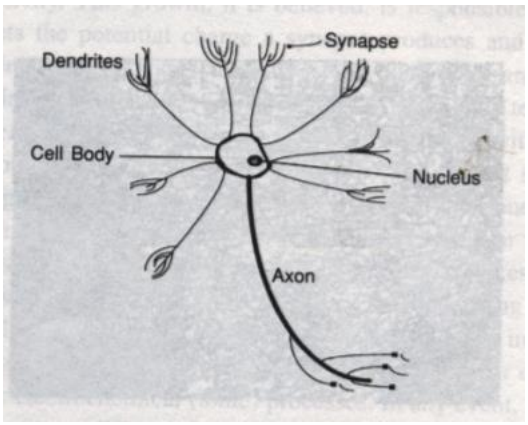
Inteligencia Artificial

Tema 7 – Introducción a las Redes Neuronales

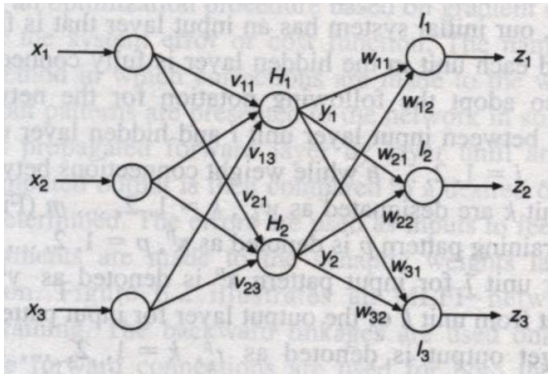
Conceptos: El aprendizaje en los sistemas biológicos se basa en redes muy complejas de neuronas interconectadas. Las neuronas son células que reciben señales electromagnéticas del exterior o de otras neuronas gracias a la llamada “sinapsis de las dendritas”.

Si la acumulación de estímulos recibidos supera un determinado umbral, la neurona se “dispara”. Esto es, emite a través de su *axón* una señal que será recibida por otras neuronas gracias a dicha conexión sináptica.



Con el desarrollo y aprendizaje, algunas conexiones de neuronas y otras se debilitan. En comparación con los ordenadores, las neuronas son relativamente lentas, pero la clave está en el paralelismo masivo que ocurre en el cerebro.

Redes neuronales: nacen inspiradas en estos procesos biológicos; dan lugar a las llamadas *redes neuronales artificiales* cómo un modelo computacional. No deja de ser un modelo matemático formal basado en la estructura de un grafo dirigido cuyos nodos representan neuronas artificiales.

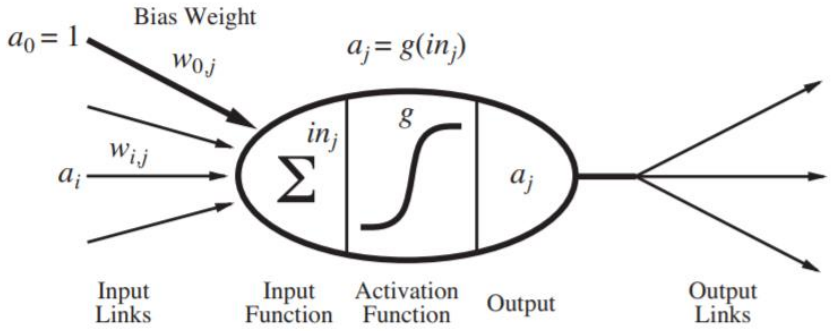


Cada nodo o unidad se conecta a otras a través de aristas dirigidas; cada arista **i → j** sirve para propagar la salida de la unidad **i** que servirá cómo entrada para la unidad **j**. Las entradas y salidas se denotan con números. Cada arista, además, tendrá asociado un peso numérico Wij que determina la fuerza y el signo de la conexión.

Cada unidad es encargada de calcular su salida en función de las entradas que recibe. La salida sirve, además, cómo entrada para otras neuronas. La red en sí recibe una serie de *entradas externas* y devuelve al exterior la salida de algunas de sus neuronas llamadas *unidades de salida*.

Cálculo unitario de una neurona o unidad:

La salida de cada unidad se calcula: $a_j = g(\sum_{i=0}^{n} w_{ij} a_i)$



, dónde:

- **g** es una función de activación.
- El sumatorio, denotado cómo **inj**, se hace sobre todas las unidades **i** que envían su salida a la unidad **j**.
- El caso **i = 0** es una excepción considerada una entrada ficticia, dónde **a0 = 1 a0 = 1** y el peso **W0j** representa el umbral (o bias).

Intuitivamente el umbral **W0j** de cada unidad se interpreta cómo el opuesto de una cantidad cuya entrada debe superar. La función de activación **g** introduce cierta componente no lineal que hace que la red no se comporte cómo una función lineal y aumenta así la expresividad del modelo. Las funciones de activación más comunes son:

- Función umbral: umbral(x) { 1 si x>0 ó 0 en otro caso }
- Sigmoide: σ(x) = 1 / (1+e^-x)
- ReLU (Rectified Lineal Unit): ReLU(x) = max{0,x} derivable en todo x menos en 0
- Tangente hiperbólica: tanh(x) = (2 / 1+e^-2x) – 1

Avance en redes neuronales: capa de entrada, capas intermedias y capa de salida:

- Diferenciamos capas de entrada, capas ocultas o intermedias y capas de salida.
- Cada capa representa conjuntos de unidades; para unidades de una misma capa, la función de activación será la misma, entre distintas capas sin embargo puede variar.
- Usualmente en capas intermedias se emplea ReLU.
- La capa de salida variará su función de activación en base al uso que se busque de la red:

- Regresión (predicción de valores): sin función de activación.
- Clasificación binaria: una sola unidad en la capa de salida con función umbral (o sigmoide).
- Clasificación multiclase: sin función de activación (pero luego se aplican correcciones).

Redes neuronales cómo clasificadores: una red neuronal con *n* unidades en la capa de entrada y *m* unidades en la capa de salida la entendemos cómo una función de R^n en R^m. Distinguimos entre clasificación binaria (dos clases) y clasificación multiclase (más de dos clases).

Redes neuronales cómo clasificadores binarios

Para clases 0 y 1, tomamos en la capa de salida una sola unidad y el sigmoide cómo función de activación. El valor de salida se interpreta cómo la probabilidad de pertenece a la clase 1.

Redes neuronales cómo clasificadores multiclase

En general, para clasificadores con *m* posibles valores, cada unidad de salida corresponde con un valor de clasificación. Se interpreta que la unidad con mayor salida es la que indica el valor de la clasificación.

Es habitual que las salidas sean normalizadas con *softmax* e interpretar la salida en cada unidad cómo la probabilidad de pertenecer a la correspondiente clase.

$$softmax(a_1, \ldots, a_m) = \frac{1}{\sum_k e^{a_k}}(e^{a_1}, \ldots, e^{a_m})$$

Redes neuronales y aprendizaje: al hablar de aprendizaje (o entrenamiento) de redes neuronales, hablamos de encontrar los pesos de las conexiones entre unidades, de manera que la red se comporte de una determinada manera (descrita por un conjunto de entrenamiento).

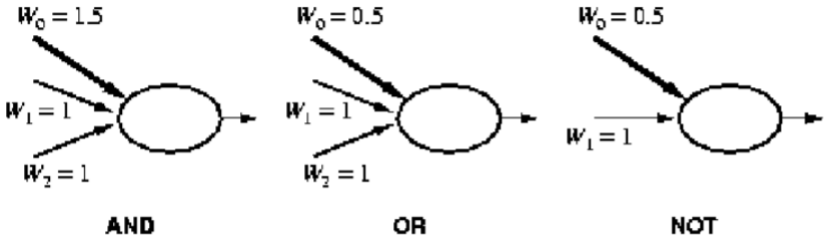
Para redes neuronales planteamos la siguiente tarea de aprendizaje supervisado:

- Dado un conjunto de entrenamiento **D = {(Xd, Yd) : Xd ∈ R^n, Yd ∈ R^m, d = 1,…,k}**
- Y una red neuronal de la que sólo conocemos su estructura: capas, nº de unidades en cada capa, función de activación en cada capa.
- Encontrar un conjunto de pesos **Wij** 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.
- Necesitamos especificar que es eso de “lo mejor posible”.

Aplicaciones prácticas de redes neuronales:

- Resolver problemas que puedan expresarse numéricamente ya sean discretos o continuos.
- Dominios en los que el volumen de los datos sea muy alto y pueda presentar ruido (cámaras, micrófonos, tratamiento de imágenes digitales, etc).
- Problemas en los que interesa la solución, pero no el "por qué de la solución".
- Problemas en los que se asume previamente que se necesita de mucho tiempo para el entrenamiento de la red.
- Problemas en los que se requiera de muy poco tiempo para evaluar nuevas instancias (Ej: coche conducción autónoma).

Perceptrones: el caso más simple de red neuronal → sólo una capa de entrada y una de salida. Este tipo de red se denomina perceptrón. Un perceptrón con función de activación umbral es capaz de representar las funciones booleanas básicas:



Un perceptrón *n* unidades de entrada, pesos *Wi(i=0,…,n)* y función de activación umbral, clasifica cómo positivos a aquellos (x1,…,xn) tal que: $\sum_{i=0}^{n} w_i x_i > 0$

- La ecuación anterior representa un hiperplano, con x0 = 1.
- Una función booleana 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; esto se denomina *linealmente separable*.
- Por ejemplo, las funciones AND y OR son linealmente separables, pero no así la función XOR.

Algoritmo de entrenamiento del perceptrón (umbral):

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

Algoritmo

1) Considerar unos pesos iniciales generados aleatoriamente $\vec{w} \leftarrow (w_0, w_1, \ldots, w_n)$
2) Repetir hasta condición de terminación:
   1) Para cada $(\vec{x}, y)$ del conjunto de entrenamiento hacer
      1) Calcular $o = umbral(\sum_{i=0}^{n} w_i x_i)$ (con $x_0 = 1$)
      2) Para cada peso w_i hacer: $w_i \leftarrow w_i + \eta(y - o)x_i$
3) Devolver $\vec{w}$

, dónde **η** es una constante positiva pequeña (0.1 por ejemplo), llamada factor de aprendizaje, que modera las actualizaciones de los pesos.

En cada actualización, si **y = 1** y **o = 0**, entonces **y – o = 1 > 0**, y por tanto los **Wi** correspondientes a **Xi** positivos aumentarán (y disminuirán los correspondientes a **Xi** negativos), lo que aproximará **o** (salida real) a **y** (salida esperada). Esto ocurre análogamente cuando **o = 1** e **y = 0**. Cuando **y = o**, los **Wi** no se modifican.

El algoritmo anterior converge en un nº finito de pasos a un vector de pesos **W** que clasifica correctamente todos los ejemplos de entrenamiento siempre que estos sean linealmente separables y **η** suficientemente pequeño.

Otro algoritmo de entrenamiento del perceptrón distinto: descenso por el gradiente:

- Cuando el conjunto de entrenamiento no es linealmente separable, la convergencia del algoritmo anterior no está garantizada.
- En tal caso, no es posible encontrar un perceptrón que, sobre TODOS los elementos del conjunto de entrenamiento, devuelva la salida esperada.
- En su lugar, intentaremos minimizar el error cuadrático:

$$E(\vec{w}) = \frac{1}{2}\sum_d (y_d - o_d)^2 = \frac{1}{2}\sum_d [y_d - g(w_0x_0 + w_1x_1 + \cdots + w_nx_n)]^2$$

, dónde:

- **g** es la función de activación y **Yd** es la salida esperada para la instancia **(Xd, Yd) ∈** $\mathbb{D}$.
- **Od** es la salida obtenida por el perceptrón.
- **E** es función de **w** y tratamos de encontrar un **w** que minimice **E**.
- En este caso, **g** suele ser el sigmoide.

La idea es comenzar con un **w** aleatorio y modificarlo sucesivamente en pequeños desplazamientos en la dirección opuesta al gradiente, esto es **w ← w + Δw**, siendo **Δw = −ηΔE(w)**, y **η** el factor de aprendizaje.

Algoritmo de entrenamiento de descenso por el gradiente:

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

Algoritmo

1) Considerar unos pesos iniciales generados aleatoriamente $\vec{w} \leftarrow (w_0, w_1, \ldots, w_n)$
2) Repetir un número de veces prefijado (*epochs*) lo siguiente:
   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 $\vec{w}$

Descenso por el gradiente: *batch vs estocástico*: el anterior algoritmo calcula el error que se comete con todos los ejemplos; para cada actualización de pesos tiene que recorrer TODOS los ejemplos, por eso se llama *batch*; sin embargo, es preferible usar una versión estocástica del algoritmo en la que cada vez que se trata un ejemplo, se produce una actualización de los pesos.

Descenso estocástico por el gradiente: regla delta:

- En lugar de tratar de minimizar el error cuadrático cometido sobre todos los ejemplos de **D**, procede incrementalmente tratando de descender el error cuadrático **Ed(w) = 1/2(y – o)^2**, cometido sobre el ejemplo **(x,y) ∈ D** que se esté tratando en cada momento.

Algoritmo de entrenamiento de descenso por el gradiente con la regla delta:

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

Algoritmo

1) Considerar unos pesos iniciales generados aleatoriamente $\vec{w} \leftarrow (w_0, w_1, \ldots, w_n)$
2) Repetir un número de veces igual a *epochs* lo siguiente:
   1) Para cada $(\vec{x}, y) \in D$ (tomados en orden aleatorio):
      1) Calcular $in = \sum_{i=0}^{n} w_i x_i$ y $o = g(in)$
      2) Para cada peso w_i, hacer
         $w_i \leftarrow w_i + \eta(y - o)g'(in)x_i$
3) Devolver $\vec{w}$

Casos particulares de la regla delta:

- Perceptrones con función de activación lineal:
  - **g'(in) = C** (constante)
  - La regla delta queda (transformando **η**): **Wi ← Wi + η (y – o)Xi**
- Perceptrones con función de activación sigmoide:
  - **g'(in) = g(in)(1 – g(in)) = o(1 – o)**
  - La regla delta queda: **Wi ← Wi + η(y – o)o (1 – o)Xi**

Redes Multicapa: dado que los perceptrones tienen una capacidad expresiva limitada, introducimos las redes multicapas. Una red multicapa es una red cuyas unidades se estructuran en capas; cada una de ellas recibe la entrada de las unidades de la capa anterior (capa de entrada, oculta o intermedia y salida). La expresividad de la red aumenta en tanto combinemos más unidades en distintas capas (siempre que la función de activación no sea lineal).

Entrenamiento en redes multicapa:

- Análogamente al caso del perceptrón, tenemos un conjunto de entrenamiento D tal que cada **(x,y) ∈ D** contiene una salida esperada **y ∈ R^m** para la entrada **x ∈ R^n**.
- Se parte de una red multicapa con una estructura dada y queremos encontrar los pesos de la red que se ajusten mejor a los ejemplos; esto lo haremos mediante un proceso de actualizaciones sucesivas llamado algoritmo de retropropagación, basado en las ideas del descenso por el gradiente.

Notación:

- Suponiendo 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 **L** la de salida.
- Cada unidad de una capa ***l*** está conectada con todas las unidades de la capa ***l + 1***.
- Suponemos funciones de activación ***Gi*** que pueden ser distintas en cada capa.
- El peso de la arista entre la unidad ***i*** y la unidad ***j*** se denota ***Wij***.
- Dado un ejemplo **(x,y) ∈ D**:
    - Si **i** es una unidad de la capa de entrada, notaremos por **Xi** la componente de **X** correspondiente a dicha unidad.
    - Si **k** es una unidad de la capa de salida, notaremos por **Yk** la componente de **Y** correspondiente a dicha unidad.
- Al calcular la salida real que la red obtiene al recibir como entrada un ejemplo **X**, notaremos **INi** a la entrada que recibe una unidad **i** cualquiera y **Ai** a la salida de la misma unidad **i**.
    - Si **i** es una unidad de entrada, entonces **ai = xi.**
    - Si **i** es una unidad de una capa **I != 1**, entonces **ai = gi(INi)** y $in_i = \sum_j w_{ji} a_j$

Algoritmo de retropropagación para entrenamiento en redes multicapa:

Entrada: Un conjunto de entrenamiento D (con ejemplos de la forma $(\vec{x}, \vec{y})$, con $\vec{x} \in R^n$ e $\vec{y} \in R^m$), un factor de aprendizaje η y una estructura de *red*

Algoritmo

1) Inicializar los pesos de la *red* (aleatoriamente, usualmente con valores cercanos a cero, positivos o negativos)
2) Repetir hasta que se satisfaga el criterio de parada
   1) Para cada ejemplo $(\vec{x}, \vec{y}) \in D$ hacer:
      1) Calcular la salida a_i de cada unidad i, propagando valores hacia adelante
      2) Calcular los errores Δ_i de cada unidad i y actualizar los pesos w_{ji}, propagando valores hacia detrás
3) Devolver *red*

Punto 2.1.1) Calcular la salida ai de cada unidad i, propagando valores hacia adelante:

Procedimiento

1) Para cada nodo i de la capa de entrada hacer $a_i \leftarrow x_i$
2) Para l desde 2 hasta L hacer
   1) Para cada nodo i de la capa l hacer $in_i \leftarrow \sum_j w_{ji} a_j$ y $a_i \leftarrow g_l(in_i)$ (donde en el sumatorio anterior hay un sumando por cada unidad j de la capa $l-1$)

Punto 2.1.2) Calcular los errores Δi de cada unidad i y actualizar los pesos Wij, propagando valores hacia detrás:

Procedimiento

1) Para cada unidad i en la capa de salida hacer $\Delta_i \leftarrow g_l'(in_i)(y_i - a_i)$
2) Para l desde $L-1$ hasta 1 (decrementando l) hacer
   1) Para cada nodo j en la capa l hacer
      1) $\Delta_j \leftarrow g_l'(in_j) \sum_i w_{ji} \Delta_i$ (donde el sumatorio anterior tiene un sumando por cada unidad i de la capa $l+1$)
      2) Para cada nodo i en la capa $l+1$ hacer $w_{ji} \leftarrow w_{ji} + \eta a_j \Delta_i$

Momentum en el algoritmo de retropropagación: existe el problema de los mínimos locales; una variante muy común es introducir un sumando adicional en la actualización de pesos. Este sumando hace que en cada actualización de pesos se tenga en cuenta también la actualización realizada en la iteración siguiente. La técnica del momentum puede ser eficaz para “escapar” de pequeños mínimos locales.