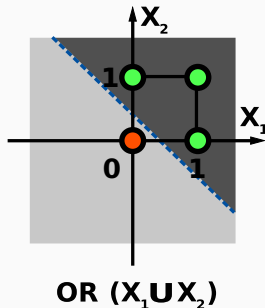
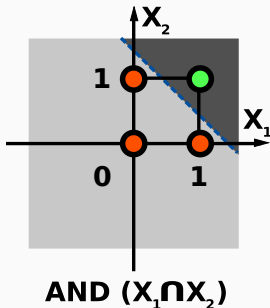
Simulación de funciones lógicas

- La salida de una neurona de umbral es binaria, mientras que las entradas pueden ser binarias o continuas.
- Si las entradas son binarias, una función de umbral implementa una función Booleana.
- El alfabeto Booleano $\{1, -1\}$ es el que habitualmente se utiliza en vez del $\{0, 1\}$. La correspondencia con el alfabeto clásico Booleano $\{0, 1\}$ se puede establecer mediante:

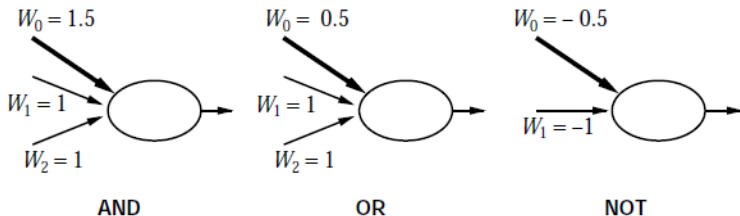
$$0 \rightarrow 1; 1 \rightarrow -1; y \in \{0, 1\}, x \in \{1, -1\} \Rightarrow x = 1 - 2y = (-1)^y$$

Neurona artificial

- Simular mediante una neurona funciones lógicas simples.
- Para el **And** y el **Or** nos basta una neurona con la función *step* y los pesos correctamente seleccionados:



Neurona artificial



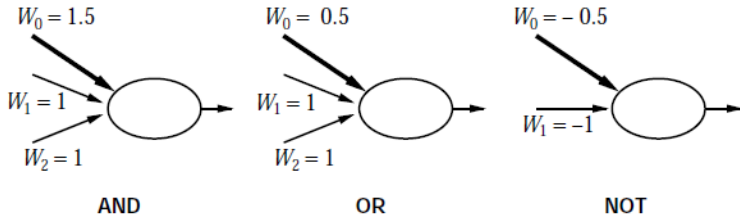
$$and(0, 0) = step_{1,5}(1 \cdot 0 + 1 \cdot 0) = step_{1,5}(0) = 0$$

$$and(0, 1) = step_{1,5}(1 \cdot 0 + 1 \cdot 1) = step_{1,5}(1) = 0$$

$$and(1, 0) = step_{1,5}(1 \cdot 1 + 1 \cdot 0) = step_{1,5}(1) = 0$$

$$and(1, 1) = step_{1,5}(1 \cdot 1 + 1 \cdot 1) = step_{1,5}(2) = 1$$

Neurona artificial



$$or(0, 0) = step_{0,5}(1 \cdot 0 + 1 \cdot 0) = step_{0,5}(0) = 0$$

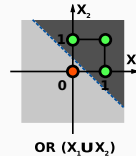
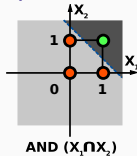
$$or(0, 1) = step_{0,5}(1 \cdot 0 + 1 \cdot 1) = step_{0,5}(1) = 1$$

$$or(1, 0) = step_{0,5}(1 \cdot 1 + 1 \cdot 0) = step_{0,5}(1) = 1$$

$$or(1, 1) = step_{0,5}(1 \cdot 1 + 1 \cdot 1) = step_{0,5}(2) = 1$$

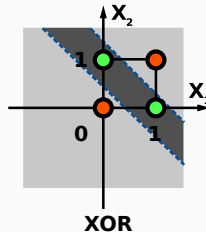
Neurona artificial

- Estos problemas se pueden resolver con una neurona porque son **linealmente separables**:



- Problema:** una sola neurona no es capaz de resolver problemas más complejos, p.ej. el XOR.

El XOR necesita 2 neuronas: una que distingue el (0,0) de los demás, otra que distingue el (1,1) de los demás y en la "intersección" de las neuronas se encuentran el resto de valores.





Introducción

Neurona artificial

Redes neuronales

Aprendizaje

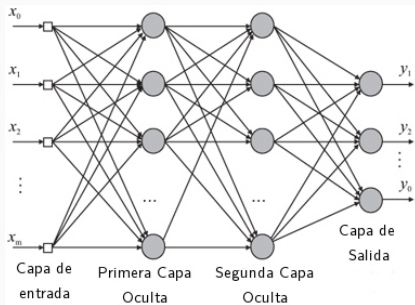
Clasificación

RNAs en Weka

Conclusiones

Redes neuronales artificiales

- Una **Red Neuronal Artificial (RNA)** es un grafo de nodos (o neuronas) conectados por enlaces.
- Las neuronas se organizan por **capas**, de manera que las salidas de las neuronas de una capa sirven como entradas para las neuronas de la siguiente capa.
- También conocidas como *MultiLayer Perceptron* (MLP).

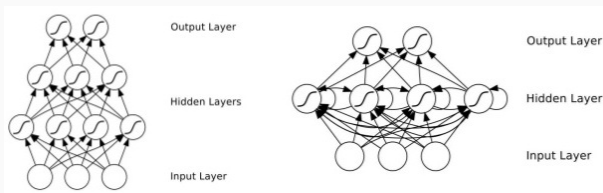


Este es un ejemplo de
"Propagación hacia adelante"

Redes neuronales artificiales

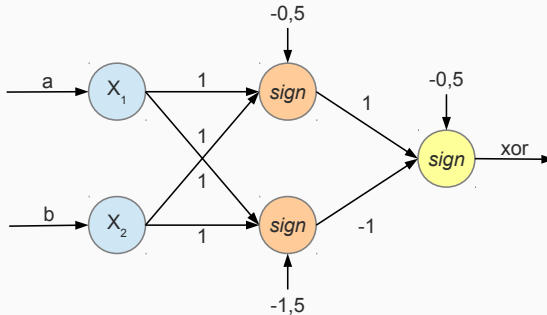
Tipos de RNAs según su organización:

- **RNA de propagación hacia delante** (*Feedforward neural networks*): las neuronas de una capa solo se enlazan con las neuronas de la capa siguiente.
 - A veces, se establecen enlaces que se saltan capas (*skip layer connections*): la capa de entrada conectada con la primera capa y con todas las capas ocultas. **Capas ocultas: las de en medio, las que no son entrada ni salida**
- **RNA recurrentes**: las neuronas de una capa se pueden conectar entre si, formando bucles (se utilizan para el modelado de series temporales). **P.ej. cuando oyes una oración, el significado de una palabra puede depender de la palabra anterior a esa.**

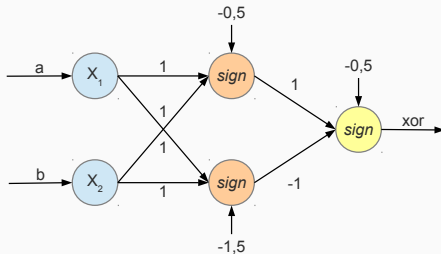


Redes neuronales artificiales

- El problema XOR se puede resolver con una RNA de una capa oculta y dos neuronas:

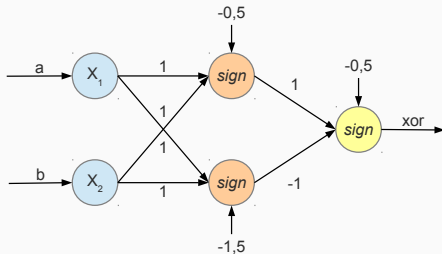


Redes neuronales artificiales



$$\begin{aligned} \text{xor}(0, 0) &= \text{step}_{0,5}(\text{step}_{0,5}(1 \cdot 0 + 1 \cdot 0) - \text{step}_{1,5}(1 \cdot 0 + 1 \cdot 0)) \\ &= \text{step}_{0,5}(\text{step}_{0,5}(0) - \text{step}_{1,5}(0)) = \text{step}_{0,5}(0 - 0) = \text{step}_{0,5}(0) = 0 \\ \text{xor}(0, 1) &= \text{step}_{0,5}(\text{step}_{0,5}(1 \cdot 0 + 1 \cdot 1) - \text{step}_{1,5}(1 \cdot 0 + 1 \cdot 1)) \\ &= \text{step}_{0,5}(\text{step}_{0,5}(1) - \text{step}_{1,5}(1)) = \text{step}_{0,5}(1 - 0) = \text{step}_{0,5}(1) = 1 \end{aligned}$$

Redes neuronales artificiales

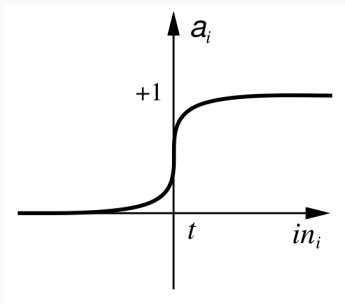


$$\begin{aligned} \text{xor}(1, 0) &= \text{step}_{0,5}(\text{step}_{0,5}(1 \cdot 1 + 1 \cdot 0) - \text{step}_{1,5}(1 \cdot 1 + 1 \cdot 0)) \\ &= \text{step}_{0,5}(\text{step}_{0,5}(1) - \text{step}_{1,5}(1)) = \text{step}_{0,5}(1 - 0) = \text{step}_{0,5}(1) = 1 \\ \text{xor}(1, 1) &= \text{step}_{0,5}(\text{step}_{0,5}(1 \cdot 1 + 1 \cdot 1) - \text{step}_{1,5}(1 \cdot 1 + 1 \cdot 1)) \\ &= \text{step}_{0,5}(\text{step}_{0,5}(2) - \text{step}_{1,5}(2)) = \text{step}_{0,5}(1 - 1) = \text{step}_{0,5}(0) = 0 \end{aligned}$$

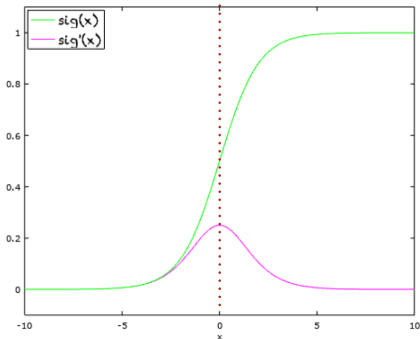
Redes neuronales artificiales

- La capa oculta pasa los datos a un espacio donde la tarea a realizar es más fácil.
- Si queremos optimizar los pesos de la red, la función *step* no es adecuada, ¿por qué?. Porque no es derivable
- Utilizamos una aproximación, la función sigmoide (con sesgo):

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



Propiedades de la función sigmoide



Plot of $\sigma(x)$ and its derivate $\sigma'(x)$

Domain: $(-\infty, +\infty)$

Range: $(0, +1)$

$$\sigma(0) = 0.5$$

Other properties

$$\sigma(x) = 1 - \sigma(-x)$$

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

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

derivada

sigma(x) es la salida de la neurona.

La derivada sería $\sigma(x) \cdot (1 - \sigma(x))$, así que si tenemos la salida calculada, sacar la derivada es muy rápido.

Source: <https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/>

Redes neuronales artificiales

- Propiedad muy importante de la derivada de $\sigma(x)$:

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

$$\begin{aligned}\sigma'(x) &= (-1) \cdot (1 + e^{-x})^{-2} (1 + e^{-x})' = \\ &= (-1) \cdot (1 + e^{-x})^{-2} \cdot (0 + (e^{-x})') \\ &= (-1) \cdot (1 + e^{-x})^{-2} \cdot (e^{-x}) \cdot (-x)' \\ &= (-1) \cdot (1 + e^{-x})^{-2} \cdot (e^{-x}) \cdot (-1) = \frac{e^{-x}}{(1 + e^{-x})^2} \\ &= \frac{1}{(1 + e^{-x})} \frac{e^{-x}}{(1 + e^{-x})} = \sigma(x) \frac{1 + e^{-x} - 1}{(1 + e^{-x})} \\ &= \sigma(x) \left(\frac{(1 + e^{-x})}{(1 + e^{-x})} - \frac{1}{(1 + e^{-x})} \right) = \sigma(x) \cdot (1 - \sigma(x))\end{aligned}$$



Introducción

Neurona artificial

Redes neuronales

Aprendizaje

Clasificación

RNAs en Weka

Conclusiones

Algoritmo de retropropagación

- Red neuronal *feedforward*: propagación hacia delante para obtener las salidas.
- Algoritmo *backpropagation*: propagación hacia atrás para obtener el error, las derivadas y ajustar los pesos.
'backpropagation' -> también llamado Regla Delta
- Dado un conjunto de entrenamiento, queremos ajustar los pesos de las conexiones para que el error cometido de clasificación o regresión en dicho conjunto sea **mínimo**.
- La idea general es **minimizar el error de entrenamiento** y el procedimiento matemático está basado en **derivar** una función de coste.
- Algoritmo de retropropagación del error o de **descenso por gradiente**.

Algoritmo de retropropagación

- Notación:

negrita → vector

no negrita → valor escalar

- Patrones de entrenamiento:

- Vector de entradas: $\mathbf{x} = (x_1, \dots, x_k)$.
 - Vector de salidas deseadas: $\mathbf{d} = (d_1, \dots, d_J)$.

- Arquitectura de la red: $\{n_0 : n_1 : \dots : n_H\}$

- n_h es el número de neuronas de la capa h .
 - $n_0 = k$, $n_H = J$.
 - $H - 1$ capas ocultas. **H - 1 → Porque empezamos en 0 y tenemos dos capas más (input y output)**

- Pesos de la red. Para cada capa h , sin contar la capa de entrada:

- Matriz con un vector de pesos de entrada por cada neurona:
 $\mathbf{W}^h = (\mathbf{w}_1^h, \dots, \mathbf{w}_{n_h}^h)$.
 - Vector de pesos de la neurona j de la capa h (incluye sesgo):
 $\mathbf{w}_j^h = (w_{j0}^h, w_{j1}^h, \dots, w_{jn_{(h-1)}}^h)$.

- Salida de la red: $\mathbf{o} = (o_1, \dots, o_J)$.

Algoritmo de retropropagación

Primero, vamos a considerar problemas de predicción (**regresión**) con una o más variables a predecir.

Vamos a considerar que todas las neuronas, salvo las de la capa de entrada, serán de tipo sigmoide.

- Si las neuronas disponen de sesgo, su fórmula será:

superíndice → capa
subíndice → neurona

$$out_j^h = \frac{1}{1 + \exp(-w_{j0}^h - \sum_{i=1}^{n_{h-1}} w_{ji}^h out_i^{h-1})}$$

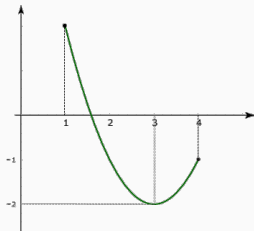
Salida de la neurona 'j' en la capa 'h'

- Si no disponen de sesgo:

$$out_j^h = \frac{1}{1 + \exp(-\sum_{i=1}^{n_{h-1}} w_{ji}^h out_i^{h-1})}$$

Algoritmo de retropropagación

- Idea subyacente:
 - Minimizar la función $f(x) = x^2 - 6x + 7$.



- Función derivada $f'(x) = 2x - 6$
- Como la función es muy simple (**un solo mínimo global**) y solo depende de **una variable** (x), para minimizar podemos igualar a cero esta derivada:

$$2x - 6 = 0 \rightarrow x = 6/2 = 3 \quad (1)$$

Algoritmo de retropropagación

- La filosofía es la misma, vamos a intentar minimizar el error cometido por la red neuronal, **tomando los pesos de la red como variables**. Para un patrón concreto, el error cuadrático medio (consideramos problemas de **regresión**) es:

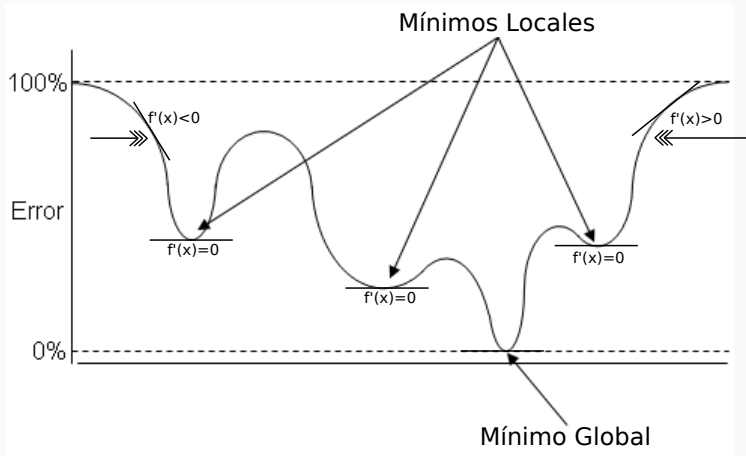
Cuánto se desvía la salida respecto a lo deseado

$$E = \frac{1}{J} \sum_{i=1}^J (d_i - o_i)^2 \quad (2)$$

'Di' es la salida deseada de la neurona 'i'
'Oi' es la salida real de la neurona 'i'

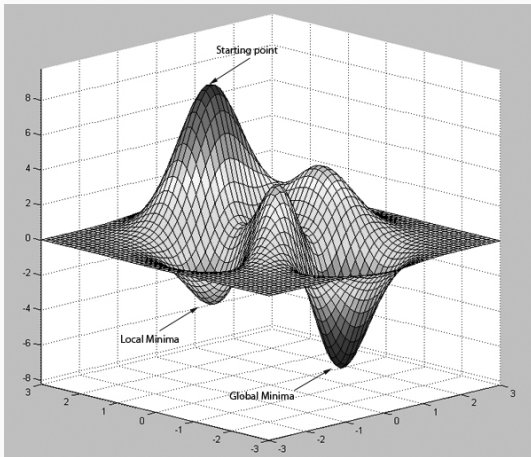
- El valor de d_i es fijo (según el patrón de entrenamiento) y está dado por el investigador, tutor o decisor (aprendizaje supervisado), pero **el valor de o_i depende de los pesos**.
- Realizamos un proceso iterativo, donde, dado un valor para los pesos actuales, movemos esos pesos intentando minimizar E .
- Evaluamos la derivada en el punto en el que estamos:
 - Si $f'(x) > 0$, disminuimos el valor de x . Si tengo una derivada positiva, tengo que bajar el peso.
 - Si $f'(x) < 0$, aumentamos el valor de x . Si tengo una derivada negativa, tengo que subir el peso.

Algoritmo de retropropagación



Algoritmo de retropropagación

- Imaginemos que la red solo tiene dos pesos, entonces, según el valor de los pesos, podemos representar su **superficie de error**:



Algoritmo de retropropagación

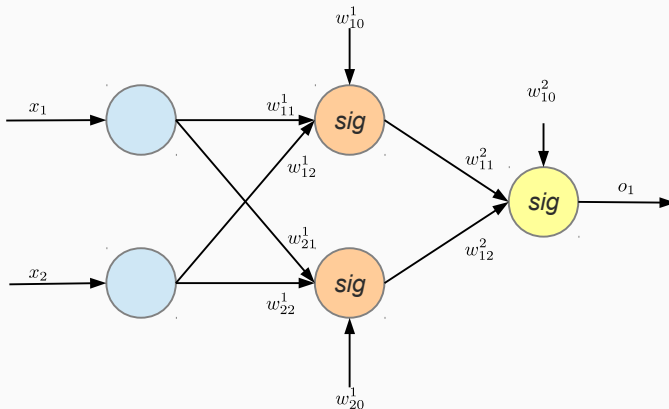
- En el caso de tener múltiples pesos, necesitamos un vector de derivadas, donde cada componente es la derivada del error respecto a cada uno de los pesos.
- Esto es lo que se conoce como el **vector gradiente**.

$$\nabla E = \left\{ \frac{\partial E}{\partial w_{10}^1}, \frac{\partial E}{\partial w_{11}^1}, \dots, \frac{\partial E}{\partial w_{n_1 k}^1}, \frac{\partial E}{\partial w_{10}^2}, \dots, \frac{\partial E}{\partial w_{J_{n(H-1)}}^H} \right\}$$

- La estructuración en capas de la red neuronal hace que estas derivadas **se puedan calcular de forma recursiva**.

Algoritmo de retropropagación

- Vamos a calcular las derivadas para un ejemplo simple y luego veremos como se calculan de forma general.



Algoritmo de retropropagación

Fase 1: Propagación hacia delante.

- Llamamos out_j^h a la salida de la neurona j en la capa h .
- Dados dos valores x_1 y x_2 de entrada, calcular la salida de cada neurona.
 - Primera capa:

$$out_1^0 = x_1; out_2^0 = x_2$$

- Segunda capa:

$$out_1^1 = \sigma(w_{10}^1 + w_{11}^1 out_1^0 + w_{12}^1 out_2^0) = \sigma(w_{10}^1 + w_{11}^1 x_1 + w_{12}^1 x_2);$$

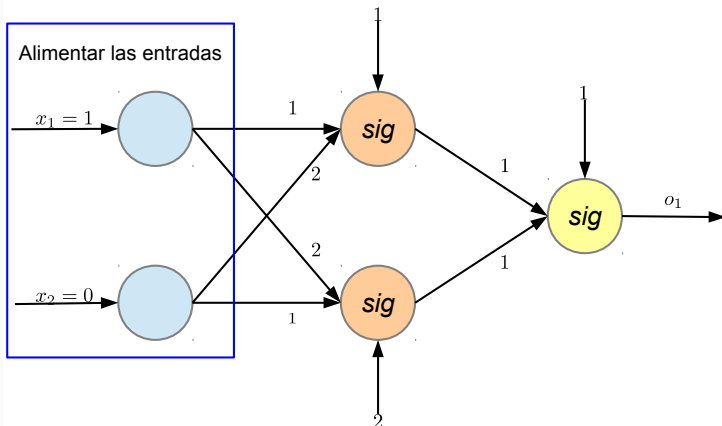
$$out_2^1 = \sigma(w_{20}^1 + w_{21}^1 out_1^0 + w_{22}^1 out_2^0) = \sigma(w_{20}^1 + w_{21}^1 x_1 + w_{22}^1 x_2);$$

- Tercera capa:

$$out_1^2 = o_1 = \sigma(w_{10}^2 + w_{11}^2 out_1^1 + w_{12}^2 out_2^1)$$

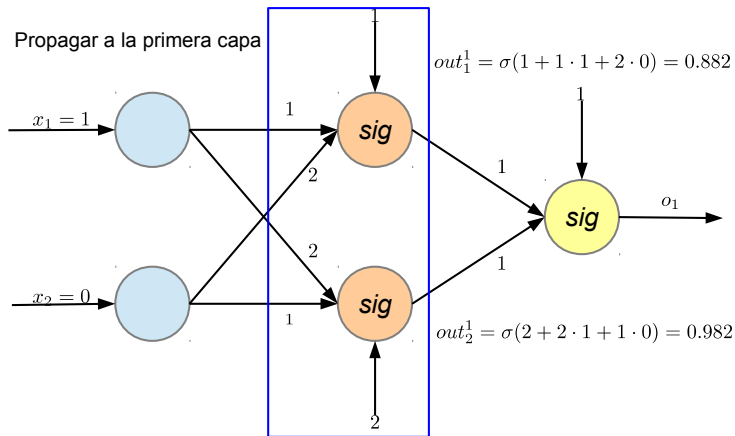
Algoritmo de retropropagación

Fase 1: Propagación hacia delante.



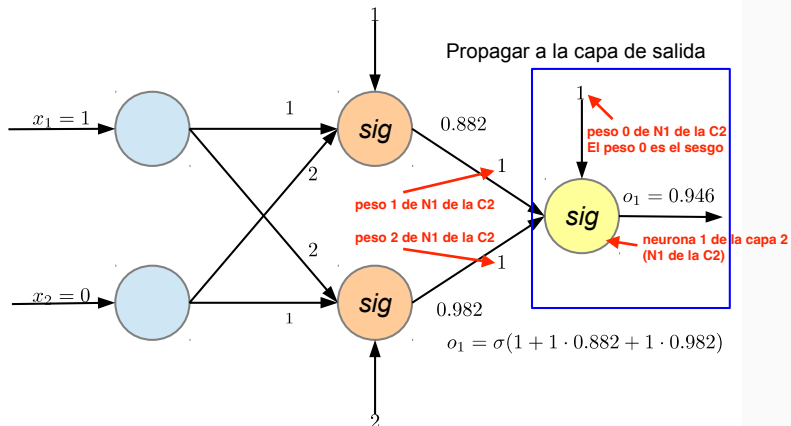
Algoritmo de retropropagación

Fase 1: Propagación hacia delante.



Algoritmo de retropropagación

Fase 1: Propagación hacia delante.



Algoritmo de retropropagación

Fase 2: Cálculo del error y de las derivadas. (Propagación hacia atrás)


- Obtenemos el valor de error cometido por la red:

$$E = (d_1 - o_1)^2$$

- Ahora derivamos ese error respecto a cada uno de los pesos:

$$\nabla E = \left\{ \frac{\partial E}{\partial w_{10}^1}, \frac{\partial E}{\partial w_{11}^1}, \frac{\partial E}{\partial w_{12}^1}, \frac{\partial E}{\partial w_{20}^1}, \frac{\partial E}{\partial w_{21}^1}, \frac{\partial E}{\partial w_{22}^1}, \frac{\partial E}{\partial w_{10}^2}, \frac{\partial E}{\partial w_{11}^2}, \frac{\partial E}{\partial w_{12}^2} \right\}$$

- De la expresión $(d_1 - o_1)^2$, los pesos solo influyen en o_1 (o_1 es una función de cada uno de los pesos). En estos casos, la regla de la cadena nos permite realizar estas derivadas de forma recursiva (lo siguiente se cumple para todos los pesos):

$$\frac{\partial E}{\partial w_{10}^2} = \frac{\partial E}{\partial o_1} \frac{\partial o_1}{\partial w_{10}^2} = -2(d_1 - o_1) \frac{\partial o_1}{\partial w_{10}^2}$$


Algoritmo de retropropagación

- Llamamos net_j^h a la suma ponderada de las entradas de la neurona j en la capa h , es decir, a la salida antes de aplicar la función de activación sigmoide:

$$out_j^h = \sigma(net_j^h)$$

'net' es como la salida pero sin haber aplicado la sigmoide

Algoritmo de retropropagación

- Recordamos:

$$o_1 = \sigma(\text{net}_1^2)$$
$$\sigma(x)' = \sigma(x)(1 - \sigma(x))$$

- Continuamos derivando con respecto a o_1 :

$$\frac{\partial E}{\partial w_{10}^2} = -2(d_1 - o_1) \frac{\partial o_1}{\partial w_{10}^2} = -2(d_1 - o_1) \cdot o_1(1 - o_1) \frac{\partial \text{net}_1^2}{\partial w_{10}^2}$$
$$\frac{\partial E}{\partial w_{11}^2} = -2(d_1 - o_1) \frac{\partial o_1}{\partial w_{11}^2} = -2(d_1 - o_1) \cdot o_1(1 - o_1) \frac{\partial \text{net}_1^2}{\partial w_{11}^2}$$
$$\frac{\partial E}{\partial w_{12}^2} = -2(d_1 - o_1) \frac{\partial o_1}{\partial w_{12}^2} = -2(d_1 - o_1) \cdot o_1(1 - o_1) \frac{\partial \text{net}_1^2}{\partial w_{12}^2}$$

Algoritmo de retropropagación

- Y ahora ya podemos escribir las derivadas completas para los pesos de la capa de salida (w_{1j}^2):

$$\begin{aligned} net_1^2 &= w_{10}^2 + w_{11}^2 out_1^1 + w_{12}^2 out_2^1 \\ \frac{\partial E}{\partial w_{10}^2} &= -2(d_1 - o_1)o_1(1 - o_1)\frac{\partial net_1^2}{\partial w_{10}^2} = \\ &= -2(d_1 - o_1)o_1(1 - o_1)1 \\ \frac{\partial E}{\partial w_{11}^2} &= -2(d_1 - o_1)o_1(1 - o_1)\frac{\partial net_1^2}{\partial w_{11}^2} = \\ &= -2(d_1 - o_1)o_1(1 - o_1)out_1^1 \\ \frac{\partial E}{\partial w_{12}^2} &= -2(d_1 - o_1)o_1(1 - o_1)\frac{\partial net_1^2}{\partial w_{12}^2} = \\ &= -2(d_1 - o_1)o_1(1 - o_1)out_2^1 \end{aligned}$$

Algoritmo de retropropagación

- Pasamos a analizar las derivadas de los pesos de la capa oculta (w_{1i}^1 y w_{2i}^1):
 - Los pesos w_{1i}^1 influyen en out_1^1 .
 - Los pesos w_{2i}^1 influyen en out_2^1 .
- Por tanto, su derivada será similar a las otras derivadas, pero cuando llegemos a $\frac{\partial net_1^2}{\partial w_{ji}^1}$ tendremos que continuar derivando.
- Para el peso w_{10}^1 (sesgo de la primera neurona en la primera capa):

$$\begin{aligned} net_1^2 &= w_{10}^2 + w_{11}^2 out_1^1 + w_{12}^2 out_2^1 \\ \frac{\partial E}{\partial w_{10}^1} &= -2(d_1 - o_1)o_1(1 - o_1)\frac{\partial net_1^2}{\partial w_{10}^1} = \\ &= -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 \frac{\partial out_1^1}{\partial w_{10}^1} \end{aligned}$$

Algoritmo de retropropagación

- Recordamos:

$$out_1^1 = \sigma(net_1^1)$$

$$net_1^1 = w_{10}^1 + w_{11}^1 x_1 + w_{12}^1 x_2$$

$$\sigma(x)' = \sigma(x)(1 - \sigma(x))$$

- Continuamos derivando con respecto a out_1^1 :

$$\begin{aligned}\frac{\partial E}{\partial w_{10}^1} &= -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 \frac{\partial out_1^1}{\partial w_{10}^1} = \\ &= -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 out_1^1(1 - out_1^1) \frac{\partial net_1^1}{\partial w_{10}^1} = \\ &= -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 out_1^1(1 - out_1^1)1\end{aligned}$$

Algoritmo de retropropagación

- Repetimos este proceso para todos los pesos de capa oculta:

$$\frac{\partial E}{\partial w_{10}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 out_1^1(1 - out_1^1)1$$

$$\frac{\partial E}{\partial w_{11}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 out_1^1(1 - out_1^1)x_1$$

$$\frac{\partial E}{\partial w_{12}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{11}^2 out_1^1(1 - out_1^1)x_2$$

$$\frac{\partial E}{\partial w_{20}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{12}^2 out_2^1(1 - out_2^1)1$$

$$\frac{\partial E}{\partial w_{21}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{12}^2 out_2^1(1 - out_2^1)x_1$$

$$\frac{\partial E}{\partial w_{22}^1} = -2(d_1 - o_1)o_1(1 - o_1)w_{12}^2 out_2^1(1 - out_2^1)x_2$$

Problema del Desvanecimiento del Gradiente: si se añaden muchas capas (más de 6 o 7), las derivadas podrían ser muy pequeñas y el modelo no conseguiría aprender nada en las primeras capas.

Si usamos la función 'relu' (rectified linear unit), no habría problema porque para esa función la derivada es 1.

Algoritmo de retropropagación

- Recapitulando:
 - Capa de salida:

$$\frac{\partial E}{\partial w_{ji}^2} = \begin{cases} -2(d_1 - o_1)o_1(1 - o_1)1, & \text{si } i = 0, \\ -2(d_1 - o_1)o_1(1 - o_1)out_i^1, & \text{si } i \neq 0. \end{cases}$$

- Capa oculta:

$$\frac{\partial E}{\partial w_{ji}^1} = \begin{cases} -2(d_1 - o_1)o_1(1 - o_1)w_{1j}^2 out_j^1(1 - out_j^1)1, & \text{si } i = 0, \\ -2(d_1 - o_1)o_1(1 - o_1)w_{1j}^2 out_j^1(1 - out_j^1)x_i, & \text{si } i \neq 0. \end{cases}$$

Algoritmo de retropropagación

- Ojo, hay partes comunes:
 - Capa de salida:

$$\frac{\partial E}{\partial w_{ji}^2} = \begin{cases} -2(d_1 - o_1)o_1(1 - o_1)1, & \text{si } i = 0, \text{ } \rightarrow \text{si es sesgo} \\ -2(d_1 - o_1)o_1(1 - o_1)out_i^1, & \text{si } i \neq 0. \end{cases}$$

- Capa oculta:

$$\frac{\partial E}{\partial w_{ji}^1} = \begin{cases} -2(d_1 - o_1)o_1(1 - o_1)w_{1j}^2out_j^1(1 - out_j^1)1, & \text{si } i = 0, \\ -2(d_1 - o_1)o_1(1 - o_1)w_{1j}^2out_j^1(1 - out_j^1)x_i, & \text{si } i \neq 0. \end{cases}$$

Algoritmo de retropropagación

- Muchas partes son comunes, el cálculo de derivadas se puede realizar de manera recursiva.
- Llamamos δ_j^h a la derivada de la neurona j de la capa oculta h con respecto al error (“cuánta culpa tiene sobre el error esa neurona”).

delta  $\delta_1^2 = -2(d_1 - o_1)o_1(1 - o_1)$

$$\delta_1^1 = w_{11}^2 \delta_1^2 out_1^1 (1 - out_1^1)$$

$$\delta_2^1 = w_{12}^2 \delta_1^2 out_2^1 (1 - out_2^1)$$

- A efectos de la actualización de pesos, la constante (2) puede obviarse.

Algoritmo de retropropagación

- Redefinimos las derivadas en función de estos valores δ_j^h :
(delta)
 - Capa de salida:

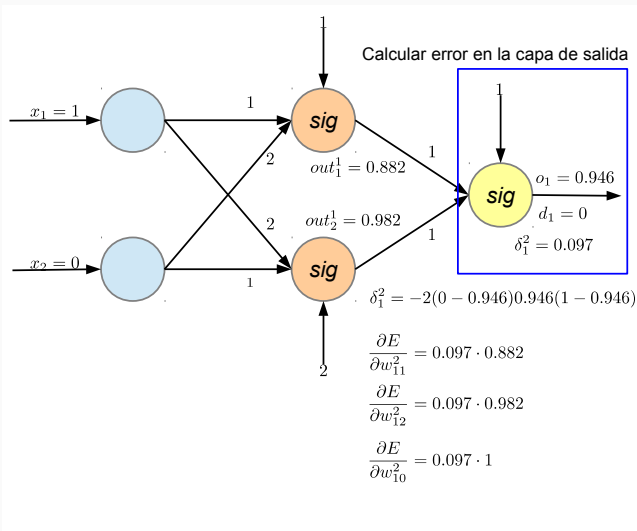
$$\frac{\partial E}{\partial w_{ji}^2} = \begin{cases} \delta_j^2 1, & \text{si } i = 0, \\ \delta_j^2 out_i^1, & \text{si } i \neq 0. \end{cases}$$

- Capa oculta:

$$\frac{\partial E}{\partial w_{ji}^1} = \begin{cases} \delta_j^1 1, & \text{si } i = 0, \\ \delta_j^1 x_i, & \text{si } i \neq 0. \end{cases}$$

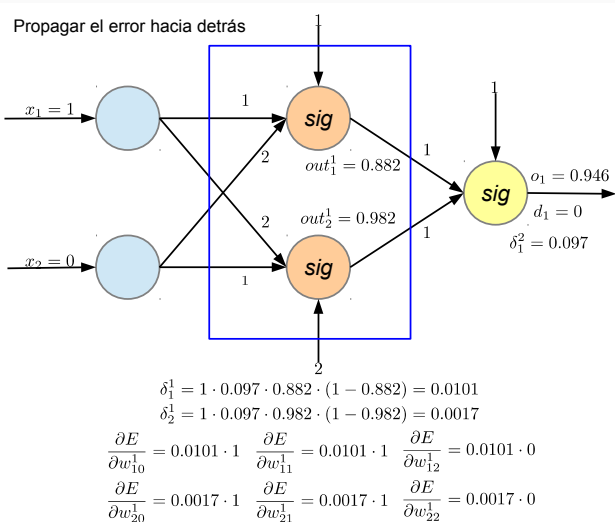
Algoritmo de retropropagación

Fase 2: Propagación hacia atrás.



Algoritmo de retropropagación

Fase 2: Propagación hacia atrás.

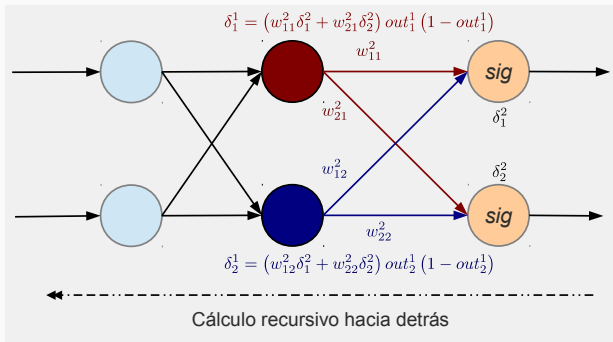


Algoritmo de retropropagación

Fase 2: Propagación hacia atrás.

Si hay varias neuronas en la siguiente capa, cada δ_j^h recibe su valor a partir de las neuronas con las que esté conectado:

$$\delta_j^h \leftarrow \left(\sum_{i=1}^{n_{h+1}} w_{ij}^{h+1} \delta_i^{h+1} \right) \cdot out_j^h \cdot (1 - out_j^h)$$



Algoritmo de retropropagación

Fase 3: Actualización de pesos.

- Una vez hemos obtenido el vector gradiente, debemos actualizar los pesos.
- Se utiliza el propio valor de la derivada (sobre todo por su signo), multiplicado por una constante *eta* (η) que controla que los pasos que se van dando no sean muy pequeños o muy grandes (*tasa de aprendizaje*).
- Fórmula general:

$$w_{ji}^h = w_{ji}^h - \eta \Delta w_{ji}^h$$

$$\Delta w_{ji}^h = \frac{\partial E}{\partial w_{ji}^h} = \begin{cases} \delta_j^h \cdot 1, & \text{si } i = 0 \\ \delta_j^h \cdot out_i^{h-1}, & \text{si } i \neq 0 \end{cases}$$

Algoritmo de retropropagación

Fase 3: Actualización de pesos.

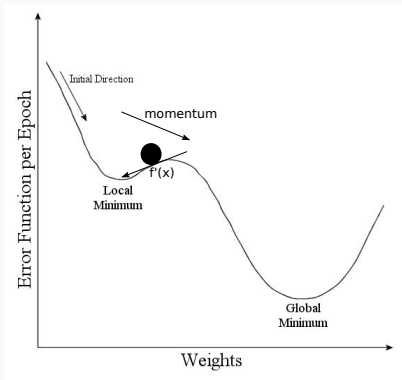
- Ajustar el valor de η es difícil:
 - Si es demasiado grande, podemos provocar oscilaciones.
 - Si es demasiado pequeño, necesitaremos muchas iteraciones.
- Utilizaremos el concepto de **momento**, para mejorar la convergencia:
 - Los cambios anteriores deberían influir en la dirección de movimiento actual:

$$w_{ji}^h = w_{ji}^h - \eta \Delta w_{ji}^h - \mu (\eta \Delta w_{ji}^h (t - 1))$$

- Así, los pesos que empiezan a moverse en una determinada dirección, tienden a moverse en esa dirección.
- El parámetro μ (μ) controla el efecto del momento.

Algoritmo de retropropagación

- La idea es que en un ejemplo como el siguiente, el momento haga que se pueda escapar del óptimo local.
- Pese a que la derivada te diga que vayas a la izquierda, la “*inercia*” puede llevarte a seguir para la derecha:



Algoritmo de retropropagación

- Hasta ahora, hemos visto como ajustar los pesos según el error cometido en un solo patrón.
- Existen diferentes formas de adaptar la red de acuerdo con la forma en que se use la información de entrenamiento (patrones).
 - Aprendizaje **Off-line** (*batch*):
 - Cada vez que actualizamos los pesos de las conexiones del modelo, consideramos todos los patrones de entrenamiento.
 - Usualmente llamado época (*epoch*).
 - Costoso si hay muchos datos
 - Aprendizaje **On-line**:
 - Por cada patrón de entrenamiento, realizamos una actualización de pesos.
 - Problema de “olvido” de patrones “viejos”.
 - Métodos intermedios \Rightarrow adaptación cada p patrones (en mini lotes, *mini batches*).
 - \Rightarrow Problema: hay que definir el tamaño de *batch*.

Algoritmo de retropropagación

Algoritmo de retropropagación *on-line*

Inicio

1. $w_{ji}^h \leftarrow U[-1, 1]$ // Aleatorios entre -1 y $+1$
2. **Repetir**
 - 2.1 **Para** cada patrón con entradas \mathbf{x} , y salidas \mathbf{d}
 - 2.1.1 $\Delta w_{ji}^h \leftarrow 0$ // Se aplicarán cambios por cada patrón
 - 2.1.2 $out_j^0 \leftarrow x_j$ // Alimentar entradas
 - 2.1.3 forwardPropagation() // Propagar las entradas ($\Rightarrow \Rightarrow$)
 - 2.1.4 backPropagation() // Retropropagar el error ($\Leftarrow \Leftarrow$)
 - 2.1.5 accumulateChange() // Calcular ajuste de pesos
 - 2.1.6 weightAdjustment() // Aplicar el ajuste calculado
 - Fin Para**
- Hasta** (CondicionParada)
3. **Devolver** matrices de pesos.

Fin

Algoritmo de retropropagación

Algoritmo de retropropagación *off-line*

Inicio

1. $w_{ji}^h \leftarrow U[-1, 1]$ // Aleatorios entre -1 y $+1$
2. **Repetir**
 - 2.1 $\Delta w_{ji}^h \leftarrow 0$ // Se aplicarán cambios al final
 - 2.2 **Para** cada patrón con entradas \mathbf{x} , y salidas \mathbf{d}
 - 2.2.1 $out_j^0 \leftarrow x_j$ // Alimentar entradas
 - 2.2.2 forwardPropagation() // Propagar las entradas ($\Rightarrow \Rightarrow$)
 - 2.2.3 backPropagation() // Retropropagar el error ($\Leftarrow \Leftarrow$)
 - 2.2.4 accumulateChange() // Acumular ajuste de pesos

Fin Para

- 2.3 weightAdjustment() // Aplicar el ajuste calculado

Hasta (CondicionParada)

3. **Devolver** matrices de pesos.

Fin

Algoritmo de retropropagación: funciones

Función de entrada: suma ponderada.

Función de activación: $g(x)$.

- Sigmoide: $g(x) = \frac{1}{1+\exp(-x)}$, $g'(x) = g(x) (1 - g(x))$.
- Tangente hiperbólica: $g(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$,
 $g'(x) = (1 - (g(x))^2)$ **Demostrar**.

Algoritmo de retropropagación: funciones

forwardPropagation()

Inicio

1. **Para** h de 1 a H // *Para cada capa ($\Rightarrow \Rightarrow$)*

1.1 **Para** j de 1 a n_h // *Para cada neurona de la capa h*

1.1.1 $net_j^h \leftarrow w_{j0}^h + \sum_{i=1}^{n_{h-1}} w_{ji}^h out_i^{h-1}$

1.1.2 $out_j^h \leftarrow g(net_j^h)$

Fin Para

Fin Para

Fin

Algoritmo de retropropagación: funciones

backPropagation()

Inicio

1. **Para** j de 1 a n_H // Para cada neurona de salida
 - 1.1 $\delta_j^H \leftarrow -(d_j - out_j^H) \cdot g'(net_j^H)$ // Hemos obviado la constante (2), ya que el resultado será el mismo

Fin Para

2. **Para** h de $H - 1$ a 1 // Para cada capa ($\Leftarrow \Leftarrow$)
 - 2.1 **Para** j de 1 a n_h // Para cada neurona de la capa h
 - 2.1.1 $\delta_j^h \leftarrow (\sum_{i=1}^{n_{h+1}} w_{ij}^{h+1} \delta_i^{h+1}) \cdot g'(net_j^h)$ // Pasa por todas las neuronas de la capa $h + 1$ conectadas con j

Fin Para

Fin Para

Fin

Algoritmo de retropropagación: funciones

acummulateChange()

Inicio

1. **Para** h de 1 a H // *Para cada capa ($\Rightarrow \Rightarrow$)*
 - 1.1 **Para** j de 1 a n_h // *Para cada neurona de la capa h*
 - 1.1.1 **Para** i de 1 a n_{h-1} // *Para cada neurona de la capa $h - 1$*
$$\Delta w_{ji}^h \leftarrow \Delta w_{ji}^h + \delta_j^h \cdot out_i^{h-1}$$

Fin Para
 - 1.1.2 $\Delta w_{j0}^h \leftarrow \Delta w_{j0}^h + \delta_j^h \cdot 1$ // *Sesgo*

Fin Para

Fin Para

Fin

Algoritmo de retropropagación: funciones

weightAdjustment()

Inicio

1. **Para** h de 1 a H // *Para cada capa ($\Rightarrow\Rightarrow$)*
 - 1.1 **Para** j de 1 a n_h // *Para cada neurona de la capa h*
 - 1.1.1 **Para** i de 1 a n_{h-1} // *Para cada neurona de la capa $h - 1$*
$$w_{ji}^h \leftarrow w_{ji}^h - \eta \Delta w_{ji}^h - \mu (\eta \Delta w_{ji}^h (t - 1))$$

Fin Para
 - 1.1.2 $w_{j0}^h \leftarrow w_{j0}^h - \eta \Delta w_{j0}^h - \mu (\eta \Delta w_{j0}^h (t - 1))$ // *Sesgo*

Fin Para

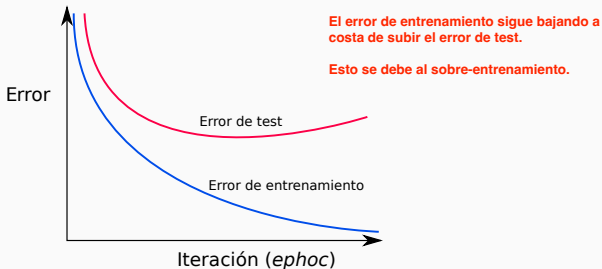
Fin Para

Fin

Algoritmo de retropropagación: problemas

- La complejidad del algoritmo es polinomial en función del número de pesos de la red.
- La **arquitectura de la red** (número de capas y número de nodos en cada capa), junto con la **tipología de los nodos** (tipos de funciones de activación a considerar), son parámetros decisivos del algoritmo que hay que buscar por prueba y error o por validación cruzada.
- La inicialización de los pesos, puede llevar a que el algoritmo quede atrapado en mínimos locales.
 - En la versión *online*, podemos aleatorizar el orden de presentación de los patrones.
 - Podemos introducir ruido en los patrones de entrenamiento (además previene contra el sobre-entrenamiento).

Algoritmo de retropropagación: problemas



- El sobre-entrenamiento en redes neuronales suele venir provocado por dos causas:
 - Entrenamiento demasiado largo (condición de parada inadecuada).
 - Redes demasiado complejas (muchas neuronas o muchas capas).

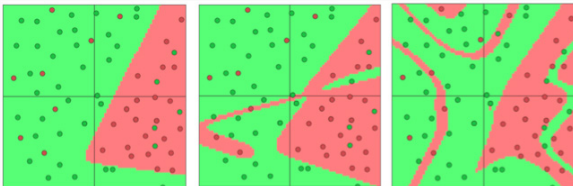
Los modelos tienen que llegar a un compromiso entre buena varianza y buen sesgo para no provocar sobre-entrenamiento

Algoritmo de retropropagación: problemas

Una capa con 3, 6 y 20 neuronas:



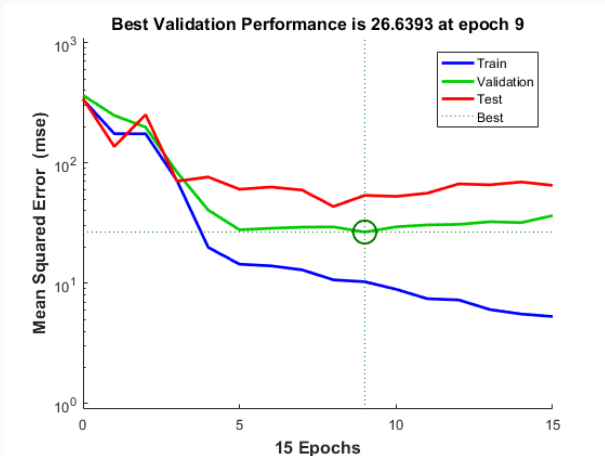
Una, dos y cuatro capas, con 3 neuronas cada una:



Algoritmo de retropropagación: problemas

Early stopping: mecanismo para detectar cuando se sobre-entrena.

(parada temprana)



Algoritmo de retropropagación: problemas


Early stopping: mecanismo para detectar cuando se sobre-entrena.

1. Dividimos los datos en tres partes: **entrenamiento** (p.ej. 60 %), **validación** (p.ej. 20 %) y **test** (p.ej. 20 %).
2. Ajustamos los pesos con el conjunto de **entrenamiento**.
3. Evaluamos la red en el conjunto de **validación**.
4. Si el error de **validación** baja en un valor mayor que t (tolerancia), volvemos al paso 2. En caso contrario, paramos el entrenamiento.
5. Evaluar el modelo final sobre el conjunto de **test**.

Algoritmo de retropropagación: problemas

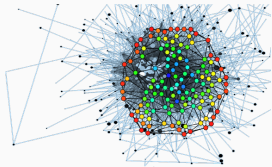
Regularización: mecanismo para evitar sobre-entrenamiento (minimiza la magnitud de los pesos). Evitar sobreaprendizaje: intentar mantener pesos bajos

- Regularización L2:

$$E = \frac{1}{J} \sum_{i=1}^J (d_i - o_i)^2 + \lambda \sum_h \sum_j \sum_i (w_{ji}^h)^2 \quad (3)$$


- La modificación de las derivadas es directa:

$$\frac{\partial E}{\partial w_{ji}^h} = \begin{cases} \delta_j^h \cdot 1 + 2\lambda w_{ji}^h, & \text{si } i = 0 \\ \delta_j^h \cdot out_i^{h-1} + 2\lambda w_{ji}^h, & \text{si } i \neq 0 \end{cases}$$



Introducción

Neurona artificial

Redes neuronales

Aprendizaje

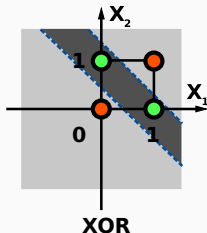
Clasificación

RNAs en Weka

Conclusiones

- **Motivación:** A menudo nos encontramos problemas del mundo real donde el objetivo es “*categorizar*” o “*clasificar*” según un conjunto de características.
- Necesitamos un modelo predictivo que, a partir de una base de datos, sea capaz de obtener el valor de una variable categórica o nominal.
- Por ejemplo:
 - Color de ojos: {azul, verde, marron}.
 - Éxito o fracaso de un tratamiento: {si, no}.
 - Presencia de cáncer en un órgano fotografiado: {si, no}.

- El problema del XOR también es un problema de clasificación:




- En realidad, la clasificación es una tarea intrínsecamente no lineal porque intentamos poner cosas que no son iguales en el mismo grupo, es decir, una diferencia en el vector de entradas no provoca una diferencia en la salida del modelo.

Representación de los valores de clase

- Representación de los valores de clase (color de ojos):

Tipo	Clase azul	Clase verde	Clase marrón
Valores enteros	1	2	3

- Utilizando esta representación, podríamos emplear el perceptrón multicapa de regresión, de forma que la clase predicha sería el entero más cercano al valor predicho por el modelo.
- **Inconvenientes:**
 - Se ha asumido que existe un orden entre las clases.
 - Se ha asumido una distancia entre cada una de las clases.



Hemos dicho que 'azul' es más cercano a 'verde' que a 'marrón'

Representación de los valores de clase

- **Representación 1-de- J** , donde J es el número de clases.
 - Por cada patrón, tendremos un vector de J elementos, donde el elemento i -ésimo será igual a 1 si el patrón pertenece a esa clase y a 0 si no pertenece.
 - Es decir, el vector contendrá 0 en todas las posiciones menos en la posición que corresponde a la clase correcta (en la que habrá un 1).
- Representación de los valores de clase (color de ojos):

Tipo	Clase azul	Clase verde	Clase marrón
1-de- k	$\{1, 0, 0\}$	$\{0, 1, 0\}$	$\{0, 0, 1\}$

- Utilizando esta representación, el perceptrón multicapa modelará cada una de las J variables binarias por separado.
- 1 neurona de salida por clase (modelar tres variables a la vez).

Representación de los valores de clase

- Clase predicha: neurona con el **máximo** valor de salida.
- Si usamos una función sigmoide en la capa de salida, aseguramos que los valores predichos estarán entre 0 y 1.

d_1	d_2	d_3	o_1	o_2	o_3	Clase predicha
0	0	1	0,1	0,2	0,8	3
0	0	1	1,0	0,2	0,8	1
0	1	0	0,2	0,9	0,1	2

Da 1 en lugar de 3 (que es lo que tendría que salir) porque no está mirando lo que pasa en las clases vecinas, solo en la propia.

- **Problema:** inconsistencias, ya que las variables se están modelando de forma independiente.
- **Solución:** incorporar un **significado probabilístico** a las salidas.

Interpretación probabilística

- Si lo pensamos, la representación 1-de- J puede verse como una representación probabilística de una serie de eventos:
 - J clases: $\{C_1, C_2, \dots, C_J\}$.
 - J eventos: el patrón pertenece a cada una de las clases del problema, es decir, " $\mathbf{x} \in C_1$ ", " $\mathbf{x} \in C_2$ ", ..., " $\mathbf{x} \in C_J$ ".
 - La salida deseada (\mathbf{d}) es predecir que el patrón pertenece a la clase correcta con la mayor probabilidad (salida 1-de- J):

$$d_j = \begin{cases} 1 & \text{Si } \mathbf{x} \in C_j \\ 0 & \text{Si } \mathbf{x} \notin C_j \end{cases} \quad (4)$$

- De esta forma, lo que modelamos y predecimos es la probabilidad de pertenecer a cada una de las clases:


$$o_j = \hat{P}(\mathbf{x} \in C_j | \mathbf{x}) \quad (5)$$

La función softmax

- Ahora necesitamos que las salidas de la red neuronal (**o**) sean “consistentes”, desde un punto de vista probabilístico:

$$\sum_{j=1}^J o_j = 1 \quad (6)$$

- Para ello, podemos utilizar la función **softmax** que es una función de **normalización** que asegura que las salidas estarán entre 0 y 1 y que su suma será 1:

suma ponderada 

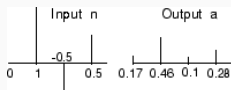
$$net_j^H = w_{j0} + \sum_{i=1}^{n_H-1} w_{ji} out_i^{H-1}, \quad (7)$$

$$out_j^H = \frac{\exp(net_j^H)}{\sum_{l=1}^{n_H} \exp(net_l^H)} \quad (8)$$

La suma de todas las salidas debe ser 1

Normaliza la salida

La función softmax



$$o_j = out_j^H = \frac{\exp(net_j^H)}{\sum_{l=1}^{n_H} \exp(net_l^H)} \quad (9)$$

- Es una aproximación **plausible** (desde el punto de vista biológico) y **derivable** a la función máximo.
- La función exponencial (\exp) asegura que trataremos cantidades positivas y “exagera” mucho las salidas, para que el resultado **se parezca a la función máximo** (un 1 para el máximo y un 0 para el resto).

La función softmax

- El denominador, $\sum_{l=1}^{n_H} \exp(\text{net}_l^H)$, **normaliza** estas cantidades positivas, ya que es la suma de todas ellas. De esta forma, conseguimos que se cumpla el axioma de cálculo de probabilidades:

$$\sum_{j=1}^k o_j = 1 \quad (10)$$

- Por tanto, produce salidas correctas desde el punto de vista probabilístico.
- La clase predicha será el índice de la neurona de salida con valor más alto:

$$C(\mathbf{x}) = \arg \max_j o_j \quad (11)$$

La función softmax

net_1^H	net_2^H	net_3^H	$e^{net_1^H}$	$e^{net_2^H}$	$e^{net_3^H}$	$\sum_{l=1}^{n_H} e^{net_l^H}$
-1	0	3	0,368	1,000	20,086	21,454
1	4	-1	2,718	54,598	0,368	57,684
0,4	0	3	1,492	1,000	20,086	22,578

out_1^H	out_2^H	out_3^H	Clase predicha
0,017	0,047	0,936	3
0,047	0,947	0,006	2
0,066	0,044	0,890	3

La función softmax: derivadas

$$o_j = out_j^H = \frac{\exp(net_j^H)}{\sum_{l=1}^{n_H} \exp(net_l^H)} \quad (12)$$


- Para simplificar, tomemos $net_j = n_j$ y obviemos los límites del sumatorio:

$$o_j = \frac{e^{n_j}}{\sum_l e^{n_l}} \quad (13)$$

- Queremos calcular $\frac{\partial o_j}{\partial n_i}$.
- A la hora de calcular las derivadas, todos los n_l forman parte de la salida de cada neurona.

La función softmax: derivadas

Distinguimos dos casos:

1. $i = j$, es decir, $\frac{\partial o_j}{\partial n_j}$.  El peso influye porque está en una neurona vecina
2. $i \neq j$, es decir, $\frac{\partial o_i}{\partial n_j}$

Primer caso ($i = j$). Derivada de la salida (o_j) respecto a algo que está por detrás de la neurona j :

$$\begin{aligned}\frac{\partial o_j}{\partial n_j} &= \frac{\partial}{\partial n_j} \frac{e^{n_j}}{\sum_l e^{n_l}} = \frac{\partial}{\partial n_j} (\sum_l e^{n_l})^{-1} e^{n_j} = \\ &= \left((-1) (\sum_l e^{n_l})^{-2} e^{n_j} \right) e^{n_j} + (\sum_l e^{n_l})^{-1} e^{n_j} = \\ &= -\frac{(e^{n_j})^2}{(\sum_l e^{n_l})^2} + \frac{e^{n_j}}{\sum_l e^{n_l}} = -o_j^2 + o_j = o_j(1 - o_j)\end{aligned}$$

La función softmax: derivadas

Segundo caso ($i \neq j$). Derivada de la salida (o_j) respecto a algo que está por detrás de cualquier neurona i que no sea j :

$$\begin{aligned}\frac{\partial o_j}{\partial n_i | i \neq j} &= \frac{\partial}{\partial n_i} \frac{e^{n_j}}{\sum_l e^{n_l}} = e^{n_j} \frac{\partial}{\partial n_i} (\sum_l e^{n_l})^{-1} = \\ &= e^{n_j} \left((-1) (\sum_l e^{n_l})^{-2} e^{n_i} \right) = \\ &= - \frac{e^{n_j}}{(\sum_l e^{n_l})} \cdot \frac{e^{n_i}}{(\sum_l e^{n_l})} = -o_j o_i\end{aligned}$$

Se puede resumir ambos del siguiente modo:

$$\frac{\partial o_j}{\partial n_i} = o_j \left(I(i=j) - o_i \right)$$

Comprueba la condición:
si true \rightarrow 1
si false \rightarrow 0

donde $I(cond)$ será 1 si $cond$ es cierto y 0 en caso contrario.

La función softmax: derivadas

- Para una neurona de tipo *softmax*, el valor δ_j^h debe obtenerse sumando las derivadas con respecto a todos los net_j^h :

$$\delta_j^h = \sum_{i=1}^{n_h} out_i^h (I(i=j) - out_i^h) \quad (14)$$

donde out_j^h es la transformación *softmax*:

$$out_j^h = \frac{\exp(net_j^h)}{\sum_{l=1}^{n_h} \exp(net_l^h)} \quad (15)$$

Función de rendimiento: CCR

- En una tarea de clasificación, nuestro objetivo debería ser que el clasificador acertase casi siempre la clase.
- *Correctly Classified Ratio* o porcentaje de patrones bien clasificados:

$$CCR = 100 \times \frac{1}{N} \sum_{p=1}^N (I(y_p = y_p^*)) \quad (16)$$

- N : número de patrones.
- y_p : clase deseada para el patrón p , $y_p = \arg \max_o d_{po}$.
 - Índice del valor máximo del vector \mathbf{d}_p .
- y_p^* clase obtenida para el patrón p , $y_p^* = \arg \max_o o_{po}$.
 - Índice del valor máximo del vector \mathbf{o}_p o la neurona de salida que obtiene la máxima probabilidad para el patrón p .

Función de rendimiento: *CCR*

- Podríamos entrenar el perceptrón intentando maximizar esta cantidad, pero tiene un problema:
 - Para obtener y_p e y_p^* hay que aplicar la función **arg máx** que es **no derivable**.
 - Además, el *CCR* mejora a saltos, lo cual no nos permitiría ir ajustando poco a poco los pesos \Rightarrow **Convergencia difícil**.
- Se puede utilizar, de nuevo, el *MSE* como función de error (a minimizar) tomando la codificación 1-de- J para las salidas y las probabilidades predichas por la función *softmax*:

$$MSE = \frac{1}{N} \sum_{p=1}^N \left(\frac{1}{J} \sum_{o=1}^J (d_{po} - o_{po})^2 \right) \quad (17)$$

Función de error: entropía cruzada

- El error cuadrático medio (MSE) no es la función natural de error cuando tenemos salidas probabilísticas, ya que **trata por igual cualquier diferencia de error**.
- Para problemas de clasificación, deberíamos penalizar más los errores cometidos para la clase correcta ($d_j = 1$) que para la incorrecta ($d_j = 0$).
- La entropía cruzada ($-\ln$ verosimilitud) es más adecuada para problemas de clasificación ya que compara las dos distribuciones de probabilidad:

$$L = -\frac{1}{N \cdot J} \sum_{p=1}^N \left(\sum_{o=1}^J d_{po} \ln(o_{po}) \right) \quad (18)$$

Función de error: entropía cruzada

- Dado el conjunto de entrenamiento, entrenar un algoritmo de clasificación minimizando esta función de error supone **estimar los parámetros que maximizan la verosimilitud de mis parámetros** (*Maximum likelihood estimation*).
- La derivada se obtiene de forma similar (para un solo patrón y obviando la constante $\frac{1}{J}$):

$$L = - \sum_{l=1}^J d_l \ln(o_l)$$

$$(\ln(x))' = \frac{1}{x}$$

$$\frac{\partial L}{\partial w_{ji}^h} = - \sum_{l=1}^J \frac{\partial L}{\partial o_l} \frac{\partial o_l}{\partial w_{ji}^h} = - \sum_{l=1}^J \left(\frac{d_l}{o_l} \right) \frac{\partial o_l}{\partial w_{ji}^h}$$

Resumen cálculo de δ_j^h

- Derivadas para neuronas de tipo sigmoide o tanh:

- Capa de salida:

- Error *MSE*:

$$\delta_j^H \leftarrow - (d_j - out_j^H) \cdot g'(net_j^H)$$

- Entropía cruzada:

$$\delta_j^H \leftarrow - (d_j / out_j^H) \cdot g'(net_j^H)$$

- Capas ocultas:

$$\delta_j^h \leftarrow \left(\sum_{i=1}^{n_{h+1}} w_{ij}^{h+1} \delta_i^{h+1} \right) \cdot g'(net_j^h)$$

- Derivadas para neuronas de tipo *softmax*:

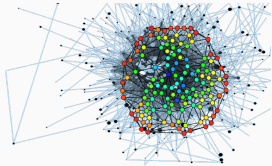
- Capa de salida:

- Error *MSE*:

$$\delta_j^H \leftarrow - \sum_{i=1}^{n_H} ((d_i - out_i^H) \cdot out_j^H (I(i=j) - out_i^H))$$

- Entropía cruzada:

$$\delta_j^H \leftarrow - \sum_{i=1}^{n_H} ((d_i / out_i^H) \cdot out_j^H (I(i=j) - out_i^H))$$



Introducción

Neurona artificial

Redes neuronales

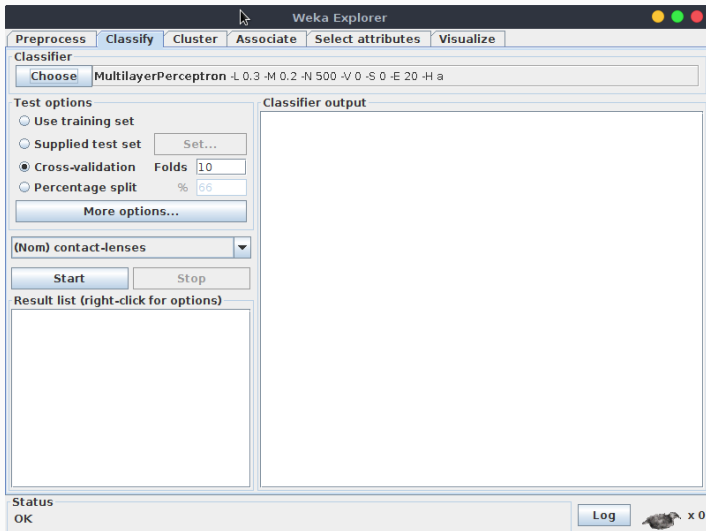
Aprendizaje

Clasificación

RNAs en Weka

Conclusiones

functions → MultilayerPerceptron



Parámetros MultilayerPerceptron

weka.gui.GenericObjectEditor

weka.classifiers.functions.MultilayerPerceptron

About
A Classifier that uses backpropagation to classify instances.

[More](#)
[Capabilities](#)

GUI ☐

autoBuild ☒

debug ☐

decay ☐

doNotCheckCapabilities ☐

hiddenLayers

learningRate

momentum

nominalToBinaryFilter ☒

normalizeAttributes ☒

normalizeNumericClass ☒

reset ☒

seed

trainingTime

validationSetSize

validationThreshold

[Open...](#) [Save...](#) [OK](#) [Cancel](#)

Parámetros MultilayerPerceptron

- GUI: permite usar una interfaz interactiva en la construcción de la red (si usamos el GUI, autoBuild nos construye la red).
- decay: decrementa el factor de aprendizaje conforme avanzan las épocas.
- hiddenLayers: arquitectura de la red.
 - 20: una capa oculta de 20 neuronas.
 - 4,4: dos capas ocultas de 4 neuronas.
 - Algunas heurísticas incluidas en Weka:
 - a: $(\text{attrs} + \text{classes}) / 2$.
 - t: $\text{attrs} + \text{classes}$.
- learningRate: tasa de aprendizaje (η).
- momentum: factor de momento (μ).
- nominalToBinaryFilter: convertir atributos nominales en binarios.
- normalizeAttributes: normalizar en el intervalo $[-1, 1]$.

Parámetros MultilayerPerceptron

- `normalizeNumericClass`: para problemas de regresión, normalizar la variable a predecir (puede ayudar).
- `reset`: evitar divergencia. Si durante el proceso de aprendizaje, las respuestas divergen demasiado de los valores estimados, entonces reseteamos el algoritmo bajando el valor de la tasa de aprendizaje.
- `seed`: semilla números aleatorios.
- `trainingTime`: número máximo de épocas.
- `validationSetSize`: si es 0, no usamos *early stopping*. Sino, este valor indica el porcentaje de la base de datos que utilizaremos como conjunto de validación en el *early stopping*.
- `validationThreshold`: durante cuantas iteraciones tiene que subir el error de validación para que paremos el algoritmo.



Introducción

Neurona artificial

Redes neuronales

Aprendizaje

Clasificación

RNAs en Weka

Conclusiones

Conclusiones

- Ventajas:
 - Habitualmente gran tasa de acierto en la predicción.
 - Robustez ante la presencia de errores (ruido, outliers, ...).
 - Gran versatilidad: salida nominal, numérica, vectores...
 - Eficiencia (rapidez) en la evaluación de nuevos casos.
 - Mejoran su rendimiento mediante aprendizaje y éste puede continuar después de que se haya aplicado al conjunto de entrenamiento.
- Desventajas:
 - Necesitan mucho tiempo para el entrenamiento.
 - Muchos parámetros. El entrenamiento es, en gran parte, ensayo y error (arquitectura, tasas de aprendizaje, momento).
 - Poca interpretabilidad del modelo (modelos de caja negra).
 - Es difícil de incorporar conocimiento previo del dominio.
 - Los atributos de entrada deben ser numéricos.
 - Pueden producir sobreaprendizaje.

Conclusiones: un poco de historia

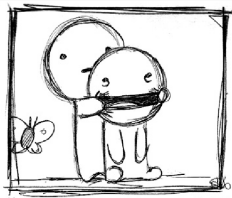
- Año 58: surgimiento del perceptrón simple, entusiasmo inicial.
- Años 70: abandono de las redes (limitaciones con problemas complejos).
- Años 80-90: Boom de las redes neuronales gracias al perceptrón multicapa y al algoritmo de retropropagación, utilización en multitud de campos.
- Años 90-2010: su uso se vio desbancado por otros modelos (árboles de decisión, máquinas de vectores soporte). Razones:
 - Proceso de aprendizaje costoso (en CPU y en tiempo de experto).
 - Modelos pocos interpretables.
 - No era posible entrenar redes de muchas capas (desvanecimiento del gradiente al aplicar la regla de la cadena).
- Años 2010-Actualidad: resurgimiento de las redes neuronales, *“Deep learning”*.

Conclusiones: deep learning

- Conjunto de técnicas que permiten usar redes neuronales de muchas capas y con muchas neuronas.
- Aplicaciones muy importantes en reconocimiento de objetos y en clasificación de imágenes.
- Impulsado también por la disponibilidad de recursos computacionales y el uso de arquitectura hardware específicas (basadas en GPU).
- Ejemplo: GoogleNet Inception v4 (arquitectura convolucional, adaptada para imágenes):
 - Red neuronal entrenada con 1.28 millones de imágenes para distinguir entre 1000 categorías de objetos.
 - Más de 60 millones de pesos en la red.

¿Preguntas?

¡Gracias!





Christopher M. Bishop.

Pattern Recognition and Machine Learning.

Springer, 1st ed. 2006. corr. 2nd printing edition, August 2007.



Fernando Berzal.

Redes Neuronales & Deep Learning.

Edición independiente, 1 edition, 2018.



Christopher M. Bishop.

Neural Networks for Pattern Recognition.

Oxford University Press, USA, 1 edition, January 1996.