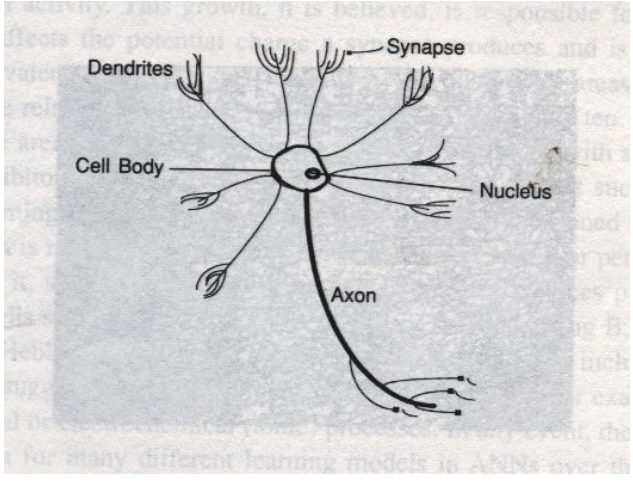
Tema 7: Introducción a las redes neuronales

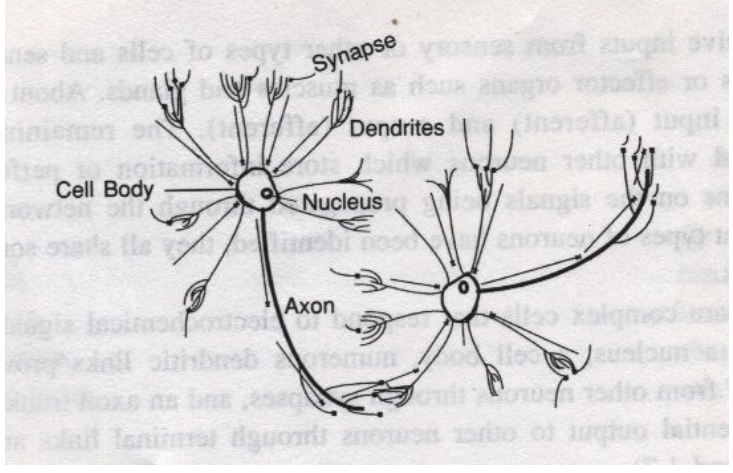
F. J. Martín Mateos
J. L. Ruiz Reina

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

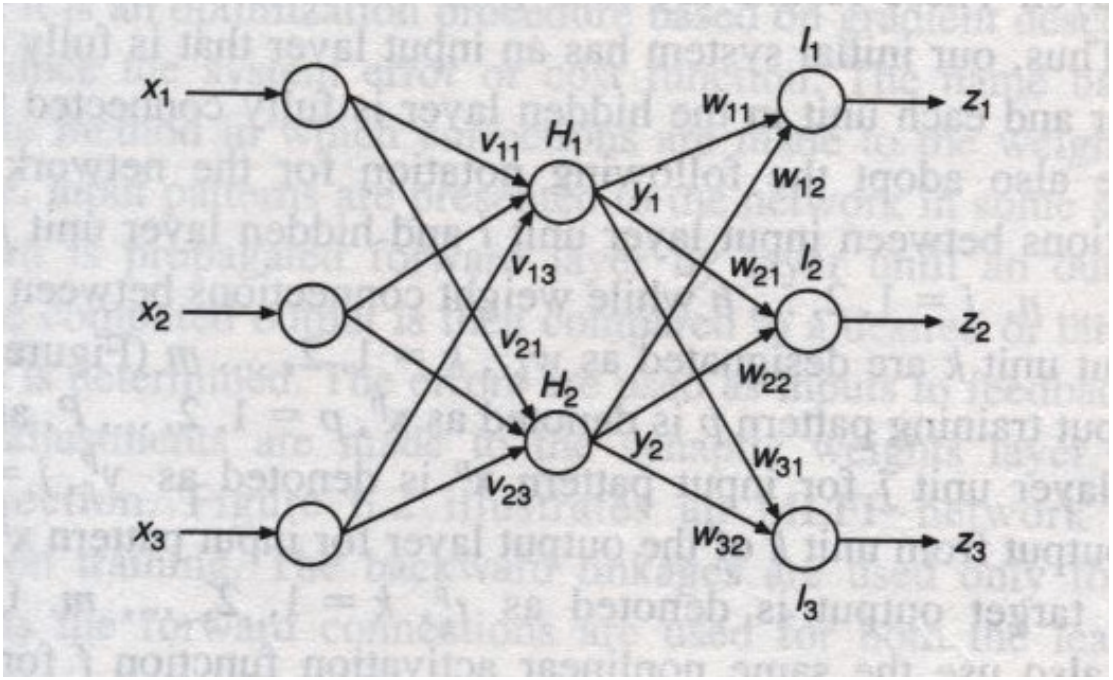
- El área de la *conexión sináptica* puede potenciar o debilitar la señal recibida. Las conexiones sinápticas son dinámicas. Con el desarrollo y el aprendizaje algunas conexiones se potencian, otras se debilitan
- Cerebro humano: red de neuronas interconectadas
  - Aproximadamente 10^{11} neuronas con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

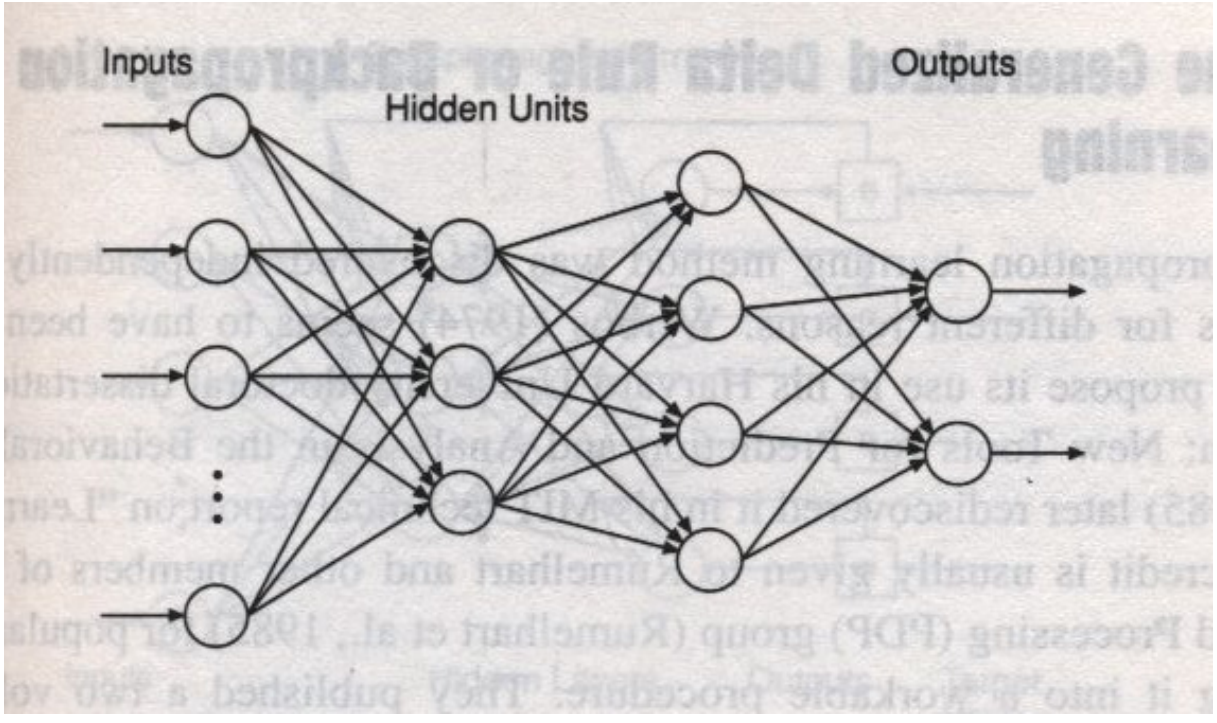
Redes Neuronales Artificiales (RNA)

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



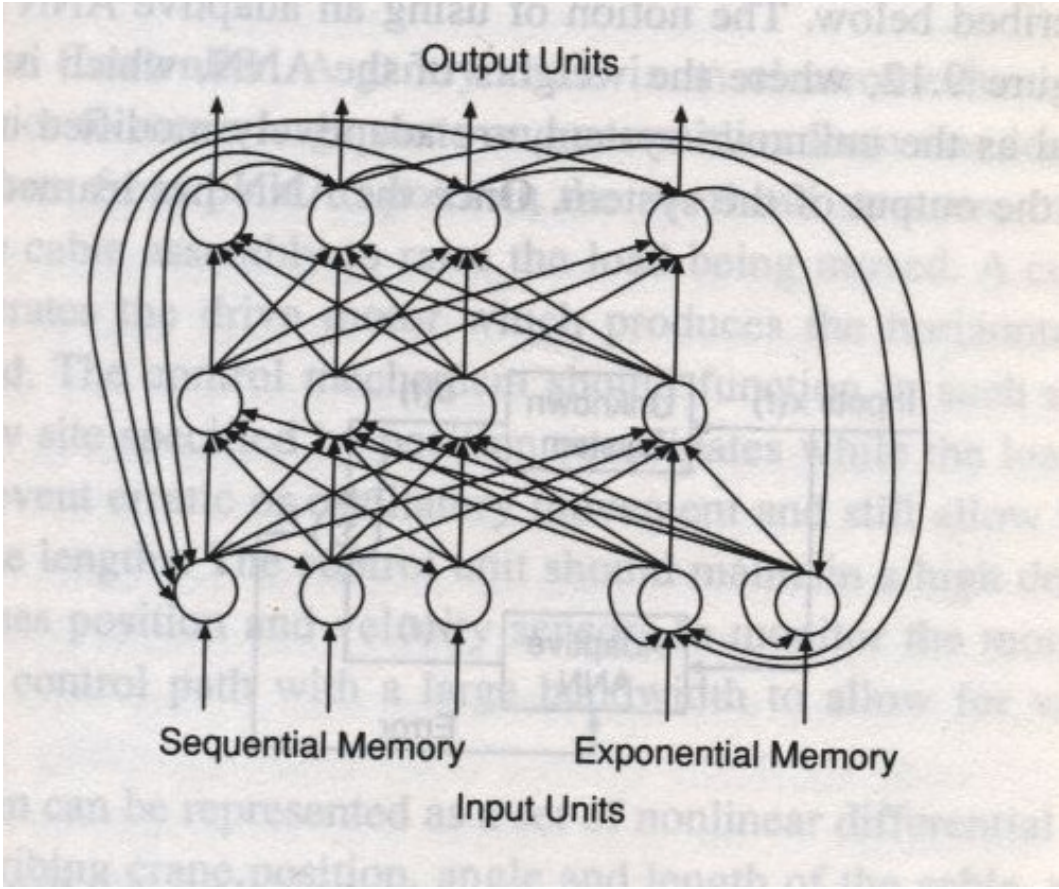
Redes Neuronales Artificiales (RNA)

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



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

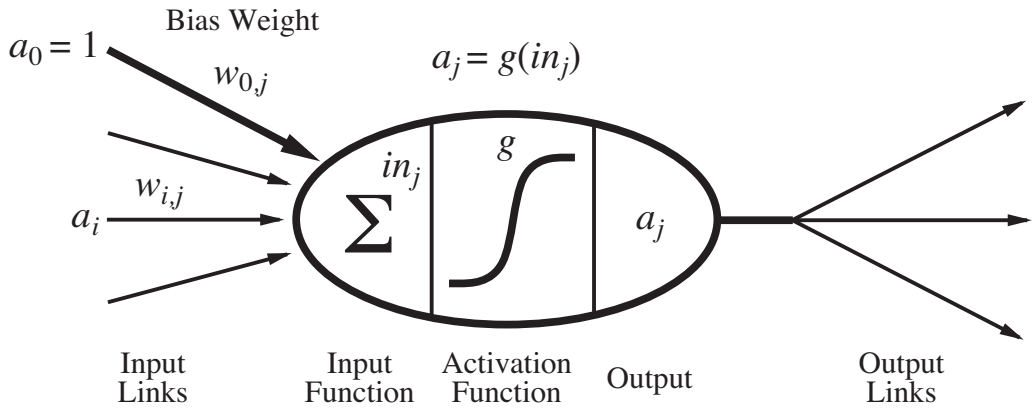
- Cada nodo o *unidad* (*neurona artificial*), se conecta a otras unidades a través de *arcos dirigidos*
- Cada arco $i \rightarrow j$ sirve para propagar la *salida* de la unidad i (notada a_i) que servirá como una de las *entradas* para la unidad j. Las entradas y salidas son *números*
- Cada arco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

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: $a_j = g(\sum_{i=0}^{n} w_{ij} a_i)$



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

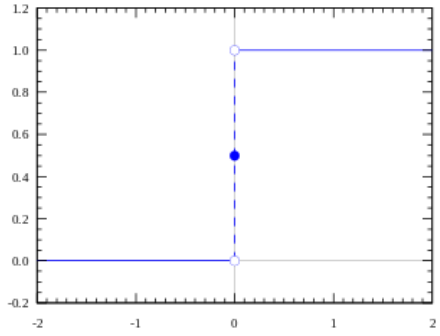
Umbral y funciones de activación

- Intuitivamente, el umbral w_{0j} de cada unidad se interpreta como el opuesto de una cantidad cuya entrada debe superar
- La función de activación g introduce cierta componente no lineal, hace que la red no se comporte simplemente como una función lineal, y aumenta la expresividad del modelo.
- Funciones de activación más usadas:
  - Función umbral: $umbral(x) = \begin{cases} 1 & \text{si} \quad x > 0 \\ 0 & \text{si} \quad x \leq 0 \end{cases}$
  - Sigmoide: $\sigma(x) = \frac{1}{1+e^{-x}}$
    - La función sigmoide es derivable y $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
  - ReLU (*Rectified Linear Unit*): $ReLU(x) = max\{0, x\}$
    - ReLU es derivable (execepto en 0) y su derivada es la función umbral
  - Tangente hiperbólica: $tanh(x) = \frac{2}{1+e^{-2x}} - 1$

Funciones de activación

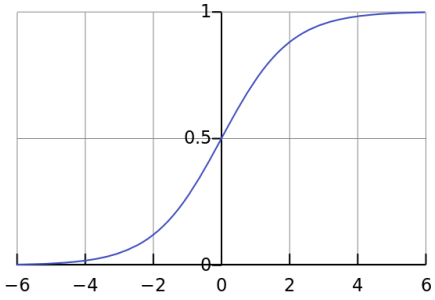
Función umbral

$$umbral(x) = \begin{cases} 1 & \text{si} \quad x > 0 \\ 0 & \text{si} \quad x \leq 0 \end{cases}$$



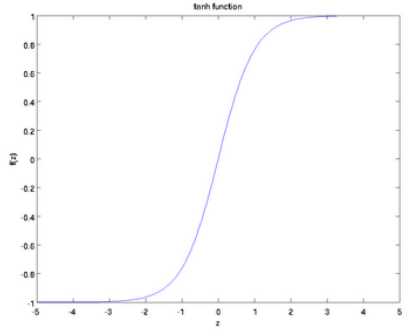
Función sigmoide

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



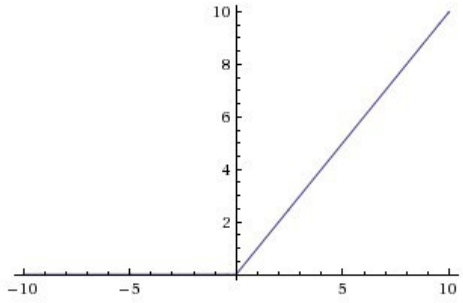
Tangente hiperbólica

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



ReLU

$$ReLU(x) = max\{0, x\}$$



Redes neuronales hacia adelante

- Capa de entrada, capas ocultas (intermedias) y capa de salida
  - La capa de entrada recoge simplemente la entrada
- Para unidades de la misma capa, la función de activación usada es la misma; entre capas distintas puede variar.
- Función de activación en capas intermedias: usualmente ReLU
- La capa de salida variará la función de activación en función del uso que se le quiera dar a la red:
  - Regresión (predicción de valores): sin función de activación
  - Clasificación binaria: una sólo unidad en la capa de salida con función umbral o mejor aún, sigmoide
  - Clasificación multiclase: sin función de activación (aunque luego se aplica *softmax*)

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 R^n en R^m
- Cuando se usan para clasificar, lo obtenido en la capa de salida se interpreta como una predicción de la pertenecia del dato de entrada a una clase.
- Distinguimos entre clasificación binaria (dos clases) y multiclase (más de dos)

Redes neuronales como clasificadores (binario)

- Para clasificación binaria (clases 1 y 0), tomar en la capa de salida una sola unidad y el sigmoide como función de activación
  - El valor de salida se interpreta como la *probabilidad* de pertenecer a la clase 1

Redes neuronales como clasificadores (multiclase)

- En general, para clasificaciones 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 clasificación
  - Es habitual normalizar las salidas con la función $softmax(a_1, \ldots, a_m) = \frac{1}{\sum_k e^{a_k}}(e^{a_1}, \ldots, e^{a_m})$, e interpretar la salida como en cada unidad como la probabilidad de pertenecer a la correspondiente clase.

Redes Neuronales y Aprendizaje

- Cuando hablamos de *aprendizaje* o *entrenamiento* de redes neuronales estamos hablando 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
- 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, número de unidades en cada capa y función de activación 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
- Tendremos que concretar lo que significa "lo mejor posible"

Aplicaciones prácticas de redes neuronales

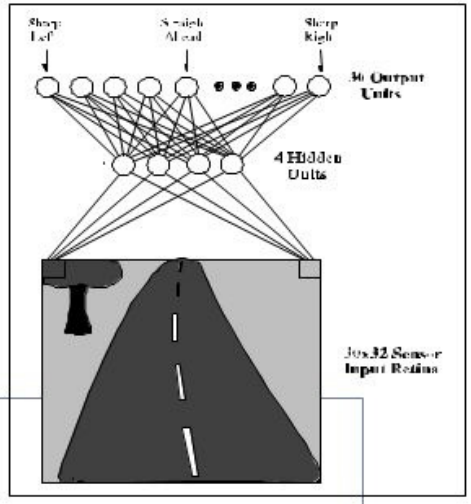
- Para problemas que se pueden expresar numéricamente (discretos o continuos)
- Se suelen utilizar en dominios en los que el volumen de datos es muy alto, y puede presentar *ruido*: cámaras, micrófonos, imágenes digitalizadas, etc
- En los que interesa la solución, pero no *el por qué* de la misma
-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

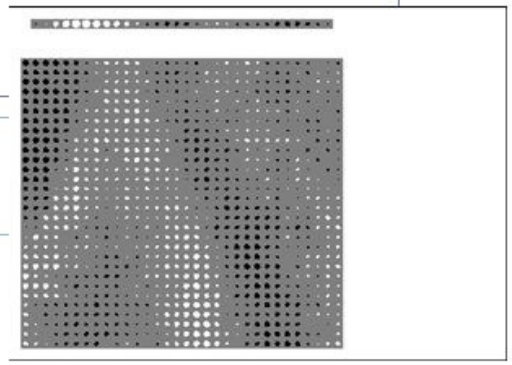
ALVINN

Coche autónomo de 1989

Neural Net example: ALVINN

- Autonomous vehicle controlled by Artificial Neural Network
- Drives up to 70mph on public highways





Note: most images are from the online slides for Tom Mitchell's book "Machine Learning"

ALVINN: un ejemplo de aplicación

- RNA entrenada para conducir un vehículo, a 70 Kms/h, en función de la percepción visual que recibe de unos sensores
- Entrada a la red (simplificación) : La imagen de la carretera digitalizada como un array de 30×32 pixels. Es decir, 960 datos de entrada
- Salida de la red (simplificación): 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

ALVINN: un ejemplo de aplicación

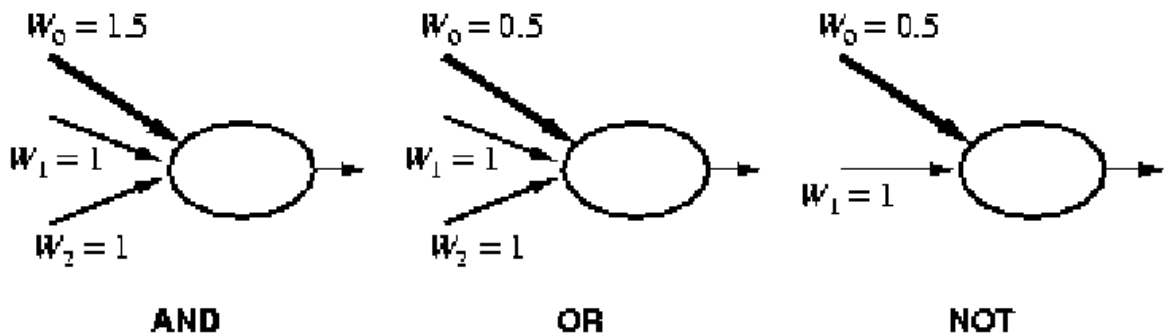
- Entrenamiento: mediante un conductor humano, que conduce el vehículo una y otra y otra vez
- Los sensores de visión registran la imagen que el conductor ve (secuencias de 960 datos cada una)
- Otros sensores registran simultáneamente las acciones (movimientos del volante) que éste realiza
- Una vez codificada 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

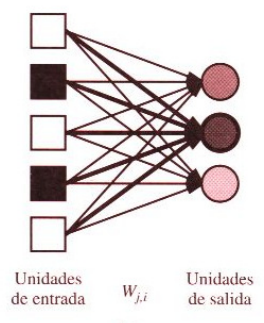
Perceptrones

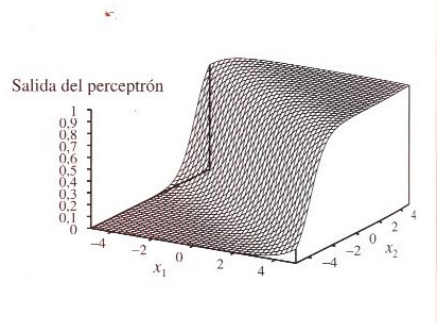
- Empezamos estudiando el caso más simple de red neuronal: sólo una capa de entrada y una de salida
  - Puesto que cada salida es independiente, podemos centrarnos en una única unidad en la capa de salida
- Este tipo de red se denomina *perceptrón*
- Un perceptrón con función de activación umbral es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



Perceptrones: limitaciones expresivas

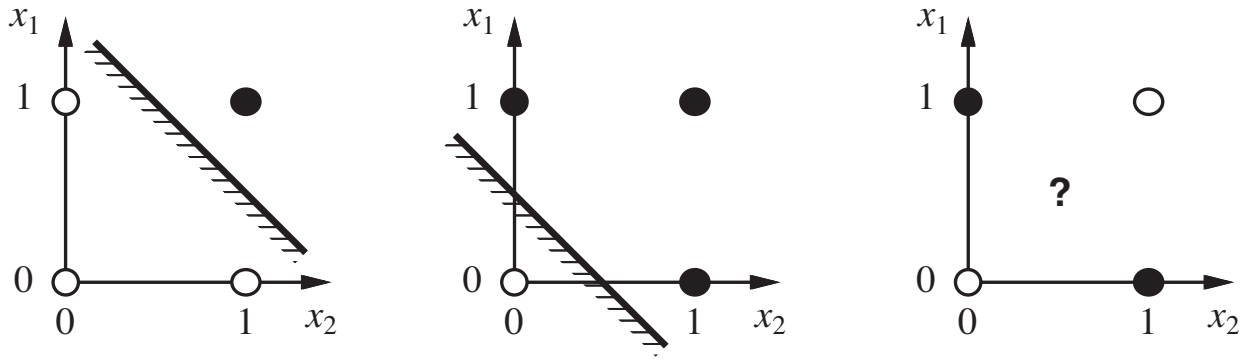
- Un perceptrón con n unidades de entrada, pesos $w_i (i = 0, \ldots, n)$ y función de activación umbral, clasifica como positivos a aquellos $(x_1, \ldots, x_n)$ tal que $\sum_{i=0}^{n} w_i x_i > 0$ (donde $x_0 = 1$)
  - La ecuación $\sum_{i=0}^{n} w_i x_i = 0$ representa un *hiperplano* en R^n
  - 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*)
- En espacios de muchas dimensiones esa restricción puede no ser importante
- Los perceptrones con activación sigmoide tienen capacidades expresivas similares (aunque "suavizadas")





Perceptrones: limitaciones expresivas

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



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_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 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 $\vec{w}$

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 actualización, si $y = 1$ y $o = 0$, entonces $y - o = 1 > 0$, y por tanto los w_i correspondientes a x_i positivos aumentarán (y disminuirán los correspondientes a x_i 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_i no se modifican

Comentarios sobre el algoritmo

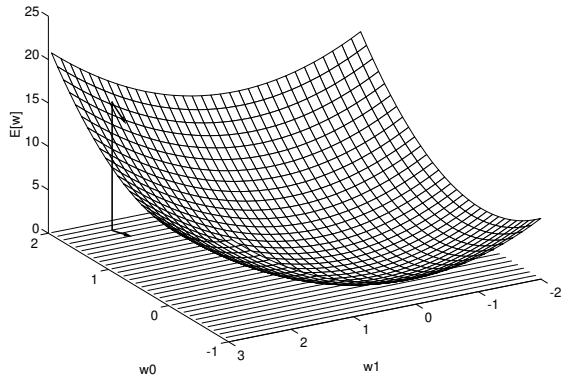
- Teorema: El algoritmo anterior *converge* en un número finito de pasos a un vector de pesos $\vec{w}$ que clasifica correctamente todos los ejemplos de entrenamiento, siempre que éstos sean *linealmente separables* y η *suficientemente pequeño* (Minsky and Papert, 1969)
- Por tanto, en el caso de conjuntos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

Otro algoritmo de entrenamiento: descenso por el 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
- En su lugar intentaremos minimizar el *error cuadrático*: $E(\vec{w}) = \frac{1}{2}\sum_d (y_d - o_d)^2 = \frac{1}{2}\sum_d [y_d - g(w_0 x_0 + w_1 x_1 + \cdots + w_n x_n)]^2$
  - Donde g es la función de activación, y_d es la salida esperada para la instancia $(\vec{x}_d, y_d) \in D$, y o_d 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 este caso g suele ser el sigmoide o la identidad.

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.

- 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*

Derivación de la regla de descenso por el 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 $x_{i,d}$ la componente i-ésima del ejemplo d-ésimo (y $x_{0,d} = 1$) y por $in^{(d)} = \sum_{i=0}^{n} w_i x_{i,d}$, entonces:

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

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

$$w_i \leftarrow w_i + \eta \sum_d (y_d - o_d) g'(in^{(d)}) x_{i,d}$$

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 gradiente del error que se comete con todos los ejemplos
  - Para cada actualización de pesos, necesita recorrer todos los ejemplos
  - Es por esto que se suele llamar versión *batch* del descenso por el gradiente
  - *Epoch:* cada recorrido completo del conjunto de entrenamiento
- 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 $E_d(\vec{w}) = \frac{1}{2}(y - o)^2$, cometido sobre el ejemplo $(\vec{x}, y) \in D$ que se esté tratando en cada momento
  - De esta forma, $\frac{\partial E_d}{\partial w_i} = (y - o)g'(in)(-x_i)$, y siendo $\Delta w_i = -\eta \frac{\partial E_d}{\partial w_i}$, tendremos $\Delta w_i = \eta(y - o)g'(in)x_i$, 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 descenso estocástico por el gradiente y la fórmula de actualización como *Regla Delta*
  - Estocástico: porque en cada epoch los ejemplos se toman en un orden aleatorio

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, 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*:
  - 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:

$$w_i \leftarrow w_i + \eta(y - o)o(1 - o)x_i$$

Algunos comentarios sobre descenso por el gradiente

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

Algunos comentarios sobre descenso por el gradiente

- Se puede demostrar que haciendo el valor de η suficientemente pequeño, la versión estocástica se puede aproximar arbitrariamente a la versión batch
- 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

Regla Delta y perceptrones con umbral

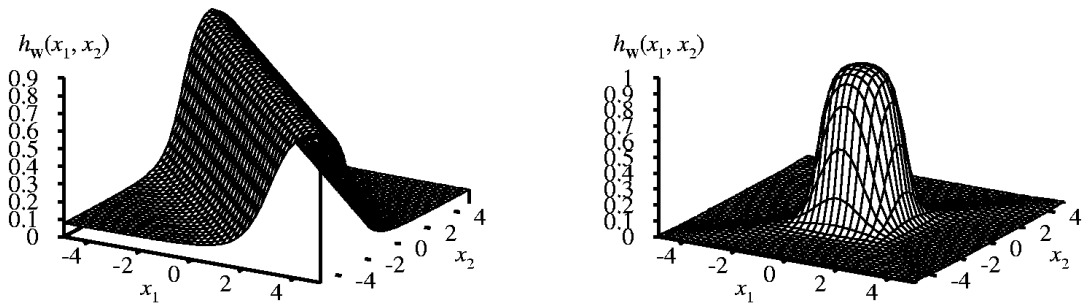
- La regla de actualización del perceptrón con umbral, y la Regla Delta para el entrenamiento de perceptrones lineales, son aparentemente la misma: $w_i \leftarrow w_i + \eta(y - o)x_i$, pero:
  - Las funciones de activación son distintas
  - Las propiedades de convergencia también:
    - Umbral: converge 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 (posiblemente 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
  - Capa de *entrada* es aquella en la que se sitúan las unidades de entrada
  - Capa de *salida* es la de las unidades cuya salida sale al exterior
  - Capas *ocultas* son aquellas que no son ni 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.

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 funciónes de activación g_l, que pueden ser distintas en cada capa
- El peso de la conexión entre la unidad i y la unidad j se nota w_{ij}
- 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 una unidad de una capa $l \neq 1$, entonces $in_i = \sum_j w_{ji} a_j$ y $a_i = g_l(in_i)$ (donde el sumatorio anterior se realiza en 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 $\Delta_i = g_i'(in_i)(y_i - a_i)$ (*error modificado* en la unidad i)
  - Entonces $w_{ji} \rightarrow w_{ji} + \eta a_j \Delta_i$
- 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: $\Delta_j = g_l'(in_j) \sum_i w_{ji} \Delta_i$ y por tanto $w_{kj} \rightarrow w_{kj} + \eta a_k \Delta_j$
- 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 la conexión entre ellas
- 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, pero no veremos aquí la demostración

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

Propagación hacia atrás

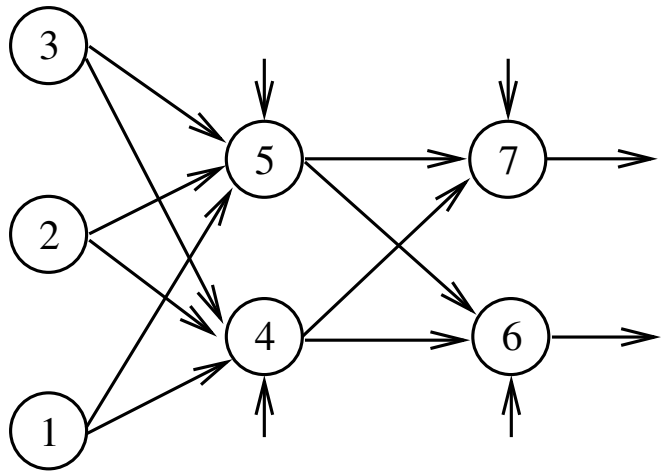
Una vez calculados en el punto 2.1.1 los valores de in_i y a_i correspondientes al ejemplo $(\vec{x}, \vec{y}) \in D$, desarrollamos con más detalle el punto 2.1.2, propagar hacia atrás de los errores y actualizar los pesos

Procedimiento

1) Para cada unidad i en la capa de salida hacer
   $\Delta_i \leftarrow g_l'(in_i)(y_i - a_i)$
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
         $w_{ji} \leftarrow w_{ji} + \eta a_j \Delta_i$

Un ejemplo de Retropropación

- 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 in_i y a_i en cada unidad $i = 1, \ldots, 7$
  - Suponemos g_2 y g_3 las respectivas funciones de activación en la capa oculta y de salida, respectivamente

Traza de la Retropropagación del error

Capa	Unidad	Cálculos que se realizan
Salida	7	$\Delta_7 = g_3'(in_7)(y_7 - a_7)$
		$w_{0,7} \longleftarrow w_{0,7} + \eta a_0 \Delta_7$
	6	$\Delta_6 = g_3'(in_6)(y_6 - a_6)$
		$w_{0,6} \longleftarrow w_{0,6} + \eta a_0 \Delta_6$
Oculta	5	$\Delta_5 = g_2'(in_5)[w_{5,6}\Delta_6 + w_{5,7}\Delta_7]$
		$w_{0,5} \longleftarrow w_{0,5} + \eta a_0 \Delta_5$
		$w_{5,6} \longleftarrow w_{5,6} + \eta a_5 \Delta_6$
		$w_{5,7} \longleftarrow w_{5,7} + \eta a_5 \Delta_7$
	4	$\Delta_4 = g_2'(in_4)[w_{4,6}\Delta_6 + w_{4,7}\Delta_7]$
		$w_{0,4} \longleftarrow w_{0,4} + \eta a_0 \Delta_4$
		$w_{4,6} \longleftarrow w_{4,6} + \eta a_4 \Delta_6$
		$w_{4,7} \longleftarrow w_{4,7} + \eta a_4 \Delta_7$
Entrada	3	$w_{3,4} \longleftarrow w_{3,4} + \eta a_3 \Delta_4$
		$w_{3,5} \longleftarrow w_{3,5} + \eta a_3 \Delta_5$
	2	$w_{2,4} \longleftarrow w_{2,4} + \eta a_2 \Delta_4$
		$w_{2,5} \longleftarrow w_{2,5} + \eta a_2 \Delta_5$
	1	$w_{1,4} \longleftarrow w_{1,4} + \eta a_1 \Delta_4$
		$w_{1,5} \longleftarrow w_{1,5} + \eta a_1 \Delta_5$

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 muy 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 recorre *varias veces* el conjunto de entrenamiento
  - Y cada recorrido completo del conjunto de entrenamiento (epoch), considera los ejemplos en un orden aleatorio
  - Podría incluso parar y recomenzar posteriormente el entrenamiento a partir de pesos ya entrenados, con nuevos ejemplos
- Se pueden usar diferentes criterios para elegir el número de epochs adecuado.
  - Por ejemplo, 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 validación independiente para comprobar la evolución del error

Aprendiendo la 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
- Lo más usual es hacer búsqueda experimental de la mejor estructura, medida sobre un conjunto de prueba independiente
    - Más unidades y capas aumenta la expresividad, pero también el riesgo de sobreajuste
    - En teoría, una sóla capa oculta es suficiente; en la práctica, cuantas más capas, menos unidades se necesitan
    - Idea: ir aumentando progresivamente el número de capas, hasta detectar sobreajuste

Bibliografía

- Russell, S. y Norvig, P. *Artificial Intelligence (A modern approach)* (Second edition) (Prentice Hall, 2003) (*o su versión en español*)
  - Sec. 18.7: "Artificial Neural Networks"
- Alpaydin, E. *Introduction to Machine Learning* (third edition) (The MIT Press, 2014)
  - Cap. 11: "Multilayer Perceptrons"
- Géron, A. *Hands-on Machine Learning* (second edition edition) (O'Reilly, 2019)
  - Cap. 10: "Introduction to Artificial Neural Networks with Keras"