Las redes neuronales artificiales

Jordi Casas Roma

1 crédito
xxx/xxxxx/xxx

Índice

Introducción 5

Objetivos 6

1. Principios y fundamentos 7
- 1.1. Las neuronas 7
- 1.2. Arquitectura de una red neuronal 13
- 1.3. Entrenamiento de una red neuronal 16
- 1.4. Ejemplo de aplicación 22
- 1.5. El problema de la desaparición del gradiente 24

2. El algoritmo de Retropropagación (*Backpropagation*) 26
- 2.1. Caso particular con un único ejemplo 27
- 2.2. Caso general con varios ejemplos 33

3. Optimización del proceso de aprendizaje 36
- 3.1. Técnicas relacionadas con el rendimiento de la red 37
- 3.2. Técnicas relacionadas con la velocidad del proceso de aprendizaje 42
- 3.3. Técnicas relacionadas con el sobreentrenamiento 45
- 3.4. Relación de técnicas y rendimiento de la red 49

4. Autoencoders 50
- 4.1. Estructura básica 50
- 4.2. Entrenamiento de un *autoencoder* 52
- 4.3. Pre-entrenamiento utilizando *autoencoders* 53
- 4.4. Tipos de *autoencoders* 55

Resumen 57

Glosario 58

Bibliografía 59

Introducción

En este módulo didáctico se describen los conceptos introductorios de las redes neuronales. Iniciaremos este módulo con un repaso a los conceptos fundamentales, tales como la estructura de una neurona, las principales funciones de activación, etc. A continuación, presentaremos las técnicas de entrenamiento de redes neuronales, y finalizaremos este primer bloque comentando el problema de la desaparición y explosión del gradiente.

El segundo capítulo de este texto está dedicado al algoritmo de Retropropagación (*Backpropagation*, en inglés), donde profundizaremos en su detalle matemático con la finalidad de comprender su funcionamiento a bajo nivel.

A continuación, en el tercer capítulo de este módulo didáctico, veremos un conjunto de técnicas diseñadas para mejorar el rendimiento de las redes neuronales, aumentar la velocidad del proceso de aprendizaje de las redes durante el entrenamiento, e intentar evitar el problema del sobreentrenamiento o sobreajuste, que dificulta la capacidad de generalización de las redes ante nuevos datos.

Finalizaremos este módulo didáctico viendo un tipo particular de redes neuronales, los *autoencoders*. En concreto, nos centraremos en sus particularidades, aplicaciones y diferencias respecto a las redes vistas hasta ese punto, así como las peculiaridades de su proceso de funcionamiento y entrenamiento.

Objetivos

En los materiales didácticos de este módulo encontraremos las herramientas indispensables para asimilar los siguientes objetivos:

1. Comprender los fundamentos y principios de funcionamiento de las redes neuronales artificiales.
2. Conocer las principales arquitecturas de redes neuronales.
3. Comprender el funcionamiento del algoritmo de Retropropagación (en inglés, *Backpropagation*).
4. Conocer las principales técnicas de optimización del aprendizaje.
5. Conocer la estructura básica y el funcionamiento de los autoencoders.

1. Principios y fundamentos

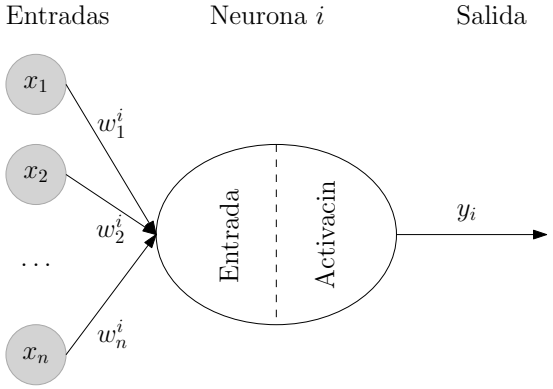
La redes neuronales artificiales permiten realizar tareas de clasificación en juegos de datos etiquetados y tareas de regresión en juegos datos continuos, aunque también permiten realizar tareas de segmentación en juegos de datos no etiquetados a partir del establecimiento de similitudes entre los datos de entrada.

En general, las redes neuronales relacionan entradas con salidas, es decir, el juego de datos inicial con la salida o resultado del algoritmo aplicado sobre los mismos. Algunos autores se refieren a las redes neuronales como "aproximadores universales", porque son capaces de aprender y aproximar con precisión la función $f(x) = y$ donde x se refiere a los datos de entrada y $f(x)$ se refiere al resultado del algoritmo.

1.1. Las neuronas

Una red neuronal artificial consiste en la interconexión de un conjunto de unidades elementales llamadas **neuronas**. Cada una de las neuronas de la red aplica una función determinada a los valores de sus entradas procedentes de las conexiones con otras neuronas, y así se obtiene un valor nuevo que se convierte en la salida de la neurona.

Figura 1. Esquema básico de una neurona artificial



La Figura 1 presenta el esquema básico de una neurona artificial. Cada neurona tiene un conjunto de entradas, denotadas como $X = \{x_1, x_2, \ldots, x_n\}$. Cada una de estas entradas está ponderada por el conjunto de valores $W^i = \{w_1^i, w_2^i, \ldots, w_n^i\}$. Debemos interpretar el valor w_j^i como el peso o la importancia del valor de entrada x_j que llega a la neurona i procedente de la neurona j.

Cada neurona combina los valores de entrada, aplicando sobre ellos una **función de entrada** o **combinación**. El valor resultante es procesado por una **función de**

activación, que modula el valor de las entradas para generar el valor de salida y_i. Este valor, generalmente, se propaga a las conexiones de la neurona i con otras neuronas o bien es empleado como valor de salida de la red.

1.1.1. Función de entrada o combinación

Como hemos visto, cada conexión de entrada x_j tiene un peso determinado w_j^i que refleja su importancia o estado. El objetivo de la función de entrada es combinar las distintas entradas con su peso y agregar los valores obtenidos de todas las conexiones de entrada para obtener un único valor.

Para un conjunto de n conexiones de entrada en el que cada una tiene un peso w_j^i, las funciones más utilizadas para combinar los valores de entrada son las siguientes:

- La función suma ponderada:

$$z(x) = \sum_{j=1}^{n} x_j w_j^i \tag{1}$$

- La función máximo:

$$z(x) = \max(x_1 w_1^i, \ldots, x_n w_n^i) \tag{2}$$

- La función mínimo:

$$z(x) = \min(x_1 w_1^i, \ldots, x_n w_n^i) \tag{3}$$

- La función lógica AND ($\wedge$) o OR ($\vee$), aplicable sólo en el caso de entradas binarias:

$$z(x) = (x_1 w_1^i \wedge \ldots \wedge x_n w_n^i) \tag{4}$$

$$z(x) = (x_1 w_1^i \vee \ldots \vee x_n w_n^i) \tag{5}$$

La utilización de una función u otra está relacionada con el problema y los datos concretos con los que se trabaja. Sin embargo, generalmente, la suma ponderada suele ser la más utilizada.

1.1.2. Función de activación o transferencia

Merece la pena detenernos en este punto para plantear la siguiente reflexión. Si la red neuronal consistiera simplemente en pasar de nodo en nodo distintas combinaciones lineales de sus respectivos datos de entrada, sucedería que a medida que incrementamos el valor X también incrementaríamos el valor resultante Y, de modo que la red neuronal sólo sería capaz de llevar a cabo aproximaciones lineales.

Las funciones de activación o transferencia toman el valor calculado por la función de entrada o combinación y lo modifican antes de pasarlo a la salida.

Algunas de las funciones de activación más utilizadas son:

- La **función escalón**, que se muestra en la Figura 2 (a) y cuyo comportamiento es:

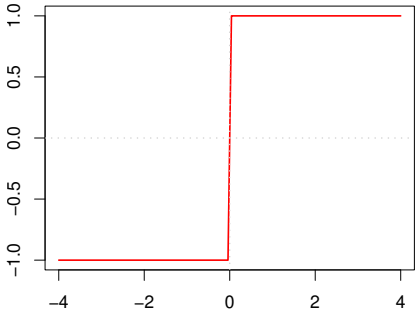
$$y(x) = \begin{cases} 1 & \text{si } x \geq \alpha \\ -1 & \text{si } x \leq \alpha \end{cases} \tag{6}$$

  donde α es el valor umbral de la función de activación. La salida binaria se puede establecer en los valores $\{-1, 1\}$, pero también se utilizan a menudo los valores $\{0, 1\}$.

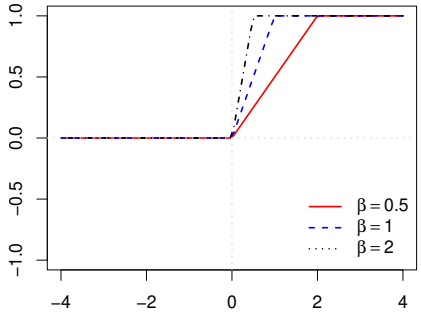
- La **función lineal**, que permite generar combinaciones lineales de las entradas. En su forma más básica se puede ver en la Figura 2 (b).

$$y(x) = \beta x \tag{7}$$

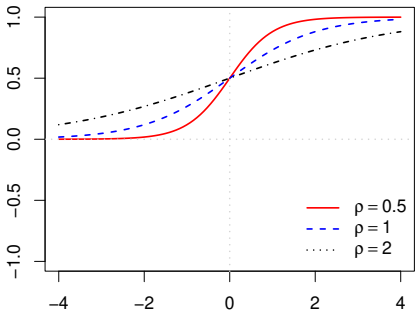
Figura 2. Representación de las funciones escalón, lineal, sigmoide, hiperbólica y rectificadora



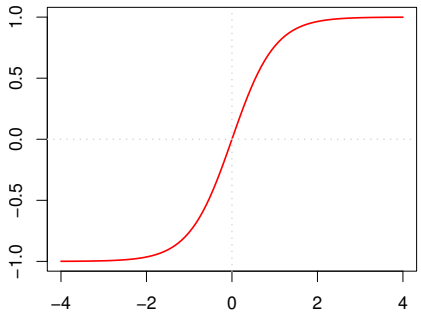
(a) Función escalón



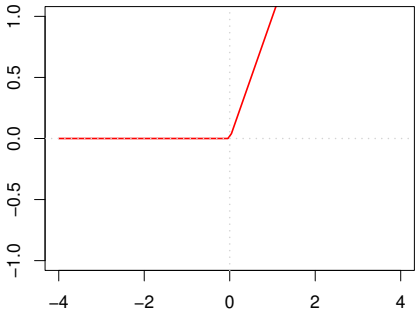
(b) Función lineal



(c) Función sigmoide



(d) Tangente hiperbólica



(e) Función rectificadora

- La **función sigmoide** o **función logística**, que se muestra en la Figura 2 (c) y cuya fórmula incluye un parámetro (ρ) para determinar la forma de la curva, pudiendo actuar como un separador más o menos "suave" de las salidas.

$$y(x) = \frac{1}{1 + e^{\frac{-x}{\rho}}} \tag{8}$$

- La **tangente hiperbólica**, que describimos a continuación y podemos ver en la Figura 2 (d).

$$y(x) = tanh(x) = \frac{senh(x)}{cosh(x)} = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{9}$$

- La **función rectificadora** se calcula obteniendo la parte positiva de su argumento, como podemos ver en la Figura 2 (e), y su fórmula de cálculo es:

$$y(x) = max(0, x) \tag{10}$$

Las funciones sigmoides e hiperbólicas satisfacen los criterios de ser diferenciables y monótonas. Además, otra propiedad importante es que su razón de cambio es mayor para valores intermedios y menor para valores extremos.

También es importante destacar que las funciones sigmoides e hiperbólicas presentan un comportamiento no lineal. Por lo tanto, cuando se utilizan estas funciones en una red neuronal, el conjunto de la red se comporta como una función no lineal compleja. Si los pesos que se aplican a las entradas se consideran los coeficientes de esta función, entonces el proceso de aprendizaje se convierte en el ajuste de los coeficientes de esta función no lineal, de manera que se aproxime a los datos que aparecen en el conjunto de las entradas.

Lectura complementaria

X. Glorot, A. Bordes, Y. Bengio. "Deep sparse rectifier neural networks". In AISTATS, Vol. 15 of JMLR Proceedings, pp. 315-323, 2011.

La función rectificadora se conoce también como "función rampa" y es análoga a la rectificación de media onda en ingeniería eléctrica. Esta función de activación se popularizó al permitir un mejor entrenamiento de redes neuronales profundas (Glorot *et al.*, 2011). Actualmente, es la función de activación más popular para redes profundas.

1.1.3. Casos concretos de neuronas

Como hemos visto, la neurona es una unidad que se puede parametrizar en base a dos funciones principales: la función de entrada o combinación y la función de activación o transferencia. Fijando estas dos funciones obtenemos algunas neuronas que son especialmente útiles y que merece la pena conocer su funcionamiento.

El perceptrón

El perceptrón (*perceptron*) es una estructura propuesta por Rosenblatt (1962). Actualmente no se utiliza demasiado, pero consideramos interesante revisar su estructura por su relevancia histórica. Se caracterizada por las siguientes funciones:

Lectura complementaria

F. Rosenblattd. "Principles of neurodynamics; perceptrons and the theory of brain mechanisms". Spartan Books, 1962.

- La función de entrada de la neurona i es la suma ponderada de las entradas (x_j) y los pesos (w_j^i), como puede verse en la Ecuación 1.

- La función de activación se representa mediante la función escalón, presentada en la Ecuación 6. En consecuencia, la salida de un perceptrón es un valor binario.

Si analizamos un poco el comportamiento de un perceptrón vemos que la salida tomará el valor 0 o 1 dependiendo de:

$$y_i = \begin{cases} 0 & \text{si } x_1 w_1^i + \ldots + x_n w_n^i - \alpha \leq 0 \\ 1 & \text{si } x_1 w_1^i + \ldots + x_n w_n^i - \alpha > 0 \end{cases} \tag{11}$$

donde el parámetro α es el **umbral** o **sesgo** empleado en la función escalón. En general, hablamos de umbral (*threshold*) cuando se encuentra en la parte derecha de la desigualdad; mientras que hablamos de sesgo (*bias*) cuando lo incorporamos en la parte izquierda de la desigualdad y se puede ver como un valor de entrada fijo. Para simplificar, y haciendo uso de la notación vectorial, sustituiremos la expresión $x_1 w_1^i + \ldots + x_n w_n^i - \alpha$ por wx. Es interesante notar que hemos añadido el parámetro de sesgo en el mismo vector que los pesos, lo que se conoce como notación extendida. Podemos ver el sesgo como un peso más del vector de pesos asociado a una entrada que siempre es igual a 1.

La neurona sigmoide

La neurona sigmoide (*sigmoid*) es muy utilizada en la actualidad, y se caracteriza por:

- La función de entrada de la neurona i es la suma ponderada de las entradas (x_j) y los pesos (w_j^i), como puede verse en la Ecuación 1.

- La función de activación emplea la función sigmoide (ver Ecuación 8). Por lo tanto, su salida no es binaria, si no un valor continuo en el rango $[0, 1]$.

La salida de la neurona sigmoide puede parecer muy similar a la de un perceptrón, pero con una salida más "suave" que permite valores intermedios. Precisamente, la suavidad de la función sigmoide es crucial, ya que significa que los cambios pequeños Δw_j en los pesos y en el sesgo $\Delta\alpha$ producirán un cambio pequeño Δy en la salida de la neurona. Matemáticamente, se puede expresar de esta forma:

$$\Delta y = \sum_{j=1}^{n} \frac{\partial y}{\partial w_j} \Delta w_j + \frac{\partial y}{\partial \alpha} \Delta\alpha \tag{12}$$

donde $\frac{\partial y}{\partial w_j}$ denota la derivada parcial de la salida respecto a w_j y $\frac{\partial y}{\partial \alpha}$ representa la

derivada parcial de la salida respecto a α.

Así, aunque que las neuronas sigmoides tienen cierta similitud de comportamiento cualitativo con los perceptrones, las primeras permiten que sea mucho más fácil definir cómo afectará a la salida el cambio en los pesos y sesgos.

La unidad lineal rectificada (ReLU)

La unidad lineal rectificada (*rectified linear unit* o ReLU) es, actualmente, una de las neuronas más importantes y utilizadas en las redes neuronales profundas. Se caracteriza por:

- La función de entrada de la neurona i es la suma ponderada de las entradas (x_j) y los pesos (w_j^i), como puede verse en la Ecuación 1.

- La función de activación es la función rectificadora (ver Ecuación 10).

Por lo tanto, su salida no es binaria, si no un valor continuo en el rango $[0, +\infty]$.

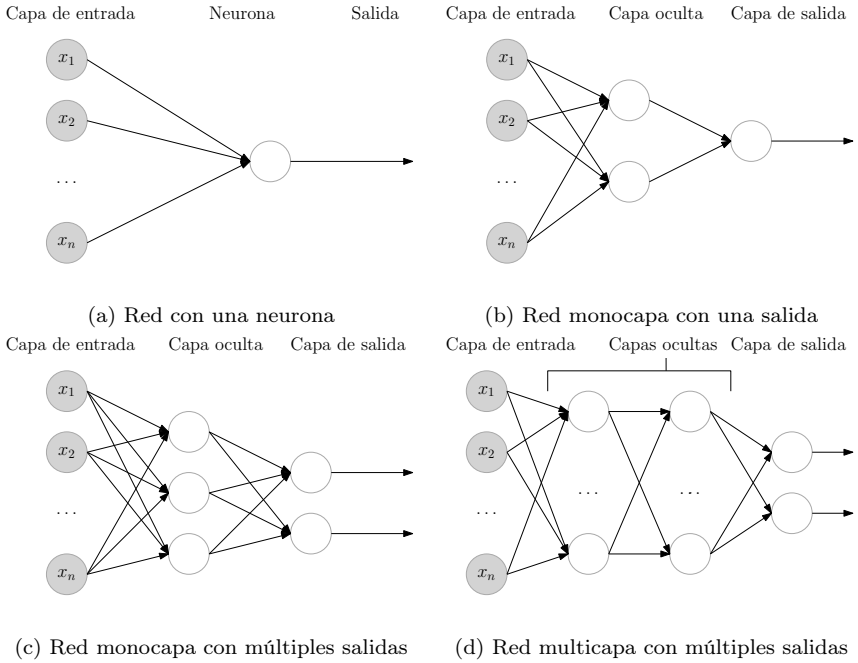
1.2. Arquitectura de una red neuronal

La arquitectura o topología de una red neuronal se define como la organización de las neuronas en distintas capas, así como los parámetros que afectan la configuración de las neuronas, tales como las funciones de entrada o activación. Cada arquitectura o topología puede ser válida para algunos tipos de problemas, presentando diferentes niveles de calidad de los resultados y, también, diferentes niveles de coste computacional.

La red neuronal más simple está formada por una única neurona, conectada a todas las entradas disponibles y con una única salida. La arquitectura de esta red se muestra en la Figura 3 (a). Este tipo de redes permite efectuar funciones relativamente sencillas, equivalentes a la técnica estadística de regresión no lineal.

La segunda arquitectura más simple de redes neuronales la forman las **redes monocapa**. Estas redes presentan la capa entrada; una capa oculta de procesamiento, formado por un conjunto variable de neuronas; y finalmente, la capa de salida con una o más neuronas. La Figura 3 (b) muestra la arquitectura monocapa con una única salida, mientras que la Figura 3 (c) presenta una red monocapa con múltiples salidas (dos en este caso concreto). Este tipo de redes son capaces de clasificar patrones de entrada más complejos o realizar predicciones sobre dominios de dimensionalidad más alta. El número de neuronas de la capa oculta está relacionado con la capacidad de procesamiento y clasificación, pero un número demasiado grande de neuronas en esta capa aumenta el riesgo de sobreespecialización en el proceso de entrenamiento.

Figura 3. Ejemplos de arquitecturas de redes neuronales



(a) Red con una neurona

(b) Red monocapa con una salida

(c) Red monocapa con múltiples salidas

(d) Red multicapa con múltiples salidas

Finalmente, el número de neuronas de la capa de salida depende, en gran medida, del problema concreto que vamos a tratar y de la codificación empleada. Por ejemplo, en casos de clasificación binaria, una única neurona de salida suele ser suficiente, pero en casos de clasificación en n grupos se suele emplear n neuronas, donde cada una indica la pertenencia a una determinada clase.

Finalmente, se puede añadir un número indeterminado de capas ocultas, produciendo lo que se conoce como **redes multicapa**, como se puede ver en la Figura 3 (d). Cuando se añaden neuronas y capas a una red se aumenta, generalmente, su poder de predicción, la calidad de dicha predicción y la capacidad de separación. Sin embargo, también se aumenta su tendencia a la sobreespecialización y el coste computacional y temporal del proceso de entrenamiento.

Hasta ahora, hemos estado discutiendo redes neuronales donde la salida de una capa se utiliza como entrada a la siguiente capa. Tales redes se llaman **redes neuronales prealimentadas** (*Feedforward Neural Networks*, FNN). Esto significa que no hay bucles en la red. Es decir, la información siempre avanza, nunca se retroalimenta.

Sin embargo, hay otros modelos de redes neuronales artificiales en las que los bucles de retroalimentación son posibles. Estos modelos se llaman **redes neuronales recurrentes** (*Recurrent Neural Networks*, RNN). La idea en estos modelos es tener neuronas que se activan durante un tiempo limitado. Esta activación puede estimular otras neuronas, que se pueden activar un poco más tarde, también por una duración limitada. Es decir, la redes recurrentes reutilizan todas o parte de las salidas de la capa i como entradas en la capa $i-q$ tal que $q \geq 1$. Algunos ejemplos interesantes son las redes de Hopfield, que tienen conexiones simétricas, o las máquinas de Boltzmann (McClelland & Rumelhart, 1986).

Lectura complementaria

J. L. McClelland, D. E. Rumelhart. "Parallel Distributed Processing". MIT press, 1986.

1.2.1. Dimensiones de una red neuronal

A continuación discutiremos la importancia y parametrización de las dimensiones de la red neuronal artificial. En este sentido, debemos considerar:

- La dimensión de la capa de entrada
- La dimensión de la capa de salida
- La topología y dimensiones de las capas ocultas.

La dimensión de la **capa de entrada** se fija inicialmente con el número de atributos que se consideran del conjunto de datos. Es relevante recordar que las estrategias de selección de atributos o reducción de dimensionalidad permiten reducir el número de elementos a considerar sin una pérdida excesiva en la capacidad de predicción del modelo. Adicionalmente, el tipo de codificación empleado en los parámetros de entrada de la red, lógicamente, también influirán en la dimensión de la capa de entrada.

La **capa de salida** de una red neuronal se define a partir del número de clases y de la codificación empleada para su representación. Generalmente, en un problema de clasificación, se utiliza una neurona en la capa de salida para cada clase a representar, de modo que sólo una neurona debería de "activarse" para cada instancia. Alternativamente, cada neurona en la capa de salida nos puede proporcionar el valor de pertenencia a una determinada clase en problemas de clasificación difusa.

Lectura complementaria

M. Mezard, J-P Nadal. "Learning in feedforward layered networks: the tiling algorithm". Journal of Physics A: Mathematical and General, Vol. 22(12):2191, 1989.

La topología y dimensión de la **capa oculta** no es un problema trivial, y no existe una solución única y óptima *a priori* para este problema. Existe literatura específica dedicada a la obtención de la arquitectura óptima para una red neuronal artificial (Mezard & Nadal, 1989).

En general, añadir nuevas neuronas en las capas ocultas implica:

1) Aumentar el poder predictivo de la red, es decir, la capacidad de reconocimiento y predicción de la red. Pero también aumenta el peligro de sobreajuste a los datos de entrenamiento.

2) Disminuir la posibilidad de caer en un mínimo local.

3) Alterar el tiempo de aprendizaje. Éste varía de forma inversa al número de neuronas de las capas ocultas.

Por el contrario, una red con menos neuronas en las capas ocultas permite:

1) Reducir el riesgo de sobreespecilización a los datos de entrenamiento, ya que al disponer de menos nodos se obtiene un modelo más general.

En la práctica, se suele utilizar un número de neuronas en las capas ocultas que se encuentre entre una y dos veces el número de entradas de la red. Si se detecta que la red está sobreajustando, debemos reducir el número de neuronas en las capas ocultas. Por el contrario, si la evaluación del modelo no es satisfactoria, debemos aumentar el número de neuronas en las capas ocultas.

1.3. Entrenamiento de una red neuronal

En esta sección discutiremos los principales métodos de entrenamiento de una red neuronal. En primer lugar, en la Sección 1.3.1, analizaremos el funcionamiento de una neurona tipo perceptrón. En la Sección 1.3.2 veremos las bases teóricas del método del descenso del gradiente, que plantea las bases necesarias para el método de entrenamiento más utilizado en la actualidad, conocido como el método de Retropropagación, que analizaremos en la Sección 1.3.3.

1.3.1. Funcionamiento y entrenamiento de una neurona

Como hemos visto en las secciones anteriores, el funcionamiento genérico de una neurona viene determinado por la función:

$$y = \sigma(wx - \alpha) \tag{13}$$

donde σ representa la función de activación de la neurona, por ejemplo la función escalón en el caso de un perceptrón.

El proceso de aprendizaje de las neuronas se basa en la optimización de dos grupos de variables:

- El conjunto de pesos de la entrada de cada neurona $W^i = \{w_1^i, w_2^i, \ldots, w_n^i\}$.
- El valor umbral de la función de activación, también llamado sesgo, que denotaremos por α.

En el caso de un perceptrón, la salida del mismo viene determinada por:

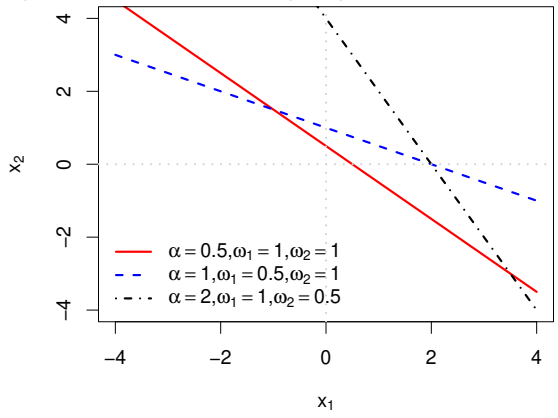
$$y = \begin{cases} 0 & \text{si } wx - \alpha \leq 0 \\ 1 & \text{si } wx - \alpha > 0 \end{cases} \tag{14}$$

En consecuencia, el borde de separación entre estas dos regiones viene dado por la expresión $wx - \alpha = 0$. Limitándonos en el espacio bidimensional, obtenemos una recta que separa las dos clases. Los puntos por encima de la línea corresponden a la clase 1, mientras que por debajo corresponden a la clase 0. Podemos ver un ejemplo de esta recta en la Figura 4 y su expresión matemática en la ecuación 15.

$$x_2 = \frac{\alpha - x_1 w_1}{w_2} \tag{15}$$

Dicha figura muestra distintos valores para los parámetros α, w_1 y w_2. Si se extiende a más dimensiones, se obtiene el hiperplano separador. Por lo tanto, podemos concluir que una neurona simple con función de entrada basada en suma ponderada y función de activación escalón es un separador lineal, es decir, permite clasificar instancias dentro de regiones linealmente separables.

Figura 4. Espacio de clasificación de un perceptrón



1.3.2. Método del descenso del gradiente

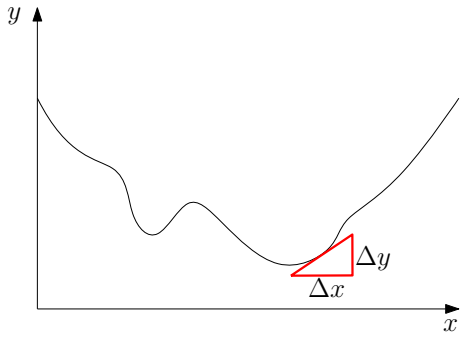
Definimos el error que comete una red neuronal como la diferencia entre el resultado esperado en una instancia de entrenamiento y el resultado obtenido. Hay varias formas de cuantificarlo, como por ejemplo, utilizando el error cuadrático:

$$\varepsilon = \frac{1}{2}(c - y)^2 \tag{16}$$

donde c es el valor de clase de la instancia de entrenamiento e y la salida de la red asociada a esta misma instancia.

Cada vez que aplicamos una instancia de entrenamiento al perceptrón, comparamos la salida obtenida con la esperada, y en el caso de diferir, deberemos modificar los pesos y el sesgo de las neuronas para que la red "aprenda" la función de salida deseada.

Figura 5. Caracterización del mínimo de una función



La idea principal es representar el error como una función continua de los pesos e intentar llevar la red hacia una configuración de valores de activación que nos permita encontrar el mínimo de esta función. Para determinar la magnitud y dirección del cambio que debemos introducir en el vector de pesos para reducir el error que está cometiendo, utilizaremos el concepto de "pendiente", ilustrado en la Figura 5 y expresado por la ecuación:

$$\frac{\Delta y}{\Delta x} \tag{17}$$

que indica que la pendiente de un punto dado es el gradiente de la tangente a la curva de la función en el punto dado. La pendiente es cero cuando nos encontramos en un punto mínimo (o máximo) de la función, donde no hay pendiente.

La función que nos permite modificar el vector de pesos w a partir de una instancia de entrenamiento $d \in D$ se conoce como la **regla delta** y se expresa de la siguiente forma:

$$\Delta w = \eta(c - y)d \tag{18}$$

donde c es el valor que indica la clase en el ejemplo de entrenamiento, y el valor de la función de salida para la instancia d, y η la **tasa** o **velocidad de aprendizaje** (*learning rate*).

El método utilizado para entrenar este tipo de redes, y basado en la regla delta, se

muestra en el Algoritmo 1.

Algoritmo 1 Pseudocódigo del método del descenso del gradiente

Entrada: W (conjunto de vectores de pesos) y D (conjunto de instancias de entrenamiento)

Salida: El conjunto de vectores de pesos W

```
1: while (y ≉ c_i ∀(d_i, c_i) ∈ D) do
2:     for all ((d_i, c_i) ∈ D) do
3:         Calcular la salida y de la red cuando la entrada es d_i
4:         if (y ≉ c_i) then
5:             Modificar el vector de pesos w' = w + η(c − y)d_i
6:         end if
7:     end for
8: end while
```

Es importante destacar que para que el algoritmo converja es necesario que las clases de los datos sean linealmente separables.

Funciones de activación no continuas

En el caso de que la función de activación neuronal no sea continua, como por ejemplo en el perceptrón u otras neuronas con función escalón, la función de salida de la red no es continua. Por lo tanto, no es posible aplicar el método de descenso del gradiente sobre los valores de salida de la red, pero si es posible aplicarlo sobre los valores de activación. Este técnica se conoce como *elementos adaptativos lineales* (*Adaptative Linear Elements* o ADALINE).

Por lo tanto, modificamos la ecuación anterior del error para adaptarla a las funciones no continuas:

$$\varepsilon = \frac{1}{2}(c - z)^2 \tag{19}$$

donde z es el valor de activación, resultado de la función de entrada de la neurona.

Aplicando el descenso del gradiente en esta función de error, obtenemos la siguiente regla delta:

$$\Delta w = \eta(c - z)d \tag{20}$$

El método utilizado para entrenar este tipo de redes, y basado en la regla delta, se muestra en el Algoritmo 2.

Algoritmo 2 Pseudocódigo del método ADALINE

Entrada: W (conjunto de vectores de pesos) y D (conjunto de instancias de entrenamiento)

Salida: El conjunto de vectores de pesos W

```
1: while (a ≉ c_i ∀(d_i, c_i) ∈ D) do
2:     for all ((d_i, c_i) ∈ D) do
3:         Calcular el valor de activación z de la red cuando la entrada es d_i
4:         if (a ≉ c_i) then
5:             Modificar el vector de pesos w' = w − η(c − z)d_i
6:         end if
7:     end for
8: end while
```

1.3.3. Método de retropropagación

En las redes multicapa no podemos aplicar el algoritmo de entrenamiento visto en la sección anterior. El problema aparece con los nodos de las capas ocultas: no podemos saber *a priori* cuáles son los valores de salida correctos.

En el caso de una neurona j con función sigmoide, la regla delta es:

$$\Delta w_i^j = \eta \sigma'(z^j)(c^j - y^j)x_i^j \tag{21}$$

donde:

- $\sigma'(z^j)$ indica la pendiente (derivada) de la función sigmoide, que representa el factor con que el nodo j puede afectar al error. Si el valor de $\sigma'(z^j)$ es pequeño, nos encontramos en los extremos de la función, donde los cambios no afectan demasiado a la salida (ver Figura 2 (c)). Por el contrario, si el valor es grande, nos encontramos en el centro de la función, donde pequeñas variaciones pueden alterar considerablemente la salida.

- $(c^j - y^j)$ representa la medida del error que se produce en la neurona j.

- x_i^j indica la responsabilidad de la entrada i de la neurona j en el error. Cuando este valor es igual a cero, no se modifica el peso, mientras que si es superior a cero, se modifica proporcionalmente a este valor.

La delta que corresponde a la neurona j puede expresarse de forma general para toda

neurona, simplificando la notación anterior:

$$\delta^j = \sigma'(z^j)(c^j - y^j) \tag{22}$$

$$\Delta w_i^j = \eta \delta^j x_i^j \tag{23}$$

A partir de aquí debemos determinar qué parte del error total se asigna a cada una de las neuronas de las capas ocultas. Es decir, debemos definir como modificar los pesos y tasas de aprendizaje de las neuronas de las capas ocultas a partir del error observado en la capa de salida.

El método de Retropropagación (*backpropagation*) se basa en un esquema general de dos pasos:

1) Propagación hacia adelante (*Feedforward*), que consiste en introducir una instancia de entrenamiento y obtener la salida de la red neuronal.

2) Propagación hacia atrás (*Backpropagation*), que consiste en calcular el error cometido en la capa de salida y propagarlo hacia atrás para calcular los valores delta de las neuronas de las capas ocultas.

El Algoritmo 3 muestra el pseudocódigo del método de Retropropagación, aunque profundizaremos en los detalles y su derivación matemática en el capítulo 2 de este módulo didáctico.

El entrenamiento del algoritmo se realiza ejemplo a ejemplo, es decir, una instancia de entrenamiento en cada iteración. Para cada una de estas instancias se realizan los siguientes pasos:

1) El primer paso, que es la propagación hacia adelante, consiste en aplicar el ejemplo a la red y obtener los valores de salida (línea 3).

2) A continuación, el método inicia la propagación hacia atrás, empezando por la capa de salida. Para cada neurona j de la capa de salida:

a) Se calcula, en primera instancia, el valor δ^j basado en el valor de salida de la red para la neurona j (y^j), el valor de la clase de la instancia (c^j) y la derivada de la función sigmoide ($\sigma'(z^j)$) (línea 5).

Algoritmo 3 Pseudocódigo del método de Retropropagación

Entrada: W (conjunto de vectores de pesos) y D (conjunto de instancias de entrenamiento)

Salida: El conjunto de vectores de pesos W

1: **while** (error de la red $>\varepsilon$) **do**
2: **for all** $((d_i, c_i) \in D)$ **do**
3: Calcular el valor de salida de la red para la entrada d_i
4: **for all** (neurona j en la capa de salida) **do**
5: Calcular el valor δ^j para esta neurona: $\delta^j = \sigma'(z^j)(c^j - y^j)$
6: Modificar los pesos de la neurona siguiendo el método del gradiente: $\Delta w_i^j = \eta \delta^j x_i^j$
7: **end for**
8: **for all** (neurona k en las capas ocultas) **do**
9: Calcular el valor δ^k para esta neurona: $\delta^k = \sigma'(z^k) \sum_{j \in S_k} \delta^k w_k^j$
10: Modificar los pesos de la neurona siguiendo el método del gradiente: $\Delta w_i^k = \eta \delta^k x_i^k$
11: **end for**
12: **end for**
13: **end while**

b) A continuación, se modifica el vector de pesos de la neurona de la capa de salida, a partir de la tasa de aprendizaje (η), el valor delta de la neurona calculado en el paso anterior (δ^j) y el factor x_i^j que indica la responsabilidad de la entrada i de la neurona j en el error (línea 6).

3) Finalmente, la propagación hacia atrás se aplica a las capas ocultas de la red. Para cada neurona k de las capas ocultas:

a) En primer lugar, se calcula el valor δ^k basado en la derivada de la función sigmoide ($\sigma'(z^k)$) y el sumatorio del producto de la delta calculada en el paso anterior (δ^k) por el valor w_k^j, que indica el peso de la conexión entre la neurona k y la neurona j (línea 9). El conjunto S_k está formado por todos los nodos de salida a los que está conectada la neurona k.

b) En el último paso de la iteración, se modifica el vector de pesos de la neurona k, a partir de la tasa de aprendizaje (η), el valor delta de la neurona calculado en el paso anterior (δ^k) y el factor x_i^k que indica la responsabilidad de la entrada k de la neurona j en el error (línea 10).

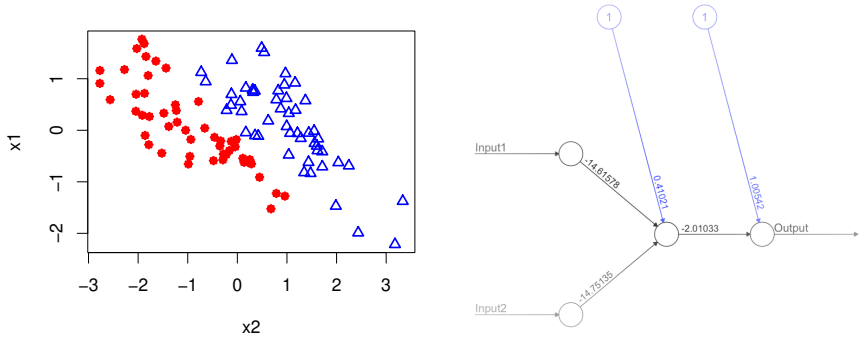
La idea que subyace a este algoritmo es relativamente sencilla, y se basa en propagar el error de forma proporcional a la influencia que ha tenido cada nodo de las capas ocultas en el error final producido por cada una de las neuronas de la capa de salida.

1.4. Ejemplo de aplicación

En primer lugar, veremos un ejemplo sencillo con un conjunto de datos linealmente separables (Figura 6 (a)). Como ya hemos comentado anteriormente, este tipo de

conjuntos puede ser clasificado correctamente con un modelo de red neuronal basado en una única neurona, que "aprende" la función necesaria para separar ambas clases. En este caso concreto, estamos hablando de un espacio de dos dimensiones, con lo cual el modelo genera una recta que divide el espacio de datos en dos partes, una para cada clase. También hemos comentado que el modelo generado por una red neuronal, una vez entrenado, incluye la arquitectura concreta (número de neuronas y capas), el tipo de función de entrada y activación, y los pesos correspondientes a todas las conexiones de la red. En este ejemplo, el modelo generado para la clasificación de estos puede verse en la Figura 6 (b). Este modelo clasifica correctamente todas las instancias del conjunto de entrenamiento y test.

Figura 6. Ejemplos basado en datos lineales



(a) Ejemplo de datos linealmente separables (b) Red monocapa con una salida

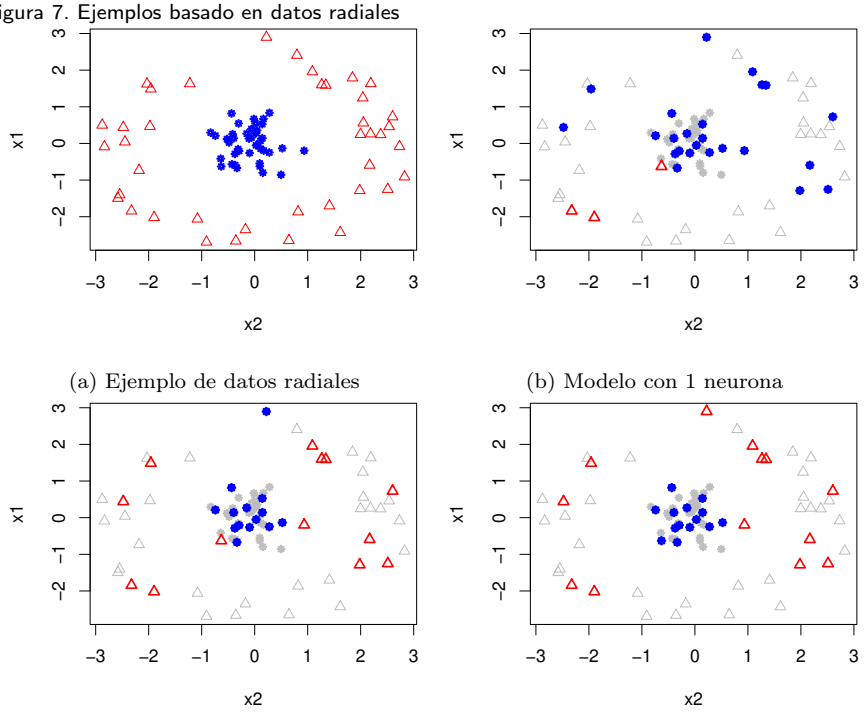
A continuación veremos un ejemplo utilizando datos en un espacio bidimensional, pero a diferencia del caso anterior, éstos presentan una estructura claramente radial y no pueden, por lo tanto, ser clasificados correctamente por una red como la vista en el caso anterior.

Los datos del ejemplo se pueden ver en la Figura 7 (a). La clase del centro se muestra con círculos azules (clase 1) y la otra mediante triángulos rojos (clase 2). Aunque ya hemos comentado que una única neurona no es capaz de separar datos que no sean linealmente separables, vamos a ver el resultado que obtendría una arquitectura similar a la utilizada en el ejemplo anterior, basada en una única neurona en la capa oculta. En este caso, la precisión del modelo se reduce a 0.592, es decir, 59,2 % de instancias correctamente clasificadas. La Figura 7 (b) muestra la clasificación propuesta por el modelo de las instancias de test. Las instancias de entrenamiento se muestran en color gris, para facilitar la visualización de los resultados. Podemos ver que se han clasificado muchas instancias como clase 1 cuando realmente pertenecen a la clase 2.

Si ampliamos a una red con dos neuronas en la capa oculta, el conjunto de la red será capaz de usar dos rectas separadoras para la clasificación de los datos. En este caso la precisión sube hasta 0.888 añadiendo una sola neurona en la capa oculta. La visualización de los resultados obtenidos sobre las instancias de test, Figura 7 (c), permite identificar de forma aproximada los dos hiperplanos (en este caso concreto, rectas en el espacio de dos dimensiones) que se han utilizado para la clasificación. Ampliando hasta 10 neuronas en una única capa oculta nos permite mejorar hasta

una precisión de 0.963, es decir, sólo una instancia de test incorrectamente clasificada. Los resultados se pueden ver en la Figura 7 (d). El resultado es exactamente el mismo que si creamos una red de dos capas, con arquitectura $(3, 2)$, es decir, tres neuronas en la primera capa oculta y dos en la segunda.

Figura 7. Ejemplos basado en datos radiales



(a) Ejemplo de datos radiales

(b) Modelo con 1 neurona

(c) Modelo con 2 neuronas

(d) Modelo con 10 neurona

1.5. El problema de la desaparición del gradiente

Muchos de los problemas existentes pueden ser resueltos mediante redes neuronales con una sola capa oculta, más la capa de entrada y la capa de salida. Aún así, se espera que las redes con un número mayor de capas ocultas puedan "aprender" conceptos más abstractos, y por lo tanto, pueden aproximar problemas más complejos o mejorar las soluciones obtenidas con arquitecturas basadas en una única capa oculta.

Cuando intentamos entrenar redes con múltiples capas ocultas empleando los algoritmos de entrenamiento vistos hasta el momento encontramos el problema conocido como la desaparición del gradiente (*the vanishing gradient problem*) (Hochreiter et al., 2001).

Lectura complementaria

S. Hochreiter, Y. Bengio, P. Frasconi. "Gradient flow in recurrent nets: the difficulty of learning long-term dependencies". In J. Kolen and S. Kremer, editors, Field Guide to Dynamical Recurrent Networks. IEEE Press, 2001.

El gradiente, empleado en el algoritmo de entrenamiento que hemos visto, se vuelve inestable en las primeras capas de neuronas cuando tratamos con redes con múltiples capas ocultas, produciendo una explosión del aprendizaje (*exploding gradient problem*) o una desaparición del mismo (*vanishing gradient problem*). En ambos casos, dificulta enormemente el aprendizaje de las neuronas en las primeras capas de la red. Esta inestabilidad es un problema fundamental para el aprendizaje basado en gradientes en redes neuronales profundas, es decir, en redes neuronales con múltiples capas ocultas.

El problema fundamental es que el gradiente en las primeras capas se calcula a partir de una función basada en el producto de términos de todas las capas posteriores. Cuando hay muchas capas, produce una situación intrínsecamente inestable. La única manera de que todas las capas puedan aprender a la misma velocidad es equilibrar todos los productos de términos que se emplean en el aprendizaje de las distintas capas.

En resumen, el problema es que las redes neuronales sufren de un problema de gradiente inestable, ya sea por desaparición o explosión del mismo. Como resultado, si usamos técnicas de aprendizaje basadas en gradiente estándar, las diferentes capas de la red tenderán a aprender a velocidades muy diferentes, dificultando el proceso de aprendizaje de la red. Este problema ha propiciado la aparición de las redes neuronales profundas (*deep neural networks*).

2. El algoritmo de Retropropagación (*Backpropagation*)

En este capítulo, discutiremos en profundidad el algoritmo de propagación hacia atrás (retropropagación o *backpropagation*, en inglés) para el cálculo del gradiente de la función de coste en una red neuronal completamente conectada.

Supongamos que tenemos una red neuronal de L capas completamente conectadas. En esta sección emplearemos la siguiente notación:

- n_l es el número de neuronas de la capa l. Por convención, consideramos n_0 la dimensión de los datos de entrada.

- $X \in \mathcal{M}_{n_0 \times m}(\mathbb{R})$ es la matriz de datos de entrada, donde cada columna representa un ejemplo y cada fila representa un atributo.

- $W^{[l]} \in \mathcal{M}_{n_l \times n_{l-1}}(\mathbb{R})$ denota la matriz de pesos que conecta la capa $l-1$ con la capa l. Más concretamente, el elemento de $W^{[l]}$ correspondiente a la fila j, columna k, denotado $w^{[l]}_{jk}$, es un escalar que representa el peso de la conexión entre la neurona j de la capa l y la neurona k de la capa $l-1$.

- $b^{[l]} \in \mathcal{M}_{n_l \times 1}(\mathbb{R})$ es un vector que denota el bias de la capa l. El bias correspondiente a la neurona j de la capa l lo denotamos $b^{[l]}_j$.

- $z^{[l]} \in \mathcal{M}_{n_l \times 1}(\mathbb{R})$ denota la combinación lineal de la entrada a la capa l con los parámetros $W^{[l]}$ y $b^{[l]}$.

- $g : \mathbb{R} \to \mathbb{R}$ es una función no lineal (como relu o sigmoide). Si $M \in \mathcal{M}_{c \times d}(\mathbb{R})$ es una matriz (o un vector), denotaremos $g(M) \in \mathcal{M}_{c \times d}(\mathbb{R})$ la matriz (o el vector) que se obtiene aplicando la función g a cada coordenada de M.

- $a^{[l]} \in \mathcal{M}_{n_l \times 1}(\mathbb{R})$ denota el vector salida de la capa l de la red neuronal. En particular $a^{[L]}$ denota la salida de la red neuronal. Por convención, denotaremos $a^{[0]}$ el vector de atributos de entrada a la red neuronal.

- Denotaremos el producto de matrices con el símbolo $*$, el producto de escalares con el símbolo $\cdot$ y el producto componente a componente con el símbolo $\odot$.

2.1. Caso particular con un único ejemplo

Empezaremos considerando el caso de un único ejemplo, en esta sección, para luego extender el caso general de varios ejemplos.

2.1.1. Propagación hacia delante

Si tenemos un único ejemplo, entonces $m = 1$ y X es un vector columna de longitud n_0, el número de atributos, que coincide con la dimensión de la entrada a la red neuronal. Entonces, según la convención que estamos utilizando, $a^{[0]} = X$.

En este caso, las ecuaciones para la propagación hacia delante son:

$$z_j^{[l]} = \sum_{k=1}^{n_{l-1}} w_{jk}^{[l]} \cdot a_k^{[l-1]} + b_j^{[l]} \tag{24}$$

$$a_j^{[l]} = g(z_j^{[l]}) \tag{25}$$

Para poder calcular todas las componentes a la vez es posible escribir las fórmulas anteriores en versión matricial de la siguiente forma:

$$z^{[l]} = W^{[l]} * a^{[l-1]} + b^{[l]} \tag{26}$$

$$a^{[l]} = g(z^{[l]}) \tag{27}$$

Una vez hemos aplicado las fórmulas anteriores a todas las capas obtenemos la salida de la red, $a^{[L]}$. Para determinar si la salida de la red es adecuada, podemos definir una función que calcule el error que comete la red neuronal en la salida, $\mathcal{L}(y, a^{[L]})$, donde y es la etiqueta correcta asociada al ejemplo con el que trabajamos. Si la etiqueta es binaria, es decir siempre vale 0 o 1, entonces la salida de la red neuronal está formada por un único valor (esto es, $n_L = 1$) y una posible función de error es el *log-loss*:

$$\mathcal{L}(y, a^{[L]}) = -(y \cdot \log(a^{[L]}) + (1 - y) \cdot \log(1 - a^{[L]})) \tag{28}$$

Es importante notar que hemos multiplicado por -1 la expresión dentro de los paréntesis para que el error sea positivo y minimizar el error se corresponda con minimizar la función de coste.

2.1.2. Propagación hacia atrás

Observemos que en el cálculo de $a^{[L]}$ intervienen todos los parámetros $W^{[l]}, b^{[l]}$, con $l = 1, \ldots, L$. Por lo tanto, la función de coste $\mathcal{L}(y, a^{[L]})$ también depende de los parámetros $W^{[l]}, b^{[l]}$, para todo $l = 1, \ldots, L$.

Dado que la función de coste mide la distancia entre la salida de la red y la etiqueta correcta podemos intentar minimizar esta función para acercar lo máximo posible los valores predichos a los valores correctos. Para minimizar la función de coste debemos modificar los parámetros de la red de forma adecuada, y para ello podemos calcular el gradiente de la función de coste respecto a los parámetros de la red y utilizarlo para actualizar los valores de los parámetros.

Para calcular el gradiente de la función de coste respecto a los parámetros de la red utilizaremos el algoritmo de la propagación hacia atrás. Este algoritmo se basa en aplicar repetidamente la regla de la cadena para calcular las derivadas parciales de la función de coste respecto cualquier parámetro de la red neuronal.

Recordatorio de la regla de la cadena

Supongamos que tenemos dos funciones de una variable:

$$\begin{array}{rcccc} f & : & \mathbb{R} & \to & \mathbb{R} \\ & & x & \mapsto & f(x) \end{array} \qquad \begin{array}{rcccc} g & : & \mathbb{R} & \to & \mathbb{R} \\ & & y & \mapsto & g(y) \end{array}$$

Entonces, por la regla de la cadena, la derivada de la composición $g(f(x))$ respecto x viene dada por el producto:

$$\frac{\partial (g \circ f)}{\partial x} = \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial x} \tag{29}$$

Ahora supongamos que tenemos dos funciones en varias variables como las siguientes

$$\begin{array}{rcccl} f & : & \mathbb{R} & \to & \mathbb{R}^d \\ & & x & \mapsto & (f_1(x), \ldots, f_d(x)) \end{array} \qquad \begin{array}{rcccl} g & : & \mathbb{R}^d & \to & \mathbb{R} \\ & & (y_1, \ldots, y_d) & \mapsto & g(y_1, \ldots, y_d) \end{array}$$

Entonces, por la regla de la cadena en varias variables, la derivada de la composición $g(f(x))$ respecto x viene dada por la fórmula:

$$\frac{\partial (g \circ f)}{\partial x} = \sum_{c=1}^{d} \frac{\partial g}{\partial f_c} \cdot \frac{\partial f_c}{\partial x} \tag{30}$$

Utilizando la regla de la cadena podemos calcular la derivada parcial de la función de coste con respecto a cualquier parámetro de la red neuronal.

El primer paso es calcular el gradiente respecto a la salida de la red neuronal. Esto dependerá de la función de coste $\mathcal{L}$ que se utilice, pero se puede calcular de forma analítica. Por ejemplo, en el caso de la función de coste *log-loss*, tenemos:

$$\frac{\partial \mathcal{L}}{\partial a^{[L]}} = -\left(\frac{y}{a^{[L]}} - \frac{1-y}{1-a^{[L]}} \right) \tag{31}$$

En general, asumiremos que hemos calculado $\frac{\partial \mathcal{L}}{\partial a^{[L]}}$ y que lo podemos utilizar en la regla de la cadena. Supongamos que queremos calcular la derivada de la función de coste respecto a un peso concreto de la red neuronal $w_{jk}^{[l]}$. Para ello, asumimos que todos los demás parámetros son constantes y tenemos entonces la siguiente composición de funciones:

$$\begin{array}{ccccccc} \mathbb{R} & \to & \mathbb{R} & \to & \mathbb{R} & \to & \mathbb{R} \\ w_{jk}^{[l]} & \mapsto & z_j^{[l]} & \mapsto & a_j^{[l]} & \mapsto & \mathcal{L} \end{array}$$

Por lo que, aplicando repetidamente la regla de la cadena en una variable, obtenemos:

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{[l]}} = \frac{\partial \mathcal{L}}{\partial a_j^{[l]}} \cdot \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \cdot \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} \tag{32}$$

Si estamos trabajando con la última capa, entonces $l = L$ y, por lo tanto, ya tenemos calculado el valor de $\frac{\partial \mathcal{L}}{\partial a_j^{[l]}}$. Asumamos por ahora que, aunque l sea menor que L ya tenemos calculado el valor de $\frac{\partial \mathcal{L}}{\partial a_j^{[l]}}$, posteriormente veremos como se calcula. Entonces, utilizando las ecuaciones 24 y 25 tenemos:

$$\frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} = g'(z_j^{[l]}) \tag{33}$$

$$\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = a_k^{[l-1]} \tag{34}$$

Supongamos ahora que queremos calcular la derivada $\frac{\partial \mathcal{L}}{\partial b_j^{[l]}}$. El procedimiento es análogo a lo que hemos hecho hasta ahora, tenemos en este caso la siguiente descomposición:

$$\begin{array}{ccccccc} \mathbb{R} & \to & \mathbb{R} & \to & \mathbb{R} & \to & \mathbb{R} \\ b_j^{[l]} & \mapsto & z_j^{[l]} & \mapsto & a_j^{[l]} & \mapsto & \mathcal{L} \end{array}$$

Por lo que, de nuevo aplicando la regla de la cadena en una variable, obtenemos:

$$\frac{\partial \mathcal{L}}{\partial b_j^{[l]}} = \frac{\partial \mathcal{L}}{\partial a_j^{[l]}} \cdot \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \cdot \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} \tag{35}$$

Volviendo a asumir que tenemos calculada la derivada $\frac{\partial \mathcal{L}}{\partial a_j^{[l]}}$, podemos calcular $\frac{\partial \mathcal{L}}{\partial b_j^{[l]}}$ a partir de:

$$\frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} = g'(z_j^{[l]}) \tag{36}$$

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1 \tag{37}$$

Si queremos calcular las derivadas de la función de coste respecto a todos las componentes de una matriz o un vector a la vez, podemos utilizar la siguiente notación para la matriz de pesos:

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \left(\frac{\partial \mathcal{L}}{\partial a^{[l]}} \odot g'(z^{[l]})\right) * (a^{[l-1]})^T \tag{38}$$

Donde $\frac{\partial \mathcal{L}}{\partial a^{[l]}}, g'(z^{[l]}) \in \mathcal{M}_{n_l \times 1}(\mathbb{R})$ y $(a^{[l-1]})^T \in \mathcal{M}_{1 \times n_{l-1}}(\mathbb{R})$, por lo que $\frac{\partial \mathcal{L}}{\partial W^{[l]}} \in \mathcal{M}_{n_l \times n_{l-1}}(\mathbb{R})$.

Y la siguiente notación para el vector de sesgo (*bias*):

$$\frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{\partial \mathcal{L}}{\partial a^{[l]}} \odot g'(z^{[l]}) = \frac{\partial \mathcal{L}}{\partial z^{[l]}} \tag{39}$$

Donde, en este caso, $\frac{\partial \mathcal{L}}{\partial b^{[l]}} \in \mathcal{M}_{n_l \times 1}(\mathbb{R})$.

Por último, necesitamos poder calcular el valor de $\frac{\partial \mathcal{L}}{\partial a_j^{[l]}}$ para $l < L$. En este caso, la activación de la neurona j en la capa l afecta a todas las neuronas de la capa $l+1$, por lo que la descomposición en funciones que tenemos es la siguiente:

$$\begin{array}{ccccccc} \mathbb{R} & \to & \mathbb{R}^{n_{l+1}} & \to & \mathbb{R}^{n_{l+1}} & \to & \mathbb{R} \\ a_j^{[l]} & \mapsto & (z_1^{[l+1]}, \ldots, z_{n_{l+1}}^{[l+1]}) & \mapsto & (a_1^{[l+1]}, \ldots, a_{n_{l+1}}^{[l+1]}) & \mapsto & \mathcal{L} \end{array}$$

Y si aplicamos la regla de la cadena en varias variables obtenemos la siguiente fórmula:

$$\frac{\partial \mathcal{L}}{\partial a_j^{[l]}} = \sum_{c=1}^{n_{l+1}} \frac{\partial \mathcal{L}}{\partial a_c^{[l+1]}} \cdot \frac{\partial a_c^{[l+1]}}{\partial z_c^{[l+1]}} \cdot \frac{\partial z_c^{[l+1]}}{\partial a_j^{[l]}} \tag{40}$$

Ahora sí, por recursividad, podemos suponer que tenemos calculada la derivada $\frac{\partial \mathcal{L}}{\partial a_c^{[l+1]}}$ para todo $c = 1, \ldots, n_{l+1}$. Por lo tanto, podemos calcular completamente la derivada $\frac{\partial \mathcal{L}}{\partial a_j^{[l]}}$ utilizando:

$$\frac{\partial a_c^{[l+1]}}{\partial z_c^{[l+1]}} = g'(z_c^{[l+1]}) \tag{41}$$

$$\frac{\partial z_c^{[l+1]}}{\partial a_j^{[l]}} = w_{cj}^{[l+1]} \tag{42}$$

El cálculo de las derivadas respecto a las activaciones de las neuronas también se puede hacer para todas las componentes a la vez utilizando la siguiente notación matricial:

$$\frac{\partial \mathcal{L}}{\partial a^{[l]}} = (W^{[l+1]})^T * \left(\frac{\partial \mathcal{L}}{\partial a^{[l+1]}} \odot g'(z^{[l+1]}) \right) \tag{43}$$

Donde $(W^{[l+1]})^T \in \mathcal{M}_{n_l \times n_{l+1}}(\mathbb{R})$ y $\frac{\partial \mathcal{L}}{\partial a^{[l+1]}}, g'(z^{[l+1]}) \in \mathcal{M}_{n_{l+1} \times 1}(\mathbb{R})$ por lo que $\frac{\partial \mathcal{L}}{\partial a^{[l]}} \in \mathcal{M}_{n_l \times 1}(\mathbb{R})$.

Observemos que para hacer los cálculos que hemos especificado necesitamos saber los valores de $z^{[l]}$ y $a^{[l]}$ para todo $l = 1, \ldots, L$, que se han calculado anteriormente durante la propagación hacia delante.

A continuación, resumimos los pasos para calcular el gradiente con un único ejemplo en el conjunto de datos.

Algoritmo de propagación hacia atrás con un ejemplo

1) Calcular $\frac{\partial \mathcal{L}}{\partial a^{[L]}}$

2) Desde $l = L$ hasta 1, repetir:

a) Calcular $\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \left(\frac{\partial \mathcal{L}}{\partial a^{[l]}} \odot g'(z^{[l]}) \right) * (a^{[l-1]})^T$

b) Calcular $\frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{\partial \mathcal{L}}{\partial a^{[l]}} \odot g'(z^{[l]})$

c) Calcular $\frac{\partial \mathcal{L}}{\partial a^{[l-1]}} = (W^{[l]})^T * \left(\frac{\partial \mathcal{L}}{\partial a^{[l]}} \odot g'(z^{[l]}) \right)$

3) Devolver $\frac{\partial \mathcal{L}}{\partial W^{[l]}}$ y $\frac{\partial \mathcal{L}}{\partial b^{[l]}}$ para todo $l = 1, \ldots, L$.

En la descripción del algoritmo se puede ver por qué se llama "de propagación hacia atrás". En efecto, el algoritmo se basa en calcular las derivadas $\frac{\partial \mathcal{L}}{\partial a^{[l]}}$ en cada capa y propagar su valor hacia atrás para permitir el cálculo de las derivadas $\frac{\partial \mathcal{L}}{\partial W^{[l]}}$ y $\frac{\partial \mathcal{L}}{\partial b^{[l]}}$, que son los parámetros de la red neuronal que se pueden modificar.

2.2. Caso general con varios ejemplos

A continuación, extenderemos la formulación anterior para considerar el caso general, con múltiples ejemplos.

2.2.1. Propagación hacia delante

Asumamos ahora que tenemos varios ejemplos en la matriz de datos X, por lo que $m > 1$. Si nos fijamos en los diferentes valores que consideramos en el apartado de notación, veremos que los únicos que dependen de los datos (a parte de X), son $z^{[l]}$ y $a^{[l]}$, que son vectores columna. Podemos considerar entonces formar matrices colocando los vectores columna correspondientes a varios ejemplos uno al lado del otro. De esta forma, obtenemos:

$$Z^{[l]} = (z^{[l](1)} \quad z^{[l](2)} \quad \cdots \quad z^{[l](m)}) \tag{44}$$

$$A^{[l]} = (a^{[l](1)} \quad a^{[l](2)} \quad \cdots \quad a^{[l](m)}) \tag{45}$$

Donde $z^{[l](i)}$ y $a^{[l](i)}$ denotan los vectores $z^{[l]}$ y $a^{[l]}$ que corresponden al ejemplo i-ésimo, respectivamente, y hemos denotado con A y Z mayúsculas las matrices resultantes.

Utilizando la misma convención que anteriormente tenemos que $A^{[0]} = X$ y las ecuaciones matriciales de la propagación hacia delante se pueden escribir como:

$$Z^{[l]} = W^{[l]} * A^{[l-1]} + B^{[l]} \tag{46}$$

$$A^{[l]} = g(Z^{[l]}) \tag{47}$$

Donde $B^{[l]}$ es una matriz formada por el vector columna $b^{[l]}$ repetido m veces. De esta forma, $A^{[L]}$ denota la salida de la red, donde cada columna corresponde a la salida para cada ejemplo.

Al tener varios ejemplos la función de coste global se define como la media de la función de coste para cada error, es decir:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(y^{(i)}, a^{[L](i)}) \tag{48}$$

En este caso entonces nos interesa minimizar la función J para conseguir que la salida de la red se aproxime a las etiquetas correctas de cada ejemplo.

2.2.2. Propagación hacia atrás

Dado que la acción de derivar es una transformación lineal, para calcular el gradiente de la función J podemos calcular el gradiente de $\mathcal{L}$ para cada ejemplo por separado como hemos hecho en la sección anterior y posteriormente hacer la media.

Con esta información ya podríamos implementar el algoritmo de propagación hacia atrás completo con varios ejemplos. Sin embargo, dado que los procesadores actuales están diseñados para realizar cálculos en paralelo, es mucho más eficiente calcular el gradiente para todos los ejemplos a la vez utilizando matrices.

Para ello, podemos utilizar las ecuaciones 38, 39 y 43 y adecuarlas a las substituciones $a^{[l]} \rightarrow A^{[l]}$ y $z^{[l]} \rightarrow Z^{[l]}$.

Concretamente, para la derivada $\frac{\partial J}{\partial W^{[l]}}$ obtenemos:

$$\frac{\partial J}{\partial W^{[l]}} = \frac{1}{m} \left(\frac{\partial \mathcal{L}}{\partial A^{[l]}} \odot g'(Z^{[l]}) \right) * (A^{[l-1]})^T \tag{49}$$

Donde $\frac{\partial \mathcal{L}}{\partial A^{[l]}}, g'(Z^{[l]}) \in \mathcal{M}_{n_l \times m}(\mathbb{R})$ y $(A^{[l-1]})^T \in \mathcal{M}_{m \times n_{l-1}}(\mathbb{R})$, por lo que $\frac{\partial J}{\partial W^{[l]}} \in \mathcal{M}_{n_l \times n_{l-1}}(\mathbb{R})$. Observemos que el producto de matrices provoca que en cada componente se están sumando los valores correspondientes de todos los ejemplos, por lo que simplemente dividiendo por m obtenemos la media que necesitamos.

Para la derivada $\frac{\partial J}{\partial b^{[l]}}$ podemos hacer:

$$\frac{\partial J}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial A^{[l](i)}} \odot g'(Z^{[l](i)}) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial Z^{[l](i)}} \tag{50}$$

Donde, en este caso, estamos obteniendo los valores de cada ejemplo separados en columnas, por lo que debemos hacer la media de las columnas para obtener el valor de $\frac{\partial J}{\partial b^{[l]}} \in \mathcal{M}_{n_l \times 1}(\mathbb{R})$.

Por último, en las fórmulas anteriores se puede ver que no es necesario calcular $\frac{\partial J}{\partial A^{[l]}}$ para obtener $\frac{\partial J}{\partial W^{[l]}}$ y $\frac{\partial J}{\partial b^{[l]}}$, pero sí necesitamos las derivadas $\frac{\partial \mathcal{L}}{\partial A^{[l]}}$. Para conseguirlas podemos adaptar directamente la ecuación 43 y considerar:

$$\frac{\partial \mathcal{L}}{\partial A^{[l]}} = (W^{[l+1]})^T * \left(\frac{\partial \mathcal{L}}{\partial A^{[l+1]}} \odot g'(Z^{[l+1]}) \right) \tag{51}$$

Donde los valores correspondientes a cada ejemplo se guardan, también en este caso, separados por columnas, que es exactamente lo que nos interesa.

Finalmente, podemos resumir todo el algoritmo de propagación hacia atrás general, con cualquier número de ejemplos y en formato matricial, con el siguiente procedimiento.

Algoritmo de propagación hacia atrás

1) Calcular $\frac{\partial \mathcal{L}}{\partial a^{[L]}}$

2) Desde $l = L$ hasta 1, repetir:

a) Calcular $\frac{\partial J}{\partial W^{[l]}} = \frac{1}{m} \left(\frac{\partial \mathcal{L}}{\partial A^{[l]}} \odot g'(Z^{[l]}) \right) * (A^{[l-1]})^T$

b) Calcular $\frac{\partial J}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \mathcal{L}}{\partial A^{[l](i)}} \odot g'(Z^{[l](i)})$

c) Calcular $\frac{\partial \mathcal{L}}{\partial A^{[l]}} = (W^{[l+1]})^T * \left(\frac{\partial \mathcal{L}}{\partial A^{[l+1]}} \odot g'(Z^{[l+1]}) \right)$

3) Devolver $\frac{\partial J}{\partial W^{[l]}}$ y $\frac{J}{\partial b^{[l]}}$ para todo $l = 1, \ldots, L$.

3. Optimización del proceso de aprendizaje

En este capítulo nos centraremos en diferentes técnicas que nos permitirán optimizar, desde diferentes puntos de vista, el proceso de aprendizaje de las redes neuronales prealimentadas (*Feedforward Neural Networks*, FNN).

Veremos sólo algunas de las principales técnicas y más empleadas en la actualidad, ya que existe una gran multitud y diversidad de ellas, que hace imposible revisarlas todas en este texto.

Dividiremos la discusión sobre estas técnicas en función del objetivo principal que persiguen, que en general podemos agrupar en tres grandes bloques o problemáticas:

- Problemas de **rendimiento** (*performance*) de la red, donde el objetivo que perseguimos es mejorar la capacidad predictiva de la red.

- Problemas relacionados con la **velocidad de aprendizaje**. En este bloque revisaremos técnicas para reducir el tiempo necesario para el entrenamiento de una red neuronal.

- Problemas de **sobreentrenamiento** (*overfitting*). El sobreentrenamiento es un problema importante en redes neuronales (y otros modelos de aprendizaje automático) producido por el sobreajuste de la red a los datos de entrenamiento, causando un deterioro importante del rendimiento cuando se evalúa con otros datos distintos de los de entrenamiento. Es decir, la red no es capaz de generalizar de forma correcta.

Es importante remarcar que la mayoría de las técnicas que veremos tienen un objetivo principal, pero suelen tener efecto en más de uno. Es decir, aunque el objetivo principal de una técnica sea mejorar el rendimiento de la red, probablemente también tendrá efecto sobre la velocidad de aprendizaje o la capacidad de generalización.

Aunque veremos estas técnicas de forma individual, generalmente se aplican múltiples de ellas a la vez. Debido a la gran cantidad de opciones disponibles es tremendamente complejo de escoger los mejores parámetros para cada problema o conjunto de datos, ya que las posibles combinaciones son innombrables e imposibles de calcular si queremos explorar todas las combinaciones.

En este sentido, existen distintas técnicas para facilitar y automatizar este proceso. A continuación comentaremos dos de las principales y más simples, pero que son ampliamente utilizadas en la actualidad. Éste es un campo activo de investigación,

donde van apareciendo nuevas técnicas y mejoras sobre las anteriores.

- La técnica conocida como **búsqueda en cuadrícula** (*grid search*) (Snoek et al., 2012) crea una distribución uniforme de valores entre los distintos parámetros, de forma que se crea una "cuadrícula" uniforme sobre el espacio de posibles combinaciones.

- La **búsqueda aleatoria** (*random search*) (Bergstra & Bengio, 2012) se basa en escoger combinaciones de valores aleatorios dentro de unos rangos prefijados.

Lectura complementaria

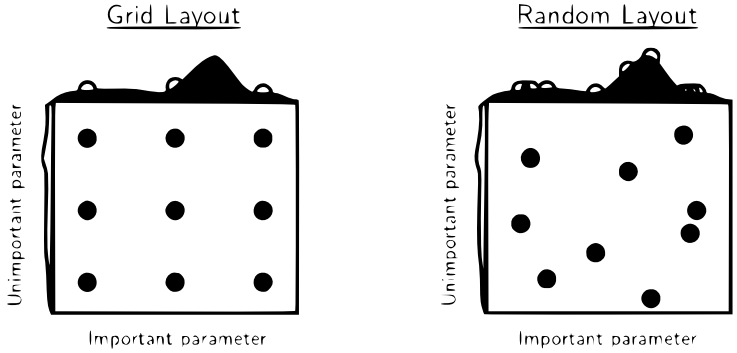
J. Snoek, H. Larochelle, R. P. Adams. "Practical bayesian optimization of machine learning algorithms". In Proc. of the NIPS '12, pp. 2951-2959, USA, 2012.

Lectura complementaria

J. Bergstra, Y. Bengio. "Random search for hyper-parameter optimization". J. Mach. Learn. Res., Vol. 13(1):281-305, 2012.

Aunque pueda parecer un contrasentido, la búsqueda aleatoria suele producir resultados similares e incluso superiores en muchas ocasiones. Por ejemplo, si suponemos que no todos los parámetros del modelo tienen la misma relevancia, y que algunos son más importantes, nos podemos encontrar en el escenario que se visualiza en la figura 8. Podemos ver que la distribución en cuadrícula del método *grid search* provoca que se analicen pocos valores de los parámetros importantes para el rendimiento del modelo, mientras que en una distribución aleatoria de los mismos es probable que se puedan alcanzar valores superiores en la función de optimización.

Figura 8. Ejemplo de comparación de los métodos *grid* y *random search* con nueve pruebas con cada uno



Fuente: (Bergstra & Bengio, 2012)

3.1. Técnicas relacionadas con el rendimiento de la red

En esta sección veremos algunas de las técnicas empleadas para mejorar el rendimiento (*performance*) de una red. Como ya hemos comentado, algunas de ellas también tienen efectos sobre el coste de la fase de entrenamiento o el sobreajuste a los datos de entrenamiento.

3.1.1. Arquitectura de la red

Hemos discutido algunos parámetros referentes al tamaño de las capas de entrada, ocultas y de salida en la Sección 1.2.1.

La flexibilidad de las redes neuronales es uno de sus principales virtudes, pero también

uno de sus principales inconvenientes. Hay muchos hiperparámetros para modificar: desde la arquitectura o topología de red (cómo se interconectan las neuronas), el número de capas, el número de neuronas por capa, el tipo de función de activación que se debe usar en cada capa, el método de inicialización de pesos, y mucho más.

Por lo tanto, una opción ampliamente utilizada consiste en usar un **método de búsqueda exhaustiva** de parámetros (como por ejemplo *grid search* o *random search*) con validación cruzada para encontrar los hiperparámetros correctos o, al menos, una buena combinación de ellos. El principal problema se debe a la gran cantidad de parámetros que debemos calibrar y el coste temporal del entrenamiento de una red neuronal. La combinación de ambos provoca que, en muchos casos, sólo podremos explorar una pequeña parte del espacio de los hiperparámetros en un tiempo razonable.

En referencia al **número de capas ocultas**, en muchos problemas se puede comenzar con una sola capa oculta y obtener unos resultados razonablemente buenos. En realidad, se ha demostrado que una red con una sola capa oculta puede modelar incluso las funciones más complejas, siempre que tenga suficientes neuronas. Durante largo tiempo, esto convenció a los investigadores de que no había necesidad de investigar redes neuronales más profundas. Sin embargo, descuidaron un detalle importante: las redes profundas tienen una eficiencia de parámetros mucho más alta que las superficiales. Es decir, pueden modelar funciones complejas utilizando exponencialmente menos neuronas que redes poco profundas, lo que las hace mucho más rápidas de entrenar.

En relación al **número de neuronas** empleadas en cada capa, es obvio que el número de neuronas en las capas de entrada y salida están determinados por el tipo de entrada y salida que requiere la tarea y la codificación empleada. En cuanto a las capas ocultas, una práctica común es dimensionarlas para formar un "embudo", es decir, la capa i tendrá (bastantes) más neuronas que la capa siguiente $i + 1$. Esta aproximación se basa en que muchas características de bajo nivel pueden unirse en muchas menos características de alto nivel. Por lo tanto, las capas iniciales trabajan con muchas características de bajo nivel, mientras que las capas ocultas cercanas a la capa de salida, trabajan con menos características de más alto nivel.

En referencia al uso de las **funciones de activación**, se suele usar la función de activación ReLU en las capas ocultas (o una de sus variantes). Esta función es un poco más rápida de computar que otras funciones de activación, y facilita la fase de entrenamiento evitando la saturación de las neuronas de las capas ocultas. Para la capa de salida, la función de activación *softmax* (que veremos en la sección 3.1.3) es generalmente una buena opción para tareas de clasificación cuando las clases son mutuamente excluyentes. Cuando no son mutuamente excluyentes (o cuando solo hay dos clases), generalmente se prefiere la función logística. Para las tareas de regresión, se suelen emplear funciones lineales de activación para la capa de salida, que permitan valores reales en la salida de la red.

3.1.2. Épocas, iteraciones y *batch*

En primer lugar, vamos a definir algunos sencillos conceptos importantes que nos resultarán de utilidad. En primer lugar, el concepto de **época** (*epoch*) se refiere a utilizar una única vez todo el conjunto de entrenamiento para entrenar la red neuronal. Veremos que pasar una única vez el conjunto de datos completo a través de una red neuronal (es decir, una época) no es suficiente y debemos pasar el conjunto de datos completo varias veces a la misma red neuronal para su entrenamiento.

Se puede utilizar un número épocas fijo, aunque no es sencillo determinar cuál es el valor óptimo. Otra estrategia más simple y eficaz consiste en terminar el proceso de aprendizaje de forma automática, por ejemplo cuando no se producen mejoras durante un número seguido de iteraciones. Debemos tener en cuenta que en algunos casos la mejora del entrenamiento puede "estancarse" durante unas cuantas iteraciones, para volver a mejorar a continuación. Es importante definir este error de forma correcta, que sea proporcional a los valores empleados en la salida y al cálculo del error de cada instancia de entrenamiento. Lógicamente, valores de umbral del error muy pequeños dificultarán el proceso de estabilización de la red, mientras que valores demasiado grandes producirán modelos poco precisos.

El descenso del gradiente (*gradient descent*) es un algoritmo de optimización que se usa a menudo para encontrar los pesos o coeficientes de los algoritmos de aprendizaje automático, como por ejemplo las redes neuronales y la regresión logística. Como hemos visto, su funcionamiento se basa en que el modelo haga predicciones sobre los datos de entrenamiento y use el error en las predicciones para actualizar el modelo, intentado reducir el error cometido en la predicción.

Existen variaciones de este algoritmo que podemos aplicar en el proceso de aprendizaje, cada uno de ellos con sus ventajas e inconvenientes:

- Descenso del gradiente estocástico (*Stochastic Gradient Descent*, SGD) es una variación que calcula el error y actualiza el modelo para cada ejemplo en el conjunto de datos de entrenamiento. Su principal problema radica en que actualizar el modelo con tanta frecuencia tiene un coste computacional muy elevado, lo que consume mucho más tiempo para entrenar a los modelos en grandes conjuntos de datos.

- Descenso del gradiente por lotes (*Batch Gradient Descent*, BGD) es otra variación que calcula el error para cada ejemplo en el conjunto de datos de entrenamiento, pero solo actualiza el modelo después de que se hayan evaluado todos los ejemplos de entrenamiento. Por lo tanto, el descenso de gradiente por lotes realiza actualizaciones del modelo al final de cada época de entrenamiento. Este método puede provocar una convergencia prematura del modelo hacia un conjunto de parámetros subóptimo.

- Descenso del gradiente por mini-lotes (*Mini-Batch Gradient Descent*, MBGD) di-

vide el conjunto de datos de entrenamiento en lotes pequeños que se utilizan para calcular el error y actualizar los coeficientes del modelo. Este método persigue el equilibrio entre la robustez del descenso de gradiente estocástico y la eficiencia del descenso de gradiente por lotes. Es la implementación más común actualmente en el campo del aprendizaje profundo. Es común utilizar un tamaño de lote (*batch size*) igual a 32, aunque puede variar dependiendo del problema y los datos concretos.

Finalmente, otro concepto importante se refiere al número de **iteraciones**, que se define como el número de lotes (*batches*) necesarios para completar una época. Por ejemplo, si dividimos un conjunto de datos de 1.600 ejemplos en lotes de 32, entonces se necesitarán 50 iteraciones para completar una época.

3.1.3. *Softmax*

Esta técnica define un nuevo tipo de neurona, a partir de la generalización de la neurona logística o sigmoide, a la que llamaremos neurona *softmax* y que está diseñada especialmente para la capa de salida de una red neuronal.

La función de entrada de una neurona *softmax* es la misma que en el caso de una neurona sigmoide. Es decir, la suma ponderada, que define el valor de entrada de la neurona j de la capa L de la siguiente forma:

$$z_j^L = \sum_k w_{jk}^L \, y_k^{L-1} + \alpha_j^L \tag{52}$$

donde w_{jk}^L representa el peso asignado a la conexión entre la neurona k de la capa $L-1$ y la neurona j de la capa L; y_k^{L-1} indica el valor de salida de la neurona k de la capa $L-1$; y α_j^L es el valor de sesgo de la neurona j de la capa L.

En este caso, en lugar de aplicar la función sigmoide al valor de entrada, i.e. $\sigma(z_j^L)$, se aplica la denominada función *softmax*:

$$y_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \tag{53}$$

donde el denominador suma sobre todas las neuronas de salida.

Los valores de salida de la capa de neuronas *softmax* tienen dos propiedades interesantes:

- Los valores de activación de salida son todos positivos, debido a que la función exponencial es siempre positiva.
- El conjunto de los valores de salida siempre suma 1, es decir, $\sum_j y_j^L = 1$.

En otras palabras, la salida de la capa *softmax* puede ser interpretada como una distribución de probabilidades. En muchos problemas es conveniente interpretar la activación de salida y_j^L como la estimación de la probabilidad de que la salida correcta sea la clase j.

3.1.4. Algoritmos de entrenamiento

Un aumento del rendimiento y de la velocidad de entrenamiento puede provenir del uso de un optimizador más rápido que el optimizador de gradiente de pendiente normal. En esta sección presentaremos algunos de los más populares. Una revisión completa queda fuera de los objetivos de este trabajo, pero puede consultarse, por ejemplo, en el trabajo de Ruder (2016).

Lectura complementaria

S. Ruder. "An overview of gradient descent optimization algorithms". CoRR, abs/1609.04747, 2016.

- **Momentum** (Qian, 1999) es un método de optimización que ayuda a acelerar el *stochastic gradient descent* (SGD) en la dirección relevante intentando "amortiguar" las oscilaciones. Esencialmente, podemos comparar este optimizador con el movimiento de una bola en una pendiente. La bola acumula impulso a medida que rueda cuesta abajo, cogiendo más y más velocidad en la dirección de máximo pendiente. Es decir, la aceleración aumenta para las dimensiones cuyos gradientes apuntan en las mismas direcciones y reduce las actualizaciones para las dimensiones cuyos gradientes cambian de dirección. Como resultado, ganamos una convergencia más rápida y una oscilación reducida.

Lectura complementaria

N. Qian. "On the momentum term in gradient descent learning algorithms". Neural Networks, Vol. 12(1):145-151, 1999.

- **Adagrad** (Duchi *et al.*, 2011) es un algoritmo para la optimización basada en gradientes que adapta la velocidad de aprendizaje a los parámetros, realizando actualizaciones más pequeñas (es decir, bajas tasas de aprendizaje) para los parámetros asociados con características frecuentes, y actualizaciones más grandes (es decir, altas tasas de aprendizaje) para los parámetros asociados con características infrecuentes. Por esta razón, es adecuado para tratar con datos dispersos.

Lectura complementaria

J. Duchi, E. Hazan, Y. Singer. "Adaptive subgradient methods for online learning and stochastic optimization". J. Mach. Learn. Res., Vol. 12:2121-2159, 2011.

- **Adadelta** (Zeiler, 2012) es una extensión de Adagrad que busca reducir su velocidad de aprendizaje agresiva y monótonamente decreciente. En lugar de acumular todos los gradientes pasados, Adadelta restringe la ventana de gradientes pasados acumulados a un tamaño de ventana fijo w.

Lectura complementaria

M. D. Zeiler. "ADADELTA: an adaptive learning rate method". CoRR, abs/1212.5701, 2012.

- **Adam** (*Adaptive Moment Estimation*) (Kingma & Ba, 2014), que representa la estimación del momento adaptativo, combina las ideas de optimización de Mo-

Lectura complementaria

D. P. Kingma, J. Ba. "Adam: A method for stochastic optimization". CoRR, abs/1412.6980, 2014.

mentum (seguimiento de un promedio de degradación exponencial decreciente de los gradientes pasados) y además mantiene un seguimiento de un promedio de decaimiento exponencial de los gradientes.

Aunque hasta hace unos años se recomendaba el uso del métodos de optimización adaptativos, como por ejemplo Adam, varios estudios recientes apuntan a que estos métodos pueden conducir a soluciones que no generalizan bien en algunos conjuntos de datos. Por lo tanto, es recomendable explorar otras opciones, como por ejemplo la optimización de Momentum.

3.2. Técnicas relacionadas con la velocidad del proceso de aprendizaje

En esta sección veremos algunas técnicas que han sido desarrolladas con el objetivo de mejorar la velocidad del proceso de entrenamiento, aunque claramente existe una correlación entre la velocidad de entrenamiento y el rendimiento del modelo.

3.2.1. Inicialización de los pesos de la red

Es habitual escoger aleatoriamente los pesos y el sesgo de las neuronas de forma aleatoria, generalmente utilizando valores aleatorios independientes de una distribución gaussiana con media 0 y la desviación estándar 1.

Una inicialización de los pesos más eficientes puede ayudar a la red, en especial a las neuronas de las capas ocultas, a reducir el efecto de aprendizaje lento, de forma similar a cómo el uso de la función de entropía cruzada (ver sección 3.2.3) ayuda a las neuronas de la capa de salida.

Una de las opciones más utilizadas consiste en inicializar los pesos de las neuronas en las capas ocultas de la red como variables aleatorias gaussianas con media 0 y desviación estándar $\frac{1}{\sqrt{n_{in}}}$, donde n_{in} es el número de entradas de la neurona. Con esto conseguiremos reducir la probabilidad de que las neuronas de las capas ocultas queden saturadas durante el proceso de entrenamiento, mejorando por lo tanto la velocidad en dicho proceso.

Es importante subrayar que la inicialización de pesos nos puede facilitar un aprendizaje más rápido, pero no implica necesariamente una mejora en el rendimiento final del proceso de entrenamiento.

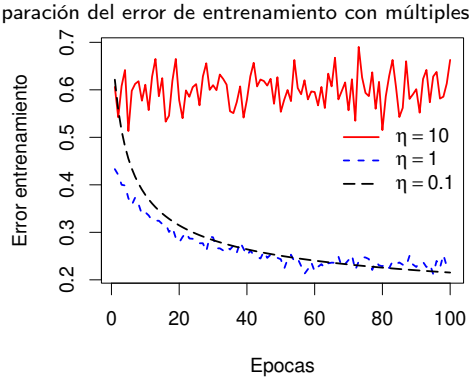
3.2.2. Velocidad de aprendizaje

Más allá de las optimizaciones vistas hasta este punto, debemos calibrar algunos parámetros, como por ejemplo la tasa de aprendizaje (η) o el parámetro de regularización (λ).

La **velocidad de aprendizaje** (determinada por el parámetro η) es un factor relevante sobre el proceso de aprendizaje de la red: valores muy altos de este parámetro pueden provocar que la red se vuelva inestable, es decir, se aproxima a valores mínimos, pero no es capaz de estabilizarse entorno a estos valores. En general, se recomienda empezar el proceso con velocidades de aprendizaje altas, e ir reduciéndolas para estabilizar la red.

Una buena estrategia, aunque no la única ni necesariamente la mejor, es empezar por la calibración de la tasa de aprendizaje. En este sentido es recomendable realizar un conjunto de pruebas con distintos valores de η, que sean sensiblemente diferentes entre ellos. Por ejemplo, la Figura 9 muestra el error en la función de coste durante el entrenamiento utilizando tres valores muy distintos para el parámetro η. Con el valor $\eta = 10$ podemos observar que el valor de error en el coste realiza considerables oscilaciones desde el inicio del proceso de entrenamiento, sin una mejora aparente. El segundo valor testado, $\eta = 1$, reduce el error de la función de coste en las primeras etapas del entrenamiento, pero a partir de cierto punto oscila suavemente de forma aleatoria. Finalmente, en el caso de utilizar $\eta = 0{,}1$ vemos que los valores de error decrecen de forma progresiva y suave durante todo el proceso de prueba. Se suele llamar "umbral de aprendizaje" al valor máximo de η que produce un decrecimiento durante las primeras etapas del proceso de aprendizaje. Una vez determinado este valor de umbral, es aconsejable seleccionar valores de η inferiores, en general, una o dos magnitudes inferiores, para asegurar un proceso de entrenamiento adecuado.

Figura 9. Comparación del error de entrenamiento con múltiples valores de η



Una vez se ha determinado la tasa de aprendizaje, ésta puede permanecer fija durante todo el proceso. Aún así, tiene sentido reducir su valor para realizar ajustes más finos en las etapas finales del proceso de entrenamiento. En este sentido, se puede establecer una tasa de aprendizaje variable, que decrezca de forma proporcional al error de la función de coste.

El **parámetro de regularización** (λ) se suele desactivar para realizar el calibrado de la tasa de aprendizaje, i.e. $\lambda = 0$. Una vez se ha establecido la tasa de aprendizaje, podemos activar la regularización con un valor de $\lambda = 1$ e ir incrementando y decrementando el valor según las mejoras observadas en el rendimiento de la red.

3.2.3. Función de entropía cruzada

En algunos casos, el proceso de aprendizaje es muy lento en las primeras etapas de entrenamiento. Esto no significa que la red no esté aprendiendo, pero lo hace de forma muy lenta en las primeras etapas de entrenamiento, hasta alcanzar un punto donde el aprendizaje se acelera de forma considerable. Cuando usamos la función de coste cuadrática, el aprendizaje es más lento cuando la neurona produce resultados muy dispares con los resultados esperados. Esto se debe a que la neurona sigmoide de la capa de salida se encuentra saturada en los valores extremos, ya sea en el valor 0 o 1, y por lo tanto, los valores de sus derivadas en estos puntos son muy pequeños, produciendo pequeños cambios en la red que generan un aprendizaje lento.

En estos casos, es preferible utilizar otra función de coste, que permita que las neuronas de la capa de salida puedan aprender más rápido, aún estando en un estado de saturación. Definimos la **función de coste de entropía cruzada** (*cross-entropy cost function*, en inglés) para una neurona como:

$$C = -\frac{1}{n}\sum_{d}[c\ ln(y) + (1-c)ln(1-y)] \tag{54}$$

donde n es el número total de instancias de entrenamiento, d es cada una de las instancias de entrenamiento, c es la salida deseada correspondiente a la entrada d, e y es la salida obtenida.

Generalizando la función para una red de múltiples neuronas y capas, obtenemos la siguiente expresión para la función de coste:

$$C = -\frac{1}{n}\sum_{d}\sum_{j}[c_j\ ln(y_j^L) + (1-c_j)ln(1-y_j^L)] \tag{55}$$

donde $c = c_1, c_2, \ldots$ son los valores deseados y $y_1^L, y_2^L, \ldots$ son los valores actuales en las neuronas de la capa de salida.

Utilizando la función de coste de entropía cruzada, el aprendizaje es más rápido cuando la neurona produce valores alejados de los valores esperados. Este comportamiento no está relacionado con la tasa de aprendizaje, con lo cual no se resuelve modificando éste valor.

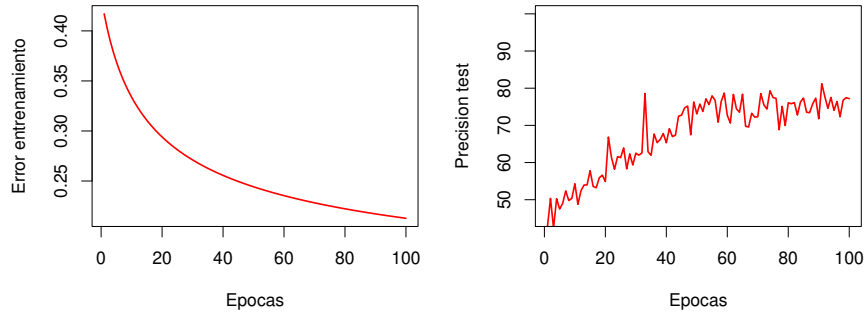
En general, utilizar la función de coste basada en la entropía cruzada suele ser la

mejor opción, siempre que las neuronas de salida sean neuronas sigmoides, dado que en la inicialización de pesos puede producir la saturación de neuronas de salida cerca de 1, cuando deberían ser 0, o viceversa. En estos casos, si se utiliza la función de coste cuadrático el proceso de entrenamiento será más lento, aunque no nulo. Pero obviamente, es deseable un proceso de entrenamiento más rápido en las primeras etapas.

3.3. Técnicas relacionadas con el sobreentrenamiento

Supongamos que tenemos una red y que analizamos el error en el conjunto de entrenamiento durante las 100 primeras épocas del proceso (*epochs*, en inglés). La Figura 10 (a) muestra un posible resultado, que *a priori* parece bueno, ya que el error de la red desciende de forma gradual conforme avanza el proceso de entrenamiento. Pero si también analizamos la precisión del conjunto de test asociado a cada etapa del proceso de entrenamiento, podemos obtener unos datos similares a los presentados por la Figura 10 (b). En este caso vemos que la precisión de la red mejora hasta llegar a la época 50, aproximadamente, donde empieza a fluctuar pero no se aprecia una mejora significativa en la precisión de la red.

Figura 10. Ejemplo de sobreentrenamiento



(a) Error en el conjunto de entrenamiento (b) Precisión en el conjunto de test

Por lo tanto, nos encontramos ante un ejemplo donde el error de entrenamiento nos dice que la red está mejorando a cada etapa de entrenamiento, mientas que el conjunto de test nos dice que a partir de cierto punto no hay mejora alguna en la precisión de la red. Nos encontramos ante un caso de **sobreespecialización** o **sobreentrenamiento** (*overfitting* o *overtraining*, respectivamente) de la red.

La red reduce el error de entrenamiento porqué se va adaptando a los datos del conjunto de entrenamiento, pero ya no es capaz de generalizar correctamente para datos nuevos, en este caso el conjunto de test, y los resultados que produce no mejoran con los nuevos datos.

El sobreentrenamiento es un problema importante en las redes neuronales. Esto es especialmente cierto en las redes modernas, que a menudo tienen un gran número de pesos y sesgos. Para entrenar con eficacia, necesitamos una forma de detectar cuando se está sobreentrenando.

Aumentar la cantidad de datos de entrenamiento es una forma de reducir este problema. Otro enfoque posible es reducir el tamaño de la red. Sin embargo, las redes grandes tienen el potencial de ser más poderosas que las redes pequeñas.

Afortunadamente, existen otras técnicas que pueden reducir el sobreentrenamiento, incluso cuando tenemos una red fija y datos de entrenamiento fijos. Estas técnicas son conocidas como **técnicas de regularización**. A continuación veremos algunas de las más empleadas y que mejores resultados proporcionan, aunque no es una lista extensiva ni completa de todas las técnicas de regularización existentes.

3.3.1. Regularización L2

A continuación veremos una de las técnicas de regularización más utilizadas, conocida como regularización L2 (*L2 regularization* o *weight decay*, en inglés).

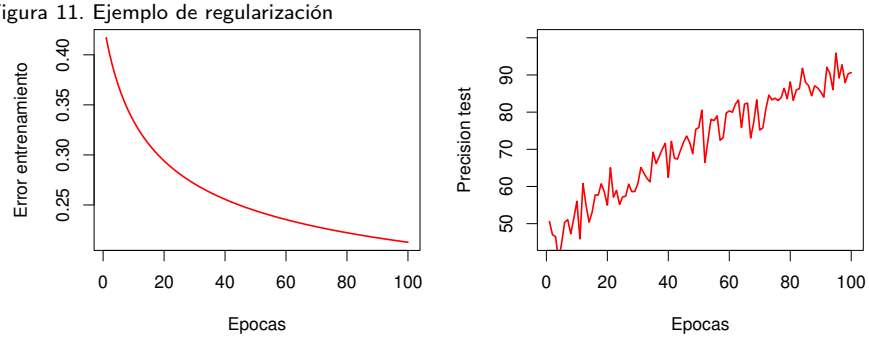
La idea de la regularización L2 es añadir un término extra a la función de coste, llamado el término de regularización:

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2 \tag{56}$$

donde C_0 es la función de coste original, i.e. no regularizada.

El objetivo de la regularización es que la modificación de pesos en la red se realice de forma gradual. Es decir, la regularización puede ser vista como una forma de equilibrio entre encontrar pesos pequeños y minimizar la función de coste. La importancia relativa de los dos elementos del equilibrio depende del valor de λ: cuando éste es pequeño, se minimiza la función de coste original; en caso contrario, se opta por premiar los pesos pequeños.

Figura 11. Ejemplo de regularización



(a) Error en el conjunto de entrenamiento (b) Precisión en el conjunto de test

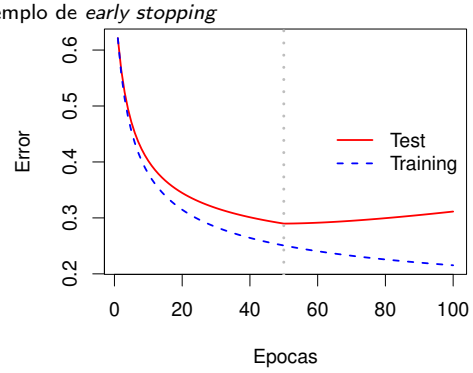
La Figura 11 muestra, empíricamente, que la regularización está ayudando a que la red generalice mejor y reduzca, por lo tanto, los efectos del sobreentrenamiento. Como se puede ver en la figura, la precisión en el conjunto de test continua mejorando junto con el error en el conjunto de entrenamiento, signo de que no se está produciendo sorbreentrenamiento. Se ha demostrado, aunque sólo de forma empírica, que las redes neuronales regularizadas suelen generalizar mejor que las redes no regularizadas.

Existen muchas otras técnicas de regularización distintas de la regularización L2, como por ejemplo la regularización L1 (*L1 regularization*).

3.3.2. *Early stopping*

Una técnica simple, pero muy efectiva, consiste en parar el proceso de aprendizaje en el momento en que el error en el conjunto de validación alcanza el valor mínimo. Esta técnica es conocida como *early stopping*, e implica que durante el proceso de entrenamiento se deberá ir testeando el modelo con los datos de validación cada cierto tiempo, para poder detectar cuando el error en este conjunto de validación aumenta.

Figura 12. Ejemplo de *early stopping*



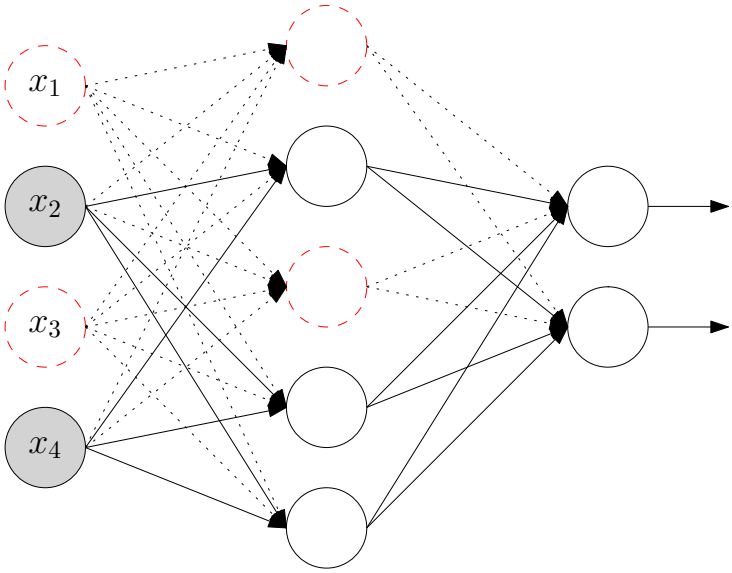
La Figura 12 muestra un ejemplo, en el que un modelo ha sido entrenado durante un cierto período. Al inicio del entrenamiento, el error en el conjunto de datos de entrenamiento se va reduciendo, al igual que el mismo error en el conjunto de datos de validación. Pero llega un momento en el cual el error sobre el conjunto de datos de validación deja de disminuir y, a continuación, empieza a aumentar. Este indica que el modelo se está sobreajustando a los datos de entrenamiento y pierde capacidad de generalización. Utilizando la técnica de *early stopping*, el modelo debe terminar el entrenamiento cuando detecta que el error en los datos de validación está aumentando.

3.3.3. Dropout

La técnica conocida como *dropout* (Srivastava *et al.*, 2014), aunque extremadamente simple, ofrece unos resultados muy destacados. El proceso es el siguiente: en cada etapa de entrenamiento se asigna una probabilidad p a cada una de las neuronas (incluyendo las de la capa de entrada, pero excluyendo las neuronas de la capa de

salida) de ser "temporalmente eliminadas", tal y como se muestra en la Figura 13. Es decir, un subconjunto de las neuronas de la red será ignoradas en cada etapa del entrenamiento. El resultado es que en cada etapa del entrenamiento se modifica la arquitectura de la red. El parámetro p se denomina *dropout rate* y, aunque su valor depende de la arquitectura y el conjunto de datos, un valor utilizado habitualmente es $p = 0{,}5$.

Figura 13. Ejemplo de *dropout*. Las neuronas mostradas en linea discontinua están desactivadas



Lectura complementaria

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". Journal of Machine Learning Research, Vol. 15:1929- 1958, 2014.

Es importante remarcar que este proceso sólo se aplica durante el entrenamiento. Después de éste, todas las neuronas de la red se mantienen activas. Este proceso permite aumentar sensiblemente la capacidad de generalización de la red.

3.3.4. Expansión artificial del conjunto de datos

Una última técnica de regularización, conocida como conocida como expansión artificial del conjunto de datos (*artificially expanding the training data* o *data augmentation*), consiste en generar nuevas instancias de entrenamiento a partir de las existentes, aumentando artificialmente el tamaño del conjunto de entrenamiento.

Para que esta técnica nos ayude a reducir el sobreentrenamiento es importante generar instancias de entrenamiento realistas. Por ejemplo, si el modelo está destinado a clasificar imágenes de animales, podemos desplazar, rotar y cambiar el tamaño de cada imagen en el conjunto de entrenamiento en varias cantidades y agregar las imágenes resultantes al conjunto de entrenamiento. Esto facilita que el modelo sea más tolerante con la posición, la orientación y el tamaño de los objetos en la imagen. Si deseamos que el modelo sea más tolerante a las condiciones de iluminación, también podemos generar imágenes con distintos contrastes e intensidades de luz. Al combinar estas transformaciones, puede aumentar considerablemente el tamaño del conjunto de entrenamiento.

3.4. Relación de técnicas y rendimiento de la red

La Tabla 1 sintetiza los objetivos de las técnicas de optimización que hemos revisado en este capítulo.

Tabla 1. Principales objetivos de las técnicas de optimización

Técnicas	Mejoras rendimiento	Velocidad aprendizaje	Overfitting
Arquitectura de la red	X	X	X
Épocas, iteraciones y batch		X	
Softmax	X		
Algoritmos de entrenamiento	X	X	
Inicialización pesos de la red		X	
Velocidad de aprendizaje	X	X	
Función de entropía cruzada		X	
Regularización L2			X
Early stopping			X
Dropout			X
Expansión conjunto de datos	X		X

4. Autoencoders

Los *autoencoders* son un tipo especial de redes neuronales *fully-connected* que funcionan intentando reproducir los datos de entrada en la salida de la red. Aunque este proceso pueda sonar "trivial" e incluso inútil, la arquitectura de estas redes permite que puedan realizar algunas tareas muy interesantes que las diferencian de las redes neuronales que hemos visto hasta ahora.

En concreto, tres de sus principales aplicaciones son:

- Reducción de la dimensionalidad. Las redes se entrenan con conjuntos de datos no etiquetados (no supervisados) para aprender una representación eficiente y reducida de los datos de entrada, llamada *codings*, que permite reducir la dimensión de los datos de entrada.

- Pre-entrenamiento de redes neuronales. Los *autoencoders* pueden facilitar la detección de los atributos más relevantes, y pueden ser empleadas para pre-entrenamiento no supervisado de redes neuronales.

- Finalmente, también pueden ser utilizados para la generación de nuevos datos sintéticos que permitan aumentar el conjunto de datos de entrenamiento. Es decir, pueden ser empleados para generar datos "similares" a los que reciben como entrada, de tal forma que luego se pueden emplear para el entrenamiento de otras redes neuronales (u otros algoritmos de aprendizaje automático).

En este capítulo veremos el funcionamiento general de un *autoencoder*, así como sus principales arquitecturas y aplicaciones concretas.

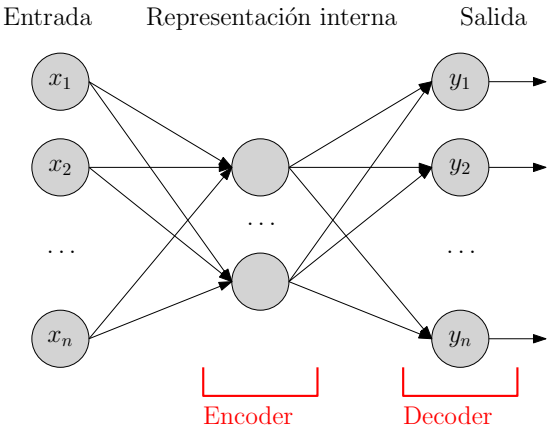
4.1. Estructura básica

Un *autoencoder* siempre presenta dos partes claramente diferenciadas:

- **Codificador** (*encoder*), que es el encargado de convertir las entradas a una representación interna, generalmente de menor dimensión que los datos de entrada. A veces también recibe el nombre de "red de reconocimiento".

- **Decodificador** (*decoder*), que se encarga de transformar la representación interna a la salida de la red. También puede recibir el nombre de "red generativa".

Figura 14. Estructura básica de un *autoencoder*



La figura 14 muestra la estructura básica de un *autoencoder*. Aunque pueda parecer similar a la estructura de una red como las que hemos visto anteriormente (ambas son *feed forward* y *fully-connected*), tiene algunas diferencias relevantes. En primer lugar, los *autoencoders* tienen el mismo número de neuronas en la capa de salida que en la capa de entrada, ya que en *autoencoder* intentará reproducir en la salida la entrada que ha recibido. En segundo lugar, la capa oculta (o capas ocultas) deben tener un número de neuronas inferior a las capas de entrada y salida, ya que en caso contrario la tarea de reproducir la entrada en la salida sería trivial. Por lo tanto, la representación interna debe preservar la información de entrada en un formato de menor dimensionalidad.

Los conceptos vistos anteriormente sobre la inicialización de parámetros, funciones de activación, regularización, etc. también son aplicables en el caso de los *autoencoders*. La principal diferencia es que en este caso no se utiliza la clase o valor objetivo del conjunto de entrenamiento. La salida deseada (y que emplearemos para calcular el error que comete la red) será la misma entrada. Por este motivo, no se suelen emplear neuronas *softmax* en la capa de salida.

Figura 15. Ejemplos de reconstrucción de datos mediante un *autoencoder* con una sola capa oculta de 32 neuronas. La fila superior muestra los datos originales, mientras que la fila inferior muestra los datos reconstruidos

La figura 15 muestra el resultado de la reconstrucción de datos del conjunto de dígitos MNIST empleando un *autoencoder* con una sola capa oculta de 32 neuronas. Es decir, en este ejemplo los datos de entradas (784 atributos por cada imagen) se reducen hasta emplear sólo 32 atributos, para luego volver a expandirse hasta la misma dimensión que los datos originales.

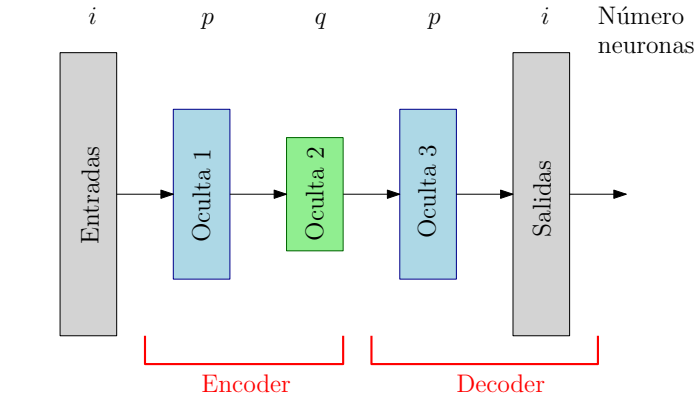
Conjunto de datos

La base de datos MNIST es una gran base de datos de dígitos escritos a mano que se usa comúnmente para entrenar varios sistemas de procesamiento de imágenes.

4.1.1. *Stacked autoencoders*

Al igual que las redes neuronales que hemos visto anteriormente, un *autoencoder* puede tener múltiples capas ocultas. En este caso, reciben el nombre de *autoencoders* apilados (*stacked autoencoders*) o *autoencoders* profundos (*deep autoencoders*).

Figura 16. Estructura básica de un *stacked autoencoder*



Generalmente, las dimensiones de las capas ocultas suelen ser simétricas respecto a la capa central (que contiene la representación más comprimida y que suele recibir el nombre de *codings*) y suelen reducirse hasta llegar a la capa central, para expandirse luego hasta la salida de la red. Es decir, según la estructura básica representada en la figura 16, se debe cumplir que $i > p > q$.

El aumento de las capas ocultas de los *autoencoders*, al igual que en las redes neuronales que hemos visto, permite crear representaciones más complejas y abstractas de los datos, pero en exceso también puede provocar problemas de sobreentrenamiento, que se traducen en una generalización pobre ante nuevos datos no vistos en el período de entrenamiento.

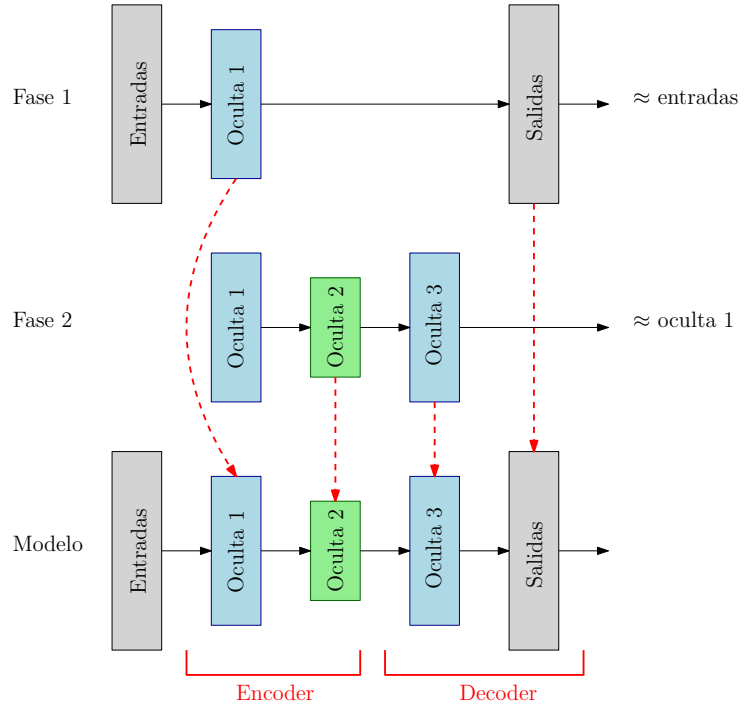
Una técnica común en los *stacked autoencoders* es utilizar los mismos pesos y *bias* en las capas simétricas, de esta forma se reduce a la mitad las variables que la red debe entrenar, aumentando la velocidad del entrenamiento y reduciendo el riesgo de sobreentrenamiento.

4.2. Entrenamiento de un *autoencoder*

El entrenamiento de un *autoencoder* que sólo contenga una (o pocas) capas ocultas (a veces también conocidos como *shallow autoencoders*) se suele realizar de la forma similar al entrenamiento visto en los capítulos anteriores.

En el caso de *autoencoders* que contengan múltiples capas ocultas (*stacked autoencoders*) se suele aplicar un proceso de entrenamiento por partes que permite reducir el tiempo de entrenamiento. Es decir, se van entrenando las capas más profundas a partir de los resultados del entrenamiento previo de las capas más superficiales.

Figura 17. Esquema del entrenamiento iterativo de un *stacked autoencoder*



La figura 17 muestra las fases de entrenamiento iterativo (o por partes) de *stacked autoencoder* y el resultado final. Como se puede ver, en la primera fase de entrenamiento sólo se incluyen las capas de entrada, la primera capa oculta y la capa de salida. En esta fase en *autoencoder* aprende a reconstruir las entradas a partir de la codificación de la primera capa oculta. A continuación, en la segunda fase de entrenamiento, se emplea la salida de la primera capa oculta para entrenar la segunda capa oculta, de menor dimensión que la primera. El objetivo es que la salida de la tercera capa oculta sea similar a los valores obtenidos en la primera capa, de forma que realiza una "compresión" mayor de los datos, ya que la dimensión de la segunda capa oculta es menor que la primera y tercera.

Finalmente, para construir el *stacked autoencoder*, se copian los valores de pesos y sesgo (*bias*) obtenidos tras el entrenamiento para construir el *autoencoder* final (líneas punteadas en la figura 17). De esta forma, se "apilan" las capas previamente entrenadas para construir el modelo final. De aquí el nombre de *stacked autoencoder*.

4.3. Pre-entrenamiento utilizando *autoencoders*

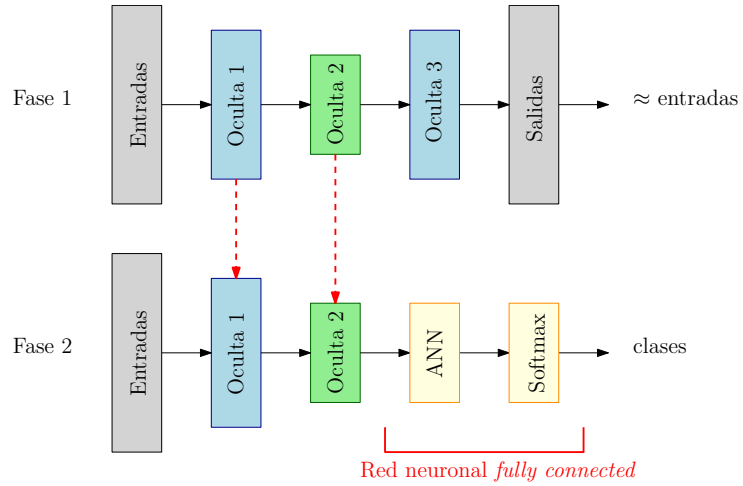
El pre-entrenamiento con *autoencoders* es una técnica que permite reducir la dimensionalidad de los datos antes del entrenamiento del modelo que se encargará de la tarea de clasificación o regresión.

Aunque tradicionalmente se ha aplicado esta técnica para reducir el tiempo de entrenamiento (de forma similar al uso de algoritmos de reducción de dimensionalidad), los avances y mejoras en la capacidad de cálculo han provocado que actualmente no se utilice demasiado para mejorar la velocidad de los procesos de entrenamiento. Aún

así, esta técnica continua siendo muy popular cuando tenemos un conjunto de datos con pocas muestras (n) en relación al número de atributos (m). Esta técnica permite mejorar la relación entre registros y atributos.

La idea de este esquema, tal y como se muestra en la figura 18, es bastante simple, aunque también muy potente. En primer lugar, se debe entrenar el *autoencoder*, ya sea con una o más capas ocultas, empleando todos los datos disponibles (ya sean etiquetados o no).

Figura 18. Esquema de pre-entrenamiento utilizando un *autoencoder*



Una vez finalizado en entrenamiento del *autoencoder*, se copian los valores de pesos y *bias* de las capas iniciales, hasta de capa de menor dimensionalidad (que hemos llamado *codings*). A la salida de la capa de *codings* se construye la red neuronal (generalmente una red *fully connected*, aunque podría ser otro modelo de aprendizaje automático). La salida de esta segunda red serán las clases o valores objetivo del problema a resolver, y por lo tanto, el entrenamiento se realizará de forma similar a cómo hemos visto anteriormente. Es importante destacar que durante el entrenamiento de esta segunda red, los valores obtenidos (pesos y *bias*) del *autoencoder* suelen mantenerse fijos.

Es interesante remarcar que en este esquema estamos mezclando el entrenamiento no supervisado del *autoencoder* (fase 1) con el entrenamiento supervisado del modelo final (fase 2).

Este proceso de pre-entrenamiento puede ser muy útil en diferentes contextos, como por ejemplo:

- Cuando se dispone de un conjunto pequeño de datos etiquetados con una gran dimensionalidad. Al reducir el número de atributos con los que trabaja la red neuronal, facilitamos su tarea de aprendizaje.

- Cuando de dispone de un conjunto de datos grande, pero en el cual sólo una parte está etiquetada. En estos casos es posible emplear todo el conjunto de datos para

entrenar el *autoencoder* (y conseguir una buena representación de los datos) y luego entrenar el modelo empleando sólo los datos etiquetados disponibles.

En el caso de redes profundas, el pre-entrenamiento mediante métodos no supervisados es especialmente relevante, y ha sido uno de los elementos que ha contribuido a la popularización de estos modelos (Bengio et al., 2007).

Lectura complementaria

Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle. "Greedy layer-wise training of deep networks". Advances in Neural Information Processing Systems 19, pp. 153-160. MIT Press, 2007.

4.4. Tipos de *autoencoders*

A partir de la descripción general que hemos visto, aparecen muchos otros tipos de *autoencoders*, que a partir de variaciones en el esquema o en el entrenamiento, pretenden resolver las mismas problemáticas que en los casos anteriores.

Aunque queda fuera del alcance de este texto una revisión exhaustiva de todos los tipos existentes, a continuación veremos algunos de los principales.

4.4.1. *Denoising autoencoders*

Una forma de intentar forzar a los *autoencoders* para que aprendan codificaciones más robustas de los datos de entrada, consiste en añadir ruido de forma aleatoria en las entradas. El objetivo es que el modelo sea capaz de reproducir la entrada, eliminando del ruido que se ha introducido, a partir de los patrones de los datos (Vincent *et al.*, 2010).

Lectura complementaria

P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, P.-A. Manzagol. "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion". J. Mach. Learn.Res., Vol. 11:3371-3408, 2010.

Básicamente existe dos formas de introducir ruido en los datos de entrada:

- Añadiendo ruido aleatorio en la capa de entrada. Por ejemplo, empleando un generador de ruido gaussiano.
- Eliminando algunas entradas de forma aleatoria. Este proceso es similar al *dropout* que hemos visto anteriormente, pero se emplea en la capa de entrada.

4.4.2. *Variational autoencoders*

Los *variational autoencoders* (Kingma & Welling, 2013) tienen dos diferencias importantes respecto a los demás modelos vistos hasta ahora.

Lectura complementaria

D. P. Kingma, M. Welling. "Auto-encoding variational bayes". CoRR, abs/1312.6114, 2013.

- En primer lugar, son modelos probabilísticos. Es decir, la salida de estos modelos es, en parte, estocástica incluso después del entrenamiento.
- En segundo lugar, el hecho de ser estocásticos les permite funcionar como modelos generativos, es decir, son capaces de generar nuevas instancias de datos similares

a los datos del conjunto de entrenamiento.

La estructura general de estos modelos es similar a los esquemas vistos anteriormente, pero presenta diferencias importantes en la capa central (*coding*). En lugar de producir una codificación para la instancia de entrada, produce una codificación media (μ) y una desviación estándar (σ). Entonces, la codificación se genera a partir de una distribución gaussiana con media μ y desviación estándar σ. A partir de aquí el proceso es similar al visto anteriormente, y se procede a la parte de decodificación según hemos presentado anteriormente.

4.4.3. Otros tipos de *autoencoders*

Hemos revisado los principales modelos de *autoencoders* existentes, aunque existen muchos otros que buscan mejorar los resultados obtenidos en problemas o funciones específicas. A pesar de que una revisión extensa queda fuera del alcance de este texto, a continuación incluimos algunos modelos adicionales que son relevantes y que es interesante conocer:

- *Sparse autoencoders*, que pretenden reducir el número de neuronas activas en la capa central (*codings*) para que la red aprenda a representar la información de la entrada empleando un número menor de neuronas.

- El *Contractive autoencoder* (Rifai *et al.*, 2011) intenta forzar al modelo para que genere codificaciones similares para valores de entrada similares.

- *Stacked convolutional autoencoders* (Masci *et al.*, 2011), especializado en extraer atributos visuales de imágenes a partir de capas convolucionales.

- Las *Generative stochastic networks* (GSN) (Alain *et al.*, 2016) son una generalización de los *denoising autoencoders* que permiten la generación de nuevos datos.

Lectura complementaria

S. Rifai, P. Vincent, X. Muller, X. Glorot, Y. Bengio. "Contractive auto-encoders: Explicit invariance during feature extraction". In Proc of the ICML '11, pp. 833-840, USA, 2011.

Lectura complementaria

J. Masci, U. Meier, D. Ciresan, J. Schmidhuber. "Stacked convolutional auto-encoders for hierarchical feature extraction". In Proc. of the ICANN '11, pp. 52-59, Berlin, Heidelberg, 2011.

Lectura complementaria

G. Alain, Y. Bengio, L. Yao, J. Yosinski, E. Thibodeau-Laufer, S. Zhang, P. Vincent. "GSNs: generative stochastic networks". Information and Inference, 2016.

Resumen

En este texto hemos presentado una introducción a las redes neuronales. Hemos iniciado este módulo con un repaso a los conceptos fundamentales, tales como la estructura de una neurona, las principales funciones de activación, etc. Esto nos ha permitido comprender los fundamentos y principios de funcionamiento de las redes neuronales artificiales, incluyendo las principales arquitecturas de redes neuronales. Las redes neuronales artificiales permiten resolver problemas relacionados con la clasificación y predicción. Una de sus principales ventajas es que son capaces de lidiar con problemas de alta dimensionalidad y encontrar soluciones basadas en hiperplanos no lineales. Por el contrario, dos de sus principales problemas o inconvenientes son: en primer lugar, el conocimiento está "oculto" en las redes, y en determinados casos puede ser complejo explicitar este conocimiento de forma clara. En segundo lugar, la preparación de los datos de entrada y salida no es una tarea trivial. Es recomendable asegurarse que los datos de entrada se encuentren en el intervalo $[0, 1]$, lo cual implica unos procesos previos de transformación de los datos.

En el segundo capítulo nos hemos centrado en el algoritmo de retropropagación (*back-propagation*), donde hemos entrado en todo el detalle matemático sobre su formulación y funcionamiento.

A continuación, en el tercer capítulo de este módulo didáctico hemos visto un conjunto de técnicas diseñadas para optimizar el proceso de aprendizaje, incluyendo mejoras en el rendimiento de las redes neuronales, mejoras en la velocidad del proceso de aprendizaje, y técnicas para reducir o evitar el problema del sobreentrenamiento (*overfitting*).

Para finalizar este módulo didáctico, hemos presentado un tipo particular de redes neuronales, los *autoencoders*. En concreto, hemos introducido la estructura básica y el funcionamiento de los autoencoders, así como las peculiaridades de su proceso de entrenamiento y uso de modelos pre-entrenados.

Glosario

Aprendizaje automático El aprendizaje automático (más conocido por su denominación en inglés, *machine learning*) es el conjunto de técnicas, métodos y algoritmos que permiten a una máquina aprender de manera automática en base a experiencias pasadas.

Aprendizaje no supervisado El aprendizaje no supervisado se basa en el descubrimiento de patrones, características y correlaciones en los datos de entrada, sin intervención externa.

Aprendizaje supervisado El aprendizaje supervisado se basa en el uso de un componente externo, llamado supervisor, que compara los datos obtenidos por el modelo con los datos esperados por éste, y proporciona retroalimentación al modelo para que vaya ajustándose y mejorando las predicciones.

MNIST La base de datos MNIST es una gran base de datos de dígitos escritos a mano que se usa comúnmente para entrenar varios sistemas de procesamiento de imágenes.

Bibliografía

Y. Bengio, I. Goodfellow, A. Courville (2016). *Deep Learning*. Cambridge, MA: MIT Press.

Simon O. Haykin (2009). *Neural Networks and Learning Machines,3rd Edition*. Pearson.

Michael A. Nielsen (2015). *Neural Networks and Deep Learning*, Determination Press.

Aurélien Géron (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition*, O'Reilly Media, Inc.