FACULTAD DE MATEMÁTICAS

Trabajo fin de Grado

Grado en Matemáticas

Desarrollo de una librería Haskell sobre redes neuronales

Realizado por
Antonio Ramírez de Arellano Marrero

Dirigido por
Francisco Jesús Martín Mateos

Departamento
Ciencias de la Computación
e Inteligencia Artificial

Sevilla, Junio de 2020

Abstract

Artificial Neural Networks are a computational model whose objective is to simulate the learning system of the human brain, this is achieved by designing and optimizing a network through pre-established examples. In this Final Degree Project we present an introductory theoretical vision on this, in addition to an implementation of a Haskell library, based on the Neurolab Python library, easy to understand for any beginning user in this field. We also add some practical examples as an user's manual for it. We conclude this work by proposing future improvements for this type of library and commenting on some that already exist.

Índice general

Índice general **III**

Índice de figuras **V**

Índice de código **VII**

1 Redes neuronales artificiales **1**
1.1 Inspiración biológica y Motivación de las redes neuronales 1
1.2 Perceptrones 2
1.2.1 El Poder Representativo de los perceptrones 3
1.2.2 La regla de entrenamiento del "Perceptrón" 4
1.2.3 Descenso del Gradiente y la Regla Delta 5
1.2.4 Visualización del espacio de hipótesis 6
1.2.5 Derivación de la regla del descenso del gradiente 7
1.2.6 Aproximación Estocástica del Descenso del Gradiente 9
1.2.7 Observaciones 10
1.3 Redes multicapa y el algoritmo de retropropagación 10
1.3.1 Unidad Diferenciable Acotada 11
1.3.2 El algoritmo de retropropagación 12
1.3.3 Derivación del algoritmo de Retropropagación 15
1.4 Observaciones en el Algoritmo de Retropropagación 18
1.4.1 Convergencia y Mínimo local 18
1.4.2 Poder de representación de las redes neuronales 19
1.4.3 Búsqueda en el espacio de hipótesis y sesgo inductivo 19
1.4.4 Representaciones de las Capas Ocultas 20
1.4.5 Criterios de Parada 21

2 Diseño de la aplicación **23**
2.1 Librerías 23
2.2 Nuevo tipo de Dato 23
2.3 Funciones Constructoras 24
2.3.1 Iniciación por Pesos Fijos 24
2.3.2 Iniciación por Pesos Aleatorios 25
2.4 Funciones de Activación 26
2.5 Funciones de Error 28
2.6 Métodos de Entrenamiento 30

2.6.1 Entrenamiento del Perceptrón Simple 31
2.6.2 Entrenamiento de Redes Nueronales Multicapa 32
2.7 Criterios de parada 33

3 Manual de uso **35**
3.1 Perceptrón Simple 35
3.2 Perceptrón Multicapa 37
3.2.1 Función XOR 37
3.2.2 Función Continua 39

4 Conclusiones **43**
4.1 Ampliaciones futuras 43

Bibliografía **45**

Índice de figuras

1.1 Comparación de una neurona biológica con una red neuronal 2
1.2 Ejemplos visuales 4
1.3 Error de diferentes hipótesis. Para una unidad lineal de dos pesos. 7
1.4 La unidad sigmoide 11
1.5 Función $\sigma(x)$ 11
1.6 Representación de puntos de dato con entrada $x \in \{0{,}5, 1, 3\}$. Gracias al sesgo inductivo que tenemos, el algoritmo tenderá a etiquetar los valores intermedios como positivos. 20
1.7 Representación de una capa oculta que ya ha aprendido. Hemos entrenado esta red de 8 x 3 x 8 para aprender la función identidad, hemos usado los 8 ejemplos de entrenamiento. Después de 5000 entrenamientos, las unidades ocultas han codificado las ocho unidades de entrada con la codificación de la derecha. Ejemplo sacado de Tom Mitchell [13] 20

3.1 Gráfica de la función booleana AND 35
3.2 Perceptrón Unicapa para la función AND 36
3.3 Gráfica de la función XOR 38
3.4 Red Neuronal Multicapa para la función XOR 38
3.5 Representación Gráfica de f 39
3.6 Red Neuronal Multicapa para la función f 40

Índice de código

2.1 Librerías Importadas 23
2.2 Tipo de dato para las funciones de activación 24
2.3 Tipo de dato para las FNN 24
2.4 Tipo de dato para las funciones Error 24
2.5 Función constructora del Perceptrón simple con pesos nulos 24
2.6 Función constructora de la red neurnal multicapa con un valor fijo v 25
2.7 Funciones constructoras con pesos nulos 25
2.8 Matriz con entradas aleatorias 25
2.9 Función constructora del Perceptrón simple con entradas aleatorias 25
2.10 Función auxiliar para la lista de matrices peso 25
2.11 Función constructora de la red neuronal multicapa con entradas aleatorias . 26
2.12 Función de activación y su derivada 26
2.13 Función de Activación Lineal 26
2.14 Función de Activación Lineal Saturada 26
2.15 Función de Activación sigmoide 27
2.16 Función de Activación Tangente Hiperbólica 27
2.17 Función de Activación ReLU 27
2.18 Función de Activación leaky ReLU 28
2.19 Función de Activación Umbral 28
2.20 Función de error basada en entropía con su derivada 28
2.21 Función de error absoluto medio con su derivada 29
2.22 Función de error cuadrático medio con su derivada 29
2.23 Función de suma de errores absolutos con su derivada 29
2.24 Función de suma de errores cuadráticos con su derivada 30
2.25 Función salida de una red 30
2.26 Función Auxiliar para los valores de las unidades 30
2.27 Entrenamiento del Perceptrón simple con regla de aprendizaje Delta 31
2.28 Entrenamiento del Perceptrón simple con la regla de aprendizaje Batch . . 31
2.29 Función eliminatoria de los pesos w_0 32
2.30 Función producto componente a componente 32
2.31 Función matrices de error por capa 32
2.32 Entrenamiento de tipo gradiente Delta multicapa 33
2.33 Entrenamiento de tipo gradiente Batch multicapa 33
2.34 Entrenamiento unicapa iterando n veces 33
2.35 Entrenamiento iterando n veces 33
2.36 Error de la red sobre los ejemplos de entrenamiento 34
2.37 Entrenamiento unicapa con error acotado 34

2.38 Entrenamiento multicapa con error acotado 34
3.1 Función auxiliar peso 35
3.2 Ejemplos de entrenamiento para la función AND 36
3.3 Perceptrón simple con pesos iniciales nulos 36
3.4 Rutinas de entrenamiento con método de parada Epsilon 36
3.5 Matrices peso de ambas rutinas 37
3.6 Valores de Salida de ambas redes 37
3.7 Entrenamiento *Epsilon* para AND 37
3.8 Valores de Salida con el método de parada Epochs 37
3.9 Red y ejemplos de entrenamiento para la función XOR 38
3.10 Rutina de entrenamiento para la función XOR con el criterio de parada *Epsilon* 39
3.11 Salida de la red para la función XOR con el criterio de parada *Epsilon* . . . 39
3.12 Red para la función XOR con el criterio de parada *Epochs* 39
3.13 Salida de la red para la función XOR con el criterio de parada *Epochs* . . . 39
3.14 Ejemplos para la función f 40
3.15 Red neuronal para la función f 40
3.16 Entrenamiento Batach para la función f bajo 500 iteraciones 40
3.17 Error de entrenamiento 41
3.18 Entrenamiento Delta para la función f bajo 500 iteraciones 41
3.19 Error de entrenamiento 41
3.20 Entrenamiento Batch con el método de parada *Epsilon* 41
3.21 Error de entrenamiento 41

CAPÍTULO 1

Redes neuronales artificiales

En este capítulo se presenta el concepto de red neuronal artificial, inspirado en las redes neuronales biológicas. Estas redes se ajustan a los datos proporcionados mediante algoritmos de aprendizaje que modifican ciertos parámetros de la red con el objetivo de disminuir el error cometido en su uso. Describimos el algoritmo de aprendizaje de descenso por el gradiente y se comentan algunas mejoras del mismo.

1.1– Inspiración biológica y Motivación de las redes neuronales

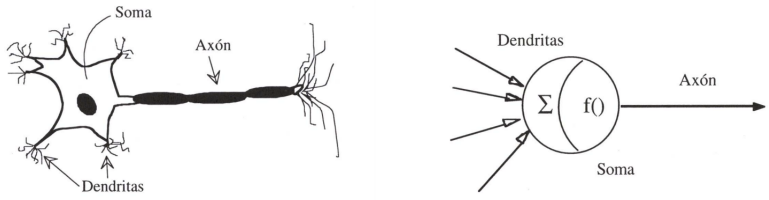
Una neurona de nuestro cerebro recibe señales electromagnéticas, provenientes del exterior (10 %), o de otras neuronas (90 %), a través de la sinapsis de las dendritas. Si la acumulación de estímulos supera cierto umbral, la neurona se *dispara*, es decir; emite a través del axón una señal, que será recibida por otras neuronas a través de las conexiones sinápticas de otras dendritas. Las conexiones sinápticas son dinámicas, con el desarrollo y aprendizaje algunas conexiones se potencian o se debilitan.[10]

En el cuerpo humano poseemos unas 10^{11} neuronas con 10^4 conexiones cada una. Las neuronas más rápidas tardan aprox. 10^{-3} segundos en intercambiar información, estas son mucho mas lentas que los ordenadores (10^{-10} seg.), sin embargo; los humanos podemos llegar a hacer tareas mucho más rápidas que los ordenadores. Por ejemplo, tardamos en reconocer visualmente a nuestra madre unos 10^{-1} seg., es decir, en sólo unos cientos de pasos.

¿A qué se debe que nuestro cerebro sea tan eficaz en algunas de las tareas? Las claves son: el proceso de paralelismo masivo y que, a diferencia de los ordenadores, los cuales devuelven valores de salida numéricos, las neuronas biológicas pueden devolver valores de salida más complejos.

Las redes neuronales artificiales fueron descritas por primera vez en [11]. Se trata de un modelo computacional inspirado en la estructura neurológica del cerebro para la resolución de problemas, tanto reales como discretos, basados en la aproximación de unas funciones objetivo.

Una Neuronal Artificial recibe como entrada un vector de datos numéricos a partir de los cuales se obtiene una combinación lineal con unos pesos asociados a cada conexión. Sobre el valor de esta combinación lineal se evalúa una función de activación que genera



(a) Neurona biológica (b) Neurona Artificial

Figura 1.1: Comparación de una neurona biológica con una red neuronal

la salida de la Neurona Artificial. Una Red Neuronal Artificial es el resultado de conectar varias Neuronas Artificiales de forma que la salida de alguna de ellas sea entrada de otras.

Los pesos de las conexiones en una Red Neuronal Artificial se ajustan en un proceso de aprendizaje para el que se utiliza un conjunto de entrenamiento formado por ejemplos de entradas junto con las salidas esperadas. En este contexto se denomina función objetivo a la función que a partir de las entradas de la red genera una salida tal y como se indica en los ejemplos de entrenamiento.

Problemas apropiados para el aprendizaje de redes neuronales. La aplicación de las redes neuronales artificiales es adecuada en situaciones donde los ejemplos de entrenamiento se corresponden con sonidos o datos de sensores que provienen de aparatos tales como cámaras y micrófonos. También es aplicable a problemas donde se suelen usar representaciones simbólicas como un árbol de decisión por ejemplo.

El algoritmo de retropropagación es la técnica de aprendizaje de redes neuronales más usada. Se recomienda usarlo en problemas con las siguientes características:

- Queremos averiguar la relevancia de algunos atributos para el valor de salida del problema. La función objetivo está definida de manera que podemos describirla por un vector de características predefinidas. Estos atributos de entrada pueden estar altamente relacionados o independientes uno del otro.
- La función objetivo de salida puede ser de valores discretos, reales o un vector de varios valores reales y discretos.
- Los ejemplos de entrenamiento pueden contener errores.
- Se permiten tiempos de entrenamiento largos.
- Se requiere que la función objetivo aprendida se pueda evaluar rápidamente.
- La capacidad de comprensión humana sobre el funcionamiento de la función objetivo es irrelevante.

1.2– Perceptrones

Existen distintos tipos de redes neuronales artificiales (en inglés, Artificial Neural Network (ANN)), una de la más básicas está basada en un unidad denominada perceptrón. Un perceptrón utiliza un vector de entradas reales, calcula una combinación lineal de esas entradas, y entonces como salida devuelve un 1 si el resultado es mejor que un cierto

umbral (predefinido de antemano) y 0 en caso contrario. Dicho de otra forma, dada la entrada $x_1, ..., x_n$, la salida $o(x_1, ..., x_n)$ calculada por el perceptrón es

$$o(x_1, ..., x_n) = \begin{cases} 1 & \text{si } w_0 + w_1 x_1 + ... + w_n x_n > 0 \\ 0 & \text{en caso contrario} \end{cases}$$

donde cada w_i es una constante real (llamada peso), que determina la relevancia de la entrada x_i con respecto a la salida del perceptrón. Es importante observar que nuestro umbral viene dado por $-w_0$ el cual debe ser sobrepasado por $w_1 x_1 + ... + w_n x_n$ para devolver 1.

Para simplificar notación, imaginamos una constante de entrada adicional $x_0 = 1$, el cual nos permite escribir la condición de activación como $\sum_{i=0}^{n} w_i x_i > 0$, o en forma vectorial como $\vec{w} \cdot \vec{x} > 0$. Para abreviar, escribiremos algunas veces la función de salida del perceptrón como:

$$o(\vec{x}) = sg(\vec{w} \cdot \vec{x})$$

donde

$$sg(y) = \begin{cases} 1 & \text{si } y > 0 \\ 0 & \text{en caso contrario} \end{cases}$$

Para enseñar a un perceptrón deberemos ir elegiendo los valores de los pesos $w_0, ..., w_n$. Por lo tanto, podemos hablar del espacio H de hipótesis candidatas, que es el conjunto de todos los posibles vectores peso con valores reales.

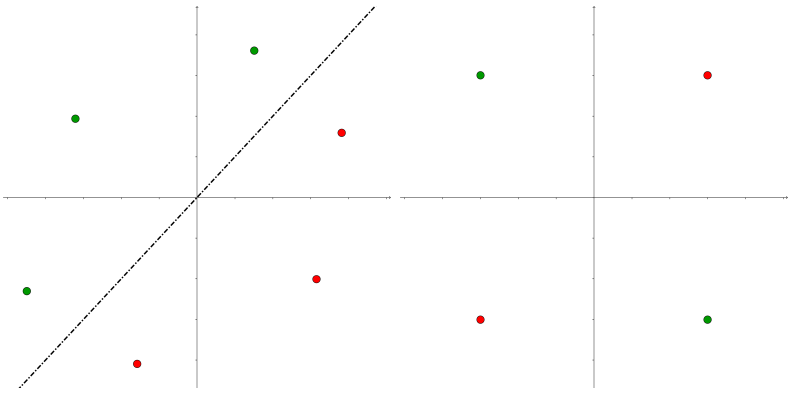
$$H = \{\vec{w} | \vec{w} \in \mathbb{R}^{n+1}\}$$

1.2.1. El Poder Representativo de los perceptrones

Podemos representar el perceptrón como un hiperplano de decisión en el espacio n-dimensional. El perceptrón devuelve 1 si está a un lado del hiperplano y 0 si está al otro lado. La ecuación de este hiperplano de decisión es $\vec{w} \cdot \vec{x} = 0$. Por supuesto, algunos conjuntos de ejemplos negativos y positivos no podemos separarlos por ningún hiperplano. A los que pueden ser separados los llamamos conjuntos de ejemplos linealmente separables.

Un sólo perceptrón puede usarse para representar muchas funciones booleanas. Por ejemplo, suponiendo que los valores 1 y 0 representan Verdadero y Falso respectivamente, una manera de usar un percetrón de dos entradas para implementar la función AND (conjunción) es dando los pesos $w_0 = -8$ y $w_1 = w_2 = 5$. Este mismo perceptrón también puede representar la función OR (disyunción) cambiando w_0 por -3. De hecho, AND y OR pueden verse como casos especiales de funciones m-n, es decir; funciones en las que al menos m de las n entradas del perceptrón deben ser ciertas. La función OR corresponde a $m = 1$ y la AND a $m = n$. Cualquier función m-n se puede representar mediante un perceptrón poniendo los pesos con el mismo valor y ajustando el w_0 adecuadamente.

Los perceptrones pueden representar las funciones booleanas AND, OR, NAND ($\neg$ AND) y NOR ($\neg$ OR). Pero algunas funciones booleanas no pueden representarse por un sólo perceptrón, como la función XOR la cual vale 1 si y sólo si $x_1 \neq x_2$ (es un claro ejemplo de conjunto no linealmente separable).



(a) Linealmente separable (b) No linealmente separable

Figura 1.2: Ejemplos visuales

1.2.2. La regla de entrenamiento del "Perceptrón"

A pesar de que estamos interesados en las redes de aprendizaje de varias unidades conectadas, vamos a empezar por comprender como se aprende a actualizar los pesos para un solo perceptrón. Aquí el problema se basa en determinar un vector peso que haga que el perceptrón devuelva el $\{1, 0\}$ correctamente para cada uno de los ejemplos de entrenamiento que le demos.

Hoy en día se conocen varios algoritmos distintos para solucionar este problema, aquí consideraremos dos: la regla perceptrón y la regla delta. Estos dos algoritmos nos garantizan convergencia bajo hipótesis y condiciones bastante aceptables. Ambos son muy importantes en las ANNs puesto que son la base de los algoritmos de aprendizaje de varias unidades.

Una manera aceptable de iniciar el método es con valores de peso aleatorios, a continuación usamos el perceptrón para cada ejemplo de entrenamiento, modificando sus pesos cuando clasifique mal el ejemplo. Este proceso lo repetimos, iterando tantas veces como necesitemos para que clasifique bien todos los ejemplos de entrenamiento. Los pesos los modificaremos en cada iteración bajo la regla de entrenamiento "perceptrón", que revisa el peso w_i asociado a la entrada x_i de acuerdo con la siguiente regla

$$w_i \leftarrow w_i + \Delta w_i$$

donde

$$\Delta w_i = \eta(t - o)x_i$$

- t : El verdadero valor de salida del ejemplo de entrenamiento.
- o : El valor de salida generado por el perceptrón.
- η : Constante positiva llamada "tasa de aprendizaje", que sirve para moderar el grado de cambio de los pesos en cada paso.

Para entender la convergencia del proceso, pensemos en casos específicos:

- En el primer caso supongamos que el perceptrón ha clasificado correctamente el ejemplo de entrenamiento, entonces $(t - o) = 0$; luego $\Delta w_i = 0$, así que el peso no cambia.

- Supongamos ahora que el perceptrón devuelve un 0 cuando el verdadero valor es 1. Para hacer que el perceptrón devuelva un 1 en vez de 0, los pesos deben alterarse de manera que aumente el valor de $\vec{w} \cdot \vec{x}$. Por ejemplo, si $x_i > 0$, entonces aumentando el w_i va a acercar al perceptrón al valor que queremos en este ejemplo. Observemos que en esta regla de entrenamiento va a aumentar w_i en este caso, porque $(t - o)$, η y x_i son todos positivos.

- Para el caso en que $t = 0$ y $o = 1$ los pesos que $x_i > 0$ van a decrecer en vez de crecer.

Además podemos hacer que converja en un número finito de pasos, bajo la hipótesis de que todos los ejemplos de entrenamiento son linealmente separables y usando un η suficientemente pequeño [12]. Si los datos no son linealmente separables, la convergencia no está asegurada.

1.2.3. Descenso del Gradiente y la Regla Delta

A pesar de que la regla perceptrón encuentra, de manera efectiva, el vector peso adecuado cuando los ejemplos de entrenamiento son linealmente separables, puede fallar la convergencia en los ejemplos que no lo sean. Para sobrepasar esta dificultad usaremos una segunda regla de entrenamiento, llamada la Regla Delta. Si el ejemplo de entrenamiento no es linealmente separable, la regla delta converge hacia la mejor aproximación que encaje en el valor objetivo.

La idea clave que hay detrás de la regla delta es usar el descenso por el gradiente de la función de error, para buscar en el espacio de hipótesis de posibles vectores peso los que mejor encajen con los ejemplos de entrenamiento. Esta regla es importante puesto que el descenso por el gradiente nos proporciona la base del algoritmo de retropropagación, el cual puede aprender los pesos de redes con varias unidades interconectadas. También es importante porque el descenso por el gradiente puede servir de base para algoritmos de aprendizaje que deben buscar en espacios de hipótesis que contienen varios tipos distintos de hipótesis con parametrización continua.

La regla de entrenamiento delta se entiende mejor considerando la tarea de entrenamiento de una unidad lineal definida de la siguiente manera:

$$o(\vec{x}) = \vec{w} \cdot \vec{x} \tag{1.1}$$

Ahora empecemos por definir la medida del "error de entrenamiento"de una hipótesis, relativa a un ejemplo de entrenamiento. A pesar de que hay muchas maneras de definir este error, una de las medidas más comunes que será especialmente conveniente es

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \tag{1.2}$$

donde D es el conjunto de los ejemplos de entrenamiento, t_d es el valor objetivo para el ejemplo de entrenamiento d, y o_d es el valor de la unidad lineal para el ejemplo d. Bajo esta definición, $E(\vec{w})$ es simplemente la mitad del cuadrado de la diferencia entre el valor

objetivo t_d y el valor de salida de la unidad lineal o_d, sumando esto sobre todos los ejemplos de entrenamiento. Básicamente estamos hablando de la Suma de Errores Cuadráticos.

Aquí caracterizamos E como una función de $\vec{w}$ porque la unidad lineal que devuelve o depende exclusivamente del vector peso. Obviamente E también depende en particular del conjunto de ejemplos de entrenamiento, pero asumimos que estos ya están fijados durante el entrenamiento.

Para estudiar la convergencia, seguiremos usando la suma de errores cuadráticos, pero actualmente se usan otras muchas medidas de error, algunas de estas medidas son los siguientes:

- Error basado en entropía:

$$-\sum_{d\in D} t_d \log(o_d) + (1-t_d)\log(1-o_d)$$

- Error absoluto medio:

$$\frac{1}{|D|}\sum_{d\in D} |t_d - o_d|$$

- Suma de errores absolutos:

$$\sum_{d\in D} |t_d - o_d|$$

- Error cuadrático medio:

$$\frac{1}{|D|}\sum_{d\in D} (t_d - o_d)^2$$

1.2.4. Visualización del espacio de hipótesis

Para comprender el algoritmo de descenso por el gradiente, puede ser de gran ayuda visualizar todo el espacio de hipótesis de los posibles vectores peso y su relación con los valores de E, como podemos observar en la figura 1.3, la función error debe ser siempre un paraboloide con un mínimo global (gracias a nuestra elección de la función error). Cada paraboloide va a depender de los ejemplos de entrenamiento.

Fijándonos más detenidamente en la figura 1.3 el espacio de hipótesis H es el plano XY asociado a w_0 y w_1 respectivamente, el eje vertical representa el error del vector peso correspondiente. La flecha indica la dirección opuesta a la del gradiente, que señala el mayor descenso de la función de error. Luego una manera de disminuir el error en cada paso puede ser guiándonos por el vector opuesto al gradiente de la función error, la longitud de paso en cada iteración la podremos ajustar para asegurar que no nos "pasamos" del mínimo.

El descenso del gradiente determina el vector peso que minimiza E empezando por un vector peso arbitrario y modificándolo en pequeños pasos. En cada paso, el vector peso lo alteramos en la dirección que produzca el mayor descenso sobre la superficie de error. Este proceso es reiterado hasta que alcancemos el mínimo global.

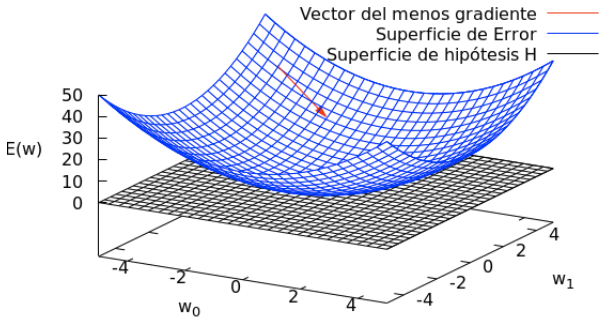


Figura 1.3: Error de diferentes hipótesis. Para una unidad lineal de dos pesos.

1.2.5. Derivación de la regla del descenso del gradiente

Ahora nos preguntamos: ¿Cómo podemos calcular la dirección que dé el salto con mayor descenso posible sobre la superficie de error? Esta dirección la podemos encontrar derivando la función E respecto a cada componente del vector $\vec{w}$. Este vector derivada se denomina Gradiente de E respecto de $\vec{w}$, denotado $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, ..., \frac{\partial E}{\partial w_n}\right] \tag{1.3}$$

Observemos que $\nabla E(\vec{w})$ es un vector en sí, cuyas componentes son las derivadas parciales de E respecto cada w_i. Interpretándolo como vector del espacio de los pesos, el gradiente nos da la dirección que produce el mayor aumento en E. Por lo tanto $-\nabla E$ nos da la dirección del mayor decrecimiento.

Puesto que el gradiente produce esta información, definimos la regla de entrenamiento para el descenso por el gradiente como

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

donde

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \tag{1.4}$$

Aquí η es una constante positiva que volveremos a llamar "tasa de aprendizaje", que determina el tamaño del paso en el descenso del gradiente. Obviamente el signo negativo está para que nos movamos hacia el mayor descenso. Está regla también la podemos expresar componente a componente

$$w_i \leftarrow w_i + \Delta w_i$$

donde

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} \tag{1.5}$$

Para construir un algoritmo práctico para ir actualizando iterativamente los pesos de acuerdo a (1.5), necesitamos una manera eficiente de calcular el gradiente en cada paso, afortunadamente, no es complicado conseguirlo. Derivando E componente a componente según su definición tenemos que

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}(\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2) \\
&= \frac{1}{2}\sum_{d\in D}\frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_{d\in D}2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_{d\in D}(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_{d\in D}(t_d - o_d)(-x_{id}) \qquad (1.6)
\end{aligned}$$

donde x_{id} denota la entrada de la componente x_i para el ejemplo de entrenamiento d. Acabamos de conseguir una ecuación que nos da $\frac{\partial E}{\partial w_i}$ en términos de x_{id}, o_d, y el valor objetivo t_d asociado con los ejemplos de entrenamiento. Sustituyendo la ecuación 1.6 en la ecuación (1.5) nos queda

$$\Delta w_i = \eta \sum_{d\in D}(t_d - o_d)x_{id} \qquad (1.7)$$

En resumidas cuentas, el algoritmo nos queda de la siguiente forma:

DESCENSO-GRADIENTE($ejemplo\_entrenamiento, \eta$)

Para cada ejemplo de entrenamiento de la forma $\langle\vec{x}, t\rangle$, donde $\vec{x}$ es el vector de los valores de entrada, t es el valor objetivo de salida y η es la tasa de aprendizaje.

- Asignamos a cada w_i un valor inicial aleatorio
- Hasta que se llegue a la condición terminal, hacer
  - Dar a cada Δw_i el valor 0.
  - Para cada $\langle\vec{x}, t\rangle$ en *ejemplo_entrenamiento*, hacer
    - Dar el valor $\vec{x}$ a la unidad y calcular el valor de salida o
    - Para cada peso w_i, hacer

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \qquad (1.8)$$

  - Para cada peso w_i, hacer

$$w_i \leftarrow w_i + \Delta w_i \qquad (1.9)$$

Cabe destacar que el algoritmo converge porque la superficie de error contiene un sólo mínimo global, además éste convergerá al mínimo error independientemente de que los ejemplos de entrenamiento sean o no separables, siempre que el η sea suficientemente pequeño, puesto que el descenso del gradiente cuenta con el riesgo de que el salto sea demasiado grande y nos saltemos el mínimo en la superficie de error en vez de alcanzarlo. Por ésta razón, una modificación que suele hacerse en el algoritmo es reducir el valor de η conforme vayamos aumentando las iteraciones.

1.2.6. Aproximación Estocástica del Descenso del Gradiente

El método del descenso del gradiente es una estrategia para la búsqueda en grandes, o incluso espacios infinitos de hipótesis. En general se puede aplicar cuando el espacio en el que trabajamos está parametrizado de forma continua y podemos derivar la función error respecto de estos parámetros continuos. Las dificultades prácticas más importantes que nos podemos encontrar al aplicarlo son:

1. La convergencia puede llegar a ser muy lenta, es decir, puede llegar a necesitar varios miles de pasos.

2. Si existen varios mínimos locales en la superficie de error, no hay garantías de que alcancemos el mínimo global.

Una variación común para aliviar estas dificultades es el descenso del gradiente estocástico. Mientras que el método del descenso del gradiente (usando la ecuación (1.7)) realiza la actualización de los pesos después de sumar los incrementos sobre todos los ejemplos de entrenamiento D, la idea detrás del método de descenso por el gradiente estocástico es actualizar los pesos con más frecuencia, ajustándolos individualmente para cada ejemplo. Esta modificación del método de entrenamiento es igual a la ecuación (1.7) exceptuando que ahora cada vez que iteremos en cada ejemplo de entrenamiento actualizamos los pesos de acuerdo a la siguiente regla:

$$\Delta w_i = \eta(t - o)x_i \tag{1.10}$$

donde t, o y los x_i son los valores objetivo, la unidad de salida y la i-ésima entrada para el ejemplo de entrenamiento en cuestión, respectivamente. Para modificar el anterior algoritmo basta reemplazar la ecuación (1.8) por

$$w_i \leftarrow \eta(t - o)x_i$$

Una manera de visualizar el descenso del gradiente estocástico es considerando una función de error distinta $E_d(\vec{w})$ definida para cada uno de los ejemplo de entrenamiento d de la siguiente forma

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d) \tag{1.11}$$

donde t_d y o_d son el valor objetivo y el valor de la unidad de salida para el ejemplo de entrenamiento d, respectivamente. El descenso por el gradiente estocástico itera sobre cada ejemplo de entrenamiento d en D, en cada iteración actualizamos los pesos de acuerdo con el gradiente del $E_d(\vec{w})$ correspondiente. Esta secuencia de iteraciones proporciona una aproximación razonable para descender por el gradiente de nuestra función de error original $E(\vec{w})$. Haciendo η suficientemente pequeño, podemos conseguir que el descenso del gradiente estocástico funcione incluso mejor que el estándar. Las diferencias clave entre el método estándar y el estocástico son:

- En la versión estándar, sumamos el error sobre todos los ejemplos antes de actualizar los pesos, mientras que el estocástico los vamos actualizando en cada ejemplo de entrenamiento.

- Sumar sobre varios ejemplos distintos en la versión estándar requiere de mucha más computación por cada actualización de los pesos. Por el otro lado, como este sí usa el verdadero gradiente, la versión estándar normalmente usa una longitud de paso más grande que el estocástico.

- En casos donde haya varios mínimos locales respecto de $E(\vec{w})$, algunas veces el método estocástico puede evitar caer en estos mínimos locales porque usa varios $\nabla E_d(\vec{w})$ en vez de sólo un $\nabla E(\vec{w})$ para guiarnos en la búsqueda.

En la práctica se usan ambos métodos con bastante frecuencia.

La regla de entrenamiento en la ecuación (1.10) la llamamos regla delta (cuya nomenclatura usaremos en la librería) o también regla LMS (least-mean-square). También aclarar que a la regla estándar la llamaremos regla Batch en la implementación. Observemos que la regla delta en la ecuación (1.10) es similar a la regla perceptrón, pero aunque ambas expresiones aparentan ser idénticas, estas son diferentes puesto que en la regla delta o se refiere a la unidad lineal $o(\vec{x}) = \vec{w} \cdot \vec{x}$, mientras que la regla perceptrón se refiere a la función umbral $o(\vec{x}) = sg(\vec{w} \cdot \vec{x})$.

1.2.7. Observaciones

Hemos considerado dos algoritmos similares para el aprendizaje iterativo del perceptrón. La diferencia clave se refleja en sus diferentes propiedades de convergencia. La regla perceptrón converge después de un número finito de pasos hacia una hipótesis que clasifica perfectamente los datos de entrenamiento, siempre y cuando los ejemplos de entrenamientos sean linealmente separables. La regla delta converge sólo asintóticamente hacia el mínimo error, posiblemente esto requiera un tiempo no acotado, pero converge sin importar si los datos de los ejemplos de entrenamiento son linealmente separables o no [17].

Un tercer algoritmo para el aprendizaje del vector de pesos es la programación lineal. La programación lineal es un método genérico y eficiente que se usa para resolver conjuntos definidos por inecuaciones lineales. Esto es aplicable en este contexto si planteamos que cada ejemplo de entrenamiento corresponde a una desigualdad de la forma $\vec{w} \cdot \vec{x} > 0$ o $\vec{w} \cdot \vec{x} \leq 0$, y su solución es el vector deseado. Desafortunadamente este acercamiento sólo nos da una solución si los ejemplos de entrenamiento son linealmente separables, sin embargo; [3] sugiere una formulación mas sutil que funciona en los casos no separables. En cualquier caso, el acercamiento por programación lineal no escala al entrenamiento de redes multicapa, el cual es nuestro objetivo principal. En cambio, el acercamiento por el descenso del gradiente, base de la regla delta; podemos extenderlo a redes multicapa, como vamos a ver en la siguiente sección.

1.3– Redes multicapa y el algoritmo de retropropagación

Como hemos visto en el apartado anterior, los perceptrones unicapa sólo pueden expresar superficies de decisión lineales. En cambio, el tipo de redes multicapa que aprenden por el algoritmo de retropropagación son capaces de expresar una rica variedad de superficies de decisión no lineales.

En esta sección discutiremos como ajustar las redes multicapa usando algoritmos de tipo gradiente de forma similar al planteado en la sección anterior.

1.3.1. Unidad Diferenciable Acotada

¿Qué tipo de unidad deberíamos usar como base para la construcción de redes multicapa? Lo primero que podríamos intentar es elegir las unidades lineales discutidas en la anterior sección, para la cual ya hemos derivado una relga del descenso del gradiente. De cualquier forma, múltiples capas de unidades lineales también producen funciones lineales, y preferimos redes capaces de representar funciones no lineales más complejas. La unidad perceptrón es otra posibilidad, pero su valores de salida son discretos, luego no son diferenciables, y por lo tanto no se portan bien en el descenso por el gradiente. Lo que necesitamos es una unidad cuya salida sea una función no lineal pero que sea también diferenciable respecto a los valores de entrada. Una solución es la unidad sigmoide, una unidad muy parecida al perceptrón, pero basada en una función diferenciable y suavizada.

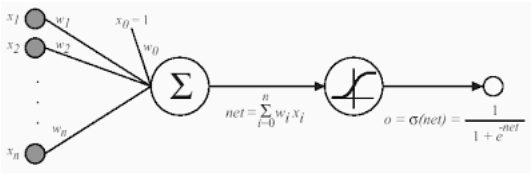


Figura 1.4: La unidad sigmoide

Esta unidad sigmoide es una función continua respecto a sus entradas, su salida o se calcula como

$$o = \sigma(\vec{w} \cdot \vec{x})$$

donde

$$\sigma(y) = \frac{1}{1 + e^{-y}} \tag{1.12}$$

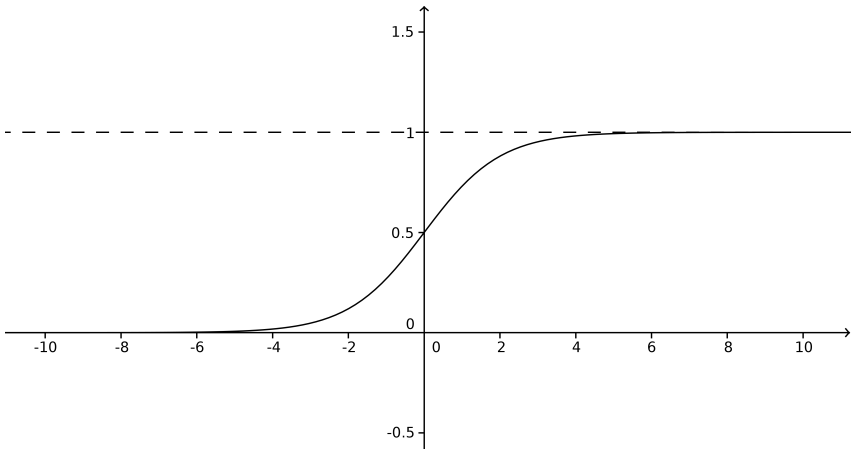


Figura 1.5: Función $\sigma(x)$

A σ la llamaremos función sigmoide o función logística. Observando la gráfica 1.5, vemos que su rango queda comprendido entre 0 y 1, además es monótona creciente respecto a su variable. Como puede partir de grandes dominios y va al $(0, 1)$, la función también se suele

llamar función aplastamiento a la unidad. La función sigmoide tiene la útil propiedad de que podemos expresar fácilmente su derivada respecto a su valor de salida ($\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$). Como podemos observar el método del descenso del gradiente hace uso de esta derivada. Algunas veces se usan otras funciones diferenciables con derivadas faciles de calcular. Por ejemplo, el término e^{-y} lo podemos reemplazar algunas veces por por e^{-ky} donde k es una constante positiva que nos determina la longitud de paso del límite.

En el ámbito teórico y a lo largo de este capítulo seguiremos con esta unidad de activación, pero existen diversas opciones en la práctica, algunas a destacar son:

- Activación Lineal (o identidad):
$$\sigma(y) = y$$
- Activación Lineal Saturada:
$$\sigma(y) = \begin{cases} 0 & \text{si } y < 0 \\ y & \text{en caso contrario} \end{cases}$$
- Tangente Hiperbólica:
$$\sigma(y) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
- Unidad Lineal Rectificada:
$$\sigma(y) = \text{máx}\{0, y\}$$
- Unidad Lineal Rectificada "Leaky":
$$\sigma(y) = \begin{cases} 0{,}01y & \text{si } y < 0 \\ y & \text{en caso contrario} \end{cases}$$

1.3.2. El algoritmo de retropropagación

El algoritmo de retropropagación aprende los pesos de una red multicapa con un conjunto fijo de unidades e interconexiones. Utiliza el método de descenso por el gradiente para intentar minimizar el error entre los valores de la red de salida y los valores objetivo de estas salidas. Aquí presentaremos este algoritmo además de algunas posibles mejoras.

Como estamos considerando ahora redes con múltiples salidas en vez de unidades simples como antes, empezaremos por redefinir la función de error E como la suma de los errores sobre todas las unidades de salida de la red

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in salidas} (t_{kd} - o_{kd})^2 \tag{1.13}$$

donde *salidas* es el conjunto de unidades de salida de la red, y t_{kd} y o_{kd} son los valores objetivos y de salida asociados a la k-ésima unidad de salida en el ejemplo de entrenamiento d.

El problema de aprendizaje al que se enfrenta la retropropagación es la búsqueda de un gran espacio de hipótesis definida por todos los posibles valores peso de todas las unidades de la red. Esta situación la podemos visualizar en términos de la superficie de error de manera similar a la de la imagen 1.3. Reemplazamos el error en ese diagrama por nuestra nueva definición de E, y las otras dimensiones del espacio corresponden ahora a todos los

pesos asociados a todas las unidades de la red. Como en el caso del entrenamiento de una unidad simple, el descenso del gradiente pretende encontrar una hipótesis para minimizar la función de error E.

RETROPROPAGACION($ejemplos\_entrenamietno$,η,n_{in},n_{out},n_{hidden})

Cada ejemplo de entrenamiento es un par de la forma $\langle \vec{x}, \vec{t} \rangle$, donde $\vec{x}$ es el vector de los valores de entrada de la red, y $\vec{t}$ es el vector de salida con los valores objetivo de la red. η es la tasa de aprendizaje, n_{in} es el número de redes de entrada, n_{hidden} el número de unidades en las capas ocultas, y n_{out} es el número de unidades de salida.

Llamaremos a la entrada que vaya de la unidad i a la unidad j como x_{ji} y al peso que vaya de la unidad i a la unidad j lo llamaremos w_{ji}.

- Creamos una red prealimentada con n_{in} entradas, n_{hidden} unidades ocultas, y n_{out} unidades de salida.
- Asignamos a todos los pesos de la red número aleatoriamente pequeños (p.ej. $-0{,}5$ y $0{,}5$).
- Hasta llegar a la condición final, hacer
  - Para cada $\langle \vec{x}, \vec{t} \rangle$ en $ejemplos\_entrenamientos$, hacer
    - Propagar la entrada hacia delante a través de la red:
      - Evaluamos el valor $\vec{x}$ en la red y calculamos la salida o_u de cada unidad u en la red.
    - Propagar el error hacia atrás a través de la red:
      - Para cada unidad de salida de la red k, calcular su error δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \tag{1.14}$$

      - Para cada unidad oculta h, calcular su error δ_h

$$d_h \leftarrow o_h(1 - o_h) \sum_{k \in salidas} w_{kh}\delta_k \tag{1.15}$$

      - Actualizamos cada peso de la red w_{ij}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji} \tag{1.16}$$

        donde

$$\Delta w_{ji} = \eta \delta_j x_{ji} \tag{1.17}$$

Una de las diferencias notables en el caso de las redes multicapa es que la superficie de error puede tener varios mínimos locales, no como en las superficies parabólicas de las redes simples (como podemos observar en la figura 1.3) donde no nos encontramos este problema. Desafortunadamente, esto implica que el descenso del gradiente nos garantiza la convergencia a un mínimo local, y no necesariamente al global. A pesar de este obstáculo, en la práctica la retropropagación proporciona excelentes resultados en diversas aplicaciones en el mundo real.

El algoritmo que acabamos de describir comienza con una construcción de la red con el número deseado de unidades ocultas y de salida, e inicia toda la red con pesos aleatorios

pequeños. Dada esta estructura fija de la red, el bucle principal del algoritmo se repite sobre los ejemplos de entrenamiento. Para cada ejemplo de entrenamiento, se aplica la red al ejemplo, se calcula el error de la salida en ese caso, se calcula el gradiente respecto del error en esa iteración (a veces repetimos esto miles de veces con el mismo ejemplo) y se ajustan los pesos hasta que la red actúe de manera aceptable.

La actualización del peso por el método del gradiente (1.17) es similar a la regla de aprendizaje delta (1.10). Al igual que la regla delta, se actualiza cada peso en proporción a la tasa de aprendizaje η, el valor de entrada x_{ij} al cual aplicamos el peso, y el error en la salida de la unidad. La única diferencia es que el error $(t-o)$ de la regla delta se reemplaza por un error más complejo, δ_j. La forma exacta de δ_j se consigue de la derivación de la regla en la que cambiamos el peso (que veremos más adelante). Para entenderla intuitivamente, primero consideremos como estamos computando δ_k en el algoritmo para cada unidad de salida de la red k (1.14). δ_k simplemente es la forma familiar de la regla delta $(t_k - o_k)$, multiplicado por un factor $o_k(1 - o_k)$, el cual es la forma de la derivada de la función sigmoide. El δ_h para cada unidad oculta h tiene una forma similar (1.15). Sin embargo, como los ejemplos de entrenamiento nos proporcionan valores objetivos t_k solo para las salidas de la red, no tenemos valores objetivos para indicar el error de los valores de las unidades ocultas. En lugar de esto, calculamos el error para las unidades ocultas h sumando los errores δ_k que está influenciados por h, pesando cada δ_k con w_{kh}, el peso que va desde la unidad oculta h y la unidad de salida k. Este peso nos da la información de "cómo y cuánto de importante es h" en el error de la unidad de salida k.

El algoritmo actualiza los pesos de forma incremental, siguiendo la presentación de cada ejemplo de entrenamiento. Esto corresponde a una aproximación estocástica al descenso por el gradiente. Para obtener el verdadero gradiente de E deberíamos sumar los valores $\delta_j x_{ji}$ sobre todos los ejemplos de entrenamiento antes de alterar los pesos.

Podemos llegar a iterar miles de veces el bucle que hacemos para actualizar los pesos en la retropropagación en cualquier aplicación típica del mismo. Una variante para las condiciones terminales es usando un procedimiento de parada, parar en un número fijado de iteraciones (Epochs) o una vez que el error decaiga por debajo de alguna cota, o bien alcance un criterio previamente marcado. La elección de estos criterios es muy importante, puesto que con pocas iteraciones se pueden fallar a la hora de reducir suficientemente el error, y con demasiadas puede "sobreajustar"los datos de entrenamiento. Discutiremos este problema más adelante.

Añadir *momentum*

Al ser la retropropagación un algoritmo muy usado, se han desarrollado muchas variaciones del mismo. Seguramente la más común consiste en variar la actualización de los pesos (1.17) del algoritmo haciendo que la actualización del peso en la iteración n-ésima dependa parcialmente en la actualización de la iteración (n-1)-ésima de la siguiente forma

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1) \tag{1.18}$$

aquí $\Delta w_{ji}(n)$ es la actualización del peso en la iteración n-ésima que ocurre en el bucle principal del algoritmo, y $0 \leq \alpha < 1$ es una contante llamada *momentum*. Observamos que el primer término de la parte derecha de la igualdad es exactamente igual al del algoritmo de retropropagación (1.17). El segundo término de la parte derecha lo llamaremos término de *momentum*. Para ver el efecto de este término, pensemos que la trayectoria del descenso

del gradiente es como una pelota que baja rodando por la superficie de error. El efecto que tiene α es que esta bola tienda a seguir rodando en la misma dirección de una iteración a la siguiente. Esto puede conseguir que la bola siga rodando aunque pase por un mínimo local o en regiones planas (gradiente nulo), es decir, en sitios donde se detendría si usásemos el algoritmo de retropropagación. Además esto también hace que vaya aumentando la longitud de paso en regiones donde el gradiente no varía, y por lo tanto su velocidad de convergencia.

Aprendizaje en redes acíclicas arbitrarias

Si analizamos el algoritmo de retropropagación, observamos que sólo se puede aplicar a redes de dos capas. Sin embargo, este mismo algoritmo se puede generalizar a redes de profundidad arbitraria. Mantenemos la regla de actualización del peso (1.17), y lo único que cambiamos es el procedimiento al calcular los valores δ. En general, calculamos el valor δ_r para una unidad r en la capa m desde los valores de δ de la siguiente capa $m+1$ de acuerdo a la siguiente regla:

$$\delta_r = o_r(1 - o_r) \sum_{s \in \text{capa m+1}} w_{sr}\delta_s \tag{1.19}$$

Hay que destacar que esto es idéntico al paso 3 del algoritmo original. Realmente lo que estamos diciendo aquí es que este paso lo vamos a repetir para cada capa oculta de la red.

De la misma forma podemos generalizarlo a cualquier grafo acíclico dirigido, sin importar si las unidades de la red están uniformemente organizadas en capas, como hemos supuesto hasta ahora. En el caso en el que no lo estén, la regla para calcular δ para cualquier unidad interna (es decir, cualquier unidad que no sea de salida) es

$$\delta_r = o_r(1 - o_r) \sum_{s \in Siguientes(r)} w_{sr}\delta_s \tag{1.20}$$

donde $Siguientes(r)$ es el conjunto de unidades a las que se va inmediatamente desde la unidad r en la red, esto es; todas las unidades cuyas entradas incluyen la salida de la unidad r. Esta es la forma general de la regla de actualización del peso que vamos a derivar en la siguiente sección.

1.3.3. Derivación del algoritmo de Retropropagación

En esta sección vamos a presentar la derivación de la regla en la que actualizamos el peso en el algoritmo de retropropagación. En concreto, la regla del descenso por el gradiente estocástico del algoritmo de retropropagación (sección 1.3.2). Si volvemos a observar la ecuación (1.11) estamos iterando sobre los ejemplos de entrenamiento de uno en uno, para cada ejemplo de entrenamiento d calculamos el gradiente del error E_d respecto de ese ejemplo. En otras palabras, para cada ejemplo de entrenamiento d actualizamos cada peso w_{ji} añadiéndole Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \tag{1.21}$$

donde E_d es el error cometido por el ejemplo de entrenamiento d, sumado sobre todas las unidades de salida en la red

$$E_d(\vec{w}) = \frac{1}{2} \sum_{k \in salidas} (t_k - o_k)^2 \tag{1.22}$$

Aquí *salidas* es el conjunto de unidades de salida en la red, t_k es el valor objetivo de la unidad k para el ejemplo de entrenamiento d, y o_k es la unidad de salida k dado el ejemplo de entrenamiento d.

La derivación que vamos a hacer es conceptualmente sencilla, pero requiere de muchos subíndices y notación necesaria, para ello vamos a definir cómo escribiremos las cosas de ahora en adelante:

- x_{ji} = La i-ésima entrada de la unidad j.
- w_{ji} = El peso asociado a la i-ésima entrada de la unidad j.
- $red_j = \sum_i w_{ji} x_{ji}$ (la suma de los pesos de entrada para la unidad j).
- o_j = La salida obtenida en la unidad j.
- t_j = El valor objetivo de salida para la unidad j.
- σ = La función sigmoide.
- $salidas$ = El conjunto de unidades en la capa final de la red.
- $Siguientes(j)$ = El conjunto de unidades cuya entrada inmediata incluye la salida de la unidad j.

Para empezar a derivar, primero nos damos cuenta de que un peso w_{ji} puede influenciar al resto de la red solamente sobre red_j. Por lo tanto, aplicando la regla de la cadena podemos escribir

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial red_j} \frac{\partial red_j}{\partial w_{ji}} = \frac{\partial E_d}{\partial red_j} x_{ji} \tag{1.23}$$

Dada la ecuación (1.23), nuestro objetivo ahora se convierte en calcular $\frac{\partial E_d}{\partial red_j}$. Para esto consideraremos dos casos: el caso donde la unidad j es de salida de la red, y el caso en que j es una unidad interna.

Caso 1: Regla de entrenamiento para pesos de unidades de salida. Consideremos una unidad de salida de la red j, como w_{ji} puede influenciar sólo en el valor de de red_j y ésta a su vez sólo actúa sobre o_j; aplicando la regla de la cadena llegamos a que

$$\frac{\partial E_d}{\partial red_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial red_j} \tag{1.24}$$

Para empezar, consideremos únicamente el primer término de la ecuación (1.24)

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} \sum_{k \in salidas} (t_k - o_k)^2$$

Las derivadas $\frac{\partial}{\partial o_j}(t_k - o_k)^2$ van a ser cero para todas las unidades de salida k excepto cuando $k = j$. Por lo tanto sólo nos queda el sumando $k = j$.

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_j)^2 = \frac{1}{2}2(t_j - o_j)\frac{\partial(t_j - o_j)}{\partial o_j} = -(t_j - o_j) \tag{1.25}$$

Ahora consideremos el segundo término de la ecuación (1.24). Como $o_j = \sigma(red_j)$, su derivada va a ser la derivada de la función sigmoide, la cual ya hemos calculado previamente $\sigma(red_j)(1 - \sigma(red_j))$. Por lo tanto

$$\frac{\partial o_j}{\partial red_j} = \frac{\partial \sigma(red_j)}{\partial red_j} = o_j(1 - o_j) \tag{1.26}$$

Sustituyendo las expresiones obtenidas (1.25) y (1.26) en (1.24), obtenemos

$$\frac{\partial E_d}{\partial red_j} = -(t_j - o_j)o_j(1 - o_j) \tag{1.27}$$

Combinando esto con las ecuaciones (1.21) y (1.23), tenemos la regla del descenso del gradiente estocástico para unidades de salida

$$\Delta w_{ji} = -\eta\frac{\partial E_d}{\partial w_{ji}} = \eta(t_j - o_j)o_j(1 - o_j)x_{ji} \tag{1.28}$$

Nos damos cuenta de que estas ecuaciones coinciden claramente con las ecuaciones del algoritmo (1.14) y (1.17), de hecho; observamos que δ_k en la ecuación (1.14) es igual a $\frac{\partial E_d}{\partial red_k}$.

Caso 2: Regla de entrenamiento para pesos de unidades ocultas. En este caso, la derivación de la regla de entrenamiento para w_{ji} debe tener en cuenta las formas en las cuales w_{ji} influencia indirectamente a las salidas de la red, y por tanto E_d. Por esta razón va a ser útil trabajar con $Siguientes(j)$ puesto que red_j sólo puede influenciar a las salidas de la red (y por lo tanto E_d) solamente sobre las unidades que estén en $Siguientes(j)$. Así que podemos escribir (a partir de ahora llamaremos δ_i a $\frac{\partial E_d}{\partial red_i}$ para una unidad i)

$$\begin{aligned}
\frac{\partial E_d}{\partial red_j} &= \sum_{k \in Siguientes(j)} \frac{\partial E_d}{\partial red_k}\frac{\partial red_k}{\partial red_j} \\
&= \sum_{k \in Siguientes(j)} -\delta_k \frac{\partial red_k}{\partial red_j} \\
&= \sum_{k \in Siguientes(j)} -\delta_k \frac{\partial red_k}{\partial o_j}\frac{\partial o_j}{\partial red_j} \\
&= \sum_{k \in Siguientes(j)} -\delta_k w_{kj}\frac{\partial o_j}{\partial red_j} \\
&= \sum_{k \in Siguientes(j)} -\delta_k w_{kj} o_j(1 - o_j)
\end{aligned}$$

luego nos queda

$$\frac{\partial E_d}{\partial red_j} = \sum_{k \in Siguientes(j)} -\delta_k w_{kj} o_j(1 - o_j) \tag{1.29}$$

Reordenando términos y usando δ_j como $\frac{\partial E_d}{\partial red_j}$, tenemos

$$\delta_j = o_j(1 - o_j) \sum_{k \in Siguientes(j)} \delta_k w_{kj}$$

y

$$\Delta w_{ji} = \eta \ \delta_j \ x_{ji}$$

el cual es precisamente la regla general de la ecuación (1.20) para actualizar los pesos de las unidades internas en grafos acíclicos dirigidos arbitrarios. Observamos que la ecuación (1.15) es un caso especial para esta regla, en la cual $Siguientes(j) = salidas$.

1.4– Observaciones en el Algoritmo de Retropropagación

1.4.1. Convergencia y Mínimo local

Como ya hemos comentado antes, el Algoritmo de Retropropagación lo que hace es implementar una búsqueda de tipo gradiente sobre el espacio de los posibles pesos de la red, reduciendo (iterativamente) el error E que hay entre los valores de salida de la red y los valores objetivos de los ejemplos de entrenamiento. Como la superficie de error puede llegar a tener varios mínimos locales en los cuales el algoritmo puede quedarse atrapado, solo tenemos garantizada la convergencia a un mínimo local pero no necesariamente al mínimo global.

En la práctica, a pesar de la carencia que supone no tener garantizado la convergencia a un mínimo global, este algoritmo nos proporciona un método de aproximaciones de funciones altamente efectivo. Intuitivamente podemos considerar que las redes con más pesos corresponden a superficies de error de más dimensiones (una por peso). Cuando el algoritmo llegue a un mínimo local respecto de uno de estos pesos, no tiene por qué ser un mínimo local de los otros pesos. En resumidas cuentas, cuantos más pesos, más "vías de escape" va a tener el algoritmo para salir del mínimo local de uno de los pesos.

Una segunda perspectiva es considerar la manera en la que evolucionan los pesos de la red respecto del aumento de las iteraciones de entrenamiento. Nos damos cuenta de que si los pesos iniciales son cercanos a cero, entonces los primeros pasos del algoritmo van a representar una función tan suave que será aproximadamente lineal respecto de los valores de entrada. Esto se debe a que la función sigmoide se comporta de forma casi lineal cuando los pesos están cerca del 0. Sólo cuando iteremos muchas veces el algoritmo empezará a comportarse de forma no lineal. Se suele esperar que los mínimos locales se encuentren en esta segunda región (cuando la función es más compleja). Es de esperar que antes de llegar a esta región nos habremos acercado lo suficiente al mínimo global y de esta forma el mínimo local alcanzado es aceptable.

A pesar de lo que acabamos de comentar, todavía no se conoce adecuadamente el comportamiento del algoritmo de descenso por el gradiente sobre superficies de error complejas representadas por ANNs, y todavía no sabemos predecir completamente cuando van a causar dificultades los mínimos locales.

Algunas de las formas más comunes para aliviar este problema son:

- Añadir *momemtum*.
- Usar el Descenso del Gradiente Estocástico.
- Entrenar múltiples redes usando el mismo dato iniciando con distintos valores.

1.4.2. Poder de representación de las redes neuronales

¿Qué conjunto de funciones podemos representar con las redes neuronales del tipo presentado? Está claro que esto va a depender de la anchura y profundidad de las redes. Los tres casos más generales son:

- **Funciones Booleanas**. Podemos representar exactamente cada función booleana con una red de dos capas, a pesar de que el número de unidades ocultas crece exponencialmente respecto al número de entradas de la red (en el peor de los casos).
- **Funciones Continuas**. Podemos aproximar cada función continua acotada por un error arbitrariamente pequeño (bajo norma finita) por una red con dos capas [1, 8]. El número de unidades ocultas requeridas depende de la función a aproximar.
- **Funciones arbitrarias**. Las podemos aproximar con un red con 3 capas [2]. De nuevo, las capas de salida usan unidades lineales, las dos capas ocultas usan unidades sigmoides, y no se sabe, en general; el número de unidades requeridas en cada capa.

Este último resultado nos dice que con redes neuronales con pocas capas ocultas podemos llegar a conseguir funciones realmente complejas.

1.4.3. Búsqueda en el espacio de hipótesis y sesgo inductivo

Puede llegar a ser interesante comparar la búsqueda en el espacio de hipótesis de la retropropagación con la búsqueda de otros algoritmos de aprendizaje. Para el primero, el espacio de hipótesis es el espacio euclídeo n-dimensional de los n pesos de la red. Al contrario que otros algoritmos, como el de árboles de decisión, basados en representaciones discretas; el espacio de hipótesis de la retroprogación es continuo. A esto le añadimos que E es diferenciable sobre parámetros continuos y lo que obtenemos es un gradiente de E bien definido el cual nos ofrece una estructura muy fácil de organizar y manejar.

Es complicado caracterizar de manera precisa el sesgo inductivo del aprendizaje mediante retropropagación, puesto que este depende de la interacción entre la búsqueda del descenso del gradiente y la forma en la cual el espacio de los pesos va abarcando el espacio de funciones representables. De cualquier forma, podemos caracterizarlo como *una suave interpolación entre puntos de dato*. Es decir, dados dos ejemplos de entrenamiento positivos sin ejemplos negativos entre ellos, la retropropagación tiende a etiquetar también los puntos que hay entre ellos como positivos.

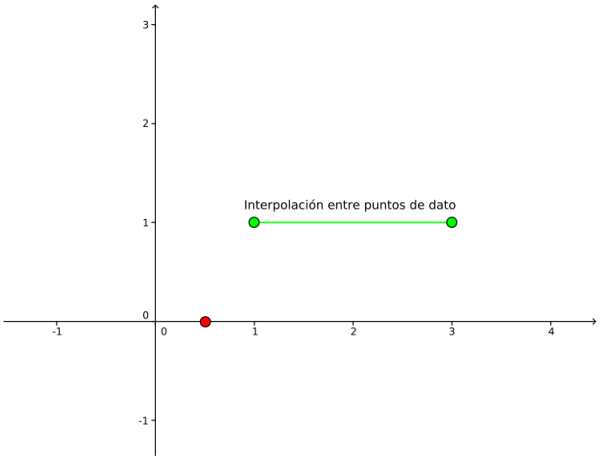
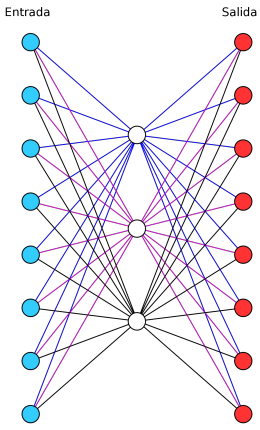


Figura 1.6: Representación de puntos de dato con entrada $x \in \{0{,}5, 1, 3\}$. Gracias al sesgo inductivo que tenemos, el algoritmo tenderá a etiquetar los valores intermedios como positivos.

1.4.4. Representaciones de las Capas Ocultas



(a) Representación Gráfica de la red

Entrada	Oculta			Salida
	Valores			
10000000	0.89	0.04	0.08	10000000
01000000	0.15	0.99	0.99	01000000
00100000	0.01	0.97	0.27	00100000
00010000	0.99	0.97	0.71	00010000
00001000	0.03	0.05	0.02	00001000
00000100	0.01	0.11	0.88	00000100
00000010	0.80	0.01	0.98	00000010
00000001	0.60	0.94	0.01	00000001

(b) Representación mediante tabla de la red una vez aprendida

Figura 1.7: Representación de una capa oculta que ya ha aprendido. Hemos entrenado esta red de 8 x 3 x 8 para aprender la función identidad, hemos usado los 8 ejemplos de entrenamiento. Después de 5000 entrenamientos, las unidades ocultas han codificado las ocho unidades de entrada con la codificación de la derecha. Ejemplo sacado de Tom Mitchell [13]

Una propiedad bastante útil de la retropropagación es su habilidad para descubrir representaciones intermedias útiles en las capas ocultas de la red, puesto que los ejemplos de entrenamiento nos dan sólo las entradas y las salidas, el procedimiento intermedio para ir alterando los pesos es libre de ir dando más relevancia o menos a las unidades ocultas para encajar los datos de salida y minimizar el error cuadrático. Esta información puede resul-

tar de gran utilidad puesto que indica qué datos de entrada son más o menos relevantes para la función objetivo.

Consideremos, por ejemplo, la red de la figura 1.7, donde podemos observar que nuestras 8 unidades de entradas están conectadas a tres unidades ocultas que a su vez están conectadas con 8 unidades de salida

1.4.5. Criterios de Parada

En este apartado vamos a estudiar cuáles pueden ser las condiciones más apropiadas para terminar el bucle del algoritmo de retropropagación. Lo primero que se nos puede venir a la cabeza puede ser el de continuar el entrenamiento hasta que el error alcance una cota prefijada, no obstante; veremos que esta estrategia resulta ser poco efectiva puesto que el algoritmo de retropropagación es susceptible a sobreajustar los ejemplos de entrenamiento a costa del decrecimiento de la precisión del mismo.

¿Por qué sucede este sobreajuste en las iteraciones más tardías y no las más tempranas? Considerando que iniciamos con pesos aleatoriamente pequeños, sólo podemos describir superficies de decisión muy suaves o simples. Conforme vayamos entrenando, algunos de los pesos irán creciendo y la complejidad de la superficie de decisión irá aumentando. En resumidas cuentas, la complejidad que podemos llegar a alcanzar aumenta con el número de iteraciones. Luego con las suficientes iteraciones, el algoritmo va a crear superficies demasiado complejas las cuales crearán un ruido en el ajuste de los datos de entrenamiento, o bien en características no representativas de ese ejemplo de entrenamiento en particular.

Existen diversas técnicas para aliviar este problema, dos de las más comunes son:

- La técnica de *decantamiento de los pesos* que consiste en disminuir cada peso por un factor pequeño en cada iteración. Básicamente redefinimos E para incluir un término penalizador sobre todos los pesos. La motivación de esta técnica es la de intentar mantener los pesos pequeños para disminuir la complejidad que puede llegar a alcanzar la superficie de decisión.

- Una de las técnicas más efectivas resulta ser la de proporcionar unos datos de validación en el algoritmo además de los datos de entrenamiento. El algoritmo monitoriza el error respecto de este conjunto de validaciones, mientras usa los ejemplos de entrenamiento para dirigir la búsqueda del descenso por el gradiente. En una típica implementación de este proceso, mantendremos dos copias de los pesos de la red: una para el entrenamiento y otra de los pesos que mejor resultado hayan hecho hasta ahora, midiéndolos respecto al error con los datos de validación. Una vez que los valores de salida hayan alcanzando un error suficientemente más grande que el que ya tenemos guardado, devolvemos este último como valor final.

- Un último criterio de parada bastante común, aunque no asegura tal efectividad como los anteriores, consiste en parar en un número prefijado de iteraciones. Esta es una manera de asegurar que la superficie de decisión no alcance una gran complejidad.

En general, este problema de sobreajuste y cómo solucionarlo es bastante delicado. De hecho, este problema es mayor si tenemos conjuntos de entrenamiento pequeños. Una variante de la técnica de validación para estos casos suele ser hacer la validación en k ocasiones diferentes, haciendo un promedio de los mejores resultados.

CAPÍTULO 2

Diseño de la aplicación

En este capítulo hablaremos de la implementación de nuestra librería sobre redes neuronales en Haskell, para la que nos hemos inspirado en la librería Neurolab de Python [14]. Esta librería desarrolla algoritmos básicos para redes neuronales, con configuraciones flexibles tanto para los algoritmos de aprendizaje, como para las redes en sí.

2.1– Librerías

Comencemos por hablar de las librerías necesarias para este proyecto. Necesitaremos:

1. Una librería para trabajar con las matrices peso como es Data.Matrix.
2. Una librería para generar los pesos aleatorios bajo cierto umbral como puede ser System.Random.
3. Una librería para pasar del tipo de dato IO (Float) a Float para poder manejar los datos aleatorios generados, como puede ser la librería System.IO.Unsafe.
4. Una librería para usar principalmente genericLength la cual daremos uso en las funciones de error, Data.list.

```
1 import qualified Data.Matrix as M
2 import qualified System.Random as S
3 import System.IO.Unsafe
4 import Data.List
```

Código 2.1: Librerías Importadas

2.2– Nuevo tipo de Dato

Vamos a implementar un nuevo tipo de dato para las FNN en haskell. La manera de implementarlas va a ser muy parecida a como lo hace la librería Neurolab en Python, pero antes vamos a hacer un tipo de dato para las funciones de activación. Como en haskell no tenemos forma de derivar de manera simbólica, nuestras funciones de activación deberán venir acompañadas de su correspondiente derivada

```
data ACT = Act (Float -> Float) (Float -> Float)
```

Código 2.2: Tipo de dato para las funciones de activación

Ahora ya estamos listos para implementar nuestro tipo de dato para las redes neuronales

```
data FNN = Net [Int] [M.Matrix Float] [ACT]
```

Código 2.3: Tipo de dato para las FNN

donde

- La primera lista describe la estructura de la red, más concretamente indica el número de unidades que tiene cada una de las capas que forman la red.
- La segunda lista almacena las matrices peso para ir de de la capa i-ésima a la capa $i+1$-ésima.
- La tercera lista contiene la función de activación con su derivada para cada capa.

Por último también implementaremos un tipo de dato para el Error de la red en los ejemplos de entrenamiento. Al igual que en las funciones de activación, también haremos uso de la correspondiente derivada, así que lo implementaremos de la siguiente forma:

```
data ERROR = Error ([Float] -> [Float] -> Float) ([Float] -> [Float] -> [Float])
```

Código 2.4: Tipo de dato para las funciones Error

Como la función Error es de tipo $\mathbb{R}^n \longrightarrow \mathbb{R}$ al hablar de su derivada, hablaremos de gradiente y por lo tanto ∇E: $\mathbb{R}^n \longrightarrow \mathbb{R}^n$, además (aunque lo demos por fijado) el error también depende de los ejemplos de entrenamiento.

2.3– Funciones Constructoras

Otros elementos base para la aplicación son las funciones constructoras para las redes neuronales. Aquí podemos diferenciar las dos principales maneras de implementar los pesos iniciales: pesos aleatorios pequeños (bajo cierto umbral) y pesos nulos.

2.3.1. Iniciación por Pesos Fijos

Comencemos por la más simple, el perceptrón simple,

```
newpValue :: Int -> ACT -> Float -> FNN
newpValue n (Act f df) v =
  Net [n,1] [M.transpose (M.fromLists [replicate (n+1) v])] [Act f df]
```

Código 2.5: Función constructora del Perceptrón simple con pesos nulos

Como podemos observar, hemos generado un perceptrón simple con n unidades de entrada y función de activación f.

Para movernos en casos más generales vamos a implementar una función para definir redes multicapa:

```
newffValue :: [Int] -> [ACT] -> Float -> FNN
newffValue ns fs v =
  Net ns (foldl (\ z (x1,x2) -> z++[M.matrix (x1+1) x2 (\ (i,j) -> v)]) []
                (zip ns (tail ns))) fs
```

Código 2.6: Función constructora de la red neurnal multicapa con un valor fijo v

Cada capa i-ésima tendrá un número de entradas definida por el i-ésimo entero de la lista ns, y con la i-ésima función de activación de la lista fs.

Un caso particular de éste es el inicio con pesos nulos el cual lo vamos a tener como construcción aparte, tanto como el perceptrón unicapa como para la red multicapa:

```
newpZero :: Int -> ACT -> FNN
newpZero n (Act f df) =
  newpValue n (Act f df) 0

newffZero :: [Int] -> [ACT] -> FNN
newffZero ns fs =
  newffValue ns fs 0
```

Código 2.7: Funciones constructoras con pesos nulos

2.3.2. Iniciación por Pesos Aleatorios

Aquí vamos a necesitar algunas funciones auxiliares, comencemos por generar las matrices con entradas aleatorias (acotadas).

```
randomMatrix :: Float -> Float -> Int -> Int -> IO (M.Matrix Float)
randomMatrix min max n m =
  do vs <- sequence [S.randomRIO (min,max) :: IO Float | i <- [1..n*m]]
     let a = M.fromList n m vs
     return a
```

Código 2.8: Matriz con entradas aleatorias

Aquí estamos generando una matriz $n \times m$ con entradas aleatorias acotadas por min y max por abajo y arriba respectivamente.

Ahora ya podemos implementar el perceptrón simple

```
newpRand :: (Float,Float) -> Int -> [ACT] -> FNN
newpRand (min,max) n fs =
 Net [n,1] [unsafePerformIO (randomMatrix min max (n+1) 1)] fs
```

Código 2.9: Función constructora del Perceptrón simple con entradas aleatorias

Para la versión de la red neuronal multicapa, daremos uso de una función auxiliar para generar la lista de matrices peso,

```
randWs :: Float -> Float -> [Int] -> [M.Matrix Float]
randWs min max xs =
  foldl (\ ws (x1,x2) -> ws ++ [unsafePerformIO(randomMatrix min max (x1+1) x2)])
       [] (zip xs (tail xs))
```

Código 2.10: Función auxiliar para la lista de matrices peso

Finalmente, la versión de la red neuronal multicapa queda:

```
newffRand :: (Float,Float) -> [Int] -> [ACT] -> FNN
newffRand (min,max) ns fs =
  Net ns (randWs min max ns) fs
```

Código 2.11: Función constructora de la red neuronal multicapa con entradas aleatorias

2.4– Funciones de Activación

En esta sección implementaremos distintos tipos de funciones de activación para nuestras redes (con sus correspondientes derivadas).

Antes de ello construyamos dos funciones para extraer la función y su derivada del tipo de dato de activación.

```
activ :: ACT -> (Float -> Float)
activ (Act f df) = f

derAct :: ACT -> (Float -> Float)
derAct (Act f df) = df
```

Código 2.12: Función de activación y su derivada

Una vez teniendo en cuenta esto, las funciones de activación con las que vamos a trabajar son las siguientes:

1. **Activación lineal**
   Esta es la función identidad

```
actLin :: Float -> Float
actLin x = x

dActLin :: Float -> Float
dActLin x = 1

pureLin :: ACT
pureLin = Act actLin dActLin
```

Código 2.13: Función de Activación Lineal

2. **Activación lineal saturada**
   Esta se va a diferenciar de la anterior cuando escapemos del intervalo (0,1).

```
actLinSat :: Float -> Float
actLinSat x =
  if x <0
  then 0
  else if x>1
       then 1
       else x

dActLinSat :: Float -> Float
dActLinSat x =
```

```
11   if x>0 && x<1
12   then 1
13   else 0
14
15 satLin :: ACT
16 satLin = Act actLinSat dActLinSat
```

Código 2.14: Función de Activación Lineal Saturada

3. **Sigmoide**
   De esta función que ya hemos hablado de manera extendida en el anterior capítulo

```
1  sigmoide :: Float -> Float
2  sigmoide x =
3    1/(1+exp(-x))
4
5  dsigmoide :: Float -> Float
6  dsigmoide x =
7    (sigmoide x) * (1-sigmoide x)
8
9  logSig :: ACT
10 logSig = Act sigmoide dsigmoide
```

Código 2.15: Función de Activación sigmoide

4. **Tangente Hiperbólica**
   De esta haskell ya tiene una implementación hecha (*tanh*), luego basta con implementar la derivada

```
1 dtanh :: Float -> Float
2 dtanh x = 1 - (tanh x)^2
3
4 tanSig :: ACT
5 tanSig = Act tanh dtanh
```

Código 2.16: Función de Activación Tangente Hiperbólica

5. **Unidad Lineal Rectificada**
   Se anula en valores negativos

```
1  actReLU :: Float -> Float
2  actReLU x =
3    max 0 x
4
5  dActReLU :: Float -> Float
6  dActReLU x =
7    if x>0 then 1 else 0
8
9  reLU :: ACT
10 reLU = Act actReLU dActReLU
```

Código 2.17: Función de Activación ReLU

6. **Unidad Lineal Rectificada Suavizada** ("leaky" en inglés).

```
actLeakyReLU :: Float -> Float
actLeakyReLU x =
  if x<0
  then 0.01*x
  else x

dActLeakyReLU :: Float -> Float
dActLeakyReLU x =
  if x<0
  then 0.01
  else 1

leakyReLU :: ACT
leakyReLU = Act actLeakyReLU dActLeakyReLU
```

Código 2.18: Función de Activación leaky ReLU

7. **Función Umbral**
   De gran utilidad para ejemplos de entrenamiento Booleanos.

```
actHardLim :: Float -> Float
actHardLim x = if x > 0 then 1 else 0

dActHardLim :: Float -> Float
dActHardLim x = 0

hardLim :: ACT
hardLim = Act actHardLim dActHardLim
```

Código 2.19: Función de Activación Umbral

2.5– Funciones de Error

En esta sección vamos a implementar las funciones error entre los valores objetivos (que llamaremos *ys*) y los valores de salida de la red (que llamaremos *os*). También añadiremos las correspondientes derivadas para un uso futuro en los métodos de entrenamiento.

Los errores con los que trabajaremos son:

1. **Error Basado en Entropía**
   Este error sólo lo podemos usar cuando los valores objetivos son booleanos {0,1}

```
cee :: [Float] -> [Float] -> Float
cee ys os =
  - sum (map (\ (y,o) -> (y*log(o)+(1-y)*log(1-o))) (zip ys os))

dcee :: [Float] -> [Float] -> [Float]
dcee ys os =
  zipWith (\ y o -> -y/o + (1-y)/(1-o)) ys os

```

```
eCEE :: ERROR
eCEE = Error cee dcee
```

Código 2.20: Función de error basada en entropía con su derivada

2. Error Absoluto Medio

```
mae :: [Float] -> [Float] -> Float
mae ys os =
  (sum (zipWith (\ y o -> abs (y-o)) ys os))/(genericLength ys)

dmae :: [Float] -> [Float] -> [Float]
dmae ys os =
  zipWith (\ y o -> if y < o then 1 else -1) ys os

eMAE :: ERROR
eMAE = Error mae dmae
```

Código 2.21: Función de error absoluto medio con su derivada

3. Error cuadrático Medio

```
mse :: [Float] -> [Float] -> Float
mse ys os =
  let n = 1/ (genericLength ys)
  in n*(sum (zipWith (\ y o -> (y-o)^2) ys os))

dmse :: [Float] -> [Float] -> [Float]
dmse ys os =
   let n = 1/(genericLength ys)
   in zipWith (\ y o -> 2*n*(o-y)) ys os

eMSE :: ERROR
eMSE = Error mse dmse
```

Código 2.22: Función de error cuadrático medio con su derivada

4. Suma de Errores Absolutos

```
sae :: [Float] -> [Float] -> Float
sae ys os =
  sum (zipWith (\ y o -> abs (y-o)) ys os)

dsae :: [Float] -> [Float] -> [Float]
dsae ys os =
  zipWith (\ y o -> (if y<o
                     then 1
                     else -1)) ys os

eSAE :: ERROR
eSAE = Error sae dsae
```

Código 2.23: Función de suma de errores absolutos con su derivada

5. **Suma de Errores cuadráticos**

```
sse :: [Float] -> [Float] -> Float
sse ys os =
  (sum (zipWith (\ y o -> (y-o)^2) ys os))/2

dsse :: [Float] -> [Float] -> [Float]
dsse ys os =
  zipWith (\ y o -> o-y) ys os

eSSE :: ERROR
eSSE = Error sse dsse
```

Código 2.24: Función de suma de errores cuadráticos con su derivada

En la actualidad existen muchas otras funciones error o versiones distintas de las antes mencionadas, pero estás son las más comunes con diferencia.

2.6– Métodos de Entrenamiento

En esta sección atacaremos dos métodos de entrenamiento para nuestras redes neuronales, la regla Delta y el modelo de entrenamiento con todos los ejemplos, también conocido como Batch.

Antes de esto cabe destacar la utilidad de dos funciones que van a servir de gran ayuda. La primera es una función que devuelve los valores de salida de una red a la que le damos unos valores de entrada xs.

```
salidaRed :: [Float] -> FNN -> [Float]
salidaRed xs (Net _ ws fs) =
  let ass= M.toList (foldl (\ x (w,Act f df) -> fmap f ((M.fromLists [[-1]] M.<|>
       x)*w))
                   (M.fromLists [xs]) (zip ws fs))
  in ass
```

Código 2.25: Función salida de una red

A la hora de los entrenamientos como el del tipo gradiente vamos a necesitar los valores de entrada de cada unidad, su valor al aplicar la función de activación y la derivada correspondiente.

```
valorEntradaSalidaDerivada :: [Float] -> FNN -> [([Float],([Float],[Float]))]
valorEntradaSalidaDerivada xs (Net _ ws fs) =
  map (\(x,y,z) -> (M.toList x,(M.toList y,M.toList z)))
  (scanl (\ (x,y,z) (w, Act f df) -> let ins = (M.fromLists [[-1]] M.<|> y)*w
                                     in (ins, fmap f ins , fmap df ins))
  ((M.fromLists [xs]),(M.fromLists [xs]),(fmap (derAct (head fs)) (M.fromLists [
      xs]))) (zip ws fs))
```

Código 2.26: Función Auxiliar para los valores de las unidades

Esta función a partir de unos valores de entrada y una red nos da una lista de ternas de vectores (a,b,c) donde

- a es el vector de valores de entrada de la capa i-ésima.

- b es el vector de valores al evaluar la función de activación i.e. $g(a)$.
- c es el vector de valores al evaluar la derivada de la función de activación i.e. $g'(a)$.

Realmente no es una terna, la salida de la función es de la forma (a,(b,c)) la cual es un par dentro de otro para extraerlos más fácilmente a posteriori con combinaciones de las funciones predefinidas fst y snd.

Debemos aclarar que nuestros ejemplos de entrenamiento serán listas de pares ordenados formados por un vector de entrada y su vector objetivo correspondiente (xs, ts).

Ya estamos a disposición de comenzar con los métodos de entrenamiento en sí. Lo dividiremos en dos secciones, unicapa y multicapa.

2.6.1. Entrenamiento del Perceptrón Simple

Dentro de que estamos aplicando el método del gradiente, utilizaremos dos reglas de aprendizaje: Delta y Batch.

La diferencia principal entre estas dos es que la regla Batch calcula el error entre todos los ejemplos de entrenamiento y los valores de salida de nuestra red y ya después actualiza los pesos de la red, mientras que la regla Delta va calculando el error y actualizando los pesos para cada ejemplo del conjunto de entrenamiento.

1. Regla de Aprendizaje Delta

```
1 entrenoPerceptronDelta :: FNN -> [([Float],[Float])] -> Float -> ERROR ->
      FNN
2 entrenoPerceptronDelta (Net ns [w] [f]) zs eta (Error error derror) =
3   Net ns [foldl (\ r z ->
4                  let ins = M.toList ((M.fromLists [(-1):(fst z)])*r)
5                      fin = map (activ f) ins
6                      [errorin] = zipWith (-) fin (snd z)
7                      nw = r - (M.transpose (fmap (*(eta*errorin))
8                                            (M.fromLists [(-1):(fst z)])))
9                  in nw) w zs] [f]
```

Código 2.27: Entrenamiento del Perceptrón simple con regla de aprendizaje Delta

donde zs es la lista de ejemplos de entrenamiento y eta es el factor de aprendizaje del cual ya hemos hablado en el capítulo 1.

2. Regla de entrenamiento Batch.

```
1 entrenoPerceptronBatch :: FNN -> [([Float],[Float])] -> Float -> ERROR ->
      FNN
2 entrenoPerceptronBatch (Net ns [w] [f]) zs eta (Error error derror) =
3   let erroresB = scanl (\ r z -> let ins = M.toList ((M.fromLists [ (-1):(
        fst z)])*w)
4                                      fin = map (activ f) ins
5                                      dfin = map (derAct f) ins
6                                      [errorin] = zipWith (-) fin (snd z)
7                                  in M.transpose (fmap (*(eta*errorin)) (M.
                                       fromLists [(-1):(fst z)]))) w zs
8   in Net ns [foldl (\ r e -> r-e) w erroresB ] [f]
```

Código 2.28: Entrenamiento del Perceptrón simple con la regla de aprendizaje Batch

2.6.2. Entrenamiento de Redes Nueronales Multicapa

Antes de presentar este algoritmo vamos a necesitar algunas funciones útiles para la implementación:

- Para conseguir algunos elementos de las unidades vamos a necesitar la matriz de pesos sin los w_0

```
quitaW0 :: M.Matrix Float -> M.Matrix Float
quitaW0 m = M.fromLists (tail (M.toLists m))
```

Código 2.29: Función eliminatoria de los pesos w_0

- Para la actualización de los pesos necesitaremos multiplicar componente a componente los vectores y no sólo el producto escalar estándar

```
prodComp :: [Float] -> [Float] -> [Float]
prodComp xs ys =
  foldl (\ r (x,y) -> r++[x*y]) [] (zip xs ys)
```

Código 2.30: Función producto componente a componente

- Por último, pero no menos importante, la función que devuelve las matrices cuyas entradas son los valores de las derivadas del error de ese ejemplo de entrenamiento concreto respecto de los pesos de cada capa.

```
matricesLd :: FNN -> ([Float],[Float]) -> ERROR -> [M.Matrix Float]
matricesLd (Net ns ws fs) (xs,ts) (Error error derror) =
  let valoresUnidades = valorEntradaSalidaDerivada xs (Net ns ws fs)
      osSalida = fst (snd (last valoresUnidades))
      deltafinal = M.transpose (M.fromLists [prodComp (derror ts osSalida) (
          snd (snd (last valoresUnidades)))])
      wsOrdenado = reverse (tail ws)
      valoresUnidadesOrdenado = map (snd) (tail (reverse valoresUnidades))
      valoresAct = map fst valoresUnidadesOrdenado
      valoresDeriv = map snd valoresUnidadesOrdenado
      deltas = scanl (\ delta (deriv,w) -> M.transpose (M.fromLists [
          prodComp (M.toList ((quitaW0 w)*delta)) deriv ])) deltafinal (zip
          valoresDeriv wsOrdenado )
  in  map (\ (o,delta) -> M.transpose (delta * (M.fromLists [(-1):o]))) (zip
       valoresAct deltas)
```

Código 2.31: Función matrices de error por capa

Como ya explicamos en el capítulo 1, calcular el error de las unidades ocultas es distinto del error de las unidades de salida. En el algoritmo de retropropagación comenzamos por estas últimas puesto que de ellas van a depender el resto de iteraciones. Cuando hablamos de delta en el código, nos referimos al producto de las derivadas parciales de la función de error multiplicadas por las derivadas de la función de activación, todo ello respecto del valor de entrada de la unidad i. Una vez tengamos estos deltas ya sólo nos queda multiplicar por el vector $(-1, os)$ para tener la matriz de los errores de los pesos.

En esta sección estamos haciendo uso del método de descenso por el gradiente, pero vamos a ver las dos versiones como en la sección anterior:

1. Regla de Aprendizaje Delta

```
entrenoGradienteDelta :: FNN -> [([Float],[Float])] -> Float -> ERROR -> FNN
entrenoGradienteDelta (Net ns ws fs) ds eta error =
  foldl (\ red d -> let matricesLD= matricesLd red d error
                        wsActualizado = map (\ (w,m) -> w - (fmap (*eta) m)) (
                            zip ws (reverse matricesLD))
                    in Net ns wsActualizado fs)
  (Net ns ws fs) ds
```

Código 2.32: Entrenamiento de tipo gradiente Delta multicapa

2. Regla de Aprendizaje Batch

```
entrenoGradienteBatch :: FNN -> [([Float],[Float])] -> Float -> ERROR -> FNN
entrenoGradienteBatch (Net ns ws fs) ds eta error =
  let matricesLD = foldl (\ actualizado d-> zipWith (+) (matricesLd (Net ns
      ws fs) d error) actualizado) (matricesLd (Net ns ws fs) (head ds)
      error) (tail ds)
      wsActualizado = map (\ (w,m) -> w - (fmap (*eta) m)) (zip ws (reverse
          matricesLD))
  in Net ns wsActualizado fs
```

Código 2.33: Entrenamiento de tipo gradiente Batch multicapa

2.7– Criterios de parada

Por último veamos los criterios de parada, de los que vamos a implementar los dos más usuales.

1. Iterar varias veces sobre los mismos ejemplos de entrenamiento (Epochs)

   - Para el Perceptrón simple

```
perceptronDeltaEpochs n net zs eta error =
   foldl (\ r y -> entrenoPerceptronDelta r zs eta error)
         net [1..n]

perceptronBatchEpochs n net zs eta error =
  foldl (\ r y -> entrenoPerceptronBatch r zs eta error)
        net [1..n]
```

Código 2.34: Entrenamiento unicapa iterando n veces

   - Para la red neuronal multicapa

```
gradienteDeltaEpochs n net zs eta error =
   foldl (\ r y -> entrenoGradienteDelta r zs eta error)
         net [1..n]

gradienteBatchEpochs n net zs eta error =
```

```
6   foldl (\ r x -> entrenoGradienteBatch r zs eta error)
7         net [1..n]
```

Código 2.35: Entrenamiento iterando n veces

2. Otro método bastante importante consiste en iterar hasta que el error de la red sea menor que cierto *epsilon* establecido, para ello necesitamos definir el error de la red, que definimos como el error medio de todos los errores cometidos en los ejemplos de entrenamiento

```
1 errorRed (Net ns ws fs) zs (Error error derror) =
2   (1/(genericLength zs))* (sum (map (\ (ins,ts) -> error ts (salidaRed ins (
      Net ns ws fs))) zs))
```

Código 2.36: Error de la red sobre los ejemplos de entrenamiento

- Para la versión del perceptrón simple

```
1  perceptronDeltaEpsilon epsilon net zs eta error =
2    if errorRed net zs error < epsilon
3    then net
4    else let newNet = entrenoPerceptronDelta net zs eta error
5         in perceptronDeltaEpsilon epsilon newNet zs eta error
6
7  perceptronBatchEpsilon epsilon net zs eta error =
8    if errorRed net zs error < epsilon
9    then net
10   else let newNet = entrenoPerceptronBatch net zs eta error
11        in  perceptronBatchEpsilon epsilon newNet zs eta error
```

Código 2.37: Entrenamiento unicapa con error acotado

- Para la versión de la red neuronal multicapa

```
1  gradienteDeltaEpsilon epsilon net zs eta error =
2    if errorRed net zs error < epsilon
3    then net
4    else let newNet = entrenoGradienteDelta net zs eta error
5         in gradienteDeltaEpsilon epsilon newNet zs eta error
6
7  gradienteBatchEpsilon epsilon net zs eta error =
8    if errorRed net zs error < epsilon
9    then net
10   else let newNet = entrenoGradienteBatch net zs eta error
11        in gradienteBatchEpsilon epsilon newNet zs eta error
```

Código 2.38: Entrenamiento multicapa con error acotado

CAPÍTULO 3

Manual de uso

En este capítulo nos centraremos en ejemplos particulares como explicación para dar uso a la librería. Lo primero que vamos a necesitar es una función auxiliar para ir obteniendo información de las matrices peso cuando lo necesitemos.

```
1 pesos (Net xs ws fs) = ws
```

Código 3.1: Función auxiliar peso

Ahora ya estamos en condiciones de empezar con los ejemplos.

3.1– Perceptrón Simple

Comencemos por la versión más sencilla, aquí trabajaremos con conjuntos linealmente separables puesto que tenemos asegurada la convergencia.

Uno de los ejemplos que ya hemos comentado en el capítulo 1 es la función AND la cual sabemos que es linealmente separable como podemos observar en la gráfica siguiente

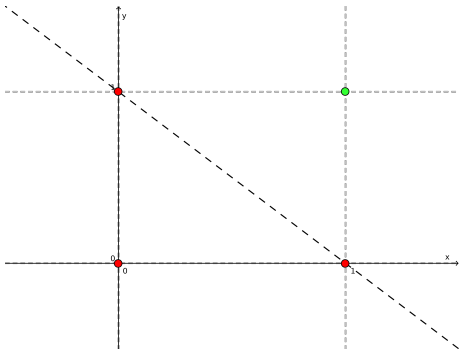


Figura 3.1: Gráfica de la función booleana AND

Al resultar ser una función booleana el dominio de esta función es el conjunto $\{(0,0),(0,1),(1,0),(1,1)\}$, luego sólo podemos considerar 4 ejemplos de entrenamiento. En este caso la convergencia es bastante rápida puesto que realmente le estamos dando los valores exactos de todo el dominio de la función.

Implementemos pues, los cuatro ejemplos de entrenamiento, para ello definamos la función AND y que actúe sobre todos los valores del dominio

```
fAND :: Float -> Float -> Float
fAND x y = if x==1.0 && y==1.0 then 1 else 0

entrada = [[0, 0], [0, 1], [1, 0], [1, 1]]

ejemplosAND = map (\ [x,y] -> ([x,y],[fAND x y])) entrada
```

Código 3.2: Ejemplos de entrenamiento para la función AND

Para el perceptrón usaremos la función de activación Umbral (2.19) así que nos queda la siguiente estructura:

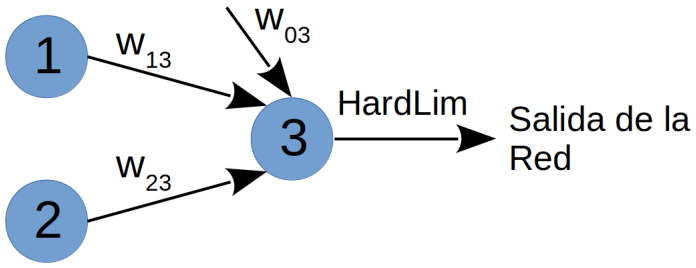


Figura 3.2: Perceptrón Unicapa para la función AND

Vamos a iniciar el modelo con pesos nulos, luego nuestro perceptrón nos queda

```
perceptronAND = newpZero 2 hardLim
```

Código 3.3: Perceptrón simple con pesos iniciales nulos

Como hemos cogido un conjunto linealmente separable tenemos la convergencia asegurada, luego podemos hacer el entrenamiento con el método de parada *Epsilon*. Veamos el entrenamiento tanto para la rutina Batch como la Delta:

```
entrenaRedfANDDeltaEpsilon =
  perceptronDeltaEpsilon 0.1 perceptronAND ejemplosAND 0.1 eSSE

entrenaRedfANDBatchEpsilon =
  perceptronDeltaEpsilon 0.1 perceptronAND ejemplosAND 0.1 eSSE
```

Código 3.4: Rutinas de entrenamiento con método de parada Epsilon

Aquí estamos entrenando con factor de aprendizaje $\eta = 0{,}1$ y nuestra cota de error va a venir dada por $\epsilon = 0{,}1$ con la función de Suma de Errores Cuadráticos.

Las matrices peso que nos salen son

```
  > pesos entrenaRedfANDDeltaEpsilon
[
    0.2
    0.2
    0.1
          ]
  > pesos entrenaRedfANDBatchEpsilon
[
    0.2
    0.2
    0.1
          ]
```

Código 3.5: Matrices peso de ambas rutinas

y efectivamente al evaluar nuestras redes con los valores de entrada, tenemos que los resultados son correctos:

```
  > map (\ x -> salidaRed x entrenaRedfANDBatchEpsilon) entrada
[[0.0],[0.0],[0.0],[1.0]]

  > map (\ x -> salidaRed x entrenaRedfANDDeltaEpsilon) entrada
[[0.0],[0.0],[0.0],[1.0]]
```

Código 3.6: Valores de Salida de ambas redes

Ahora veamos el entrenamiento con el otro criterio de parada, el del número de Epochs, para este ejemplo aplicaremos 10 iteraciones (de nuevo con $\eta = 0{,}1$).

```
entrenaRedfANDDeltaEpochs =
  perceptronDeltaEpochs 10 perceptronAND ejemplosAND 0.1 eSSE

entrenaRedfANDBatchEpochs =
  perceptronBatchEpochs 10 perceptronAND ejemplosAND 0.1 eSSE
```

Código 3.7: Entrenamiento *Epsilon* para AND

Pero aquí observamos que el error puede ser crucial, al iterar bajo tan pocos ejemplos la versión Batch aún no da la solución exacta:

```
  > map (\ x -> salidaRed x entrenaRedfANDDeltaEpochs) entrada
[[0.0],[0.0],[0.0],[1.0]]

  > map (\ x -> salidaRed x entrenaRedfANDBatchEpochs) entrada
[[0.0],[0.0],[0.0],[0.0]]
```

Código 3.8: Valores de Salida con el método de parada Epochs

3.2– Perceptrón Multicapa

3.2.1. Función XOR

Ahora ya tenemos una mayor flexibilidad para hacer los ejemplos de entrenamiento, ya hemos hablado largo y tendido de la complejidad que pueden llegar a representar los

perceptrones multicapa, empecemos con un ejemplo booleano como en la sección anterior, la función XOR.

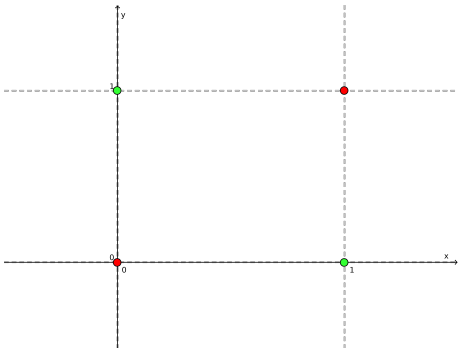


Figura 3.3: Gráfica de la función XOR

Como podemos observar en la gráfica, no es una función linealmente separable, luego no tendríamos convergencia con el perceptrón unicapa, con una sola capa oculta de tres unidades podremos conseguir una alta precisión, luego nuestro perceptrón será de la forma:

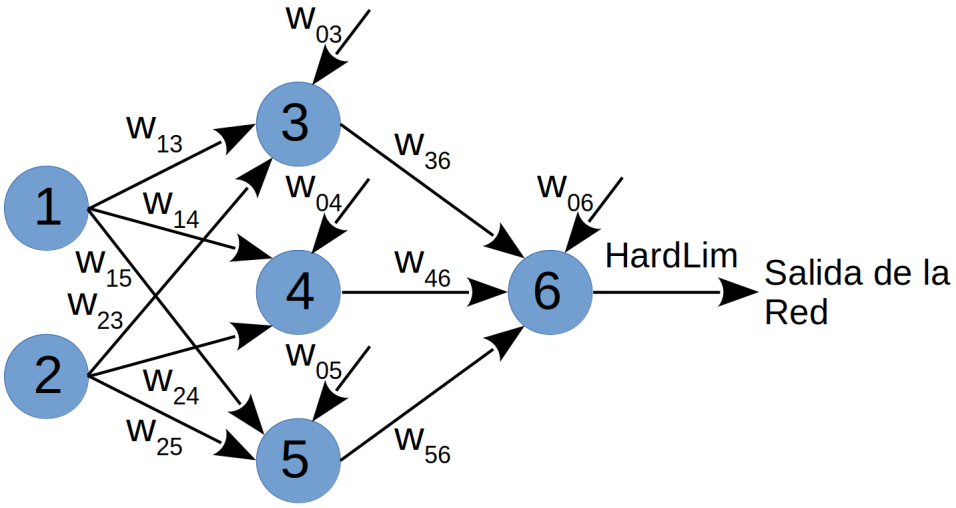


Figura 3.4: Red Neuronal Multicapa para la función XOR

Esta vez usemos el método de iniciación con valores aleatorios acotados (entre ±0,5) para visualizar un uso práctico del mismo. El dominio es el mismo que en la función AND pero con imagen diferente:

```
1 fXOR :: Float -> Float -> Float
2 fXOR x y = if x/= y then 1 else 0
3
4 ejemplosXOR = map (\ [x,y] -> ([x,y],[fXOR x y])) entrada
5
6 redXOR = newffRand (0,1) [2,3,1] [hardLim,hardLim]
```

Código 3.9: Red y ejemplos de entrenamiento para la función XOR

Ahora pues entrenemos con la rutinas de aprendizaje Batch, comencemos por el método de parada *Epsilon*

```
entrenaRedfXORBatchEpsilon =
  gradienteDeltaEpsilon 0.5 redXOR ejemplosXOR 0.1 eSSE
```

Código 3.10: Rutina de entrenamiento para la función XOR con el criterio de parada *Epsilon*

Ahora estamos entrenando con un factor de aprendizaje $\eta = 0{,}1$ y bajo el error de la Suma de Errores Cuadráticos y con cota de error $\epsilon = 0{,}5$. Observemos que realmente termina la rutina

```
  > map (\x -> salidaRed x entrenaRedfXORBatchEpsilon) entrada
[[0.0],[1.0],[0.0],[1.0]]
```

Código 3.11: Salida de la red para la función XOR con el criterio de parada *Epsilon*

Y para 2 iteraciones veamos que ya tenemos una red que nos devuelve exactamente todos los valores iguales

```
entrenaRedfXORBatchEpochs =
  gradienteBatchEpochs 2 redXOR ejemplosXOR 0.1 eSSE
```

Código 3.12: Red para la función XOR con el criterio de parada *Epochs*

Luego aquí vemos que es mejor hacer las iteraciones puesto que nos devuelve bien todos los valores de entrada

```
  > map (\x -> salidaRed x entrenaRedfXORBatchEpochs) entrada
[[0.0],[1.0],[0.0],[1.0]]
```

Código 3.13: Salida de la red para la función XOR con el criterio de parada *Epochs*

3.2.2. Función Continua

Cojamos como ejemplo ahora algo más complejo, una función continua, particularmente la función

$$\begin{aligned} f : \mathbb{R} &\longrightarrow \mathbb{R} \\ x &\longmapsto f(x) = \frac{\sin x}{2} \end{aligned}$$

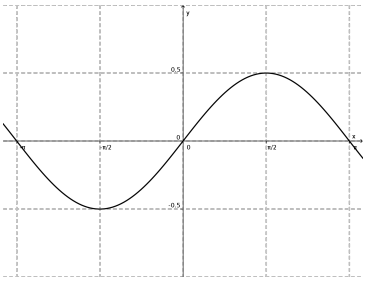


Figura 3.5: Representación Gráfica de f

Ahora, al ser el dominio finito obviamente no vamos a dar todos los puntos, usaremos la siguiente lista de puntos equidistantes entre -7 y 7 con sus respectivas imágenes:

```
lsx = [-7.0     , -6.26315789, -5.52631579, -4.78947368, -4.05263158,
       -3.31578947, -2.57894737, -1.84210526, -1.10526316, -0.36842105,
        0.36842105, 1.10526316, 1.84210526, 2.57894737, 3.31578947,
        4.05263158, 4.78947368, 5.52631579, 6.26315789, 7.0     ]

lsy = map ((/2).sin) lsx :: [Float]

ejemplos = zipWith (\ x y -> ([x],[y])) lsx lsy
```

Código 3.14: Ejemplos para la función f

Para este caso vamos a usar más unidades en la capa oculta que en el ejemplo anterior, le vamos a dar la siguiente estructura

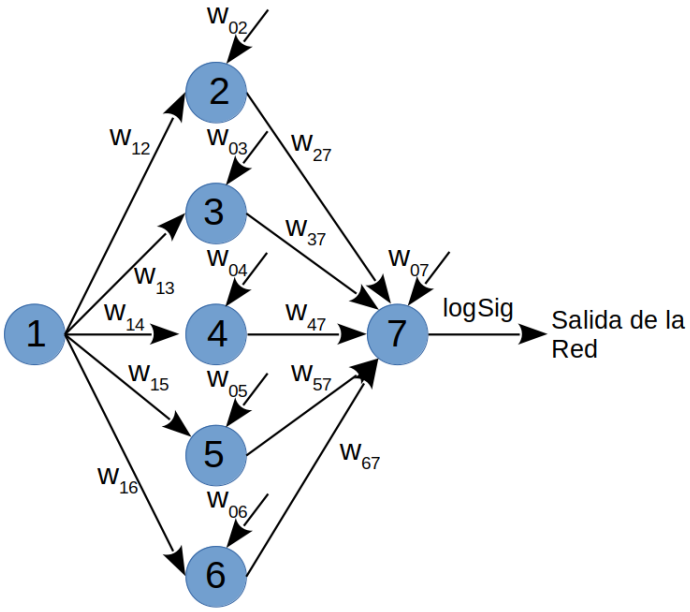


Figura 3.6: Red Neuronal Multicapa para la función f

Los pesos iniciales serán generados aleatoriamente de nuevo, esta vez en un rango entre 0 y 1, quedando de la siguiente forma

```
redSin = newffRand (0,1) [1,5,1] [logSig,logSig]
```

Código 3.15: Red neuronal para la función f

Ahora veamos las diferencias que podemos llegar a conseguir con las dos rutinas de entrenamiento con el método de parada *Epochs*, fijando 500 iteraciones, veamos primero la versión Batch

```
redSinSolBatch =
  gradienteBatchEpochs 500 redSin ejemplos 0.1 eSSE
```

Código 3.16: Entrenamiento Batach para la función f bajo 500 iteraciones

la cual nos da un error

```
1  > errorRed redSinSolBatch ejemplos eSSE
2 5.7890754e-2
```

Código 3.17: Error de entrenamiento

Ahora bien, veamos la versión delta

```
1 redSinSolDelta =
2   gradienteDeltaEpochs 500 redSin ejemplos 0.1 eSSE
```

Código 3.18: Entrenamiento Delta para la función f bajo 500 iteraciones

que genera el siguiente error

```
1  > errorRed redSinSolDelta ejemplos eSSE
2 0.124147944
```

Código 3.19: Error de entrenamiento

Como podemos observar, la diferencia entre las dos rutinas es bastante notoria para llevar 500 iteraciones, (diferencia del error entorno al 0,05) el error de la rutina Delta está doblando al error de la rutina Batch. Observando con el error de la versión Batch, tenemos asegurada la convergencia bajo una cota de error de 0,0579, es decir, que podemos entrenar la red con el método de parada *Epsilon* con $\epsilon = 0{,}1$ por ejemplo, sin temor a que no termine:

```
1 redSinSolBatchEpsilon =
2   gradienteBatchEpsilon 0.1 redSin ejemplos 0.1 eSSE
```

Código 3.20: Entrenamiento Batch con el método de parada *Epsilon*

Y efectivamente, podemos comprobar que el error es menor que la cota

```
1  > errorRed redSinSolBatchEpsilon ejemplos eSSE
2 9.135607e-2
```

Código 3.21: Error de entrenamiento

CAPÍTULO 4

Conclusiones

En está implementación intentamos acercarnos, de manera introductoria, a este complejo mundo que se sigue desarrollando hoy en día, si bien aquí hemos usado ejemplos algo básicos, la profundidad de aprendizaje de las redes puede llegar a ser realmente compleja (como ya hablamos en el capítulo 1). Lejos de estar completamente optimizada, creemos que con esta librería se podrían resolver problemas más complejos haciendo uso de ordenadores más potentes.

Aunque sobre redes neuronales la librería Neurolab resulta muy atractiva por su sencilla comprensión, a la hora de la verdad esta se queda muy corta en cuanto a la complejidad de las redes que puede modelar, para estos casos algunas de las más utilizadas son Keras [9] y TensorFlow [16]. También existen librerías en Haskell que proporcionan funcionalidades para definir y entrenar redes neuronales, tales como Neural [7].

La librería Neural de Haskell, además de poseer una gran flexibilidad a la hora de generar las redes neuronales y entrenarlas, tiene la gran ventaja de usar la derivación automática con la librería ad [4] para automatizar el algoritmo de descenso por el gradiente en el proceso de retropropagación.

4.1– Ampliaciones futuras

1. Sería muy recomendable el uso de la librería ad [4], no sólo por las ventajas que ofrecen en Neural, sino que además podríamos ahorrarnos el hecho de tener que implementar cada función de activación con su respectiva derivada (también seria más fácil para el usuario generar nuevas funciones de activación).

2. Una gran desventaja a la hora de usar funciones con el tipo de dato Data.Matrix es que al actualizar una matriz, por muy pequeño que sea el cambio, se crea una copia entera de la misma. Esto genera una gran ineficiencia puesto que estamos gastando una gran cantidad de memoria en el algoritmo. ¿Cómo podemos resolver este problema? Una de las opciones es usar la interfaz de MArray para arrays mutables [5]. Con estos arrays mutables podemos modificar la matriz de pesos entrada a entrada sin tener que genera una copia entera en cada paso.

3. El mayor inconveniente es el tiempo de convergencia. Las aplicaciones reales pueden llegar a tener miles de ejemplos en el conjunto de entrenamiento y ello requiere

días de tiempo de cálculo. Además la retropropagación es susceptible de fallar en el entrenamiento, es decir, la red puede que nunca llegue a converger. ¿Cómo podemos solucionar esto? Añadiendo a la librería algunas de las variaciones del algoritmo que ya hemos comentado en el capítulo 1, tales como añadir *momentum* o modificar el factor de aprendizaje η en cada iteración, a uno cada vez más pequeño.

Bibliografía

[1] G. Cybenko, *G. Approximation by superpositions of a sigmoidal function.* Math control Signal Systems 2 , pp.303 - 314, 1989.

[2] G. Cybenko, *Continuous Valued Neural Networks with Two Hidden Layers are Sufficient.* Department of Computer Science, Tufts University, 1988.

[3] R. O. Duda y P. E. Hart, *Pattern classification and scene analysis.* Wiley, 1973.

[4] Haskell, *ad: Automatic Differentiation.*
Consultado en https://hackage.haskell.org/package/ad

[5] Haskell, *Data.Array.MArray .* Consultado en
http://hackage.haskell.org/package/array-0.5.4.0/docs/Data-Array-MArray.html .

[6] Haskell, *Data.Matrix .* Consultado en
https://hackage.haskell.org/package/matrix-0.3.6.1/docs/Data-Matrix.html .

[7] Haskell, *Neural* Cosultado en
https://hackage.haskell.org/package/neural

[8] K. Hornik, *Approximation capabilities of multilayer feedforward networks.* Neural Networks, Volume 4, Issue 2, pp. 251-257, 1991.

[9] Keras,
Consultado en https://keras.io/

[10] F. J. Martín Mateos y J. L. Ruiz Reina, *Introducción a las redes neuronales* Apuntes de la asignatura de Inteligencia Artifical, Tema 2, 2018.

[11] W. S. McCulloch and W. H. Pitts, *A Logical Calculus of the Ideas Immanent in Nervous Activity.* The bulletin of mathematical biophysics volume 5, pp. 115–133,1943.

[12] M. Minsky and S. Papert, *Perceptrons.* M.I.T. Press, 1969.

[13] Tom M. Mitchell, *Machine Learning.* McGraw-Hill Science/Engineering/Math, 1997.

[14] Python, *Neurolab 0.3.5 documentation.* Consultado en
https://pythonhosted.org/neurolab/

[15] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2010.

[16] TensorFlow, Consultado en https://www.tensorflow.org/

[17] B. Widrow y M. E. Hoff, *Adaptative Switching Circuits.* IRE WESCON Convention Record, 1960.