Tema 4

Redes Neuronales

Gonzalo A. Aranda-Corral

Ciencias de la Computación e Inteligencia Artificial
Universidad de Huelva

noviembre 2020

Asistencia

Por favor, regístrar vuestra asistencia en la siguiente dirección:

`http://opendatalab.uhu.es/aplicaciones/asistencia`

Índice

Inspiración Biológica

- Se basa en emplear **analogías con sistemas naturales o sociales** para la resolución de problemas.
- Los algoritmos **bioinspirados** simulan el comportamiento de sistemas naturales para el diseño de **métodos heurísticos no determinísticos** de “búsqueda ”/aprendizaje/ ”comportamiento ”, ...

Inspiración Biológica

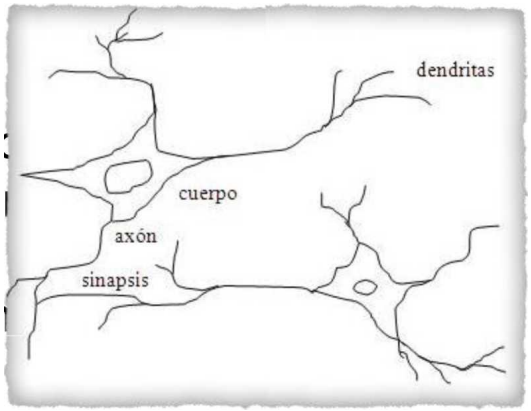
- Modelan (de forma aproximada) un fenómeno existente en la naturaleza.
- Son (en alguna forma) no determinísticos.
- A menudo presentan, implícitamente, una estructura paralela.
- Son adaptativos.
- Un ejemplo son las redes neuronales artificiales como un modelo computacional

Inspiración Biológica

- Cerebro humano: red de neuronas interconectadas.
  - Aproximadamente 10^{11} neuronas con 10^4 conexiones cada una. Las neuronas son lentas, comparadas con los ordenadores:
  - 10^{-3}s para activarse/desactivarse
- Sin embargo, los humanos hacen algunas tareas mucho mejor que los ordenadores
  - p.ej., en 10^{-1} segundos uno puede reconocer visualmente a su madre
- La clave: **paralelismo masivo**

Inspiración Biológica

- La neurona es una célula que recibe señales electromagnéticas a través de las sinapsis de las dendritas.



- Si la acumulación de estímulos recibidos supera cierto umbral, la neurona se dispara.
- Esto es, emite a través del axón un impulso que será recibido por otras neuronas
- Aprender significa (biológicamente) potenciar o debilitar algunas conexiones.

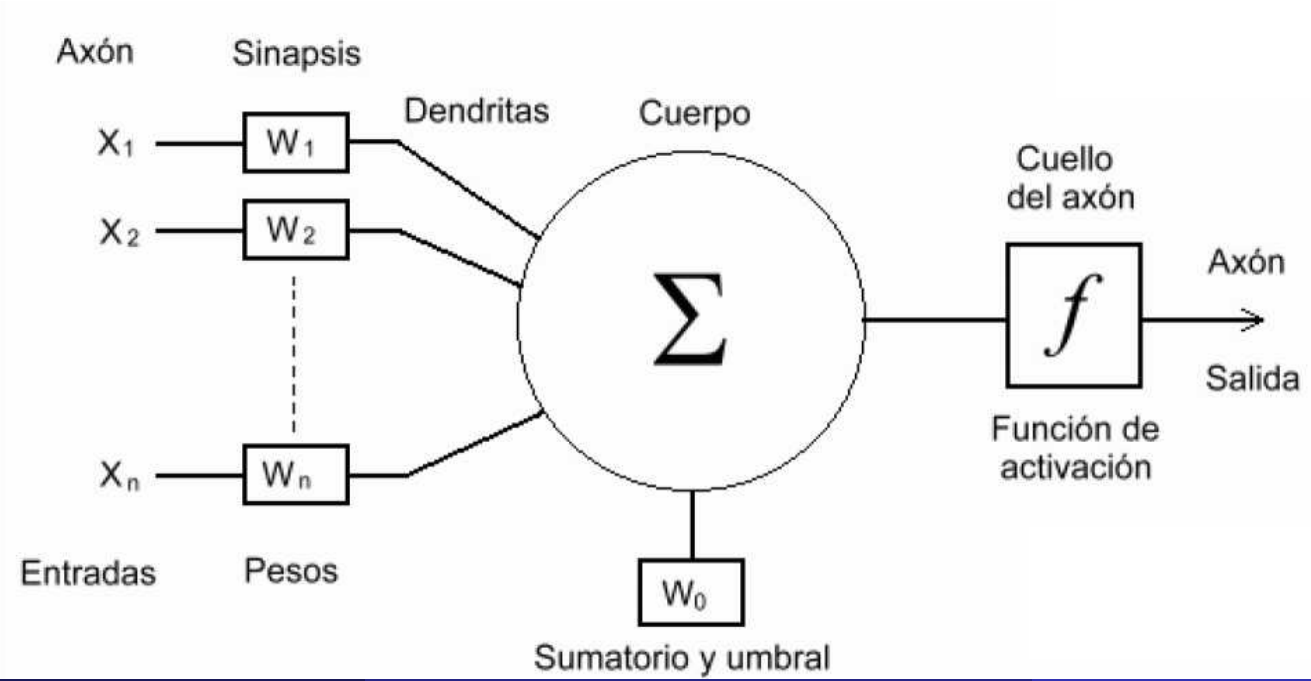
Índice

Modelo formal

- Algunas características de los sistemas biológicos no están reflejadas en el modelo computacional y viceversa
- Debemos formalizar tanto la neurona artificial como su estructura de red

Modelo formal

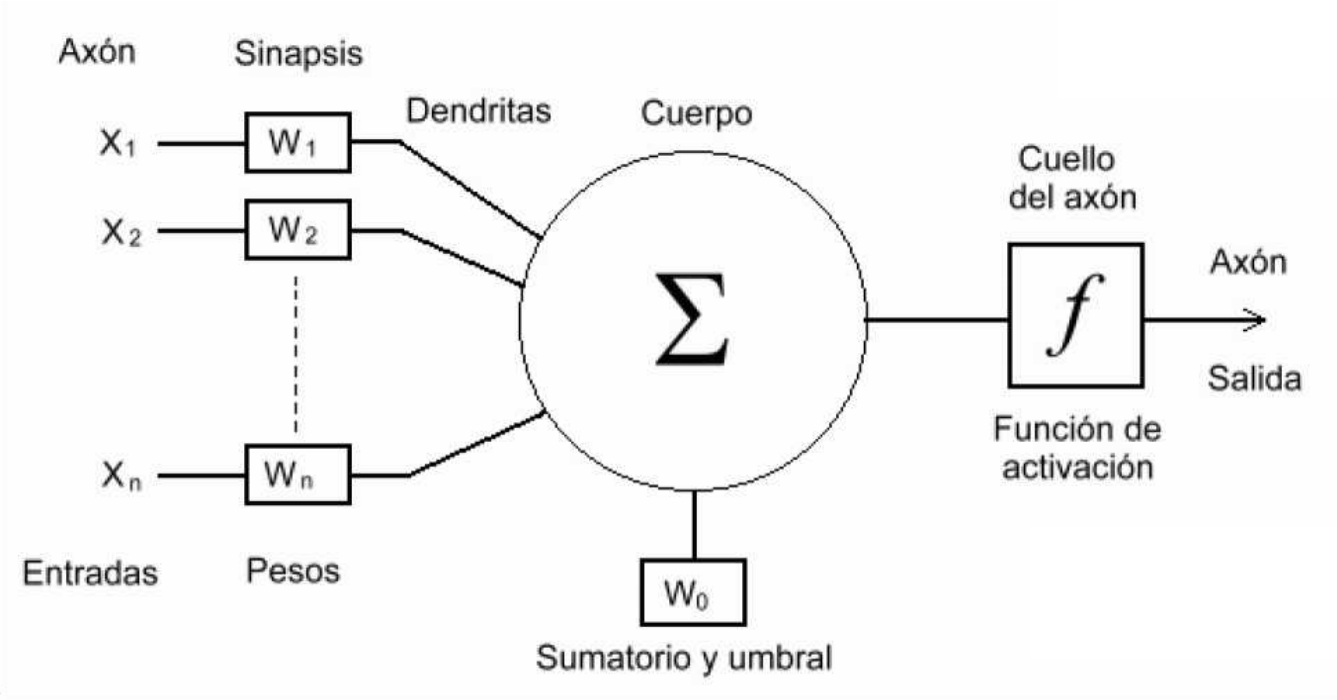
- Su funcionamiento está basado en el de la neurona natural, explicado anteriormente:
  - Entra una cantidad de impulso o información
  - Si sobrepasa un cierto límite de activación, se dispara.
  - La información sale modulada hacia las neuronas posteriores



Modelo formal

- La función de salida es la activación de la neurona en función de las entradas:

$$Salida = f\left(\sum_{i=0}^{n} w_i x_i\right)$$



Modelo formal

- La función de salida es la activación de la neurona en función de las entradas:

$$Salida = f\left(\sum_{i=0}^{n} w_i x_i\right)$$

- donde:
  - f es la función de **activación**
  - El **sumatorio** es sobre todas las entradas (atributos), incluida la x_0 $(= -1)$
  - El peso de la entrada 0 se denomina **umbral** (w_0)

Modelo formal

- El umbral, w_0, se interpreta como la cantidad de impulso que tiene que entrar para activar la neurona.
- La función de activación se puede usar para normalizar la salida (normalmente a “1”) cuando el umbral se supera.
- También provoca que el comportamiento de la neurona pueda no ser lineal

Modelo formal

- Funciones de activación más usuales:

$$f(x) = \begin{cases} x_- & x < B \\ x_+ & x \geq B \end{cases}$$

  - Función signo o bipolar: $B = 0$, $x_- = -1$, $x_+ = 1$
  - Función umbral: $B = 0$, $x_- = 0$, $x_+ = 1$

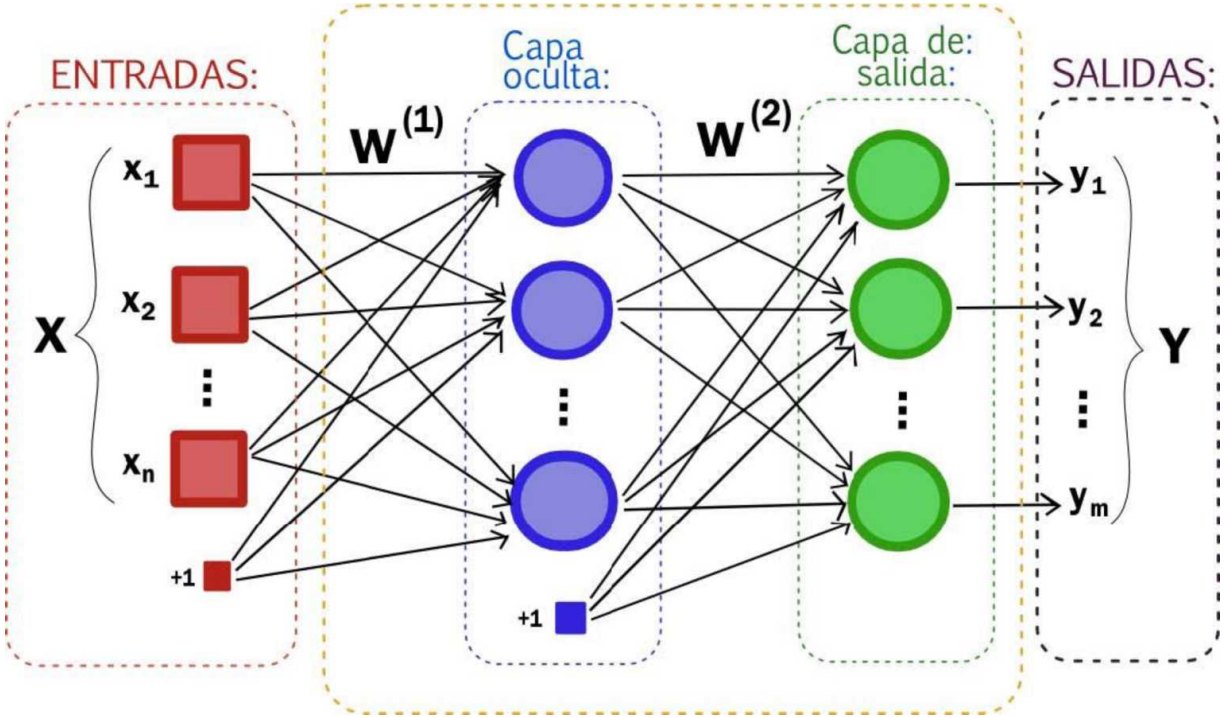
- Función sigmoide: $f(x) = \dfrac{1}{1 + e^{-\beta x}}$

Modelo formal: Estructura de red

- Una red está basada en una estructura de grafo (dirigido).
- Los nodos son neuronas artificiales.
- Los arcos (dirigidos) son las conexiones entre las neuronas
- arco $j \rightarrow i$ sirve para propagar la salida de la unidad j a la entrada de la unidad i.
- Cada arco tiene asociado un peso numérico w que determina la fuerza y el signo de la conexión

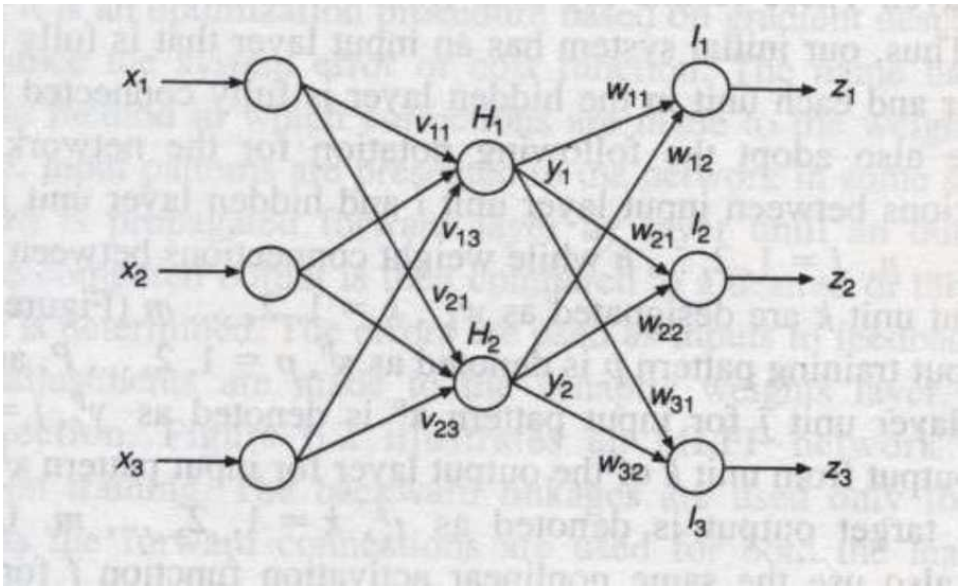
Modelo formal: Estructura de red

RED NEURONAL



Redes hacia adelante

- Cuando el grafo que representa a la red es acíclico, la red se denomina hacia adelante (las que trataremos en este tema)



Redes hacia adelante

- Las unidades en una red hacia adelante suelen estructurarse en capas, tal que cada capa recibe sus entradas de unidades de la capa inmediatamente anterior
- Capa de entrada, capas ocultas y capa de salida
- Hablamos entonces de redes multicapa
- Otras arquitecturas: redes recurrentes, en la que las unidades de salida retroalimentan a las de entrada

Índice

Redes neuronales “clasificadoras”

- Una red neuronal hacia adelante con n unidades en la capa de entrada y m unidades en la capa de salida no es más que una función

$$f :: R^n \rightarrow R^m$$

- Para clasificación binaria, tomar m=1 ... y:
  - Si se tienen funciones de activación umbral o bipolar, considerar un valor de salida (el 1, por ejemplo) como “SI” y el otro como “NO” (0 o -1)
  - Si se usa el sigmoide, considerar un valor de salida por encima de 0.5 como “SI” y un valor por debajo como “NO”

Redes neuronales “clasificadoras”

- También puede usarse como clasificador multiclase.
- Cada unidad de salida corresponde con un valor de clasificación de una clase
- Se interpreta que la unidad con mayor salida es la que indica el valor de clasificación

Aprendizaje

- Pero... ¿qué significa aprender... para redes neuronales artificiales?
  - Encontrar los pesos de las conexiones entre las unidades, y...
  - Encontrar la estructura de red adecuada (No es objetivo de este curso)
- de manera que la red se comporte adecuadamente.

Aprendizaje

- Específicamente, para redes neuronales hacía adelante, es habitual plantear la siguiente tarea de aprendizaje supervisado

Dado un conjunto de entrenamiento:

- $D = \{(x_d, y_d) : x_d \in R^n, y_d \in R^m, d = 1, \ldots, m\}$
- Y una red neuronal de la que sólo conocemos su estructura (capas y número de unidades en cada capa)
- Encontrar un conjunto de pesos w_{ij} tal que la función de R^n en R^m que la red representa se ajuste **lo mejor posible** a los ejemplos del conjunto de entrenamiento

Aplicaciones Prácticas

- Para problemas que se pueden expresar **numéricamente** (discretos o continuos)
- Se suelen utilizar en dominios en los que el volumen de datos es muy alto y puede presentar **ruido**: cámaras, micrófonos, imágenes digitalizadas, etc...
- En los que **interesa la solución**, pero **no el por qué** de la misma

Aplicaciones Prácticas

- Problemas en los que es asumible que se necesite previamente un tiempo **largo de entrenamiento** de la red
- Y en los que se requieren **tiempos cortos para evaluar** una nueva instancia.

ALVINN

- RNA entrenada para conducir un vehículo, a 70 km/h, en función de la percepción visual que recibe de unos sensores
- **Entrada** a la red: La imagen de la carretera digitalizada como un array de 30×32 pixels.
  Es decir, 960 datos de entrada.
- **Salida** de la red: Indicación sobre hacia dónde torcer el volante, codificada en la forma de un vector de 30 componentes (desde girar totalmente a la izquierda, pasando por seguir recto, hasta girar totalmente a la derecha)
- **Estructura**: una red hacia adelante, con una capa de entrada con 960 unidades, una capa oculta de 4 unidades y una capa de salida con 30 unidades

Algunos videos

- https://www.youtube.com/watch?v=H0igiP6Hg1k
- https://www.youtube.com/watch?v=0Str0Rdkxxo
- https://www.youtube.com/watch?v=mW6Y_tiiNYM
- http://www.youtube.com/watch?v=Ue_x1qzMmjw
- https://www.youtube.com/watch?v=YxHcJTs2Sxk
- DARPA: https://www.youtube.com/watch?v=PXQlpu8Y4fI

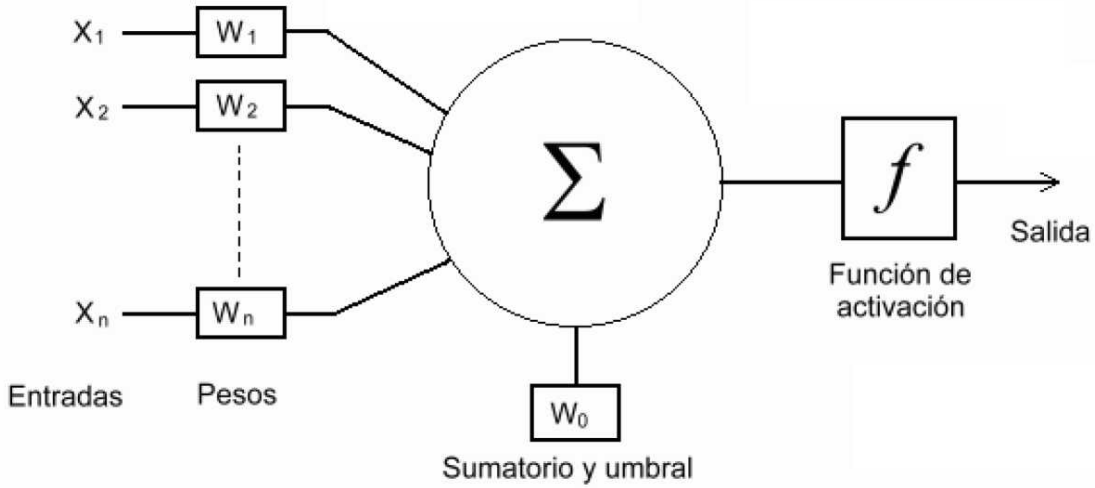
Ejemplos de aplicaciones

- Clasificación
- Reconocimiento de patrones
- Optimización
- Predicción: climatológica, de audiencias, etc
- Reconocimiento de voz
- Visión artificial, reconocimiento de imágenes
- Control, de robots, vehículos, etc
- Compresión de datos
- Diagnosis...

Índice

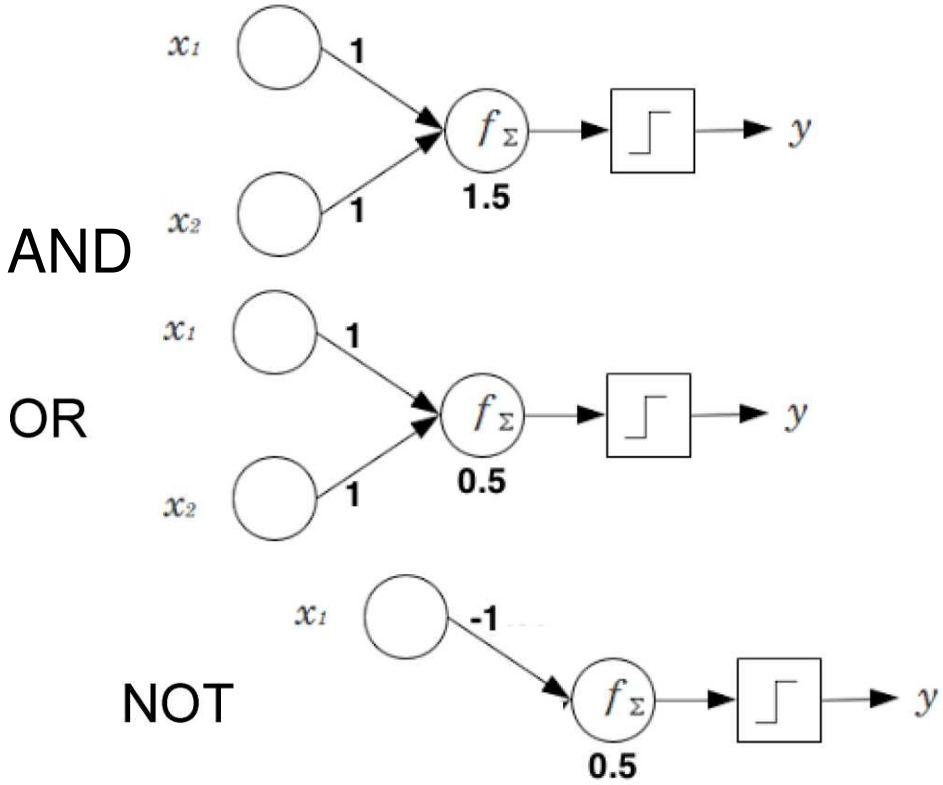
Perceptrón

- Empezamos estudiando el caso más simple de red neuronal: sólo una capa (de entrada y salida) con una sola neurona
- Este tipo de red se denomina **perceptrón**



Perceptrón

- Con un perceptrón de función de activación umbral es posible representar las funciones booleanas básicas:



Clasificador

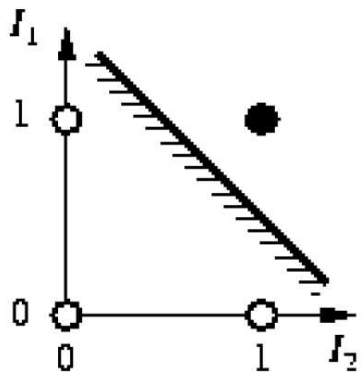
- Un perceptrón con “n” unidades de entrada, pesos $w_i (i = 0, ..., n)$ y función de activación **umbral** (o **bipolar**), clasifica como **positivos** a aquellos $(-1, x_1, \ldots, x_n)$ tal que

$$\sum_{i=0}^{n} w_i x_i > 0$$

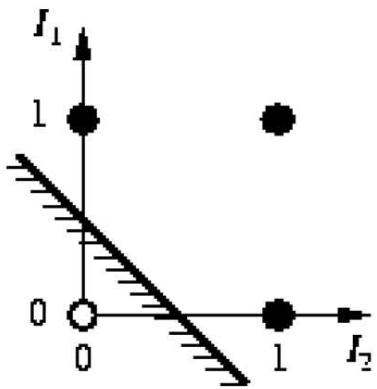
- La ecuación $\sum_{i=0}^{n} w_i x_i = 0$ representa un hiperplano en $\mathbb{R}^n$
- Es decir, una función booleana (o cualquiera) sólo podrá ser representada por un perceptrón umbral si existe un hiperplano que separa los elementos con valor 1 de los elementos con valor 0 (**linealmente separable**)

Limitaciones

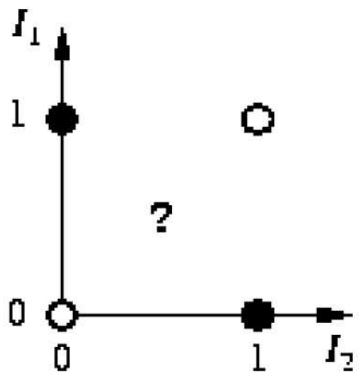
- A pesar de sus limitaciones expresivas, tienen la ventaja de que existe un **algoritmo de entrenamiento** simple para perceptrones con función de activación **umbral** capaz de encontrar un perceptrón adecuado para cualquier conjunto de entrenamiento que sea **linealmente separable**



(a) I_1 **and** I_2



(b) I_1 **or** I_2



(c) I_1 **xor** I_2

Entrenamiento (P. Umbral)

- Entrada: Un conjunto de entrenamiento D (con ejemplos de la forma $(\vec{x}, y)$, con $x \in R^n$ e $y \in \{0, 1\}$), y factor de aprendizaje α

```
Set initial weights to random w -> (w0, w1 ..... wn)
Repeat until termination condition:
    For each (~x, y) in D:
        Calculate o = threshold(sum(wi xi)) (with xO = -1)
        For each weight wi do:
            wi <- wi + alpha (y - o) xi
return ~w
```

Entrenamiento (P. Umbral)

- η es una constante positiva, usualmente pequeña (p.ej. 0.1), llamada factor de aprendizaje, que modera las actualizaciones de los pesos.
- En cada iteración, si $y = 1$ y $o = 0$, entonces $y - o = 1$, y por tanto los w_i correspondientes a x_i positivos aumentarán (y disminuirán los correspondientes a x_i negativos), lo que aproximará la salida real a la salida esperada.
- Cuando $y = o$, los w_i no se modifican.
- Para perceptrones con función de activación bipolar, el algoritmo es análogo.

Comentarios

Teorema

- El algoritmo anterior converge en un número finito de pasos a un vector de pesos $\vec{w}$ que clasifica correctamente todos los ejemplos de entrenamiento, siempre que éstos sean linealmente separables y n suficientemente pequeño (Minsky y Papert, 1969)
- Por tanto, en el caso de conjuntos de entrenamiento linealmente separables, la condición de terminación puede ser que se clasifiquen correctamente todos los ejemplos

Índice

Descenso por gradiente

- Cuando el conjunto de entrenamiento **no es linealmente separable**, la convergencia del algoritmo anterior no está garantizada
- En ese caso, no será posible encontrar un perceptrón que sobre todos los elementos del conjunto de entrenamiento devuelva la salida esperada.

Descenso por gradiente

- En su lugar intentaremos minimizar el error cuadrático.

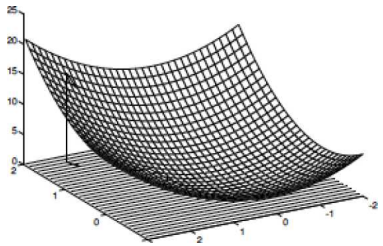
$$E(w) = \frac{1}{2}\sum_{j=1}^{m}\left(y_j - o_j\right)^2 =$$
$$= \frac{1}{2}\sum_{j=1}^{m}\left(y_j - g(w_0x_0 + w_1x_1 + \cdots + w_nx_n)\right)^2$$

  - Donde g es la función de activación, es la salida esperada para la instancia $(X^j, Y^j) \in D$, y o^j es la salida obtenida por el perceptrón
  - Nótese que E es función de $\vec{w}$ y que tratamos de encontrar un $\vec{w}$ que minimice E.

- En lo que sigue, supondremos perceptrones con función de activación g diferenciable (sigmoides, por ejemplo)
- Quedan excluidos, por tanto, perceptrones umbral o bipolares

Descenso por gradiente

- Representación gráfica de E(w) (con $\eta = 1$ y g la identidad)



- En una superficie diferenciable, la dirección de máximo crecimiento viene dada por el vector gradiente $\nabla E(\vec{w})$.
  El "negativo del gradiente" proporciona la dirección de máximo descenso hacia el mínimo de la superficie

Descenso por gradiente

- Puesto que igualar a cero el gradiente supondría sistemas de ecuaciones complicados de resolver en la práctica, optamos por un algoritmo de búsqueda local para obtener un $\vec{w}$ para e cual E($\vec{w}$) es mínimo (local),
- La idea es comenzar con un $\vec{w}$ aleatorio y modificarlo sucesivamente en pequeños desplazamientos en la dirección opuesta al gradiente, esto es $\vec{w} \leftarrow \vec{w} + \Delta w$,
  siendo $\Delta\vec{w} = -\eta\nabla E(\vec{w})$
  y η el factor de aprendizaje.

Descenso por gradiente

- El gradiente es el vector de las derivadas parciales de E "respecto de cada w_i

$$\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n}\right]$$

- Notando por x_i^j la componente i-ésima del ejemplo j-ésimo (y $x_0^j = -1$) y por $in^j = \sum_{i=0}^{n} w_i x_i^j$

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{j=1}^{m}(y^j - o^j)^2 = \sum_{j=1}^{m}(y^j - o^j)g'(in^j)(-x_i^j)$$

- Esto nos da la siguiente expresión para actualizar pesos mediante la regla de descenso por el gradiente:

$$w_i < -w_i + \eta\sum_{j=1}^{m}(y^j - o^j)g'(in^j)(-x_i^j)$$

Descenso por gradiente

- Algoritmo de entrenamiento

Algoritmo (Descenso por Gradiente)

1) Considerar unos pesos iniciales generados aleatoriamente $\vec{w} \leftarrow (w_0, w_1, \ldots, w_n)$
2) Repetir hasta que se cumpla la condición de terminación
   1) Inicializar Δw_i a cero, para $i = 0, \ldots, n$
   2) Para cada $(x, y) \in D$,
      1) Calcular $in = \sum_{i=0}^{n} w_i x_i$ y $o = g(in)$
      2) Para cada $i = 0, \ldots, n$, hacer
         $\Delta w_i \leftarrow \Delta w_i + \eta(y - o)g'(in)x_i$
   3) Para cada peso w_i, hacer $w_i \leftarrow w_i + \Delta w_i$
3) Devolver $\vec{w}$

Regla Delta

- Es una variante del método de descenso por el gradiente
- En lugar de tratar de minimizar el error cuadrático cometido sobre **todos** los ejemplos del dataset ,procede incrementalmente tratando de descender el error cuadrático $E^j(\vec{w}) = \frac{1}{2}(y^j - o^j)^2$ cometido sobre el ejemplo $(x^j, y^j) \in D$ que se esté tratando en cada momento
  - De esta forma, $\frac{\partial E^j}{\partial w_i} = (y^j - o^j)g'(in^j)(-x_i^j)$ y siendo $\Delta w_i = -\eta \frac{\partial E^j}{\partial w_i}$ tendremos $\Delta w = \eta(y - o)g'(in)x_i$ y por tanto $w_i \leftarrow w_i + \eta(y - o)g'(in)x_i$
  - Este método para actualizar los pesos iterativamente es conocido como **Regla Delta**

Regla Delta

- **Entrada**: Un conjunto de entrenamiento D (con ejemplos de la forma $(\vec{x}, y)$, con $\vec{x} \in \mathbb{R}^n$ e $y \in \mathbb{R}$), un factor de aprendizaje η y una función de activación g diferenciable.
  Algoritmo

```
Considerar unos pesos iniciales generados aleatoriamente
    w = (wo, w1, ..., wn)
Repetir hasta que se cumpla la condición de terminación
    Para cada (x, y) de D:
        Calcular
            in = sum(i=0,n) wi xi
            o = g(in)
        Para cada peso w_i, hacer:
            wi = wi + n (y-o) g'(in) xi
Devolver ~w
```

Comparación Descenso - Delta

- Algoritmo Descenso por Gradiente

```
Considerar unos pesos iniciales generados aleatoriamente
    w = (wo, w1, ..., wn)
Repetir hasta que se cumpla la condición de terminación
    Inicializar Awi = 0, para i=0,..,n
    Para cada (x, y) de D:
        Calcular
            in = sum(i=0,n) wi xi
            o = g(in)
        Para cada i=0..n:
            Awi = Awi + \eta (y - o) g'(in) xi
        Para cada peso wi:
            wi = wi + Awi
Devolver ~w
```

- Regla Delta

```
Considerar unos pesos iniciales generados aleatoriamente
    w = (wo, w1, ..., wn)
Repetir hasta que se cumpla la condición de terminación
    Para cada (x, y) de D:
        Calcular
            in = sum(i=0,n) wi xi
            o = g(in)
        Para cada peso w_i, hacer:
            wi = wi + n (y-o) g'(in) xi
Devolver ~w
```

Casos particulares

- Perceptrones con función de activación lineal:
  - En este caso $g'(in) = C$ (constante)
  - Por tanto, la Regla Delta queda (transformando η convenientemente): $w_i \leftarrow w_i + \eta(y - o)x_i$
- Perceptrones con función de activación sigmoide:
  - En ese caso, $g'(in) = g(in)(1 - g(in)) = o(1 - o)$
  - Luego la regla de actualización de pesos queda: $w_i \leftarrow w_i + \eta(y - o)o(1 - o)x_i$

Comentarios

- Tanto el método de descenso por el gradiente como la Regla Delta, son algoritmos de búsqueda local, que convergen hacia mínimos locales del error entre salida obtenida y salida esperada.
  - En descenso por el gradiente, se desciende en cada paso por el gradiente del error cuadrático de todos los ejemplos
  - En la Regla Delta, en cada iteración el descenso se produce por el gradiente del error de cada ejemplo
- Con un valor de η suficientemente pequeño, el método de Descenso por Gradiente converge (puede que asintóticamente) hacia un mínimo local del error cuadrático global.

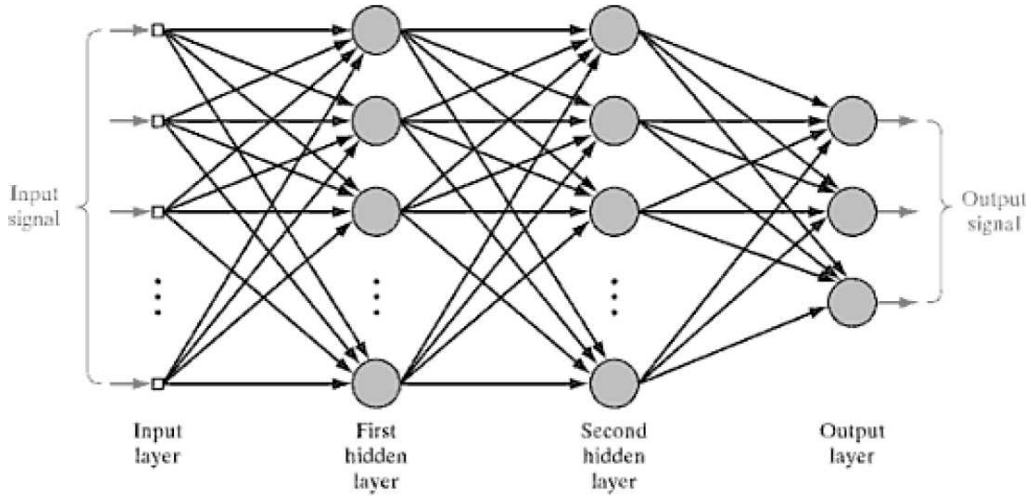
Comentarios

- Se puede demostrar que haciendo el valor de η suficientemente pequeño, la Regla Delta se puede aproximar arbitrariamente al método de descenso por el gradiente
- En la Regla Delta la actualización de pesos es más simple, aunque necesita valores de η más pequeños. Además, a veces escapa más fácilmente de los mínimos locales

Índice

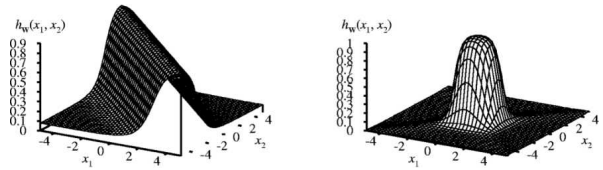
Perceptrón Multicapa

- Como hemos visto, los **perceptrones** tienen una capacidad **expresiva limitada** . Es por esto que vamos a estudiar las redes multicapa.
- En una red **multicapa**, las neuronas se estructuran en **capas** en las que cada una **recibe su entrada de la salida** de las neuronas de la capa **anterior**.



Perceptrón Multicapa

- Combinando neuronas en distintas capas (y siempre que la función de activación sea no lineal) aumentamos la capacidad expresiva de la red



(a) The result of combining two opposite-facing soft threshold functions to produce a ridge,

(b) The result of combining two ridges to produce a bump.

- Habitualmente, con una capa oculta basta para la mayoría de las aplicaciones reales.

Perceptrón Multicapa

- Definición del problema de aprendizaje:

  (Análogamente al caso del perceptrón)

  - Dado un conjunto de entrenamiento D tal que cada $(\vec{x}, \vec{y}) \in D$ contiene una salida esperada $\vec{y} \in \mathbb{R}^m$ para la entrada $\vec{x} \in \mathbb{R}^n$
  - Partiendo de una red multicapa con una estructura dada, queremos **encontrar los pesos de la red** de manera que la función que calcula la red se ajuste lo mejor posible a los ejemplos

Perceptrón Multicapa

- El aprendizaje de estos pesos se realiza mediante un proceso de actualizaciones sucesivas
- Lo vamos a basar en la misma idea del descenso por gradiente, aunque con modificaciones necesarias,
- y se llama "**Retropropagación**"
    - Backpropagation, para Google

Perceptrón Multicapa

- Supondremos una red neuronal con n unidades en la capa de entrada, m en la de salida y L capas en total
  - La capa 1 es la de entrada y la capa L es la de salida
  - Cada unidad de una capa i($< L$) está conectada con todas las unidades de la capa $i + 1$
- Supondremos una función de activación g diferenciable (usualmente, el sigmoide)

Perceptrón Multicapa

- Dado un ejemplo $(x^j, y^j) \in D$:
- Si i es una neurona de la capa de entrada, notaremos por x_i la componente de $\vec{x}$ correspondiente a dicha neurona
- Si i es una neurona de la capa de salida, notaremos por y_i la componente de $\vec{y}$ correspondiente a dicha neurona
- notaremos in_i a la entrada que recibe una neurona i cualquiera y a_i a la salida por la misma neurona i

Perceptrón Multicapa

- Si i es una neurona de entrada (es decir, de la capa 1), entonces:

$$a_i = x_i$$

- Si i es una unidad de una capa $l(>1)$, entonces:

$$in_i = \sum_{\forall j}^{capa\ ant} w_{ji} a_j$$

$$a_i = g(in_i)$$

Retropropagación

- El método de **Retropropagación** es un método de aproximaciones sucesivas.
- Formalmente, se puede considerar como descenso por el gradiente del error.

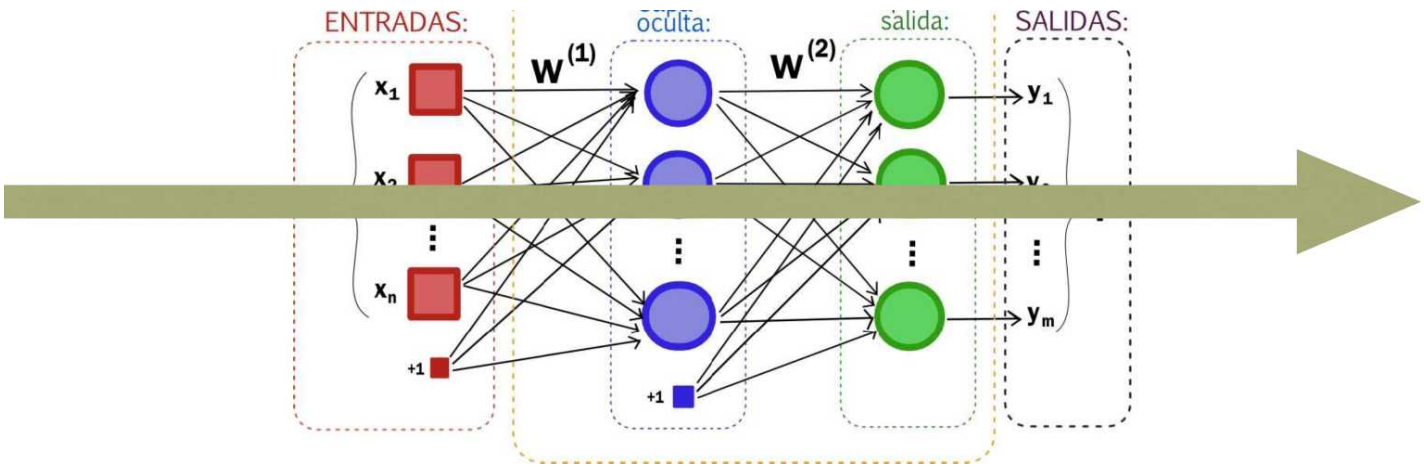
Retropropagación

Algoritmo

```
1) Inicialización aleatoria de ˜w
2) Repetir hasta criterio de parada
    Para cada ejemplo (x, y) en D hacer:
        1) Propagación de valores hacia adelante
        2) Propagación de errores hacía atrás
            Actualizando los pesos
3) Devolver ˜w
```

Retropropagación: Hacia delante

- A partir de los datos de entrada $\vec{x}$, vamos calculando todas las neuronas, desde la capa 1 hasta la capa L.
- La salida de las neuronas de la capa i, ponderada por
- los pesos correspondientes, sirve de entrada para las neuronas de la capa $i + 1$



Retropropagación: Hacia delante

```
1) Para cada nodo i de la capa de entrada hacer
    a <- xi
2) Para l desde 2 hasta L, hacer
    1) Para cada neurona i de la capa l
        Calcular su salida con:
        in_i = sum(j de la capa anterior) ( w_ji * a_j )
        a_i = g(in_i)
```

Retropropagación: Hacia atrás

- Ahora, en vez de propagar valores, vamos a propagar errores
- La dificultad está en que sólo sabemos el error en la capa final, que es donde tenemos el valor teórico de la salida ($\vec{y}$)

  En las capas anteriores, no sabemos cuanto es el valor teórico que deberían de dar.
- Creamos un símbolo Δ para significar el factor de error y es eso lo que vamos a propagar hacia detrás

Retropropagación: Hacia atrás

I. Para cada neurona i en la capa de salida, calcular:

$$\Delta_i \leftarrow g'(in_i)(y_i - a_i)$$

II. Para l, desde L - 1 hasta 1, hacer:

1) Para cada neurona j en la capa l, hacer:

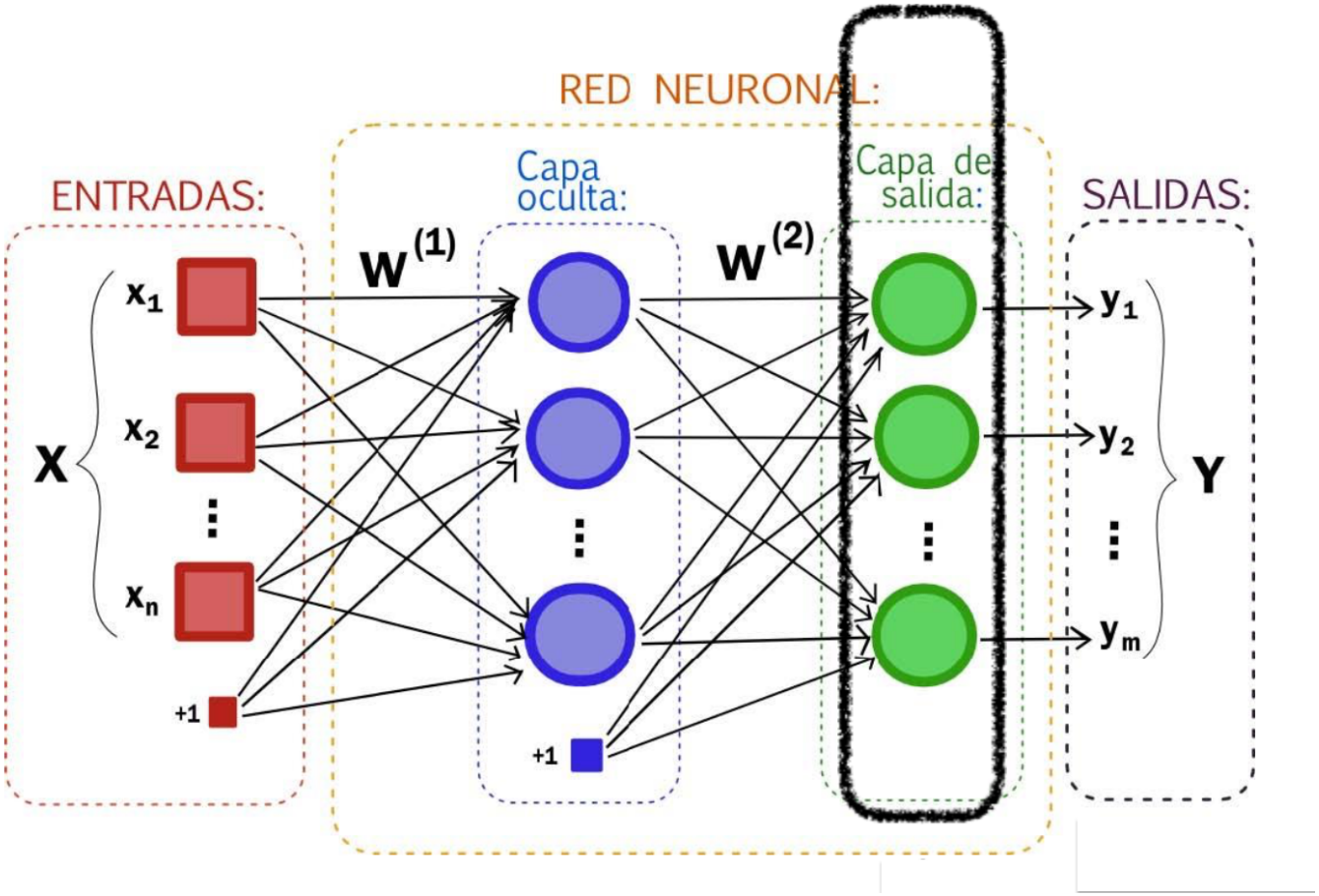
a) $\Delta_i \leftarrow g'(in_j) \sum_i^{l+1} wji\Delta_i$

b) Para cada neurona i en la capa $l + 1$ hacer:

$$w_{ji} \leftarrow w_{ji} + \eta a_j \Delta_i$$

Retropropagación: Hacia atrás

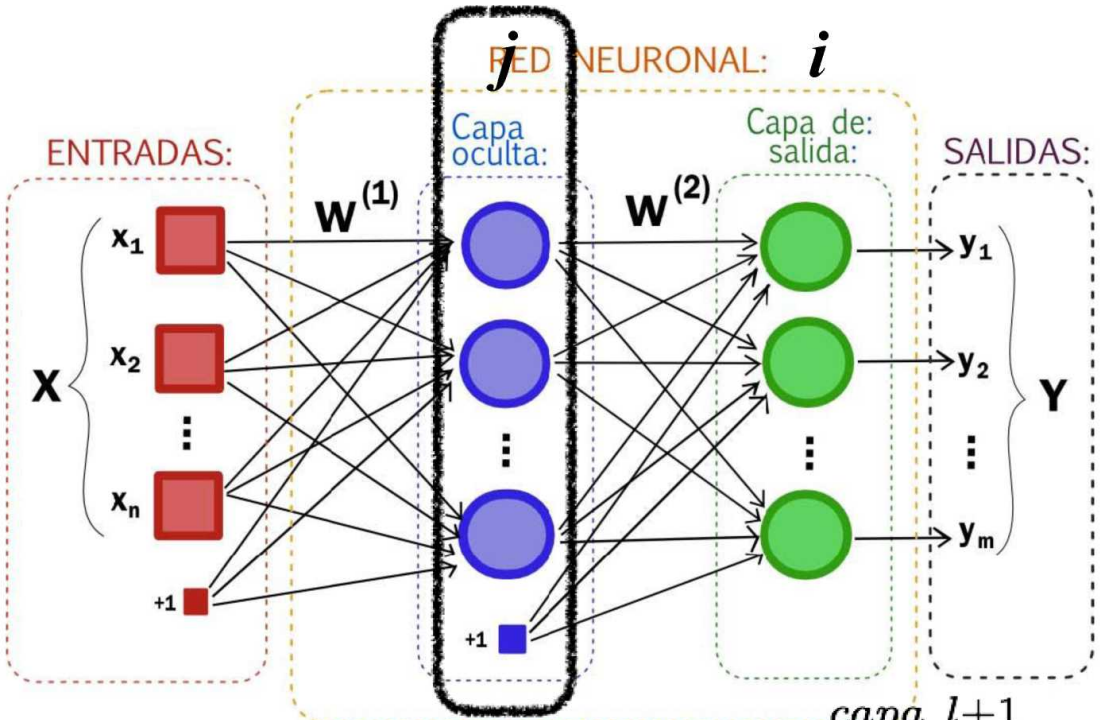
- 1. Cálculo del Δ:



$$\Delta_i \leftarrow g'(in_i)(y_i - a_i)$$

Retropropagación: Hacia atrás

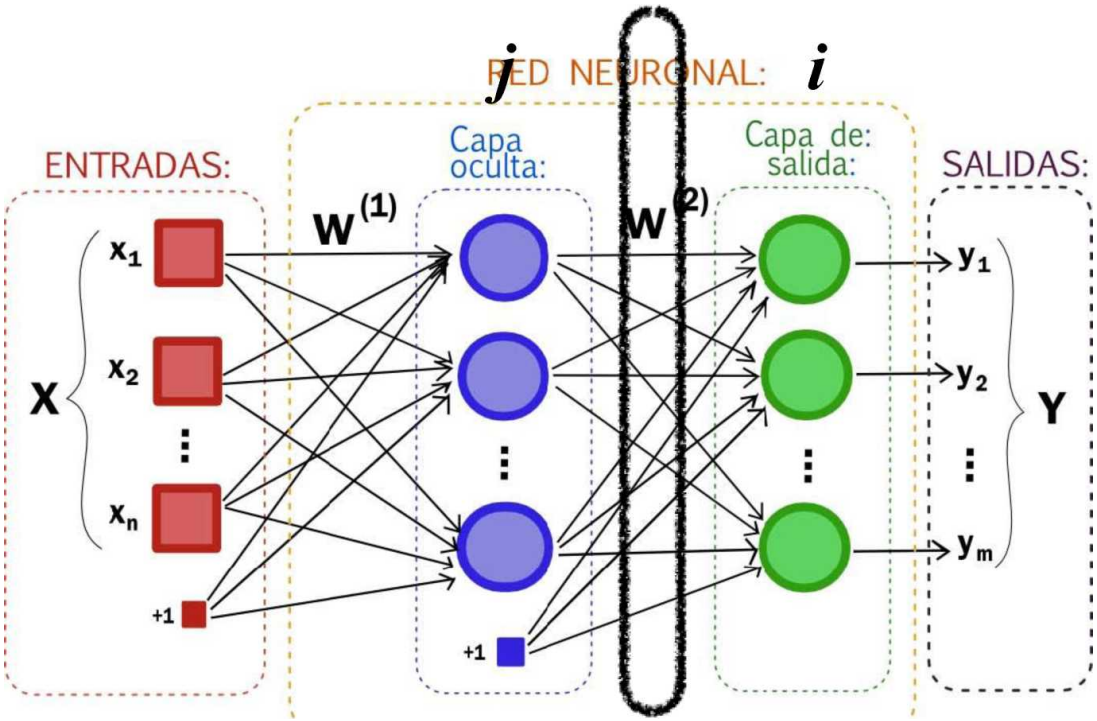
I. Cálculo del Δ en la capa interior para cada neurona



$$\Delta_j \leftarrow g'(in_j) \sum_{i \in \text{capa } l+1} w_{ji}\Delta_i$$

Retropropagación: Hacia atrás

I. Actualización de los pesos



$$w_{ji} \leftarrow w_{ji} + \eta a_j \Delta_i$$

Función de activación

- Función de activación:

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

$$g'(x) = g(x)(1 - g(x))$$

- Así, el cálculo de errores en el Paso 2 queda:
  - Para la capa de salida: $\Delta_i \leftarrow a_i(1 - a_i)(y_i - a_i)$
  - Para las capas ocultas: $\Delta_j \leftarrow a_j(1 - a_j) \sum_i w_{ji} \Delta_i$
- Esto significa que no necesitamos almacenar los in_i del Paso 1 para usarlos en el Paso 2

Momentum

- Retropropagación es un método de descenso por gradiente y, por tanto, existe el problema de mínimos locales
- Una variante muy común en el algoritmo de retropropagación es introducir un sumando adicional en la actualización de los pesos.
- Este sumando hace que en cada actualización de pesos se tenga también en cuenta la actualización realizada en la iteración anterior.

Momentum

Concretamente:

- En la iteración n-ésima, se actualizan los pesos de forma:

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}^{(n)}$$

- con:

$$\Delta w_{ji}^{(n)} = \eta a_j \Delta_i + \alpha \Delta w_{ji}^{(n-1)}$$

- α es una constante, denominada **Momentum** ($0 < \alpha < 1$)
- La técnica del momentum puede ser eficaz, a veces, para escapar de mínimos locales

Criterio parada

- Se pueden usar diferentes criterios de parada
  - Número de iteraciones prefijadas
  - Error por debajo de una cota

- En esta última, es fácil que caigamos en el sobreaprendizaje.
  - Habría que validar el resultado con un dataset independiente o alguna técnica de validación como las que hemos visto en prácticas.

Índice

Estructura de la red

- El algoritmo de retropropagación parte de una estructura de red fija
- Hasta ahora no hemos dicho nada sobre qué estructuras son las mejores para cada problema
- En nuestro caso, se trata de decidir cuántas capas ocultas se toman, y cuántas unidades en cada capa

Estructura de la red

- En general es un problema que no está completamente resuelto aún
- Lo más usual es hacer búsqueda experimental de la mejor estructura, medida sobre un conjunto de prueba independiente
- La mayoría de las veces, una sola capa oculta con pocas unidades basta para obtener buenos resultados
- Las redes grandes corren un mayor peligro de sobreajuste

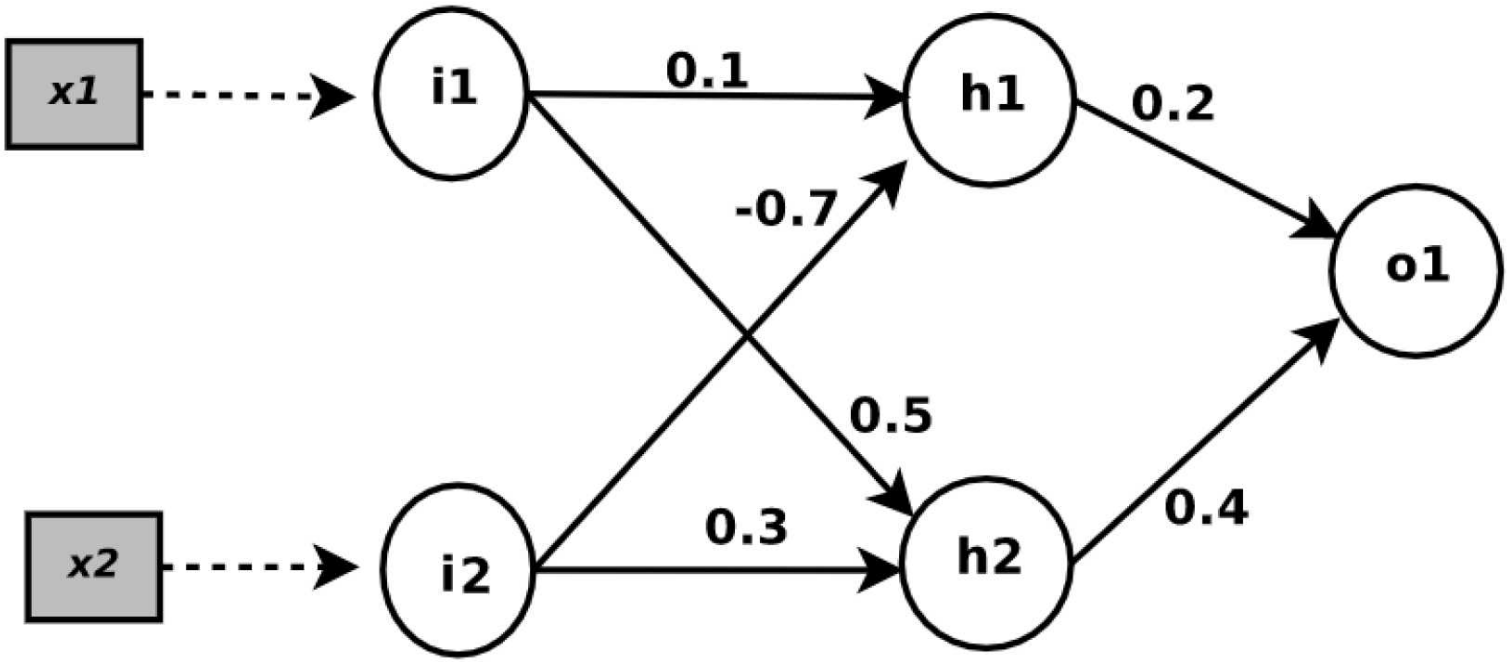
Índice

Ejemplo

- Entrenamiento de un perceptrón multicapa para realizar la operación XOR
- Descripción de la red.
- 1 capa oculta
- 2 neuronas en capa de entrada (i1, i2)
- 2 neuronas en capa oculta (h1, h2)
- 1 neurona en capa de salida (o1)
- Tasa de aprendizaje: $\eta = 0{,}25$
- Todos los pesos umbral son 0: $w_{0x} = 0{,}0$

Ejemplo

- Gráficamente



Ejemplo

Training Dataset

	x1	x2	y
e1	0	1	1
e2	1	0	1
e3	1	1	0
e4	0	0	0

Ejemplo:Hacia delante

- Seleccionamos el primer ejemplo:

$$e_1 = (0, 1, 1)$$

Ejemplo:Hacia delante

Hacia adelante. Cálculo de las “a’s” de cada neurona

- Capa de entrada

$$a_{x1} = x1 = 0$$

$$a_{x2} = x2 = 1$$

- Capa Oculta

h1

$$in_{h1} = 0{,}1 * 0 + (-0{,}7) * 1 = -0{,}7$$

$$a_{h1} = g(-0{,}7) = 1/(1 + e^{0{,}7}) = 0{,}332$$

h2

$$in_{h2} = 0{,}5 * 0 + 0{,}3 * 1 = 0{,}3$$

$$a_{h2} = g(0{,}3) = 1/(1 + e^{-0{,}3}) = 0{,}574$$

Ejemplo:Hacia delante

Hacia adelante. Cálculo de las “a’s” de cada neurona

- Capa de salida
  o1

$$in_{o1} = 0{,}2 * 0{,}332 + 0{,}4 * 0{,}574 = 0{,}296$$

$$a_{o1} = g(0{,}296) = 1/(1 + e^{-0{,}296}) = 0{,}573$$

Ejemplo:Hacia atrás

Hacia atrás

- Capa de Salida

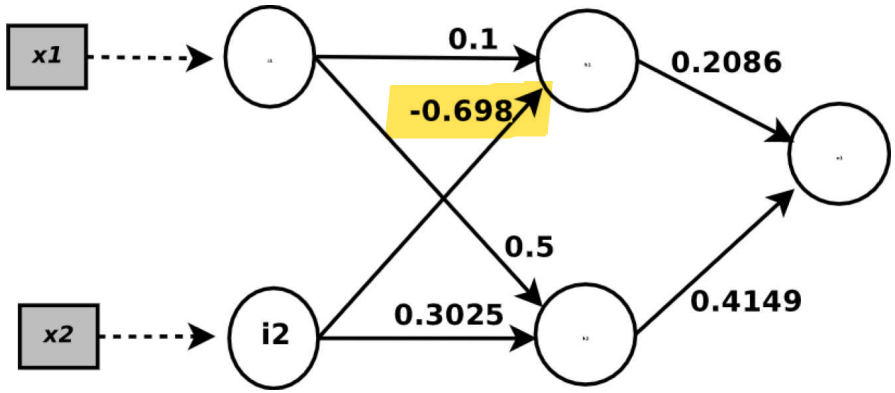
$$w_{h1o1} = w_{h1o1} + alfa * a_{h1} * \Delta_{o1} = 0{,}2 + 0{,}25 * 0{,}332 * 0{,}1044 = 0{,}2086$$

$$w_{h2o1} = w_{h2o1} + alfa * a_{h2} * \Delta_{o1} = 0{,}4 + 0{,}25 * 0{,}574 * 0{,}1044 = 0{,}4149$$

- Capa Oculta
  - $\Delta_{h1} = a_{h1}(1 - a_{h1}) * [w_{h1o1} * \Delta_{o1}] = 0{,}332(1 - 0{,}332) * [0{,}2 * 0{,}1044] = 0{,}0046$
  - $w_{i1h1} = w_{i1h1} + alfa * a_{i1} * \Delta_{h1} = 0{,}1 + 0{,}25 * 0 * 0{,}0046 = 0{,}1$
  - $w_{i2h1} = w_{i2h1} + alfa * a_{i2} * \Delta_{h1} = -0{,}7 + 0{,}25 * 1 * 0{,}0046 = -0{,}6988$

Ejemplo:Nueva red

- Obtenemos una nueva configuración de pesos que nos van a dar un comportamiento ligeramente diferente de la red
- Para el mismo ejemplo e1, la salida sería un poco mejor: 0.576 (antes 0.573)



Ejemplo

Hacia adelante (de nuevo) . Cálculo de las “a’s” de cada neurona

- Capa de entrada

$$a_{x1} = x1 = 0$$

$$a_{x2} = x2 = 1$$

- Capa Oculta

h1

$$in_{h1} = 0{,}1 * 0 + (-0{,}698) * 1 = -0{,}6988$$

$$a_{h1} = g(-0{,}6988) = 1/(1 + e^{0{,}6988}) = 0{,}3320$$

h2

$$in_{h2} = 0{,}5 * 0 + 0{,}3025 * 1 = 0{,}3025$$

$$a_{h2} = g(0{,}3025) = 1/(1 + e^{-0{,}3025}) = 0{,}5750$$

Ejemplo

Hacia adelante (de nuevo) . Cálculo de las “a’s” de cada neurona

- Capa de salida
  o1

$$in_{o1} = 0{,}2086 * 0{,}3320 + 0{,}4149 * 0{,}5750 = 0{,}3078$$

$$a_{o1} = g(0{,}3078) = 1/(1 + e^{-0{,}3078}) = 0{,}576$$

Bibliografía

- Russell, S. y Norvig, P. Artificial Intelligence (A modern approach) (Second edition) (Prentice Hall, 2003) (o su versión en español) Cap. 20: “Statistical Learning ” (disponible on-line en la web del libro)
- Mitchell, T.M. Machine Learning (McGraw-Hill, 1997) Cap. 4: “Artificial Neural Networks ”