Introducción a los modelos de redes neuronales artificiales
El Perceptrón simple y multicapa

Alba Morera Munt

Trabajo de fin de grado en Matemáticas
Universidad de Zaragoza

Director del trabajo: Tomás Alcalá Nalvaiz
9 de febrero de 2018

Prólogo

En los últimos años se ha consolidado un nuevo campo dentro de las ciencias de la computación que abarcaría un conjunto de metodologías que se caracterizan por su inspiración en los sistemas biológicos para resolver problemas relacionados con el mundo real; reconocimiento de imágenes y de voz, toma de decisiones, predicciones del tiempo atmosférico, etc.

Las **Redes Neuronales Artificiales, RNA,** son las que actualmente están causando un mayor impacto, debido a su gran aplicación práctica, lo que ha llevado a incorporarlas al conjunto de herramientas estadísticas orientadas a la clasificación de patrones y la estimación de variables continuas.

Estas redes son sistemas de procesamiento de la información cuya estructura y funcionamiento están inspirados en las redes neuronales biológicas. Consisten en un conjunto de elementos simples de procesamiento llamados neuronas conectadas entre sí por conexiones que tienen un valor numérico modificable llamado peso. La actividad que una neurona artificial realiza consiste en sumar los valores de las entradas que recibe, comparar esta cantidad con el valor umbral θ y, si lo iguala o supera, enviar una salida activada o en caso contrario, desactivada. Tanto las entradas que la unidad recibe como las salidas que envía dependen a su vez del peso de las conexiones por las cuales se realizan estas operaciones.

La arquitectura de procesamiento de la información de los sistemas de redes neuronales artificiales se distingue de la arquitectura convencional Von Neumann (base de la mayoría de los ordenadores existentes) en una serie de aspectos fundamentales.

En primer lugar, el procesamiento de la información es en paralelo, es decir, muchas unidades de procesamiento pueden estar funcionando simultáneamente; mientras que en la otra arquitectura es secuencial (una tras otra). Tiene la ventaja de que el sistema puede continuar su funcionamiento normal, aunque una pequeña parte del mismo haya resultado dañada. Además, es capaz, en ciertas circunstancias, de reconocer patrones aunque solo se le presente como entrada una parte del mismo.

En segundo lugar, la información que posee un sistema no está localizada o almacenada en compartimentos discretos, sino que en estas redes neuronales está distribuida a lo largo de todos los parámetros del sistema.

Por último, un sistema de red neuronal artificial no se programa para realizar una determinada tarea, sino que es ”entrenado” para ello; realizando dos operaciones. Primero, hay que seleccionar una muestra representativa de pares de entradas y sus correspondientes salidas. Segundo, es necesario un algoritmo o regla para ajustar los pesos de las conexiones entre las unidades en un proceso iterativo de presentación de entradas, observación de salidas y modificación de las conexiones.

Los proyectos de redes neuronales modernas suelen trabajar desde unos miles a unos pocos millones de unidades neuronales y millones de conexiones que, a pesar de ser muchas órdenes, siguen siendo de una magnitud menos compleja que la del cerebro humano.

Summary

Artificial neural networks are mathematical models that try to emulate the processes of human intelligence. Among these processes, learning, reasoning and self-correction stand out. They have been established as one of the main tools of artificial intelligence.

They are inspired by the known behavior of the human brain, mainly that referring to the neurons and their connections, and try to create artificial models that solve problems which are difficult to solve using conventional algorithmic techniques.

The neuron of McCulloch and Pitts is a unit of calculation that attempts to model the behavior of a "natural"neuron, similar to those that make up the human brain. It is the essential unit with which an artificial neural network is constructed.

The task performed by a neuron consists of, given a function f of $\mathbb{R}^n$ in $\{-1,1\}$ and an input pattern $x_1,...,x_n$, calculate an output when the weighted sum of the entries for their weights exceeds a threshold θ.

That is to say,

$$f(w_1x_1+...+w_nx_n)=\begin{cases}1 & \text{si} \quad w_1x_1+...+w_nx_n\geq\theta\\ -1 & \text{si} \quad w_1x_1+...+w_nx_n<\theta\end{cases}$$

This f function, which is called the activation function, makes a partition in the $\mathbb{R}^n$ space of the input patterns, classifying them into two classes; on the one hand there would be the patterns with output +1 and on the other those with output -1.

The objective of these neural networks is to calculate the weights $w_1,...,w_n$, which are the parameters of the model and for this we will see different types of learning algorithms.

In chapter 1, we make an introduction to neural networks. We explain the three types of artificial neurons that exist depending on the layer in which they are found; the input, the hidden and the output neurons.

The type of network that we are going to study are forward networks, although there are also recurrent networks, which contain backward connections but these are beyond the scope of this report because the number of weights increases and, therefore, learning becomes more complicated.

In the learning of a neural network, two phases are distinguished; the learning or training phase and the validation phase. Within the learning phase we will study supervised learning, in which a set of input patterns is presented to the network together with the desired output.

In chapter 2, we study the Perceptron, a concept that was introduced in 1958 by Frank Rosenblantt. It is a simple neuron model based on the McCulloch and Pitts model and a learning rule that consists in the detection and correction of the error that occurs when the estimated output y does not match the desired output z.

One of the most important characteristics of this model is its ability to learn to recognize patterns. It is the simplest model of artificial neural networks since it is constituted by a set of input sensors that receive the input patterns to recognize or classify and an output neuron that deals with classifying in two classes, depending on whether the output is 1 (activated) or 0 (disabled). Use the sign function as an activation function.

As we will see the simple Perceptron is able to implement the logical functions AND and OR but not the XOR, and this is a consequence of the separability theorem that tells us that given a set of linearly separable training patterns, a solution is found in a finite number of iterations, that is, the output of the network agrees with the desired output.

In 1960, another learning model called Adaline emerged, similar to the simple Perceptron, but it uses the identity function as an activation function and in this case the output is continuous. The learning algorithm it uses the Delta rule, which tries to determine the synaptic weights by minimizing the quadratic error function. For this, it uses the gradient descent algorithm, that is, in the same direction but opposite sign to the error function.

In chapter 3, the Multilayer Perceptron, which arises from the inability of the Simple Perceptron to implement the XOR logic function is introduced. Minsky and Papert demonstrated that this problem could be solved by introducing hidden layers between the input layer and the output layer.

In 1986 a new panorama in the field of neural networks with the rediscovery of the backpropagation algorithm was opened. It is an efficient method for training a multilayer Perceptron. The supervised learning rule for the determination of synaptic weights is very similar to that of Adaline, but now we have an input layer composed of N input neurons, a certain number of hidden layers and M neurons in the output layer. In this report we will study the networks that have a single hidden layer formed by L neurons. Now the activation function is a differentiable function and not decreasing. We consider the logistic function and the hyperbolic tangent, since they are simple functions that transform the values of the synaptic potential to the interval [0,1] and [-1,1] respectively.

In chapter 4, we see an application of these models in the R software. First we see that the simple Perceptron is able to implement the logical functions AND and OR and not the XOR, but we see how this problem is solved when we introduce a hidden layer of neurons.

Finally, we apply these algorithms to the practical case of credit scoring, with a German bank dataset. When a bank receives a loan request, according to the profile of the applicant, it has to make a decision on whether to continue with the approval of the loan or not, taking into account the risks involved in each decision. The objective of the analysis is to minimize losses from the perspective of the bank. We perform a first logistic regression model as a reference model, comparing it with a simple model of neural network with a hidden layer. Slightly higher results are obtained. Subsequently, we studied the optimal number of neurons that should be in this hidden layer to obtain a better prediction model. The resulting model is clearly superior to the logistic regression model.

Índice general

Capítulo 1

Introducción a las redes neuronales

1.1. Historia

En 1943, McCulloch y Pitts desarrollaron un modelo de redes neuronales basado en sus conocimientos de neurología, pero estos modelos se limitaron a simulaciones lógicas formales, simulando operaciones binarias $\{0, 1\}$. No fue hasta finales de los años 1950, con el apoyo de los neurocientíficos, cuando estos prometedores modelos empezaron a emerger.

En 1958, el psicólogo Frank Rosenblantt desarrolló un modelo simple de neurona basado en el modelo de McCulloch y Pitts y en una regla de aprendizaje que consiste en la corrección del error. A este modelo le llamó **Perceptrón**. Una de las características que más interés despertó de este modelo fue su capacidad de aprender a reconocer patrones. Está constituido por un conjunto de sensores de entrada que reciben los patrones de entrada a reconocer o clasificar y una neurona de salida que se ocupa de clasificarlos en dos clases, según si la salida es 1 (activada) o 0 (desactivada). Sin embargo, este modelo tenía muchas limitaciones.

En 1960, surgió otro modelo de aprendizaje desarrollado por el profesor Bernard Widrow y su alumno Ted Hoff, en la Universidad de Stanford, llamado Adaline (ADAptative LINear Element). Este particular algoritmo usa mínimos cuadrados para ajustar los pesos de la red.

Observaron que el Perceptrón simple es capaz de resolver problemas de clasificación e implementar funciones lógicas, como por ejemplo, las funciones lógicas AND y OR, pero es incapaz de implementar la XOR. Sobre estas limitaciones, en 1969, Minsky y Papert publicaron un libro titulado "Perceptrons" que supuso el abandono por parte de muchos científicos de la investigación en redes neuronales, ya que no se encontraba un algoritmo de aprendizaje capaz de implementar funciones de cualquier tipo. Las limitaciones de las redes de una sola capa hicieron que se plantease la necesidad de implementar redes en las que se aumentase el número de capas, es decir, introducir capas intermedias o capas ocultas entre la capa de entrada y la de salida de manera que se pudiese implementar cualquier función con el grado de precisión deseado. Querían desarrollar un algoritmo de aprendizaje que permitiera la determinación de los pesos sinápticos y del umbral a partir de un conjunto de patrones de entrenamiento que proporcionan las entradas y salidas de la red.

En 1986 se abrió un nuevo panorama en el campo de las redes neuronales con el redescubrimiento por parte de Rumerlhard, Hinton y Williams del algoritmo de retropropagación. La idea básica fue descubierta en 1974 por Paul Werbos en sus tesis doctoral y algoritmos similares fueron desarrollados independientemente por Bryson y Ho (1969), Parker (1985) y LeCum (1985). El algoritmo de retropropagación del error es un método eficiente para el entrenamiento de un Perceptrón multicapa. Se puede decir que puso fin al pesimismo que sobre el campo de las redes neuronales se había puesto.
Para más detalles históricos, ver [1].

1.2. Funcionamiento general

Supongamos que tenemos una función f de $\mathbb{R}^n$ en $\{-1,1\}$, que aplica un patrón de entrada $\mathbf{x} = (x_1,...,x_n)^T \in \mathbb{R}^n$ en la salida deseada z $\in \{-1,1\}$, es decir, $f(\mathbf{x}) =$z.

$$\begin{aligned} f : \mathbb{R}^n &\longrightarrow \{-1,1\} \\ \mathbf{x} = (x_1,...,x_n)^T &\mapsto f(\mathbf{x}) = \mathrm{z} \end{aligned}$$

La información que disponemos sobre esta función viene dada por p pares de patrones de entrenamiento

$$\{\mathbf{x}^1,\mathrm{z}^1\},\{\mathbf{x}^2,\mathrm{z}^2\},...,\{\mathbf{x}^p,\mathrm{z}^p\}$$

donde $\mathbf{x}^i \in \mathbb{R}^n$ y $f(\mathbf{x}^i) = \mathrm{z}^i \in \{-1,1\}, \quad i = 1,2,...,p$

Esta función f realiza una partición en el espacio $\mathbb{R}^n$ de los patrones de entrada, clasificándolos en dos clases; por una parte estarían los patrones con salida +1 y por otra los de salida -1.

A partir de un conjunto conocido de patrones de entrenamiento, vamos a construir un dispositivo sencillo que aprenda dicha función. Para ello vamos utilizar una **unidad de proceso bipolar**, que es una función matemática con dominio el conjunto n-dimensional $\{-1,1\}^n$ y rango el conjunto $\{-1,1\}$, definida por la siguiente expresión:

$$f(w_1x_1 + ... + w_nx_n) = \begin{cases} 1 & \text{si} \quad w_1x_1 + ... + w_nx_n \geq \theta \\ -1 & \text{si} \quad w_1x_1 + ... + w_nx_n < \theta \end{cases}$$

$$= \begin{cases} 1 & \text{si} \quad w_1x_1 + ... + w_nx_n - \underbrace{\theta}_{w_{n+1}=\theta,\ x_{n+1}=-1} \geq 0 \\ -1 & \text{si} \quad w_1x_1 + ... + w_nx_n - \theta < 0 \end{cases} = sgn(w_1x_1 + ... + w_nx_n - \theta)$$

Análogamente, se puede definir una **unidad de proceso binaria** como una función en el conjunto n-dimensional $\{0,1\}^n$ y rango el conjunto $\{0,1\}$.

Luego la tarea que realiza una neurona consiste en, dado un patrón de entrada , calcula una única salida cuando la suma ponderada de las entradas por sus pesos sobrepasan un umbral θ.

Veamos un esquema:

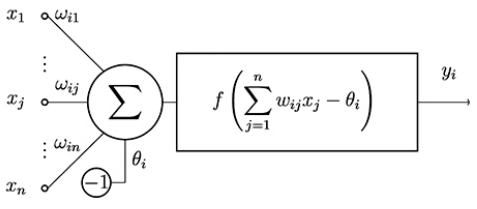

Figura 1.1: Funcionamiento general de una red neuronal. [Fuente: [2]].

donde:

- $x_1,...,x_n,x_{n+1}$ es el patrón de entrada.
  Si consideramos el umbral como el peso sináptico correspondiente a un nuevo sensor de entrada que tiene siempre una entrada igual a $x_{n+1} = -1$ y un peso el valor del umbral $w_{n+1} = \theta$, entonces la red tiene n+1 sensores, su umbral será siempre cero, los patrones de entrada serán ahora $(x_1,...,x_n,-1)$ y los pesos asociados serán $(w_1,...,w_n,\theta)$.

- w_{ij} son los **pesos sinápticos** de la red y son los parámetros del modelo.
  Cada conexión se define por un peso w_{ij} que determina el efecto que la señal de la unidad de entrada j tiene en la unidad de salida i. Estos pesos pueden tomar valores positivos, negativos o cero, indicando en este último caso que no existe conexión entre el par de neuronas.

- $w_1x_1 + ... + w_nx_n$ recibe el nombre de **potencial sináptico**.

- θ es el **umbral** de la red neuronal.

- in_j es la función de entrada y se calcula como la suma de las entradas por sus pesos.

$$in_j = \sum_{j=1}^{n+1} w_{ij}x_j \tag{1.1}$$

- $f(x)$ es la **función de activación** y determina la salida de la red.
  Puede ser de distintos tipos, si utilizamos la unidad de proceso bipolar $\{-1,1\}$ se trata de la **función signo** y si utilizamos la binaria $\{0,1\}$ entonces es la **función umbral.**

- y_i es la salida de la neurona i y se obtiene aplicando la función de activación a la función de entrada.

$$y_i = f(in_j) = f\left(\sum_{j=1}^{n+1} w_{ij}x_j\right) \tag{1.2}$$

Las entradas y salidas de una neurona vamos a considerarlas binarias, es decir, solo admiten dos valores posibles $\{-1,1\}$ o bien $\{0,1\}$. Aunque también podrían ser neuronas continuas, es decir, admiten valores dentro de un determinado rango, que en general suele definirse como [-1,1].

Podemos encontrar tres **tipos de neuronas artificiales** las cuales se agrupan en unidades funcionales llamadas capas:

- **Neuronas de entrada:** reciben la información directamente del exterior y se agrupan en la capa de entrada.

- **Neuronas ocultas:** reciben información de otras neuronas artificiales y cuyas señales de entrada y salida permanecen dentro de la red. Se agrupan en las capas ocultas.

- **Neuronas de salida:** reciben la información procesada y la devuelven al exterior. Forman la capa de salida.

Distinguimos dos **tipos de red** según el patrón de conexiones entre las unidades y la propagación de datos.

- **Redes hacia delante:** donde el flujo de entrada desde las unidades de entrada hasta las de salida son estrictamente hacia delante. El procesamiento de datos puede extenderse a través de múltiples capas de unidades pero no hay conexiones hacia atrás. (Ver figura 1.2).

- **Redes recurrentes:** contienen conexiones hacia atrás. Éstas pueden ser de una neurona con ella misma, entre neuronas de una misma capa y entre neuronas de una capa a una capa anterior. (Ver figura 1.3). Al permitir conexiones recurrentes aumenta el número de pesos, es decir, el número de parámetros ajustables de la red por lo que se complica el aprendizaje y este tipo de redes quedan fueran del alcance de esta memoria. Para más detalle, ver [3].

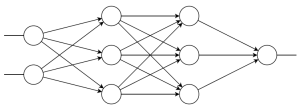

Figura 1.2: Red neuronal hacia delante.[Fuente: [4]]

.

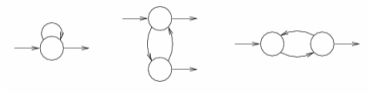

Figura 1.3: Tipos de conexiones de las redes recurrentes. [Fuente: [4]]

1.3. Aprendizaje

En el aprendizaje de una red neuronal [5] distinguimos dos fases:

1. **Fase de aprendizaje o entrenamiento.**

   En esta primera fase, la red es entrenada para realizar un determinado tipo de procesamiento. Partiendo de un conjunto de pesos sinápticos, el proceso de aprendizaje busca un conjunto de pesos que permitan a la red desarrollar correctamente una determinada tarea. Durante este proceso se va refinando iterativamente la solución hasta alcanzar un nivel de ejecución suficientemente bueno.

   A su vez, podemos dividir el proceso de aprendizaje en tres grupos según sus características:

   a) **Aprendizaje supervidado:** se presenta a la red un conjunto de patrones de entrada junto con la salida deseada. Los pesos se van modificando de manera proporcional al error que se produce entre la salida real y la salida deseada.

   b) **Aprendizaje no supervisado:** en este caso no hay información sobre la salida esperada luego se deberán ajustar los pesos en base a la correlación existente entre los datos de entrada.

   c) **Aprendizaje por refuerzo:** se encuentra entre medio de los dos anteriores ya que se le presenta a la red un conjunto de patrones de entrada y se la indica a la red si la salida obtenida es correcta o no.

2. **Fase de validación.**

   Una vez finalizada la fase de aprendizaje, la red puede ser utilizada para realizar la tarea para la que fue entrenada.

Una de las principales ventajas que posee este modelo es que la red aprende la relación existente entre los datos, adquiriendo la capacidad de generalizar conceptos. De esta manera, una red neuronal puede tratar con información que no disponía durante la fase de entrenamiento.

En esta memoria nos vamos a centrar en redes neuronales artificiales con **conexiones hacia delante**. Este tipo de red utiliza algoritmos de **entrenamiento de tipo supersivado** como son el Perceptrón simple, la red Adaline y el Perceptrón multicapa.

Capítulo 2

El Perceptrón simple

El Perceptrón simple fue introducido en 1958 por Frank Rosenblantt. Se trata del modelo más sencillo de redes neuronales artificiales, ya que consta de una sola capa de neuronas con una única salida y. (Ver figura 2.1).

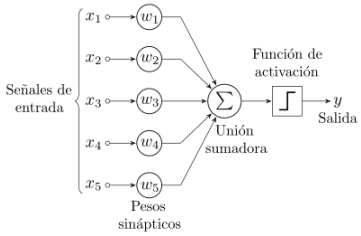

Figura 2.1: Ejemplo de Perceptrón simple con $n = 5$ entradas. [Fuente: [6]]

2.1. Regla de apredizaje

Para la determinación de los pesos sinápticos y del umbral vamos a seguir un proceso adaptativo que consiste en comenzar con unos valores iniciales aleatorios e ir modificándolos iterativamente cuando la salida de la unidad y no coincida con la salida deseada z. Vamos a tener en cuenta que el umbral se puede considerar como el peso correspondiente a un nuevo sensor de entrada que tiene siempre una entrada igual a $x_{n+1} = -1$.

La regla que vamos a seguir para modificar los pesos sinápticos se conoce con el nombre de **regla de aprendizaje del Perceptrón simple** [7] y viene dada por la expresión:

$$\begin{cases} w_j(k+1) = w_j(k) + \Delta w_j(k), \quad k = 1,2,... \\ \text{siendo } \ \Delta w_j(k) = \eta(k)[\ \mathrm{z}(k) - y(k)]x_j(k) \end{cases}$$

Es decir,

$$\boxed{w_j(k+1) = w_j(k) + \eta(k)\ [\mathrm{z}(k) - y(k)]x_j(k)} \tag{2.1}$$

Nos indica que la variación del peso w_j es proporcional al producto del error $\mathrm{z}(k) - y(k)$ por la componente j-ésima del patrón de entrada que hemos introducido en la iteración k, es decir, $x_j(k)$. La constante de proporcionalidad $\eta(k)$ es un parámetro positivo que recibe el nombre de **tasa de aprendizaje**, ya que cuanto mayor es, más se modifica el peso sináptico y viceversa. Es decir, es el parámetro

que controla el proceso de aprendizaje. Cuando es muy pequeño la red aprende poco a poco. Si se toma constante en todas las iteraciones, $\eta(k) = \eta > 0$ tendremos la regla de aprendizaje con incremento fijo.

Cuando usamos la **función signo** como función de transferencia, la regla de aprendizaje se puede escribir de la siguiente forma:

$$w_j(k+1) = \begin{cases} w_j(k) + 2\eta(k)x_j(k) & \text{si} \quad y(k) = -1 \text{ y } z(k) = 1 \\ w_j(k) & \text{si} \quad y(k) = z(k) \\ w_j(k) - 2\eta(k)x_j(k) & \text{si} \quad y(k) = 1 \text{ y } z(k) = -1 \end{cases}$$

Esta regla de aprendizaje es un método de detección del error y corrección. Solo aprende, es decir, modifica los pesos, cuando se equivoca.
Cuando tenemos un patrón que pertenece a la primera clase ($z(k) = 1$) y no es asignado a la misma ($y(k) = -1$), entonces corrige el valor sináptico añadiéndole una cantidad proporcional al valor de la entrada, es decir, lo **refuerza**, mientras que si el patrón de entrada no pertenece a esta clase ($z(k) = -1$) y el Perceptrón lo asigna a ella ($y(k) = 1$), lo que hace es **debilitar** el peso restándole una cantidad proporcional al patrón de entrada. No modificaremos los pesos cuando el valor deseado coincida con la salida de la red ($z(k) = y(k)$).

Vamos a verlo en forma de algoritmo:

2.1.1. Algoritmo de apredizaje

Paso 0: Inicialización.
Inicializar los pesos sinápticos con números aleatorios del intervalo [-1,1].
Ir al paso 1 con k = 1.

Paso 1: (k-ésima iteración).
Calcular $y(k) = \text{sgn}\left(\sum_{j=1}^{n+1} w_j x_j(k)\right)$

Paso 2: Corrección de los pesos sinápticos.
Si $z(k) \neq y(k)$ modificar los pesos sinápticos según la expresión:

$$w_j(k+1) = w_j(k) + \eta(k)\,[z(k) - y(k)]x_j(k), \quad j = 1,...,n,n+1$$

Paso 3: Parada.
Si no se han modificado los pesos en las últimas p iteraciones, es decir,

$$w_j(r) = w_j(k)\,, \quad j = 1,...,n+1, \quad r = k+1,...,k+p$$

parar. La red se ha estabilizado.
En caso contrario, ir al paso 1 con $k = k+1$.

2.1.2. Ejemplo: función lógica OR

Veamos ahora un ejemplo de la regla de aprendizaje del Perceptrón simple sobre la función lógica OR, dada por la siguiente relación:

Entradas	Salidas
(-1,-1)	-1
(-1,1)	1
(1,-1)	1
(1,1)	1

Vamos a considerar estos cuatro casos y tomamos como pesos $w_1 = 1$, $w_2 = 1$ y $\theta = 0{,}5 = w_3$, que define el separador $x_1 + x_2 - 0{,}5 = 0$ (ya que $x_3 = -1$). Tomamos $\eta = 0{,}25 > 0$.

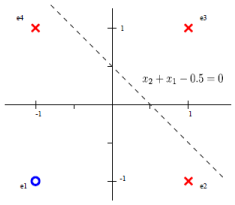

Figura 2.2: Separador $x_1 + x_2 - 0{,}5 = 0$. [Fuente: [8]]

Notar que la cruz hace referencia a la salida deseada z = 1 y el círculo a z = -1, luego se observa que algunos casos no están bien clasificados, vamos a analizarlos:

Caso 1: (-1,-1)

$$y = -1 - 1 - 0{,}5 = -2{,}5 \ < 0 \implies y = -1$$

luego (-1,-1) está bien clasificado ya que $z = y = -1$. (Ver figura 2.2).

Caso 2: (1,-1)

$$y = 1 - 1 - 0{,}5 = -0{,}5 \ < 0 \implies y = -1$$

luego (1,-1) está mal clasificado ya que $z = 1 \neq y = -1$. (Ver figura 2.2).

Aplicamos la regla del Perceptrón (2.1):

$$w_1(1) = w_1(0) + 0{,}25 \cdot [z(0) - y(0)] \cdot x_1(0) = 1 + 0{,}25 \cdot 2 \cdot 1 = 1{,}5$$
$$w_2(1) = w_2(0) + 0{,}25 \cdot [z(0) - y(0)] \cdot x_2(0) = 1 + 0{,}25 \cdot 2 \cdot (-1) = 0{,}5$$
$$\theta(1) = \theta(0) + 0{,}25 \cdot [z(0) - y(0)] \cdot x_3(0) = 0{,}5 + 0{,}25 \cdot 2 \cdot (-1) = 0$$

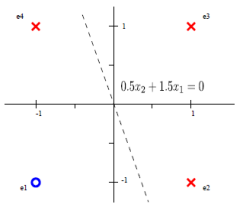

Figura 2.3: Obtenemos el separador $1{,}5\, x_1 + 0{,}5\, x_2 = 0$. [Fuente: [8]]

Ahora el caso 2 ya está bien clasificado. (Ver figura 2.3).

Caso 3: (1,1)

$$y = 1{,}5 \cdot 1 + 0{,}5 \cdot 1 = 2 \ > 0 \implies y = 1$$

luego (1,1) está bien clasificado ya que $z = y = 1$. (Ver figura 2.3).

Caso 4: (-1,1)

$$y = 1{,}5 \cdot (-1) + 0{,}5 \cdot 1 = -1 \ < 0 \implies y = -1$$

luego (-1,1) está mal clasificado ya que $z = 1 \neq y = -1$. (Ver figura 2.3).

Aplicamos la regla del Perceptrón (2.1):

$$w_1(2) = w_1(1) + 0{,}25 \cdot [z(1) - y(1)] \cdot x_1(1) = 1{,}5 + 0{,}25 \cdot 2 \cdot (-1) = 1$$
$$w_2(2) = w_2(1) + 0{,}25 \cdot [z(1) - y(1)] \cdot x_2(1) = 0{,}5 + 0{,}25 \cdot 2 \cdot 1 = 1$$
$$\theta(2) = \theta(1) + 0{,}25 \cdot [z(1) - y(1)] \cdot x_3(1) = 0 + 0{,}25 \cdot 2 \cdot (-1) = -0{,}5$$

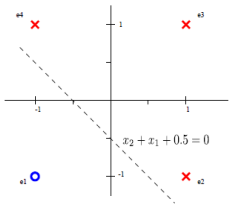

Figura 2.4: Obtenemos el separador $x_1 + x_2 + 0{,}5 = 0$. [Fuente: [8]]

Y obtenemos los cuatro casos bien clasificados. (Ver figura 2.4).

2.2. Convergencia

Vamos a ver ahora que dado un conjunto de patrones de entrenamiento, el Perceptrón simple solamente aprende a clasificarlos correctamente si los patrones son linealmente separables.
Esto reduce considerablemente el campo de aplicaciones ya que ni siquiera es capaz de implementar la función lógica XOR dada por la siquiente relación:

Entradas	Salidas
(1,1)	-1
(1,-1)	1
(-1,1)	1
(-1,-1)	-1

Sin embargo, se pueden clasificar correctamente los patrones cuando se utiliza una función no lineal (ver figura 2.5):

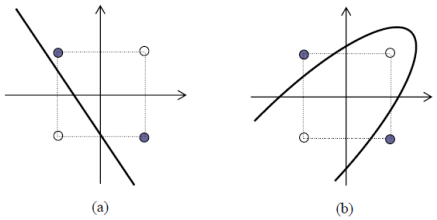

Figura 2.5: (a) Patrones separables linealmente. (b) Patrones no separables linealmente. [Fuente: [7]]

Vamos a estudiar la convergencia del Perceptrón simple, es decir, bajo que condiciones es capaz de encontrar una solución en un número finito de iteraciones.

Definición. Dos conjuntos de puntos A y B son **linealmente separables** en un espacio n-dimensional si existen $n+1$ número reales $w_1,...,w_n,\theta$ de manera que cada punto $(x_1,...,x_n) \in A$ satisface $\sum_{j=1}^{n} w_j x_j \geq \theta$ y cada punto $(x_1,...,x_n) \in B$ satisface $\sum_{j=1}^{n} w_j x_j < \theta$.

Teorema (de convergencia del Perceptrón)
Si el conjunto de patrones de entrenamiento $\{\mathbf{x}^1,\mathrm{z}^1\},\{\mathbf{x}^2,\mathrm{z}^2\},...,\{\mathbf{x}^p,\mathrm{z}^p\}$ es **linealmente separable** entonces el Perceptrón simple encuentra una **solución en un número finito de iteraciones**, es decir, consigue que la salida de la red coincida con la salida deseada para cada uno de los patrones de entrenamiento.

Demostración. Ver apéndice A. □

Valor del parámetro de aprendizaje η

Veamos cuál sería el mejor valor que debemos elegir del parámetro η para que sea más rápida la convergencia de la red. Se trata de elegir η de forma que $D(k+1) = \sum_{j=1}^{n+1}(w_j(k+1) - w_j^*)^2$ sea lo menor posible y así conseguir un mayor acercamiento de los pesos de la red a la solución.

Como $D(k+1)$ es una función cuadrática del parámetro η, solo tenemos que derivar e igualar a cero para encontrar el valor de dicho parámetro que corresponde al mínimo de la expresión.

$$E(\eta) \;=\; D(k+1) \;=\; D(k) \;+\; 4\eta^2 \sum_{j=1}^{n+1} x_j(k)^2 \;-\; 4\eta \Big|\sum_{j=1}^{n+1} w_j(k)x_j(k)\Big| \;-\; 4\eta \Big|\sum_{j=1}^{n+1} w_j^* x_j(k)\Big|$$

$$\frac{\partial E(\eta)}{\partial \eta} \;=\; 8\eta \sum_{j=1}^{n+1} x_j(k)^2 \;-\; 4\Big|\sum_{j=1}^{n+1} w_j(k)x_j(k)\Big| \;-4\,\Big|\sum_{j=1}^{n+1} w_j^* x_j(k)\Big| = 0$$

Y el valor de η que verifica esta ecuación es:

$$\eta_{opt} = \frac{|\sum_{j=1}^{n+1} w_j(k)x_j(k)| \;+\; |\sum_{j=1}^{n+1} w_j^* x_j(k)|}{2(n+1)}$$

Sin embargo, el segundo término del numerador no lo conocemos ya que no conocemos los valores de x_j^*. Si aproximamos este término por el anterior, tenemos un valor aproximado de la tasa de aprendizaje óptima:

$$\tilde{\eta}_{opt} = \frac{|\sum_{j=1}^{n+1} w_j(k)x_j(k)|}{n+1}$$

Y de la igualdad vista en la demostración: $-2|\sum_{j=1}^{n+1} w_j(k)x_j(k)| = [\mathrm{z}(k) - y(k)]\sum_{j=1}^{n+1} w_j(k)x_j(k)$

obtenemos que:

$$\boxed{\tilde{\eta}_{opt} = \frac{-[z(k) - y(k)]\sum_{j=1}^{n+1} w_j(k)x_j(k)}{2(n+1)}}$$

Sustituimos este valor del parámetro η en la regla de aprendizaje (2.1):

$$\boxed{w_j(k+1) = w_j(k) - \frac{[z(k) - y(k)]\sum_{j=1}^{n+1} w_j(k)x_j(k)}{2(n+1)}[\mathrm{z}(k) - y(k)]x_j(k)}$$

Como $[z(k)-y(k)]^2 = 4$ cuando $z(k) \neq y(k)$, ya que que si el error es cero entonces no hay que modificar los pesos, se obtiene así la **regla del Perceptrón normalizada**:

$$\boxed{w_j(k+1) = w_j(k) - 2\frac{\sum_{j=1}^{n+1} w_j(k)x_j(k)}{n+1}x_j(k)} \tag{2.2}$$

Se conoce con este nombre ya que si partimos de un vector de pesos normalizados, es decir, $||w(k)|| = 1$, entonces todos los pesos que van obteniendo según la regla de aprendizaje se mantienen normalizados.

2.3. El Adaline: la regla de aprendizaje Delta

Otro modelo clásico de redes neuronales es el Adaline o neurona con adaptación lineal. Es similar al Perceptrón simple pero utiliza como función de activación la **función identidad**, $f(x) = x$, en lugar de la función signo. La salida de el Adaline es simplemente una función lineal de las entradas ponderadas con los pesos sinápticos y ahora **la salida de la red es continua** en lugar de binaria $\{0,1\}$ y ésta sigue siendo $y = f\left(\sum_{j=1}^{n+1} w_j x_j\right)$.

Vamos a verlo en general para una función de activación f diferenciable y no decreciente y luego lo aplicaremos para el caso concreto del Adaline, con función de activación la función identidad.

La regla de aprendizaje supervisado que vamos a seguir es la **regla Delta**, que trata de determinar los pesos sinápticos de manera que se minimice la función de **error cuadrático** siguiente, que hemos dividido por dos para que se simplifique la derivada:

$$E = \frac{1}{2}\sum_{k=1}^{p}(z(k)-y(k))^2 = \frac{1}{2}\sum_{k=1}^{p}\left(z(k) - f\left(\sum_{j=1}^{n+1} w_j(k)x_j(k)\right)\right)^2 \tag{2.3}$$

Se hace mediante un proceso iterativo donde se van presentando los patrones una a uno y se van modificando los parámetros de la red mediante la **regla del descenso de gradiente** [7], es decir, en la misma dirección pero en sentido opuesto al gradiente.

Consiste en realizar un cambio en cada peso proporcional a la derivada del error respecto del peso:

$$\begin{cases} w_j(k+1) = w_j(k) + \Delta w_j(k), & j = 1,...,n+1 \\ \text{siendo } \Delta w_j(k) = -\eta\frac{\partial E}{\partial w_j(k)} \end{cases}$$

Es decir,

$$\boxed{w_j(k+1) = w_j(k) - \eta\frac{\partial E}{\partial w_j(k)}} \tag{2.4}$$

Veamos esa derivada:

$$\frac{\partial E}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{2}\sum_{k=1}^{p}\left(z(k) - f\left(\sum_{j=1}^{n+1} w_j(k)x_j(k)\right)\right)^2 = \frac{1}{2}\sum_{k=1}^{p}\frac{\partial}{\partial w_j}\left(z(k) - f\left(\sum_{j=1}^{n+1} w_j(k)x_j(k)\right)\right)^2 =$$

$$= \frac{1}{2}\sum_{k=1}^{p} 2\left(z(k) - f\left(\sum_{j=1}^{n+1} w_j(k)x_j(k)\right)\right)\frac{\partial}{\partial w_j}\left(z(k) - f\left(\sum_{j=1}^{n+1} w_j(k)x_j(k)\right)\right) =$$

$$= \sum_{k=1}^{p}\left(z(k) - f\left(\sum_{j=1}^{n+1} w_j(k)x_j(k)\right)\right) f'\left(\sum_{j=1}^{n+1} w_j(k)x_j(k)\right)(-x_j(k)) =$$

$$= -\sum_{k=1}^{p}[z(k) - y(k)]f'\left(\sum_{j=1}^{n+1} w_j(k)x_j(k)\right)x_j(k)$$

$$= -\sum_{k=1}^{p}[z(k) - y(k)]f'(h)x_j(k) \quad \text{con} \quad h = \sum_{j=1}^{n+1} w_j(k)x_j(k)$$

Por tanto, la **regla de aprendizaje general** es la siguiente:

$$\boxed{w_j(k+1) = w_j(k) + \eta[\mathrm{z}(k) - y(k)]f'(h)x_j(k), \quad j = 1,...,n+1} \tag{2.5}$$

Como en este caso f es la función identidad, tenemos la siguiente **regla de aprendizaje para el Adaline**:

$$\boxed{w_j(k+1) = w_j(k) + \eta[\mathrm{z}(k) - y(k)]x_j(k), \quad j = 1,...,n+1} \tag{2.6}$$

El parámetro η controla la longitud del paso que vamos a dar en la dirección opuesta del gradiente. Conforme mayor sea η, mayor será la cantidad por la que se modificarán los pesos sinápticos. Dicho parámetro debe ser un valor pequeño para evitar dar pasos demasiado largos, es decir, que no nos lleven a soluciones peores de las que teníamos, ya que el método del gradiente solamente garantiza el decrecimiento de la función de error si nos desplazamos en la dirección opuesta del gradiente pero en un entorno suficientemente pequeño.

En el proceso de entrenamiento hemos introducido un patrón en cada iteración, por eso diremos que hemos realizado un **aprendizaje en línea**. También podemos introducir lo p patrones directamente y comparar las salidas de la red con las salidas deseadas, pasando entonces a actualizar los pesos sinápticos, en cuyo caso diremos que el **aprendizaje es por lotes**.

La modificación de los pesos se hace tomando como función de error el **error cuadrático medio**, dividiendo por p el error total para interpreterla como error cuadrático medio por patrón, es decir,

$$E = \frac{1}{2p}\sum_{k=1}^{p}(z(k) - y(k))^2 = \frac{1}{2p}\sum_{k=1}^{p}\left(z(k) - f\left(\sum_{j=1}^{n+1} w_{j(k)}x_j(k)\right)\right)^2 \tag{2.7}$$

y así la regla de aprendizaje es:

$$\boxed{w_j(k+1) = w_j(k) + \eta\frac{1}{p}[\mathrm{z}(k) - y(k)]x_j(k)} \tag{2.8}$$

Vamos a verlo en forma de algoritmo:

2.3.1. Algoritmo de aprendizaje de el Adaline por lotes

Paso 0: Inicialización.
Inicializar los pesos de forma aleatoria e introducir los p patrones de entrada.

Paso 1: (k-ésima iteración).
Calcular la salida $y(k) = \sum_{j=1}^{n+1} w_j(k)x_j(k)$, compararla con la salida deseada $\mathrm{z}(k)$ y obtener la diferencia $\mathrm{z}(k) - y(k)$.

Paso 2: Corrección de los pesos sinápticos.
Modificar los pesos sinápticos según la expresión:

$$w_j(k+1) = w_j(k) + \eta \frac{1}{p}[\mathrm{z}(k) - y(k)]x_j(k)$$

Paso 3: Parada.
Si el error cuadrático medio es un valor reducido aceptable, termina el proceso de aprendizaje, sino, se repite otra vez desde el paso 1 con otros patrones de entrenamiento.

Por tanto, la principal diferencia entre el Perceptrón simple y el Adaline es la forma en que se usa la salida de la red en la regla de aprendizaje. El primero solo tiene en cuenta si se ha equivocado o no, mientras que el Adaline considera cuánto se ha equivocado.

Hemos visto que el Perceptrón simple solo puede clasificar correctamente problemas linealmente separables, por lo que es incapaz de implementar la función lógica XOR.

Veamos ahora cómo solucionar este problema.

Minsky y Papert demostraron que para entradas binarias, cualquier transformación puede llevarse a cabo añadiendo una capa que esté conectada a todas las entradas.
Para el problema específico XOR demuestran que al introducir **neuronas ocultas**, extendiendo así la red a un **Perceptrón multicapa**, el problema puede ser resuelto.

A pesar de esta desventaja del Perceptrón simple, este tipo de redes tienen la ventaja de que debido a linealidad del sistema, el algoritmo de aprendizaje convergerá a la solución óptima. Lo que no va a suceder en sistema no lineales como es el caso de las redes multicapa.

Capítulo 3

El Perceptrón multicapa

El Perceptrón multicapa es una red de alimentación hacia delante compuesta por una capa de N neuronas de entrada (sensores), otra capa formada por M neuronas de salida y un número determinado de capas ocultas. (Ver figura 3.1). El tamaño de éstas dependerán de la dificultad de la correspondencia a implementar.

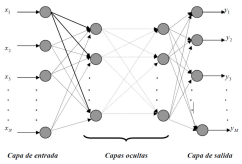

Figura 3.1: Topología de un perceptrón multicapa. [Fuente: [10]].

El objetivo que se busca con este tipo de red es el mismo, establecer una correspondencia entre un conjunto de entrada y un conjunto de salidas deseadas, de manera que:

$$(x_1,...,x_N) \in \mathbb{R}^N \longrightarrow (y_1,...,y_M) \in \mathbb{R}^M$$

Para ello se dispone de un conjunto de p pares de entrenamiento de manera que sabemos perfectamente que al patrón de entrada $(x_1^k,...,x_N^k)$ le corresponde la salida $(y_1^k,...,y_M^k)$, $k = 1,...,p$. Así, nuestro conjunto de entrenamiento es:

$$\{(x_1^k,...,x_N^k) \longrightarrow (y_1^k,...,y_M^k), \quad k = 1,...,p\}$$

Vamos a estudiar el **Perceptrón multicapa con una sola capa oculta**, formada por L neuronas ocultas.

$$y_i = f_1\left(\sum_{j=1}^{L} w_{ij}s_j\right) = f_1\left(\sum_{j=1}^{L} w_{ij}f_2\left(\sum_{r=1}^{N} t_{jr}x_r\right)\right) \tag{3.1}$$

donde:

- w_{ij} es el peso sináptico que conecta la neurona de salida i con la neurona j de la capa oculta.
- f_1 es la función de activación de las unidades de salida.
- t_{jr} es el peso sináptico que conecta la neurona oculta j con la neurona de entrada x_r.
- f_2 es la función de activación de las unidades de la capa oculta.

La función de activación va a ser una función diferenciable y no decreciente:

- Función logística:

$$f(x) = \frac{1}{1 + e^{-2\beta x}}$$

Es una función simple ya que pasa los valores del potencial sináptico al intervalo **[0,1]**.
El parámetro de ganancia β controla la pendiente de la curva, es decir, cuanto mayor es β la curva tiene más pendiente y se aproxima a la **función umbral**.
Se utiliza esta función como función de activación ya que su derivada es muy simple:

$$f^{'}(x) = 2\beta f(x)[1 - f(x)]$$

- Función tangente hiperbólica:

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

Se trata también de una función simple ya que pasa los valores del potencial sináptico al intervalo **[-1,1]**.
Cuanto mayor es el parámetro de ganancia β mayor es la pendiente de la curva y más se asemeja a la **función signo**.
Su derivada es muy sencilla ya que es una función de la propia función.

$$f^{'}(x) = \beta[1 - f(x)^2]$$

3.1. Funcionamiento del algoritmo de retropropagación del error

La regla de aprendizaje supervisado que vamos a seguir para la determinación de los pesos sinápticos es la **regla Delta** que ya hemos visto para el caso del Adaline, pero ahora la función de error cuadrático que se quiere minimizar es la siguiente:

$$E = \frac{1}{2}\sum_{k=1}^{p}\sum_{i=1}^{M}(z_i(k) - y_i(k))^2 \tag{3.2}$$

El algoritmo de retropropagación del error también utiliza la **regla del descenso del gradiente** (2.4). El funcionamiento de este algoritmo es el siguiente:

Dado un patrón de entrada que se aplica a la primera capa de neuronas de la red, se va propagando por las siguientes capas hasta que llega a la capa de salida, donde se compara la salida obtenida con la deseada. A continuación se calcula el error para cada neurona de salida y éste es transmitido hacia atrás, por todas las capas intermedias, y se van modificando sus pesos sinápticos según dicho error y los valores de las salidas de las unidades precedentes ponderadas por sus pesos hasta llegar a la capa de las unidades de entrada.

Modificación de los pesos sinápticos de la capa de salida

Esta modificación de los pesos es la misma que la que hemos realizado de forma general para el Adaline, (2.5), teniendo en cuenta que ahora los pesos son w_{ij} y que $h_i = \sum_{j=1}^{L} w_{ij}(k)s_j(k)$.

Por tanto,

$$\boxed{w_{ij}(k+1) = w_{ij}(k) + \eta[z_i(k) - y_i(k)]f_1^{'}(h_i)s_j(k), \quad j = 1,...,n+1} \tag{3.3}$$

Modificación de los pesos sinápticos de la capa oculta

De forma análoga a la anterior, utilizando la fórmula (2.4) y la derivación en cadena:

$$\Delta t_{jr} = -\eta \frac{\partial E}{\partial t_{jr}(k)} = -\eta \frac{\partial E}{\partial s_j(k)} \frac{\partial s_j(k)}{\partial t_{jr}(k)}$$

y tenemos que:

$$\boxed{t_{jr}(k+1) = t_{jr}(k) + \eta \sum_{i=1}^{M} [z_i(k) - y_i(k)] f_1'(h_i) w_{ij}(k) f_2'(u_j) x_r(k), \quad j = 1,...,n+1, \quad r = 1,...,N} \tag{3.4}$$

siendo $u_j = \sum_{r=1}^{N} t_{jr} x_r$.

El error que acabamos de calcular es el cometido al introducir el k-ésimo patrón, es decir, es un error individualizado porque se realiza para cada patrón, por ellos diremos que la regla es de **entrenamiento individualizado.**

También podemos realizar un **entrenamiento por lotes**, en cuyo caso se introducen los p patrones simultáneamente. En este caso tomamos la función de error total, parecida a (2.7):

$$E = \frac{1}{2} \frac{1}{p} \sum_{k=1}^{p} \sum_{i=1}^{M} (z_i(k) - y_i(k))^2 \tag{3.5}$$

Y así,

$$w_{ij}(k+1) = w_{ij}(k) + \eta \frac{1}{p} [z_i(k) - y_i(k)] f_1'(h_i(k)) s_j(k), \quad i = 1,...,M, \quad j = 1,...,n+1$$

$$t_{jr}(k+1) = t_{jr}(k) + \eta \frac{1}{p} \sum_{k=1}^{p} \sum_{i=1}^{M} [z_i(k) - y_i(k)] f_1'(h_i(k)) w_{ij}(k) f_2'(u_j(k)) x_r(k), \quad r = 1,...,N \tag{3.6}$$

Vamos a verlo en forma de algoritmo:

3.1.1. Algoritmo de retropropagación del error

- **Paso 0: Inicialización**

  Inicializar los pesos de forma aleatoria e introducir los N patrones de entrada.

- **Paso 1: (k-ésima iteración).**

  Calcular la salida

$$y_i(k) = f_1\left(\sum_{j=1}^{L} w_{ij}(k) s_j(k)\right) = f_1\left(\sum_{j=1}^{L} w_{ij}(k) f_2\left(\sum_{r=1}^{N} t_{jr}(k) x_r(k)\right)\right)$$

  compararla con la salida deseada $z_i(k)$ y obtener la diferencia $z_i(k) - y_i(k)$.

- **Paso 2: Corrección de los pesos sinápticos de la capa de salida y oculta según el error.**

  Modificar los pesos sinápticos según las expresiones:

$$w_{ij}(k+1) = w_{ij}(k) + \eta[z_i(k) - y_i(k)]f_1^{'}(h_i)s_j(k)$$

$$t_{jr}(k+1) = t_{jr}(k) + \eta\sum_{i=1}^{M}[z_i(k) - y_i(k)]f_1^{'}(h_i)w_{ij}(k)f_2^{'}(u_j)x_r(k)$$

- **Paso 3: Parada**

  Si el error cuadrático medio es un valor reducido aceptable, termina el proceso de aprendizaje, sino, se repite otra vez desde el paso 1 con otros patrones de entrenamiento.

3.1.2. Aprendizaje con Momentos: la regla Delta generalizada

Acabamos de ver que el algortimo de retropropagación del error se basa en el método del gradiente, con este método nos vamos acercando al mínimo de la función de error cuadrático mediante el descenso por el gradiente. Sin embargo, la función de error suele tener un gran número de mínimos locales y **el algoritmo de aprendizaje puede quedar atrapado en un mínimo local no deseado**, ya que en él, el gradiente vale cero.

Para prevenir esta situación, Hinton y Williams (1986) propusieron que se tuviera en cuenta también el gradiente de la iteración anterior y realizar un promedio con el gradiente de la iteración actual. Este promedio produce un efecto de reducción drástica de las fluctuaciones del gradiente en iteraciones consecutivas. Por tanto, se modifica la regla de retropropagación teniendo en cuenta el descenso seguido en el paso anterior y descendiendo por una dirección intermedia entre la dirección marcada por el gradiente (en sentido opuesto) y la dirección seguida en la iteración anterior, como se expresa en la siguiente ecuación:

$$\Delta w_j(k) = \alpha\Delta w_j(k-1) - \eta\frac{\partial E}{\partial w_j(k)} = \alpha\Delta w_j(k-1) + \eta f^{'}(h_i)[z_i(k) - y_i(k)]s_j(k)$$

Por tanto, el incremento del peso sináptico de la componente j es:

$$\boxed{\Delta w_j(k) = \alpha\Delta w_j(k-1) + \eta f^{'}(h_i)[z_i(k) - y_i(k)]s_j(k)} \tag{3.7}$$

α es la **constante de momentos**, $0 \leq \alpha < 1$, ya que es la que controla el grado de modificación de los pesos teniendo en cuenta la modificación en la etapa anterior. Cuando $\alpha = 0$ tenemos la **regla Delta**, por eso se conoce como regla Delta generalizada.

Por tanto, este término tiene el efecto de mantener la exploración en la misma dirección, lo que permite omitir el mínimo local y realizar pasos más largo para lograr una convergencia más rápida.

Capítulo 4

Aplicaciones prácticas en R

R es un lenguaje y entorno para computación y gráficos estadísticos. Está disponible como software libre y proporciona una amplia variedad de técnicas estadísticas (modelado lineal y no lineal, pruebas estadísticas clásicas, análisis de series de tiempo, clasificación, agrupamiento, ...) y gráficos, y es altamente extensible. Ver [11].

R forma parte de un proyecto colaborativo y abierto. Sus usuarios pueden publicar paquetes que extienden su configuración básica. Existe un repositorio oficial de paquetes [12] que consta actualmente de 2697 paquetes.

Debido a la gran cantidad, se han organizado por temas, que permiten agruparlos según su naturaleza y función.

Para el ajuste de redes neuronales aritificiales vamos a utilizar los siguientes:

- **neuralnet:** se utiliza para el entrenamiento de redes neuronales usando propagación hacia atrás. El paquete permite configuraciones flexibles a través de la elección personalizada del error y de la función de activación. Además, está implementado el cálculo de pesos generalizados.
  Creada por Stefan Fritsch, Frauke Guenther, Marc Suling y Sebastian M. Mueller. Ver [13].
- **nnet:** se utilizada en redes neuronales con conexiones hacia delante y con una sola capa oculta.
  Creada por Brian Ripley y William Venables. Ver [14].

4.1. Implementación de las funciones lógicas AND, OR y XOR

En esta parte práctica vamos a comprobar, como hemos visto en el capítulo 2, que el Perceptrón simple es capaz de implementar las funciones lógicas AND y OR pero es incapaz de implementar la XOR.

Funciones lógicas AND y OR

Construimos la red neuronal con tres neuronas de entrada, sin capas ocultas y con una neurona de salida, y las consideramos binarias $\{0, 1\}$. Utilizamos el paquete neuralnet.
Ver el Script en el Apéndice B.

El conjunto de datos de entrada es el siguiente, donde se encuentran todas las combinaciones de ceros y unos de las variables de entrada, así como sus respectivos resultados AND y OR (ver cuadro 4.1):

Var 1	Var 2	Var3	AND	OR
0	0	0	0	0
1	0	0	0	1
0	1	0	0	1
1	1	0	0	1
0	0	1	0	1
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Cuadro 4.1: Tabla de ejemplos para AND y OR.

Obtenemos la siguiente red neuronal:

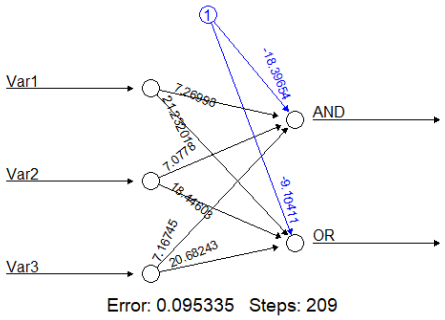

Figura 4.1: Representación gráfica de la red neuronal.

Observamos que se obtiene un error muy pequeño, 0,095335, lo que nos indica que el Perceptrón simple predice correctamente las variables lógicas AND y OR, confirmando la separabilidad lineal de estas funciones.

Función lógica XOR

Comprobemos ahora que el Perceptrón simple no es capaz de implementar la función lógica XOR.

Para ello construimos la red neuronal con dos neuronas de entrada, sin capas ocultas y una neurona de salida. La función de error es la suma de cuadrados y el parámetro de aprendizaje $\eta = 0{,}0001$. El conjunto de datos se muestra en el cuadro 4.2:

Var 1	Var 2	XOR
0	0	0
1	0	1
0	1	1
1	1	0

Cuadro 4.2: Tabla de ejemplos para XOR.

La red neuronal que se obtiene es la siguiente:

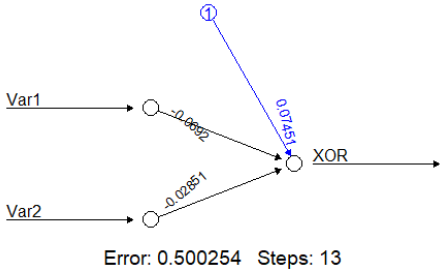

Figura 4.2: Representación gráfica de la red neuronal.

Observamos que en este caso el error es grande, 0,500254, lo que nos indica que el Perceptrón simple no predice correctamente la variable lógica XOR confirmando la no separabilidad lineal de esta función.

Veamos ahora que sucede si **añadimos una capa oculta formada por dos neuronas.**

Obtenemos la siguiente red neuronal:

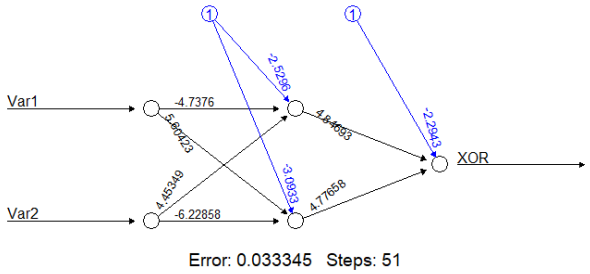

Figura 4.3: Representación gráfica de la red con una capa oculta formada por dos neuronas.

Ahora el error es más pequeño, 0,033345, y por lo tanto, la función lógica XOR con una capa oculta ya es predecible. Acabamos de comprobar, como demostraron Minsky y Papert, que el problema puede ser resuelto introduciendo neuronas ocultas, y extiendo así la red a un Perceptrón multicapa.

4.2. Aplicación a la concesión de créditos

En esta parte vamos a aplicar las redes neuronales para predecir la concesión de créditos.

Cuando un banco recibe una solicitud de préstamo, según el perfil del solicitante, tiene que tomar una decisión sobre si continuar con la aprobación del préstamo o no.

Hay dos tipos de riesgos asociados con la decisión del banco:

- Si el solicitante es un **buen riesgo de crédito**, es decir, es probable que devuelva el préstamo, entonces no aprobar el préstamo a la persona resulta en una pérdida de negocios para el banco.

- Si el solicitante tiene un **riesgo de crédito malo**, es decir, no es probable que pague el préstamo, la aprobación del préstamo a la persona genera una pérdida financiera para el banco.

El objetivo del análisis es minimizar las pérdidas desde la perspectiva del banco. Éste necesita una regla de decisión sobre a quién otorgar la aprobación del préstamo y a quién no. Los gerentes de préstamos consideran los perfiles demográficos y socioeconómicos del solicitante antes de tomar una decisión con respecto a su solicitud de préstamo.

Los datos de crédito alemanes [15] que vamos a utilizar contienen datos sobre 20 variables y la clasificación de si un solicitante se considera un riesgo de crédito bueno o malo para 1000 solicitantes de préstamos. Ver la explicación de las variables en el Apéndice C.
Se espera que un modelo predictivo desarrollado con esta información proporcione al administrador del banco una guía para tomar una decisión sobre si aprobar un préstamo a un posible candidato en función de su perfil.

Por tanto, la variable respuesta, que hemos llamado **"Concesión Credito"** es binaria, siendo 1 cuando el solicitante devuelve el préstamo y 0 en caso contrario.

Vamos a dividir los 1000 datos en dos conjunto de 500 solicitantes cada uno. El primer conjunto es el de entrenamiento de la red (train) y el otro, el conjunto de validación (test).

4.2.1. Modelo regresión logística

El modelo más simple en el contexto de concesión de créditos es el modelo de regresión logística. Ver [5]. Se ha ajustado este modelo a nuestros datos de entrenamiento, obteniendo así la siguiente fórmula:

ConcesionCredito $\sim$ DuracionCreditosMeses + ImporteCredito + Años + SaldoCuenta + EstadoCreditosAnteriores + Proposito + Ahorros + DuracionTrabajoActualAños + PorcentajeIngresosDisponibles + EstadoCivilySexo + Garante + AñosDomicilioActual + MayorActivoValioso + TipoApartamento + NumeroCreditosAnteriores + Ocupacion + NumeroPersonasQueMantiene + Telefono + TrabajadorExtranjero + OtrosCreditos

Ver Script en el Apéndice D.

Usando el **criterio de información de Akaike, AIC,** obtenemos que las variables más significativas para predecir si se debe conceder el crédito o no son las 9 siguientes:

ConcesionCredito $\sim$ DuracionCreditosMeses + ImporteCredito + SaldoCuenta + EstadoCreditosAnteriores + Ahorros + PorcentajeIngresosDisponibles + Garante + TipoApartamento + OtrosCreditos

Y con este modelo reducido de variables obtenemos un AIC = 472,55, que es menor que el obtenido con todas las variables.

Una curva ROC es una representación gráfica para un sistema clasificador binario de la razón o ratio de verdaderos positivos (VPR = Razón de Verdaderos Positivos) frente a la razón o ratio de falsos positivos (FPR = Razón de Falsos Positivos) también según se varía el umbral (valor a partir del cual decidimos que un caso es un positivo).
Vamos a utilizar el paquete ROC. [16].

Como medida resumen de la capacidad predictiva de un clasificador binario usamos el área bajo la curva ROC (AUC) y la matriz de confusión [5]. En ambos casos se usa este resumen sobre el conjunto de validación (test).

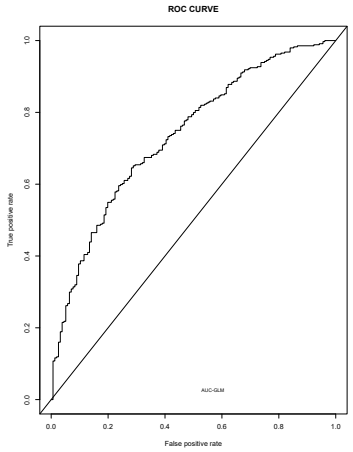

Figura 4.4: Curva ROC para el modelo de regresión logísitca.

Obtenemos un **área bajo la curva, AUC,** de 0,7334899.

La matriz de confusión que obtenemos es la siguiente, donde cada columna de la matriz representa el número de predicciones de cada clase y cada fila representa a los ejemplos en la clase real:

Real \ Predicho	0	1
0	70	86
1	44	300

Cuadro 4.3: Matriz de confusión del modelo de regresión logística.

Esta matriz nos dice que hay:

- 70 Verdaderos Negativos, **VN**, es decir, número de créditos que no se concedieron y que han sido clasificados correctamente por el modelo, ya que no se les ha concedido.
- 86 Falsos Positivos, **FP**, número de créditos que no se concedieron y que han sido clasificados incorrectamente por el modelo ya que se les ha concedido el crédito.
- 44 Falsos Negativos, **FN**, número de créditos que si se concedieron y que han sido clasificados incorrectamente por el modelo ya que no se les ha concedido el crédito.
- 300 Verdaderos Positivos, **VP**, número de créditos que si se concedieron y que han sido clasificados correctamente por el modelo.

Y el porcentaje de **correctos clasificados**, que se obtiene sumando la diagonal y dividiendo para los 500 solicitantes, $\frac{FN+VP}{500}100 = 74\,\%$.

Hemos tomado un umbral $\theta = 0,5$, es decir, las probabilidades que son mayores que este valor las clasifica como 1, mientras que son 0 si no superan ese umbral.

4.2.2. Modelo Perceptrón multicapa

Ahora vamos a utilizar el modelo del Perceptrón multicapa con una capa oculta para mejorar los resultados obtenidos. Lo aplicamos al modelo mejorado de las 9 variables.

Primero introducimos **dos neuronas en la capa oculta.**

Observamos en el siguiente gráfico que la curva ROC y el área bajo la curva, 0,7529815, mejoran ligeramente.

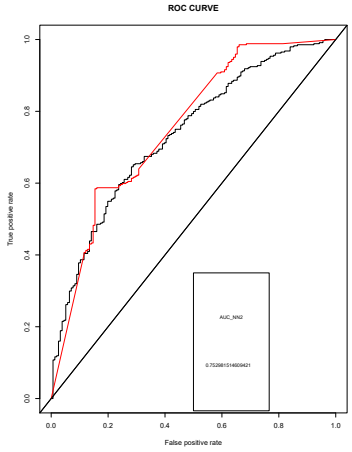

Figura 4.5: Curva ROC y AUC del Perceptrón multicapa con una capa oculta de dos neuronas (en rojo); en negro el modelo de regresión logística.

Obtenemos la siguiente matriz de confusión:

Real \ Predicho	0	1
0	54	102
1	9	335

Cuadro 4.4: Matriz de confusión del modelo Perceptrón multicapa con una capa oculta de dos neuronas.

Y ahora el porcentaje de correctos clasificados ha aumentando de 74 % a 77,8 %.

Veamos una representación gráfica de esta red; formada por 22 patrones de entrada (las variables han sido convertidas a tipo factor), 2 neuronas en la capa oculta y una salida, que es 0 ó 1, dependiendo de si se concede el crédito o no.
Se representan también todos los pesos sinápticos, y éstos son mayores cuanto más gruesa es la línea que une las neuronas de las distintas capas.

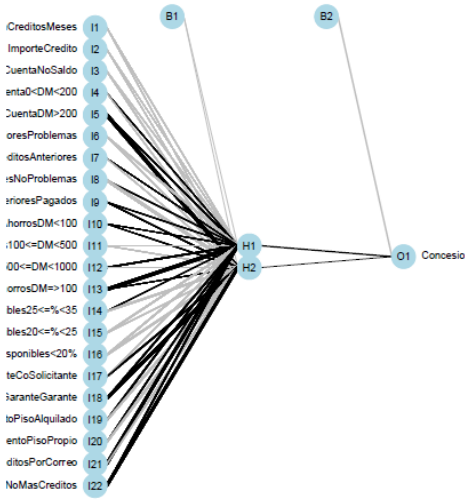

Figura 4.6: Topología de la red del Perceptrón multicapa con 2 neuronas en la capa oculta.

Realizamos ahora un estudio de simulación para ver el número de neuronas que habría que introducir en la capa oculta para que mejore el modelo.
Ver Script en el Apéndice D.

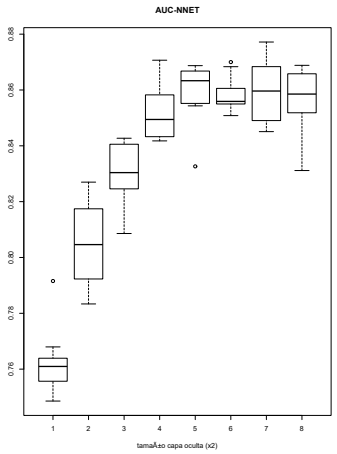

Figura 4.7: Gráfica de los AUC de la red neuronal.

Para elegir el que mayor AUC tiene vamos a considerar la mediana de los diagramas de caja obtenidos y podemos observar que el que tiene mayor mediana corresponde al tamaño $5 \cdot 2 = 10$, luego el tamaño óptimo es introducir 10 neuronas en la capa oculta.

Veamos por último como mejoran los datos si consideramos el Perceptrón multicapa con **10 neuronas en la capa oculta.** Con este modelo tenemos un AUC = 0,8533281.

La matriz de confusión que se obtiene es:

Real \ Predicho	0	1
0	122	34
1	22	322

Cuadro 4.5: Matriz de confusión del modelo con 10 neuronas en la capa oculta.

El porcentaje de correctos clasificados ha aumentado a 88,8 %, y podemos concluir que este modelo es el que mejor predice los resultados de concesión de créditos.

Veamos ahora la topología de la red con las 10 neuronas en la capa oculta:

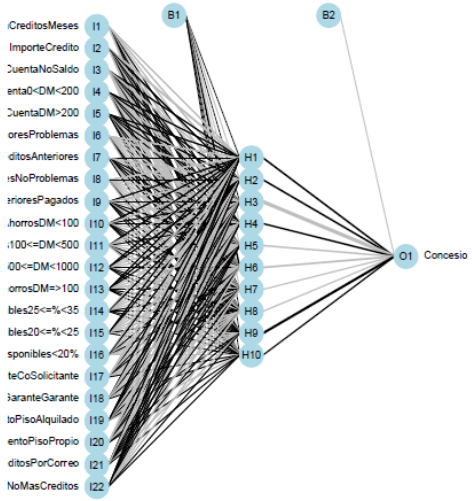

Figura 4.8: Topología de la red del Perceptrón multicapa con 10 neuronas en la capa oculta.

Anexos

Apéndice A

Demostración Teorema convergencia del Perceptrón simple

Demostración.

Como los patrones son linealmente separables existirán unos valores $w_1, ..., w_n, \theta^*$ tal que

$$\sum_{j=1}^{n} w_j^* x_j > \theta^* \text{ para los patrones de la clase 1}$$

$$\underbrace{\sum_{j=1}^{n} w_j^* x_j}_{\textit{salida deseada}} < \theta^* \text{ para los patrones de la clase 0}$$

Supongamos que en la iteración k la red tiene que modificar los pesos sinápticos según la regla de aprendizaje, ya que la salida de la red $y(k)$ no coincide con la salida deseada $\mathrm{z}(k)$, es decir, $y(k) \neq \mathrm{z}(k)$.

Tendremos que:

$\sum_{j=1}^{n+1}(w_j(k+1) - w_j^*)^2 = \sum_{j=1}^{n+1}(w_j(k) + \eta[\mathrm{z}(k) - y(k)]x_j(k) - w_j^*)^2 =$

$= \sum_{j=1}^{n+1}(w_j(k) - w_j^* + \eta[\mathrm{z}(k) - y(k)]x_j(k))^2 =$

$= \sum_{j=1}^{n+1}(w_j(k) - w_j^*)^2 + \eta^2[\mathrm{z}(k) - y(k)]^2 \sum_{j=1}^{n+1} x_j(k)^2 + 2\eta[\mathrm{z}(k) - y(k)] \sum_{j=1}^{n+1}(w_j(k) - w_j^*)x_j(k) =$

$= \sum_{j=1}^{n+1}(w_j(k) - w_j^*)^2 + \eta^2[\mathrm{z}(k) - y(k)]^2 \sum_{j=1}^{n+1} x_j(k)^2 + 2\eta[\mathrm{z}(k) - y(k)](\sum_{j=1}^{n+1}(w_j(k)x_j(k)) -$

$2\eta[\mathrm{z}(k) - y(k)](\sum_{j=1}^{n+1}(w_j^* x_j(k))$

Observamos que $[\mathrm{z}(k) - y(k)] \sum_{j=1}^{n+1} w_j(k)x_j(k) < 0$,

ya que si $\sum_{j=1}^{n+1} w_j(k)x_j(k) > 0 \implies y(k) = 1$ y como la salida es incorrecta $(y(k) \neq \mathrm{z}(k))$ tiene que ser $\mathrm{z}(k) = -1$.

y si $\sum_{j=1}^{n+1} w_j(k)x_j(k) < 0 \implies y(k) = -1$ y como la salida es incorrecta $(y(k) \neq \mathrm{z}(k))$ tiene que ser $\mathrm{z}(k) = 1$.

Este término se puede escribir de la forma $-2|\sum_{j=1}^{n+1} w_j(k)x_j(k)|$.

Y el término $[\mathrm{z}(k)-y(k)]\sum_{j=1}^{n+1} w_j^* x_j(k) > 0$,

ya que si $\sum_{j=1}^{n+1} w_j^* x_j(k) > 0 \implies$ la salida deseada $\mathrm{z}(k) = 1$ y la salida incorrecta de la red tiene que ser $y(k) = -1$.

y si $\sum_{j=1}^{n+1} w_j^* x_j(k) < 0 \implies \mathrm{z}(k) = -1$ y la salida incorrecta de la red tiene que ser $y(k) = 1$.

Este término se puede escribir de la forma $2|\sum_{j=1}^{n+1} w_j^* x_j(k)|$.

Por lo tanto, prescindiendo de un término negativo a la derecha de la expresión, tenemos que:

$$\sum_{j=1}^{n+1}(w_j(k+1)-w_j^*)^2 \leq \sum_{j=1}^{n+1}(w_j(k)-w_j^*)^2 + 4\eta^2\sum_{j=1}^{n+1} x_j(k)^2 - 4\eta|\sum_{j=1}^{n+1} w_j(k)x_j(k)|$$

Sea

$$\begin{cases} D(k+1) = \sum_{j=1}^{n+1}(w_j(k+1)-w_j^*)^2 \\ D(k) = \sum_{j=1}^{n+1}(w_j(k)-w_j^*)^2 \\ T = \mín_{1\leq k\leq p}\{|\sum_{j=1}^{n+1} w_j^* x_j(k)|\} \end{cases}$$

Como $\sum_{j=1}^{n+1} x_j(k)^2 = ||\mathbf{x}(k)||^2 = n+1$, se tiene la siguiente desigualdad

$$D(k+1) \leq D(k) + 4\eta^2(n+1) - 4\eta T$$

$$D(k+1) \leq D(k) + 4\eta[\eta(n+1) - T]$$

Si tomamos $\eta(n+1) - T < 0$, es decir, $\eta < \frac{T}{n+1}$, entonces$D(k+1) \leq D(k)$.

Esto significa que eligiendo un valor de η tal que $0 < \eta < \frac{T}{n+1}$ hace que $D(k)$ disminuya al menos en la cantidad constante $4\eta[\eta(n+1) - T]$ en cada iteración, con corrección.

Si el número de iteraciones con corrección fuese infinito entonces llegaríamos al absurdo de alcanzar en un momento determinado un valor negativo para el término $D(k)$ que evidentemente no puede ser negativo.

□

Apéndice B

Script R funciones lógicas

```
library(neuralnet)
```

Funciones lógicas AND y OR

```
AND <- c(rep(0,7),1)
OR <- c(0,rep(1,7))

binary.data <- data.frame(expand.grid(c(0,1), c(0,1), c(0,1)), AND, OR)
print(net <- neuralnet(AND+OR~Var1+Var2+Var3,  binary.data, hidden=0,
                       rep=1, err.fct="ce", linear.output=FALSE))
plot(net, rep="best")
compute(net, binary.data[,1:3])
```

Función lógica XOR sin capa oculta

```
XOR <- c(0,1,1,0)

xor.data <- data.frame(expand.grid(c(0,1), c(0,1)), XOR)
print(net.xor <- neuralnet(XOR~Var1+Var2, xor.data,
                           hidden=0, rep=5,
                           err.fct="sse",
                           learningrate=0.0001,
                           linear.output=FALSE))
compute(net.xor, binary.data[,1:2])
plot(net.xor, rep="best")
```

Añadimos a XOR una capa oculta formada por dos neuronas

```
XOR <- c(0,1,1,0)

xor.data <- data.frame(expand.grid(c(0,1), c(0,1)), XOR)
print(net.xor <- neuralnet(XOR~Var1+Var2, xor.data,
                           hidden=2, rep=5,
                           err.fct="sse",
                           learningrate=0.0001,
                           linear.output=FALSE))
compute(net.xor, binary.data[,1:2])
plot(net.xor, rep="best")
```

Apéndice C

Conjunto de datos de créditos alemanes

Vamos a explicar las 21 variables de este conjunto de datos. [15].

La unidad monetaria con la que trabajamos es el marco alemán, DM, que fue la moneda oficial de Alemania Occidental (1948-1990) y de Alemania (1990-2002) hasta la adopción del euro en 2002.
El Banco Central Europeo fijó el tipo de cambio irrevocable del marco alemán, a partir del 1 de enero de 1999, en 1,95583 DM = 1 euro.

Variables numéricas:

1. **Duración del créditos en meses**
2. **Importe del crédito**
3. **Años del solicitante**

Variables categóricas:

4. **Saldo de la cuenta**
   - No tiene cuenta corriente
   - No tiene saldo
   - $0 \leq$ DM $<$ 200
   - DM $\geq$ 200
5. **Estado de créditos anteriores**
   - Hay dudas
   - Problemas
   - No tiene créditos anteriores
   - No hay problemas
   - Pagados
6. **Propósito del crédito**
   - Otro
   - Coche nuevo
   - Coche usado

- Muebles
- TV
- Electrodomésticos
- Reparación
- Educación
- Vacaciones
- Entrenamiento
- Negocios

7. **Ahorros del solicitante**
   - No disponibles
   - DM $<$ 100
   - 100 $\leq$ DM $<$ 500
   - 500 $\leq$ DM $<$ 1000
   - DM $\geq$ 1000
8. **Duración del empleo actual en años**
   - Sin trabajo
   - $<$ 1 año
   - 1 $\leq$ años $<$ 4
   - 4 $\leq$ años $<$ 7
   - años $\geq$ 7
9. **Porcentaje de los ingresos disponibles**
   - $\geq 35\,\%$
   - $25 \leq\ \% < 35$
   - $20 \leq\ \% < 25$
   - $< 20\,\%$
10. **Estado civil y sexo**
    - Hombre divorciado o viviendo solo
    - Hombre soltero
    - Hombre casado o viudo
    - Mujer
11. **Garante**
    - Nadie
    - Co-solicitante
    - Garante
12. **Años en el domicilio actual**
    - $<$ 1 año
    - 1 $\leq$ años $<$ 4

- $4 \leq$ años < 7
- ≥ 7 años

13. **Mayor activo valioso**
    - No disponible
    - Coche
    - Seguro de vida
    - Propiedad
14. **Tipo de apartamento**
    - Gratis
    - Alquilado
    - Propio
15. **Número de créditos anteriores en este banco**
    - 1
    - 2 ó 3
    - 4 ó 5
    - 6 o más
16. **Ocupación**
    - Trabajador no cualificado sin permanencia
    - Trabajador no cualificado con permanencia
    - Trabajador cualificado
    - Funcionario superior
17. **Número de personas que mantiene**
    - 3 o más
    - de 0 a 2
18. **Telefono**
    - No
    - Si
19. **Trabajador extranjero**
    - Si
    - No
20. **Concesión del crédito**
    - No
    - Si
21. **Otros créditos**
    - En otros bancos
    - Por correo
    - No más créditos

Apéndice D

Script R concesión de créditos

```
# Preparamos los datos

vnumer<-c(1,2,3)       # indices de las variable numéricas
vresp<-c(20)           # índice de la variable respuesta Concesión Crédito


German[,vresp]<-as.numeric(German[,vresp])-1   # conversión a variable 0-1 de la respuesta
German[,vresp]<-as.factor(German[,vresp])      # conversión de la respuesta a factor
German[,vnumer]<- scale(German[,vnumer])       # tipificación de las variables numéricas


# Separamos los datos en conjunto de entrenamiento (train) y de validación (test)

d = sort(sample(nrow(German), nrow(German)*.5))   # theta = 0.5
train<-German[d,]
test<-German[-d,]

# Creamos de la fórmula

n <- names(data.frame(German[,-vresp]))
f <- as.formula(paste("ConcesionCredito ~", paste(n[!n %in% "German"], collapse = " + ")))
```

Modelo regresión logística con los datos train

```
mreglog<-glm(f,data=train,family=binomial())

summary(mreglog)
```

Quitamos las variables que no son significativas, usando criterio AIC

```
stpreglog<-step(mreglog)


# Curva ROC, se aplica a los datos test

library(ROCR)
```

```
test$score<-predict(stpreglog,type='response',test)
pred_test<-prediction(test$score,test$ConcesionCredito)
perf_test <- performance(pred_test,"tpr","fpr")
plot(perf_test, main="ROC CURVE")
abline(a=0.0,b=1.0)

# AUC(área bajo la curva)

auc_glm<-performance(pred_test,"auc")
auc_glm<-unlist(slot(auc_glm,"y.values"))
legend(.5,.15,round(auc_glm,5),title="AUC-GLM", cex=0.7, bty="n")
auc_glm


# Matriz de confusión

CM<-table(test[,vresp],predict(stpreglog, data=test, type="response") > 0.5)
CM
sum(diag(CM))/sum(CM)
```

Modelo Perceptrón multicapa con dos neuronas en la capa oculta

```
library(nnet)

fstep<-formula(stpreglog)   # comparamos sobre el mismo conjunto de variables que el GLM


# Ajuste de la NN con 2 neuronas en capa oculta

nnG2<-nnet(fstep, data=train,
           size=2,
           decay=1e-4,
           maxit=5000)


# Curva ROC y AUC sobre test

test$score<-predict(nnG2,type='raw',data=test)
pred_test<-prediction(test$score,test$ConcesionCredito)
perf_test <- performance(pred_test,"tpr","fpr")
plot(perf_test, main="ROC CURVE", add=TRUE, col=2, bty="n")
abline(a=0.0,b=1.0)

# AUC

auc_nn2<-performance(pred_test,"auc")
auc_nn2<-unlist(slot(auc_nn2,"y.values"))
legend(.5,.35,auc_nn2,title="AUC_NN2", cex=0.7)
auc_nn2
```

```
# Matriz de confusión

CM<-table(test[,vresp],predict(nnG2, data=test, type="class"))
CM
sum(diag(CM))/sum(CM)

# Representación Gráfica

library(devtools)

source_url('https://gist.githubusercontent.com/fawda123/7471137/raw/
466c1474d0a505ff044412703516c34f1a4684a5/nnet_plot_update.r')

plot.nnet(nnG2)
```

Hacemos una simulación para ver cual es el número óptimo de neuronas en la capa oculta

```
# vector para guardar los auc de una nnet de tamaño fijo
auc_mat<-matrix(0, nrow=10, ncol=8)

for( inn in 1:8){

ncv<-10
auc_cv<-rep(0,ncv)
for (i in 1:ncv){
  nnG2<-nnet(fstep, data=train,
             size=2*inn,      # tamaño 2*inn
             decay=1e-4,
             maxit=5000,
             verbose=FALSE)
  test$score<-predict(nnG2,type='raw',data=test)
  pred_test<-prediction(test$score,test$ConcesionCredito)
  perf_test <- performance(pred_test,"tpr","fpr")
  auc_nn2<-performance(pred_test,"auc")
  auc_cv[i]<-unlist(slot(auc_nn2,"y.values"))
}

auc_mat[,inn]<-auc_cv
print(inn)
}

boxplot(auc_mat)
title(main="AUC-NNET", xlab="tamaño capa oculta (x2)")
```

Modelo Perceptrón multicapa con 10 neuronas en la capa oculta

```
nnG10<-nnet(fstep, data=train,
             size=10,  #tamaño optimizado
             decay=1e-4,
```

```
              maxit=5000,
              verbose=FALSE)

# Curva ROC y AUC

test$score<-predict(nnG10,type='raw',data=test)
pred_test<-prediction(test$score,test$ConcesionCredito)
perf_test <- performance(pred_test,"tpr","fpr")
auc_nn10<-performance(pred_test,"auc")
auc_nn10<-unlist(slot(auc_nn10,"y.values"))
plot(perf_test, main="ROC CURVE", add=TRUE, col=3)
legend(.5,.55,round(auc_nn10,5),title="AUC_NN10", cex=0.7, bty="n")
auc_nn10

# Matriz de confusión

CM<-table(test[,vresp],predict(nnG10, data=test, type="class"))  ++
CM
sum(diag(CM))/sum(CM)


# Representación Gráfica

library(devtools)

source_url('https://gist.githubusercontent.com/fawda123/7471137/raw/
466c1474d0a505ff044412703516c34f1a4684a5/nnet_plot_update.r')

plot.nnet(nnG10)
```

Bibliografía

[1] M. TIM JONES, *Artificial Intelligence, A Systems Approach*, 2008.

[2] *Principales elementos de una neurona artificial*, https://www.google.es/search?q=principales+elementos+de+una+neurona+artificial&source=lnms&tbm=isch&sa=X&ved=0ahUKEwjFouygu5jZAhWK16QKHcZcDjcQ_AUICigB&biw=1242&bih=602#imgrc=dxYy85dqA53QEM:.

[3] BEN KRÖSE Y PATRICK VAN DER SMAGT, *An introduction to Neural Networks*, Eighth edition, 1996.

[4] *Creación de una estructura de red neuronal artificial Perceptrón multicapa*, https://censorcosmico.blogspot.com.es.

[5] T. HASTIE, ROBERT TIBSHIRAMI Y J. FRIEDMAN, *The Elements of Statistical Learning*, Spinger.

[6] *Perceptrón*, https://es.wikipedia.org/wiki/Perceptr%C3%B3n, disponible en https://es.wikipedia.org.

[7] J. MUÑOZ PÉREZ, *El Perceptrón Simple*, http://www.lcc.uma.es/~munozp/documentos/modelos_computacionales/temas/Tema4MC-05.pdf, disponible en http://www.lcc.uma.es.

[8] JAVIER BÉJAR, *Artificial Neural Networks*, http://www.lsi.upc.edu/~bejar/apren/docum/trans/05-ANN.pdf, disponible en http://www.lsi.upc.edu.

[9] J. MUÑOZ PÉREZ, *Redes Neurononales Multicapa*, http://www.lcc.uma.es/~munozp/documentos/modelos_computacionales/temas/Tema5MC-05.pdf, disponible en http://www.lcc.uma.es.

[10] *Inteligencia Artifical*, http://inteligenciaartificialespammfl.blogspot.com.es/2015/06/perceptron-multicapa.html.

[11] *The R Project for Statistical Computing*, https://www.r-project.org/.

[12] *Repositorio oficial de paquetes de R*, https://web.archive.org/web/20101221001753/http://cran.r-project.org/web/packages/. disponible en https://web.archive.org/.

[13] *Library neuralnet*, https://cran.r-project.org/web/packages/neuralnet/index.html.

[14] *Library nnet*, https://cran.r-project.org/web/packages/nnet/index.html.

[15] *Analysis of German dredit data*, https://onlinecourses.science.psu.edu/stat857/node/215.

[16] *Library ROC*, https://cran.r-project.org/web/packages/pROC/pROC.pdf.