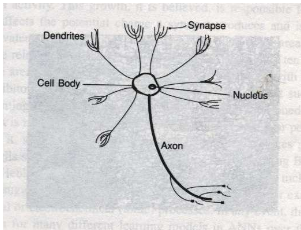
Tema 8: Introducción a las redes neuronales

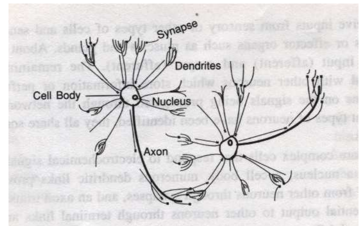
D. Balbontín Noval, F. J. Martín Mateos, J. L. Ruiz Reina,M. A. Gutiérrez-Naranjo, L. Valencia Cabrera

Dpto. Ciencias de la Computación e Inteligencia Artificial,Universidad de Sevilla

Neuronas artificiales: inspiración biológica

- El aprendizaje en los sistemas biológicos está basado en redes muy complejas de neuronas interconectadas
- La neurona es una célula que recibe señales electromagnéticas, provenientes del *exterior* (10 %), o de otras *neuronas* (90 %), a través de las *sinapsis* de las *dendritas*
- Si la acumulación de estímulos recibidos supera un cierto umbral, la neurona *se dispara*. Esto es, emite a través del *axón* una señal que será recibida por otras neuronas, a través de las *conexiones sinápticas* de las *dendritas*





Neuronas artificiales: inspiración biológica

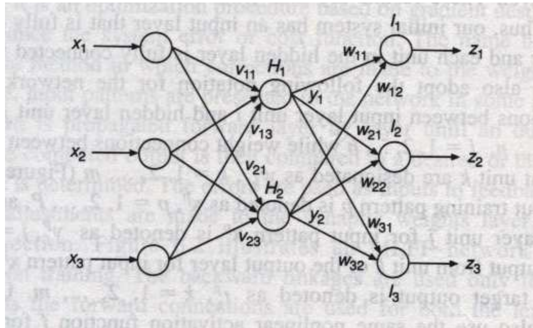
- El área de la *conexión sináptica* puede potenciar o debilitar la señal recibida. Las conexiones sinápticas son dinámicas. Con el desarrollo y el aprendizaje algunas conexiones se potencian, otras se debilitan
- Cerebro humano: red de neuronas interconectadas
  - Aproximadamente 10^{11} neuronas con 10^{4} conexiones cada una
- Las neuronas son lentas, comparadas con los ordenadores: 10^{-3} segundos 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*

Neuronas artificiales: inspiración biológica

- Inspiradas en estos procesos biológicos, surgen las *redes neuronales artificiales* como un modelo computacional
- Sin embargo, no debe de olvidarse que se trata de un modelo formal:
  - Algunas características de los sistemas biológicos no están reflejadas en el modelo computacional y viceversa
- Nosotros las estudiaremos como un modelo matemático en el que se basan potentes algoritmos de aprendizaje automático, independientemente de que reflejen un sistema biológico o no

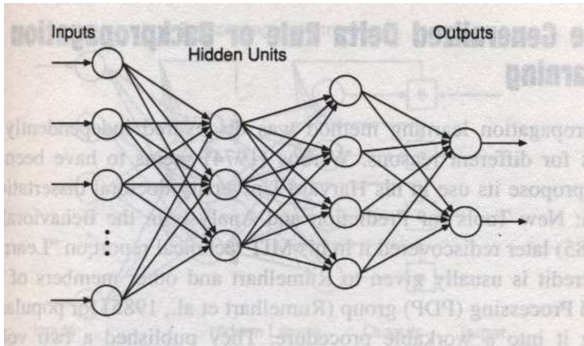
Redes Neuronales Artificiales (RNA)

- Modelo matemático basado en una estructura de grafo dirigido cuyos nodos son *neuronas artificiales*. Por ejemplo:



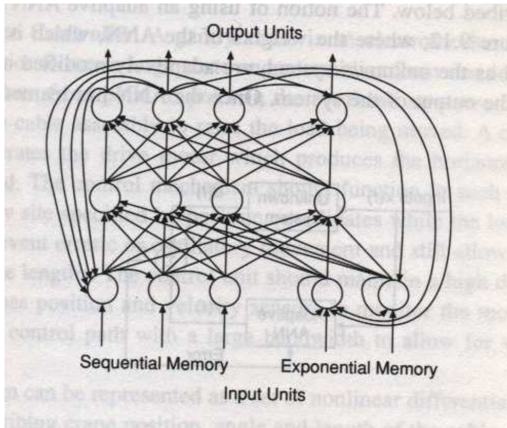
Redes Neuronales Artificiales (RNA)

- Modelo matemático basado en una estructura de grafo dirigido cuyos nodos son *neuronas artificiales*. Por ejemplo:



Redes Neuronales Artificiales (RNA)

- Modelo matemático basado en una estructura de grafo dirigido cuyos nodos son *neuronas artificiales*. Por ejemplo:



Funcionamiento general de una red neuronal

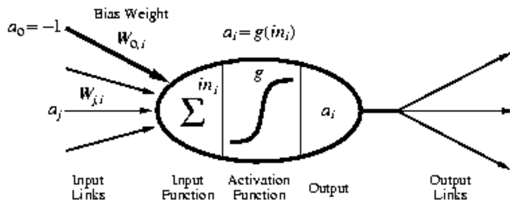
- Cada nodo o *unidad* (*neurona artificial*), se conecta a otras unidades a través de *arcos dirigidos* (modelando la *conexión axón* $\rightarrow$ *dendritas*)
- Cada arco $j \rightarrow i$ sirve para propagar la *salida* de la unidad j (notada a_j) que servirá como una de las *entradas* para la unidad i. Las entradas y salidas son *números*
- Cada arco $j \rightarrow i$ tiene asociado un peso numérico w_{ji} que determina la fuerza y el signo de la conexión (simulando la *sinapsis*)

Funcionamiento general de una red neuronal

- Cada unidad calcula su salida en función de las entradas que recibe
- La salida de cada unidad sirve, a su vez, como una de las entradas de otras neuronas
    - El cálculo que se realiza en cada unidad será muy simple, como veremos
- La red recibe una serie de entradas externas (*unidades de entrada*) y devuelve al exterior la salida de algunas de sus neuronas, llamadas *unidades de salida*

Cálculo realizado por cada unidad

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



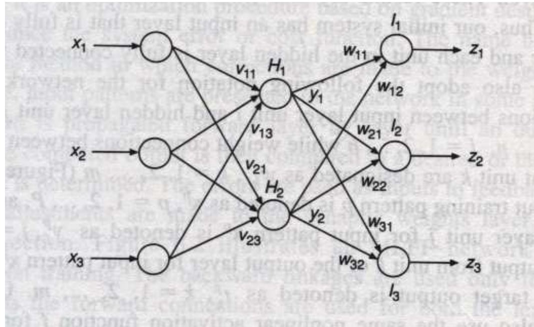
- Donde:
  - g es una *función de activación*
  - El sumatorio $\sum_{j=0}^{n} w_{ji} a_j$ (notado in_i) se hace sobre todas las unidades j que envían su salida a la unidad i
  - Excepto para $j = 0$, que se considera una entrada ficticia $a_0 = -1$ y un peso w_{0i} denominado *umbral* (ver enlace)

Umbral y funciones de activación

- Intuitivamente, el umbral w_{0i} de cada unidad se interpreta como una cantidad que debe superar la suma de las señales de entrada que recibe la unidad, para que se active
- La función de activación g tiene el papel de "normalizar" la salida (usualmente a 1) cuando el umbral de entrada se supera. Además hace que la red no se comporte simplemente como una función lineal
- Funciones de activación más usadas:
  - Función bipolar: $sgn(x) = \begin{cases} 1 & \text{si} \quad x > 0 \\ -1 & \text{si} \quad x \leq 0 \end{cases}$
  - Función umbral: $umbral(x) = \begin{cases} 1 & \text{si} \quad x > 0 \\ 0 & \text{si} \quad x \leq 0 \end{cases}$
  - Función sigmoide: $\sigma(x) = \frac{1}{1+e^{-x}}$ (ver esta hoja)
  - La función sigmoide es derivable y $\sigma'(x) = \sigma(x)(1 - \sigma(x))$

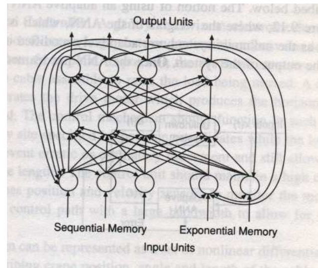
Redes neuronales hacia adelante

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



Redes neuronales 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 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



Esto es solo una introducción, hay mucho más[1].

[1] The Neural Network Zoo: `http://www.asimovinstitute.org/neural-network-zoo/`

Redes neuronales como clasificadores

- 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 de $\mathbb{R}^n$ en $\mathbb{R}^m$
- Por tanto, puede usarse como clasificador de conjuntos en $\mathbb{R}^n$:
  - Para **clasificación booleana**, tomar $m = 1$ y:
    - Si se usa función umbral o bipolar, considerar un valor de salida (*e.g.*, el 1) como "SI" y el otro como "NO"
    - Si se usa sigmoide o tanh, considerar una salida por encima de 0.5 como "SI" y un valor por debajo como "NO"
  - Para **clasificaciones con m posibles valores**:
    - Cada unidad de salida representa un valor de clasificación
    - La **unidad con mayor valor de salida** es la que indica el valor de clasificación

Redes Neuronales y Aprendizaje

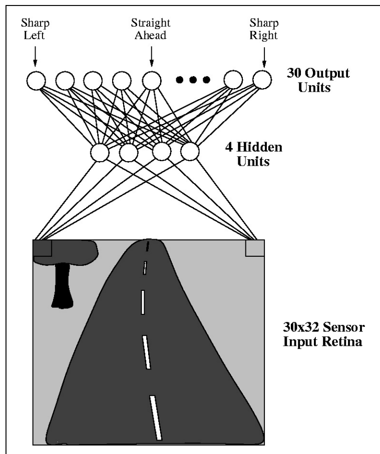
- Cuando hablamos de *aprendizaje* o *entrenamiento* de redes neuronales estamos hablando de:
  - **Encontrar los pesos** de las conexiones entre neuronas, de modo que la red se comporte *de manera similar a la dada* por un conjunto de entrenamiento
- Específicamente, para redes neuronales hacia adelante, es habitual plantear la siguiente tarea de *aprendizaje supervisado*:
  - Dado un conjunto de entrenamiento $D = \{(\vec{x_d}, \vec{y_d}) : \vec{x_d} \in R^n, \vec{y_d} \in R^m, d = 1, \ldots, k\}$
  - 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_{ji} 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
- Tendremos que concretar lo que significa "lo mejor posible"

Aplicaciones prácticas de redes neuronales

- 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
- 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: un ejemplo de aplicación

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



ALVINN: un ejemplo de aplicación

- Entrenamiento: mediante un conductor humano, que conduce el vehículo una y otra y otra vez...
  - Los sensores de visión registran la imagen que el conductor ve (secuencias de 960 datos cada una)
  - Otros sensores registran simultáneamente las acciones (movimientos del volante) que éste realiza
- Una vez codificadas ambas informaciones adecuadamente, disponemos de distintos pares (secuencias) de la forma $(\vec{x}, \vec{y})$, donde $\vec{x} = (x_1, x_2, \ldots, x_{960})$ e $\vec{y} = (y_1, y_2, \ldots, y_{30})$, constituyen ejemplos de *entrada/salida* para la red
- Objetivo: encontrar los valores de los pesos w_{ji} asociados a cada arco $j \to i$ de la red de tal forma que para cada dato de entrada $\vec{x}$, que propaguemos a lo largo de la red el valor obtenido en la salida coincida con el valor $\vec{y}$ correspondiente (o *se parezca lo más posible*)

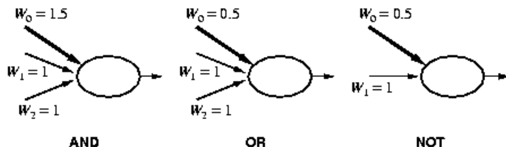
Ejemplos de aplicaciones prácticas

- Clasificación
- Reconocimiento de patrones
- Optimización
- Predicción: climatológica, de audiencias, etc
- Interpretación de datos sensoriales del mundo real
  - Reconocimiento de voz
  - Visión artificial, reconocimiento de imágenes
- Satisfacción de restricciones
- Control de robots, vehículos, etc
- Compresión de datos
- Diagnosis

Puede ver algunos ejemplos ilustrativos experimentales en este enlace.

Perceptrones

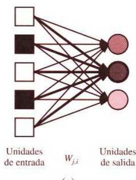
- Empezamos estudiando el caso más simple de red neuronal: sólo una capa de entrada y una de salida
  - Puesto que cada salida es independiente, podemos centrarnos en una única unidad en la capa de salida
- Este tipo de red se denomina *perceptrón*
- Un perceptrón con función de activación umbral es capaz de representar las funciones booleanas básicas:

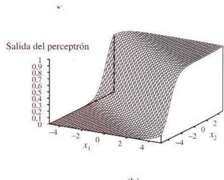


Nota: *recordemos que, como vimos en clase, la función NOT no está correctamente representada por los pesos dados en el perceptrón del gráfico de la derecha, en el que se deben invertir los signos de ambos pesos.*

Perceptrones: limitaciones expresivas

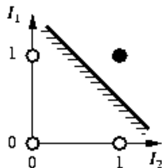
- Un perceptrón con n unidades de entrada, pesos $w_j (j = 0, \ldots, n)$ y función de activación umbral (o bipolar), clasifica como positivos a aquellos $(x_1, \ldots, x_n)$ tal que $\sum_{j=0}^{n} w_j x_j > 0$ (donde $x_0 = -1$)
  - La ecuación $\sum_{j=0}^{n} w_j x_j = 0$ representa un *hiperplano* en R^n
  - Es decir, 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 (*linealmente separable*)
- Los perceptrones con activación sigmoide tienen limitaciones expresivas similares (aunque "suavizadas")



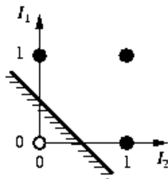


Perceptrones: limitaciones expresivas

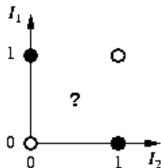
- Por ejemplo, las funciones AND y OR son linealmente separables pero no la función XOR:



(a) I_1 and I_2



(b) I_1 or I_2



(c) I_1 xor I_2

- 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

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 que se cumpla la condición de terminación
   2.1) Para cada $(\vec{x}, y)$ del conjunto de entrenamiento hacer
      a) Calcular $o = umbral(\sum_{i=0}^{n} w_j x_j)$ (con $x_0 = -1$)
      b) Para cada peso w_j hacer: $w_j \leftarrow w_j + \eta(y - o)x_j$
3) Devolver $\vec{w}$

Comentarios sobre el algoritmo

- 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 $y - o = 1 > 0$, y por tanto los w_j correspondientes a x_j positivos aumentarán (y disminuirán los correspondientes a x_j negativos), lo que aproximará o (salida real) a y (salida esperada)
- Análogamente ocurre si es $o = 1$ e $y = 0$
- Cuando $y = o$, los w_j no se modifican
- Para perceptrones con función de activación bipolar, el algoritmo es análogo

Comentarios sobre el algoritmo

- 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 η *suficientemente pequeño* (Minsky and 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

Otro algoritmo de entrenamiento: la Regla Delta

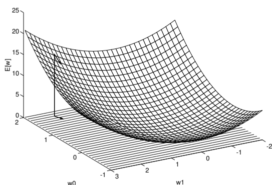
- Cuando el conjunto de entrenamiento no es linealmente separable, la convergencia del algoritmo anterior no está garantizada
  - El perceptrón no podrá asegurar devolver la salida esperada sobre *todos* los ejemplos del conjunto de entrenamiento
  - En su lugar intentaremos **minimizar el error cuadrático** acumulado sobre los ejemplos[2]:
    $E(\vec{w}) = \frac{1}{2}\sum_d (y_d - o_d)^2 = \frac{1}{2}\sum_d [y_d - g(w_0 x_{0d} + w_1 x_{1d} + \cdots + w_n x_{nd})]^2$

    con g la función de activación, y_d salida esperada para la instancia $(\vec{x}_d, y_d) \in D$, y o_d salida obtenida por el perceptrón
  - Siendo E función de $\vec{w}$, buscamos **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

[2] Se toma la mitad (del error acumulado) para facilitar cálculos posteriores

Idea del método del descenso por el gradiente

- Representación gráfica de $E(\vec{w})$ (con $n = 1$ 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.
¿No queda claro? Pinche aquí.

- 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 el 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 $\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$, siendo $\Delta\vec{w} = -\eta\nabla E(\vec{w})$, y η el *factor de aprendizaje*

Puede ver una explicación ilustrada y simplificada aquí, y una hoja con un ejemplo simplificado analizando la influencia del factor de aprendizaje aquí.

Derivación de la regla de descenso por el gradiente[3]

- El gradiente es el vector de las derivadas parciales de E respecto a cada w_j

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

- Sea x_{jd} la componente j-ésima del ejemplo d-ésimo e $in^{(d)} = \sum_{j=0}^{n} w_j x_{jd}$:

$$\frac{\partial E}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{2} \sum_d (y_d - o_d)^2 = \sum_d (y_d - o_d)(-g'(in^{(d)}))x_{jd}$$

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

$$w_j \leftarrow w_j + \eta \sum_d (y_d - o_d) g'(in^{(d)}) x_{jd}$$

[3] ¿Es el único método? No, puede encontrar más información aquí

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 y una función de activación g diferenciable

Algoritmo

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_j a cero, para $j = 0, \ldots, n$
   2) Para cada $(x, y) \in D$,
      1) Calcular $in = \sum_{j=0}^{n} w_j x_j$ y $o = g(in)$
      2) Para cada $j = 0, \ldots, n$, hacer $\Delta w_j \leftarrow \Delta w_j + \eta(y - o)g'(in)x_j$
   3) Para cada peso w_j, hacer $w_j \leftarrow w_j + \Delta w_j$
3) Devolver $\vec{w}$

La 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 de D, procede *incrementalmente* tratando de **descender el error** cuadrático $E_d(\vec{w}) = \frac{1}{2}(y-o)^2$, cometido **sobre el ejemplo** $(\vec{x}, y) \in D$ **tratado** en cada momento
  - De esta forma, $\frac{\partial E_d}{\partial w_j} = (y-o)(-g'(in))x_j$, y siendo $\Delta w_j = -\eta \frac{\partial E_d}{\partial w_j}$, tendremos $\Delta w_j = \eta(y-o)g'(in)x_j$, y por tanto $w_j \leftarrow w_j + \eta(y-o)g'(in)x_j$
  - Este método para actualizar los pesos iterativamente es conocido como *Regla Delta*

Entrenamiento de Perceptrones 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 y una función de activación g diferenciable

Algoritmo

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) Para cada $(\vec{x}, y) \in D$
        1) Calcular $in = \sum_{j=0}^{n} w_j x_j$ y $o = g(in)$
        2) Para cada peso w_j, hacer $w_j \leftarrow w_j + \eta(y - o)g'(in)x_j$
3) Devolver $\vec{w}$

Casos particulares de la Regla Delta

- Perceptrones con función de activación *lineal*:
  - En este caso $g'(in) = C$ (constante)
  - Por tanto, la Regla Delta queda (transformando η convenientemente):

$$w_j \leftarrow w_j + \eta(y - o)x_j$$

- 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:

$$w_j \leftarrow w_j + \eta(y - o)o(1 - o)x_j$$

Algunos comentarios sobre la Regla Delta

- 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 el gradiente converge (puede que asintóticamente) hacia un mínimo local del error cuadrático global

Algunos comentarios sobre la Regla Delta[4]

- 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

[4] Puede hacer uso de una hoja de cálculo para Descenso de Gradiente (DG) y Regla Delta (RD) en este enlace

Regla Delta y perceptrones con umbral

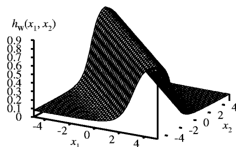
- La regla de entrenamiento del perceptrón con umbral, y la Regla Delta para el entrenamiento de perceptrones lineales, son aparentemente la misma: $w_j \leftarrow w_j + \eta(y - o)x_j$, pero:
  - Las funciones de activación son distintas
  - Las propiedades de convergencia también:
    - Umbral: converge en un número finito de pasos hacia un ajuste perfecto, siempre que el conjunto de entrenamiento sea linealmente separable
    - Regla Delta: converge asintóticamente hacia un mínimo local del error cuadrático, siempre
  - Las propiedades de separación también son distintas:
    - Umbral: busca hiperplano que separe completamente los datos
    - Regla Delta: busca un modelo de regresión, el hiperplano (posiblememente suavizado con el sigmoide) más próximo a los datos de entrenamiento

Redes multicapa (hacia adelante)

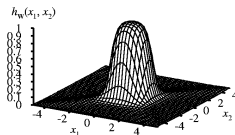
- Como hemos visto, los perceptrones tienen una capacidad expresiva limitada. Es por esto que vamos a estudiar las redes multicapa
- Recordar que en una red multicapa, las unidades se estructuran en *capas*, en las que las unidades de cada capa reciben su entrada de la salida de las unidades de la capa anterior, y de modo que (*ojo a la dificultad*):
  - Capa de *entrada* es la que contiene las unidades de entrada
  - Capa de *salida* es la de las unidades cuya salida sale al exterior
  - Capas *ocultas* son las que no son de entrada ni de salida

Redes multicapa: capacidad expresiva

- Combinando unidades en distintas capas (y siempre que la función de activación sea no lineal) aumentamos la capacidad expresiva de la red
- Es decir, la cantidad de funciones $f : R^n \rightarrow R^m$ que pueden representarse aumenta



(a)



(b)

Figure 20.23 (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.

- Para muchas aplicaciones reales basta con una capa oculta.

Entrenamiento de redes multicapa

- Análogamente al caso del perceptrón, tenemos un conjunto de entrenamiento D tal que cada $(\vec{x}, \vec{y}) \in D$ contiene una salida esperada $\vec{y} \in R^m$ para la entrada $\vec{x} \in R^n$
- Partimos de una red multicapa con una estructura dada y queremos encontrar los pesos de la red de manera que la función que calcula la red se ajuste lo mejor posible a los ejemplos
- Lo haremos mediante un proceso de actualizaciones sucesivas de los pesos, llamado *algoritmo de retropropagación*, basado en las mismas ideas de descenso por el gradiente que hemos visto con el perceptrón

Redes multicapa: notación

- 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 l está conectada con todas las unidades de la capa $l+1$
- Supondremos una función de activación g diferenciable (usualmente, el sigmoide)
- El peso de la conexión de la unidad j a la unidad i se nota w_{ji}
- Dado un ejemplo $(\vec{x}, \vec{y}) \in D$:
  - Si i es una unidad de la capa de entrada, notaremos por x_i la componente de $\vec{x}$ correspondiente a dicha unidad
  - Si k es una unidad de la capa de salida, notaremos por y_k la componente de $\vec{y}$ correspondiente a dicha unidad

Redes multicapa: notación

- Al calcular la salida real que la red obtiene al recibir como entrada un ejemplo $\vec{x}$, notaremos in_i a la entrada que recibe una unidad i cualquiera y a_i a la salida por la misma unidad i
- Es decir:
  - Si i es una unidad de entrada (es decir, de la capa 1), entonces $a_i = x_i$
  - Si i es una unidad de una capa $l \neq 1$, entonces $in_i = \sum_j w_{ji} a_j$, y $a_i = g(in_i)$ (el sumatorio recorre todas las unidades j de la capa $l-1$)

Algoritmo de Retropropagación: idea intuitiva

- Dado un ejemplo $(\vec{x}, \vec{y}) \in D$, y una unidad i de la capa de salida, la actualización de los pesos que llegan a esa unidad se hará de manera similar a como se hace con la Regla Delta:
  - Sea $\Delta_i = g'(in_i)(y_i - a_i)$ (*error modificado* en la unidad i)
  - Entonces $w_{ji} \rightarrow w_{ji} + \eta \Delta_i a_j$
- En las unidades de capas ocultas, sin embargo, no podemos hacer lo mismo
  - Ya que no sabemos cuál es el valor de salida esperado en esas unidades

Algoritmo de Retropropagación: idea intuitiva

- ¿Cómo actualizamos los pesos de conexiones con capas ocultas?
- Idea: ir hacia atrás, calculando el error Δ_j de una unidad de la capa $l-1$ a partir del error de las unidades de la capa l (con las que está conectada j)
- Esto es: $\Delta_j = g'(in_j)\sum_i w_{ji}\Delta_i$ y por tanto $w_{ji} \rightarrow w_{ji} + \eta\Delta_i a_j$
- Intuitivamente, cada unidad j es "responsable" del error que tiene cada una de las unidades a las que envía su salida
    - Y lo es en la medida que marca el peso de su conexión
- La salida de cada unidad se calcula propagando valores hacia adelante, pero el error en cada una se calcula desde la capa de salida hacia atrás (de ahí el nombre de *retropropagación*)
- El método de retropropagación se puede justificar formalmente como descenso por el gradiente del error

El Algoritmo de Retropropagación

- 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, una función de activación g diferenciable 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*

- *En las siguientes transparencias desarrollamos 2.1.1) y 2.1.2)*

Propagación hacia adelante

- Desarrollamos con más detalle el punto 2.2.1) anterior: propagación hacia adelante para un ejemplo $(\vec{x}, \vec{y}) \in D$

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 $a_i \leftarrow g(in_i)$ (donde en el sumatorio anterior hay un sumando por cada unidad j de la capa $l - 1$)

Propagación hacia atrás

- Una vez calculados en el punto 2.1.1) los valores de in_i y a_i correspondientes al ejemplo $(\vec{x}, \vec{y}) \in D$, desarrollamos con más detalle el punto 2.1.2), propagar hacia atrás de los errores y actualizar los pesos

Procedimiento

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

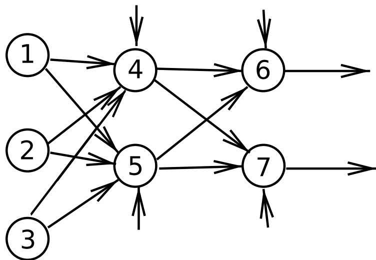
  - Para la capa de entrada ($l = 1$) no es necesario calcular los Δ_j
  - Si hubiera pesos umbral, después de calcular cada Δ_i se han de actualizar igualmente : $w_{0i} \leftarrow w_{0i} + \eta \Delta_i a_0$ (donde $a_0 = -1$)

Retropropagación con unidades sigmoide

- La versión más común del algoritmo de retropropagación es la que se realiza con redes con función de activación sigmoide
- Recordar que el sigmoide se define $\sigma(x) = \frac{1}{1+e^{-x}}$ y que $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- Por tanto, en el caso de que $g(x) = \sigma(x)$, entonces $g'(in_i) = g(in_i)(1 - g(in_i)) = a_i(1 - a_i)$
- 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

Un ejemplo de Retropropagación

- Considérese una red neuronal con la siguiente estructura en la que se usa el sigmoide como función de activación:



  - Supongamos dado un ejemplo (x_1, x_2, x_3) con salida esperada (y_6, y_7)
  - Supongamos también que *ya hemos calculado* la salida a_i en cada unidad $i = 1, \ldots, 7$

Traza de la Retropropagación del error

Capa	Unidad	Cálculos que se realizan
Salida	7	$\Delta_7 = a_7(1 - a_7)(y_7 - a_7)$
		$w_{0,7} \longleftarrow w_{0,7} + \eta \Delta_7 a_0$
	6	$\Delta_6 = a_6(1 - a_6)(y_6 - a_6)$
		$w_{0,6} \longleftarrow w_{0,6} + \eta \Delta_6 a_0$
Oculta	5	$\Delta_5 = a_5(1 - a_5)[w_{5,6}\Delta_6 + w_{5,7}\Delta_7]$
		$w_{0,5} \longleftarrow w_{0,5} + \eta \Delta_5 a_0$
		$w_{5,6} \longleftarrow w_{5,6} + \eta \Delta_6 a_5$
		$w_{5,7} \longleftarrow w_{5,7} + \eta \Delta_7 a_5$
	4	$\Delta_4 = a_4(1 - a_4)[w_{4,6}\Delta_6 + w_{4,7}\Delta_7]$
		$w_{0,4} \longleftarrow w_{0,4} + \eta \Delta_4 a_0$
		$w_{4,6} \longleftarrow w_{4,6} + \eta \Delta_6 a_4$
		$w_{4,7} \longleftarrow w_{4,7} + \eta \Delta_7 a_4$
Entrada	3	$w_{3,4} \longleftarrow w_{3,4} + \eta \Delta_4 a_3$
		$w_{3,5} \longleftarrow w_{3,5} + \eta \Delta_5 a_3$
	2	$w_{2,4} \longleftarrow w_{2,4} + \eta \Delta_4 a_2$
		$w_{2,5} \longleftarrow w_{2,5} + \eta \Delta_5 a_2$
	1	$w_{1,4} \longleftarrow w_{1,4} + \eta \Delta_4 a_1$
		$w_{1,5} \longleftarrow w_{1,5} + \eta \Delta_5 a_1$

Momentum en el algoritmo de retropropagación

- Retropropagación es un método de descenso por el gradiente y por tanto existe el problema de los mínimos locales
- Una variante común en el algoritmo de retropropagación es introducir un sumando adicional en la actualización de pesos
- Este sumando hace que en cada actualización de pesos se tenga también en cuenta la actualización realizada en la iteración anterior
- Concretamente:
  - En la iteración n-ésima, se actualizan los pesos de la siguiente manera: $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}^{(n)}$ donde $\Delta w_{ji}^{(n)} = \eta a_j \Delta_i + \alpha \Delta w_{ji}^{(n-1)}$
  - $0 < \alpha \leq 1$ es una constante denominada *momentum*
- La técnica del momentum puede ser eficaz a veces para escapar de "pequeños mínimos locales", donde una versión sin momentum se estancaría

Criterio de parada para retropropagación

- Nótese que el algoritmo podría recorrer *varias veces* el conjunto de entrenamiento
  - O podría tomar aleatoriamente el ejemplo a tratar en cada iteración
  - O incluso parar y recomenzar posteriormente el entrenamiento a partir de pesos ya entrenados
- Se pueden usar diferentes criterios de parada en el algoritmo de retropropagación. Por ejemplo:
  - Número de iteraciones prefijadas
  - Cuando el error sobre el conjunto de entrenamiento está por debajo de una cota prefijada
- En este último caso, se corre el riesgo de sobreajuste, por lo que lo más frecuente es usar un conjunto de prueba independiente para validar el error, o incluso validación cruzada

Aprendiendo la estructura de la red

- El algoritmo de retropropagación, ya parte de una **estructura** de red **fija**, pero...
  - No vimos qué estructuras son las mejores para cada problema
  - Debemos **decidir** a priori **cuántas capas ocultas** se toman, y **cuántas unidades en cada capa**
  - La elección es un problema que no está completamente resuelto aún
- ¿Cómo se toma esa decisión?
  - Usual: **búsqueda experimental** de la mejor estructura, medida sobre un conjunto de prueba independiente
  - En muchos casos una sola capa oculta con pocas unidades basta para obtener buenos resultados
  - Las redes grandes corren un mayor peligro de sobreajuste

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"