Universidad de Huelva


**Universidad de Huelva**

Grado en Ingeniería Informática

# Tema 4. Redes Neuronales

*Resumen*

Autor: Alberto Fernández Merchán
Asignatura: Aprendizaje Automático

# 1. Introducción

## 1.1. Algoritmos Bioinsipirados

Los algoritmos bioinspirados simulan el **comportamiento de sistemas naturales** para el diseño de métodos heurísticos **no determinísticos** de búsqueda, aprendizaje y/o comportamiento utilizando analogías con sistemas naturales o sociales.

Estos algoritmos presentan una estructura paralela que los hace más potentes y son adaptativos, es decir, pueden funcionar correctamente cuando cambia el entorno.

## 1.2. Redes Neuronales Artificiales

Las **redes neuronales artificiales** son un ejemplo de modelo computacional bioinspirado. Están basadas en el funcionamiento del **cerebro humano**. Este está compuesto por $10^{11}$ neuronas con $10^4$ conexiones cada una. Dichas neuronas son lentas, para activarse requieren $10^{-3}s$, mucho más lento que los ordenadores. Sin embargo, el cerebro humano posee **paralelismo masivo** (partes del cerebro ya "cableadas"que son sensibles a algunos patrones) que le permite realizar algunas tareas más rápido (reconocer a su madre visualmente en $10^{-1}s$).

### 1.2.1. Funcionamiento de las neuronas

Una **neurona** es una célula que recibe señales electromagnéticas a través de la sinapsis de las dendritas. Si la acumulación de estímulos supera cierto **umbral**, la neurona se dispara (emite a través del axón un impulso que será recibido por otras neuronas).

Aprender significa potenciar o debilitar algunas conexiones.

# 2. Modelo Formal

No todas las características de los sistemas biológicos están reflejadas en los modelos computacionales ni al contrario. Para poder computar las redes neuronales artificiales debemos formalizar tanto la **neurona** como su **estructura de red**.

El funcionamiento de las **RNA** está basado en:

- Entra una cantidad de impulso (información)
- Si sobrepasa cierto umbral de activación, se dispara.
- La información sale modulada hacia las siguientes neuronas.

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

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

donde:

- f es la función de activación. Se puede usar para **normalizar** la salida cuando se supera el umbral. Puede provocar que el funcionamiento de la neurona pueda no ser lineal.
- El sumatorio es sobre todas las etradas (incluida $x_o = -1$)
- El peso de la entrada 0 se denomina **umbral** ($w_0$). Que se interpreta como **la cantidad de impulso que tiene que entrar para activar la neurona**.

Las funciones de activación más usuales son:

- **Función de signo o bipolar**:

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

  Para $B = 0$, $x_- = -1$, $x_+ = 1$

- **Función Umbral**:

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

  Para $B = 0$, $x_- = 0$, $x_+ = 1$

- **Función Sigmoide**:

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

  Normalmente usaremos $\beta = 1$

- **Función Identidad**:

$$f(x) = x$$

### 2.1. Estructura de Red

Una **RNA** está basada en una **estructura de grafo dirigido** donde:

- Los nodos son neuronas artificiales.
- Los argos son las conexiones entre neuronas.
- Cada arco $(i \to j)$ sirve para propagar la salida de la neurona i a la entrada de la neurona j.
- Cada arco tiene asociado un peso numérico $(w)$ que determina la fuerza y el signo de la conexión.

Cuando el grafo de la red es **acíclico** se denomina **red con estructura hacia adelante**. En este tipo de redes las neuronas se estructuran en capas, tal que cada capa recibe en sus entradas las salidas de la capa inmediatamente anterior. Si una red tiene más de una capa se denomina **red multicapa**.

Existen diferentes tipos de capas:

- **Capa de Entrada**: Es única y tiene una neurona por cada atributo de entrada.
- **Capa Oculta**: Puede haber varias.
- **Capa de Salida**: Es única y tiene una neurona por cada clase.

## 3. Redes Neuronales Clasificadoras

Una red neuronal hacia adelante con **n** unidades en la capa de entrada y **m** unidades en la capa de salida se puede expresar como una función:

$$f :: \mathbb{R}^n \to \mathbb{R}^m$$

A la hora de clasificar dos clases (clasificación binaria) se tomará $m = 1$ y:

- Si se tienen funciones de activación **umbral o bipolar**, considerar el valor de salida 1 como positivo y $0$ *o* $-1$ como negativo.
- Si se utiliza la función **sigmoide** considerar $> 0{,}5$ como positivo y $< 0{,}5$ como negativo.

Las RNA's pueden utilizarse como clasificadores multiclase si se le coloca una neurona en la capa de salida por cada clase que queremos clasificar.
Interpretaremos que la neurona con mayor salida es la que indica el nivel de clasificación (*softmax*).

## 3.1. Aprendizaje

En las redes neuronales artificiales, aprender significa **encontrar los valores de los pesos de las conexiones entre neuronas** y encontrar la estructura de red adecuada de manera que la red se comporte correctamente.

Es habitual plantear la siguiente tarea de aprendizaje supervisado:
Dado un conjunto de entrenamiento $D = \{(x_d, y_d) : x_d \in \mathbb{R}^n, y_d \in \mathbb{R}^m, d = 1, \ldots, m\}$ Y una red neuronal de la que solo conocemos su estructura (Capas y número de neuronas en cada capa).
Debemos encontrar un conjunto de pesos $w_{ij}$ tal que ajuste lo mejor posible $f :: \mathbb{R}^n \to \mathbb{R}^m$ a los ejemplos del conjunto de entrenamiento.

## 3.2. Aplicaciones Prácticas

Se suelen utilizar en:

- Problemas que se pueden expresar numéricamente.
- Dominios con volúmenes de datos altos y con ruido.
- Problemas donde interesa la solución pero no el por qué.
- Problemas en los que se pueda asumir un tiempo largo de entrenamiento.
- Problemas en los que se requieran tiempos corto para evaluar una nueva instancia. Los cálculos se realizan en paralelo por lo que es recomendable utilizar una GPU.
- Problemas de:
  - Clasificación
  - Reconocimiento de Patrones
  - Optimización
  - Predicción climatológica.
  - Reconocimiento de voz
  - Visión artificial
  - Control de robots
  - Compresión de datos
  - Diagnosis

## 3.3. Ejemplo: ALVINN

ALVINN es una red neuronal entrenada para conducir un vehículo a 70km/h, en función de la percepción visual que recibe de unos sensores. Se caracteriza por:

- **Entrada**: Una imagen de la carretera digitalizada como un *array* de 32x32 píxeles (960 datos de entrada).
- **Salida**: Indicación sobre hacia dónde torcer el volante (vector de 30 componentes).
- **Estructura**: Red hacia adelante, con una capa de entrada de **960 unidades**, una capa oculta de **4 unidades** y una capa de salida de **30 unidades.**

# 4. Perceptrón

Es el caso más simple de red neuronal. **Sólo tiene una capa** (entrada y salida) con una sola neurona.

Con un perceptrón de **función de activación umbral** es posible representar las funciones booleanas básicas AND, OR y NOT (XOR no es linealmente separable). Clasifica como positivos aquellos valores de entrada $(x_0, x_1, \ldots, x_n)$ tal que:

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

Los valores que satisfacen $\sum_{i=0}^{n} w_i x_i = 0$ representan un hiperplano en $\mathbb{R}^n$. Es decir, una función booleana solo podrá ser representada por un **perceptrón umbral** si existe un hiperplano que separa los elementos con valor 1 de los elementos con valor 0. (Si los datos son **linealmente separables**).

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**: Los elementos que posee el anterior algoritmo son los siguientes:

```
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
```

Figura 1: Algoritmo de Entrenamiento del Perceptrón Umbral

- **alpha**: Es el factor de aprendizaje. Una constante positiva, usualmente pequeña, que modera las actualizaciones de los pesos.
- En cada **iteración** si $y = 1$ o $y = 0$, entonces $y - o = 1$ y, por tanto, los pesos correspondientes a ejemplos positivos aumentarán (y disminuirán los correspondientes a ejemplos negativos). Esto provoca que la salida real se aproxime a la salida esperada.
- Cuando $y = o$ los pesos no se modifican.
- El algoritmo es el mismo para perceptrones con **función de activación bipolar**.
- 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 sean linealmente separables y con un factor de aprendizaje lo suficientemente pequeño (Minsky y Papert, 1969).
- La condición de parada puede ser que se clasifiquen correctamente todos los ejemplos.

# 5. Descenso Por Gradiente

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

Intentaremos, entonces, **minimizar el error cuadrático**.

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

donde:

- **g** es la función de activación.
- **y** es la salida esperada para la instancia $(X^j, Y^j)$.
- $o^j$ es la salida obtenida por el perceptrón.

Necesitamos encontrar un $\vec{w}$ que minimice $E$.
Para este apartado debemos suponer perceptrones con **funciones de activación derivables**. Quedan excluidos los que tengan función de activación **umbral** o **bipolar**.

En una superficie diferenciable, la dirección del **máximo crecimiento** viene dada por el vector gradiente $\nabla E(\vec{w})$. El negativo del gradiente, por tanto, proporciona la **dirección de máximo descenso hacia el mínimo de la superficie.**
Hacer $\nabla E(\vec{w}) = 0$ supondría resolver sistemas de ecuaciones demasiado complejos, por tanto, optamos por un algoritmo de búsqueda local para obtener un $\vec{w}$ para el cual $E(\vec{w})$ sea un mínimo local. La **idea** es comenzar con un vector de pesos $(\vec{w})$ aleatorio y modificarlo en pequeños desplazamientos en la dirección opuesta al gradiente:

$$\vec{w} \leftarrow \vec{w} + \Delta w = \vec{w} \leftarrow \vec{w} + -\eta \nabla E(\vec{w})$$

donde:

- $\eta$ es el factor de aprendizaje.
- $\nabla E(\vec{w})$ 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 como $x_i^j$ a la componente iésima del ejemplo j-ésimo ($x_0^j = -1$) y por $in^j = \sum_{i=0}^{n} w_i x_i^j$. Obtenemos el vector de derivadas parciales como:

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

- Y, por tanto, la actualización de los pesos quedaría como:

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

Por tanto, el algoritmo de descenso por gradiente quedaría como:

```
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
```

Figura 2: Algoritmo de Entrenamiento por Descenso por Gradiente

# 6. Regla Delta

Es una variante del método de descenso por gradiente que, en lugar de minimizar el error cuadrático cometido sobre **todos** los ejemplos del dataset, procede incrementalmente tratando de descender por el error cuadrático de cada ejemplo $E^j(\vec{w}) = \frac{1}{2}(y^j - o^j)^2$ cometido sobre el ejemplo $(x^j, y^j) \in D$ que se esté tratando en cada momento. De esta forma:

$$\frac{\partial E^j}{\partial w_i} = \left(y^j - o^j\right) g' \cdot (in^j)(-x_i^j)$$

y siendo

$$\Delta w_i = -\eta \frac{\partial E^j}{\partial w_i} \rightarrow \Delta w = \eta(y - o)g' \cdot (in)x_i$$

por tanto, la actualización de pesos quedaría:

$$w_i = w_i + \eta(y - o)g' \cdot (in)x_i$$

El algoritmo es el siguiente:

```
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
```

Figura 3: Algoritmo regla delta

Estos dos algoritmos (descenso por gradiente y regla delta) son métodos de búsqueda local que convergen hacia mínimos locales del error entre salida obtenida y esperada. La diferencia es que:

- En **descenso por gradiente** se desciende en cada paso por el gradiente del error cuadrático de **todos** los ejemplos
- En **regla delta** en cada iteración el descenso se produce por el gradiente del error de cada ejemplo.

Con un valor del factor de aprendizaje ($\eta$) lo suficientemente pequeño, el método de descenso por gradiente **converge** (puede que asintóticamente) hacia un mínimo local del error cuadrático global.

La regla delta, con un factor de aprendizaje lo suficientemente pequeño, se aproxima, arbitrariamente, al método de descenso por gradiente. En este método, la actualización de pesos es más simple, aunque necesita valores de $\eta$ más pequeños. Puede **escapar** más fácilmente de los mínimos locales.

## 6.1. Casos Particulares

Algunos casos a tener en cuenta son:

- Perceptrones con **función de activación lineal**: En este tipo de funciones, la derivada es constante ($g'(in) = C$). Por tanto la actualización de pesos en la regla delta quedaría como $w_i \leftarrow w_i + \eta(y - o)x_i$. Hay que transformar el factor de aprendizaje convenientemente.
- Perceptrones con **función de activación sigmoide**: En este caso $g'(in) = g(in)(1 - g(in)) = o(1 - o)$. Por tanto, la regla de actualización de pesos queda como: $w_i \leftarrow w_i + \eta(y - o)o(1 - o) * x_i$

# 7. Perceptrón Multicapa

Los perceptrones tienen una capacidad expresiva limitada. Podemos extender este perceptrón y hacer más capas creando una **red multicapa**. En una red multicapa, las neuronas se estructuran en capas donde cada una recibe como entrada la salida de la capa anterior.

Combinando neuronas en distintas capas (con una función de activación no lineal) aumentamos la capacidad expresiva de la red. Normalmente es suficiente con una **capa oculta**.

El **problema de aprendizaje** es análogo al caso del perceptrón simple: Dado un conjunto de entrenamiento $D$ tal que cada ejemplo $(\vec{x}, \vec{y} \in D)$ 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.

Suponemos una red neuronal con N neuronas en la capa de entrada, M en la capa de salida y L capas en total (capa 1 es la de entrada y capa L es la de salida). Cada neurona de la capa $i$ está conectada con las neuronas de la capa $i+1$. Utilizaremos como función de activación la **función sigmoide** (derivable).

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 $\vec{x}$ corespondiente 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.
- $in_i$ será la entrada que recibe una neurona $i$ cualquiera y $a_i$ la salida de la misma neurona $i$.
- Para las neuronas de la capa de entrada: $a_i = x_i$
- Para las neuronas de las capas ocultas: $in_i = \sum_{j=0}^{capa\ anterior} w_{ji} a_j$ y la salida será: $a_i = g(in_i)$

El aprendizaje de los pesos se realiza mediante un proceso de actualizaciones sucesivas. Se basa en la misma idea que el descenso por gradiente, pero con algunas modificaciones. Esta idea se conoce como **retropropagación**. Este método es un método de aproximaciones sucesivas, se puede considerar como descenso por el gradiente del error.

El algoritmo es el siguiente:

```
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
```

Figura 4: Algoritmo de retropropagación

## 7.1. Retropropagación hacia delante

A partir de los datos de entrada $\vec{x}$ se calculan 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 $i+1$.

```
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)
```

Figura 5: Algoritmo de retropropagación hacia delante.

## 7.2. Retropropagación hacia detrás

En vez de propagar valores, se propagan **errores**. La dificultad está en que solo sabemos el error en la capa final, que es donde tenemos el valor teórico de la salida. Propagaremos entonces un **factor de error** ($\Delta$).

Para cada neurona i en la capa de salida, se calcula:

$$\Delta_i = g'(in_i)(y_i - a_i) \tag{1}$$

Para l desde L-1 hasta 1 se hace:

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

$$\Delta_j = g'(in_j) \sum_{i}^{l+1} w_{ji}\Delta_i \tag{2}$$

2. Para cada neurona i en la capa $l+1$ se realiza:

$$w_{ji} = w_j i + \eta a_j \Delta_i \tag{3}$$

- La **retropropagación** es un método de descenso por gradiente y, por tanto, existe el problema de quedar atascado en un mínimo local.
- Una variante común es introducir un sumando adicional en la actualización de los pesos que hace que, en cada actualización, se tenga en cuenta la actualización realizada en la iteración anterior (***momentum***).

### 7.2.1. Momentum

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

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

con

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

donde $\alpha$ es una constante denominada **MOMENTUM** que va de 0 a 1 ($0 < \alpha < 1$).
Esta técnica puede ser eficaz para **escapar** de mínimos locales.

Como **criterio de parada** podemos utilizar:

- Número de iteraciones prefijadas
- Error por debajo de una cota: Con esta es fácil caer en el sobreaprendizaje. Podemos validar el resultado con un dataset independiente o alguna otra técnica de validación.

## 8. Estructura de la red

Existen diferentes estructuras de red que se ajustan mejor o peor a cada problema. Decidiremos cuántas capas ocultas se crearán y cuantas neuronas tendrá cada una de ellas.
Este es un problema que no está completamente resuelto. Lo más usual es hacer una búsqueda experimental de la mejor estructura medida sobre un conjunto de prueba independiente.

Nosotros utilizaremos *grid optimization*, una técnica que prueba algunos valores de hiperparámetros y escoge el que mejor resultado de.

La mayoría de las veces con **una** capa oculta y **pocas** neuronas basta para obtener buenos resultados. Las redes grandes corren peligro de sobreajuste.