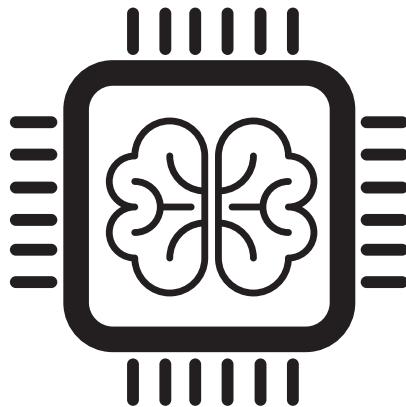


TECNOLOGÍA

ANNA BOSCH RUÉ  
JORDI CASAS ROMA  
TONI LOZANO BAGÉN

# DEEP LEARNING

## PRINCIPIOS Y FUNDAMENTOS



EDITORIAL UOC





*Deep learning*  
Principios y fundamentos

Anna Bosch Rué  
Jordi Casas Roma  
Toni Lozano Bagén

12 de julio de 2019

Director de la colección Manuales (Tecnología): Antoni Pérez

Diseño de la colección: Fundació per a la Universitat Oberta de Catalunya  
Diseño de la cubierta: Natàlia Serrano  
Pictograma de cubierta: Freepik

Primera edición en lengua castellana: diciembre 2019  
Primera edición en formato digital (PDF): enero 2020

© Anna Bosch Rué, Jordi Casas Roma, Toni Lozano Bagén, del texto  
© Fundació per a la Universitat Oberta de Catalunya, de esta edición, 2019  
Avinguda del Tibidabo, 39-43 (08035 Barcelona).  
Marca comercial: Editorial UOC  
<http://www.editorialuoc.com>

Realización editorial: Reverté-Aguilar, S. L.  
ISBN: 978-84-9180-657-8

Ninguna parte de esta publicación, incluyendo el diseño general y de la cubierta, no puede ser copiada, reproducida, almacenada o transmitida de ninguna forma ni por ningún medio, ya sea eléctrico, químico, mecánico, óptico, de grabación, de fotocopia o por otros métodos, sin la autorización previa por escrito de los titulares del *copyright*.

## ***Autores***

### **Anna Bosch Rué**

Actualmente trabaja como VP Data Intelligence en Launchmetrics. Originalmente de Girona, se graduó como ingeniera informática (2003). Posteriormente, bajo la supervisión del Dr. Xavier Muñoz y el Prof. Andrew Zisserman, realizó un doctorado conjunto entre la Universidad de Girona (UdG) y la Universidad de Oxford (2007) con la calificación de *cum laude* por unanimidad. En 2008 realizó un posdoctorado en la Universidad de Oxford. Antes de unirse al equipo Launchmetrics, trabajó en los departamentos de I+D Mediapro y Blue Room Innovation. Sus intereses en investigación se centran alrededor de analizar grandes volúmenes de datos para extraer conocimiento, aplicando técnicas de *machine learning*, *data mining* y, más recientemente, *deep learning*.

### **Jordi Casas Roma**

Licenciado en Ingeniería Informática por la Universitat Autònoma de Barcelona (UAB), máster en Inteligencia artificial avanzada por la Universidad Nacional de Educación a Distancia (UNED) y doctor en Informática por la UAB. Desde 2009 ejerce como profesor en los Estudios de Informática, Multimedia y Telecomunicación de la Universitat Oberta de Catalunya (UOC) y, recientemente, también como profesor asociado en la UAB. Es director del máster universitario en Ciencia de datos de la UOC. Sus actividades docentes se centran en la minería de datos, el aprendizaje automático (*machine learning*) y el análisis de redes o grafos (*graph mining*). Desde 2010 pertenece al grupo de investigación KISON (K-ryptography and Information Security for Open Networks). Sus intereses de investigación incluyen temas relacionados con la privacidad, el aprendizaje automático y la minería de grafos.

### **Toni Lozano Bagén**

Cursó la doble titulación de licenciatura en Matemáticas e Ingeniería Informática en la UAB, a continuación hizo el máster en Matemática avanzada y profesional en la Universidad de Barcelona (UB) y posteriormente obtuvo el doctorado en Matemáticas por la UAB (2016). Es fundador de Tengen, donde trabaja actualmente como

*data scientist* desarrollando proyectos basados en inteligencia artificial para aquellas empresas e instituciones que desean obtener el máximo provecho de sus datos. En paralelo trabaja como consultor en la UOC y como profesor asociado en la UAB, donde da clases de aprendizaje automático a nivel de grado y máster.

*A mis hijos, Emma y Jac. Ellos mantienen viva en mí la curiosidad e inquietud que todos los niños tenemos.*

~~Anna Bosch Rué~~

*A mis hijos, Arnau y Txell, por todo el tiempo que este y otros proyectos nos han robado.*

Jordi Casas Roma

*A mi familia.*

Toni Lozano Bagén



# Índice

Prefacio	13
<b>I Introducción</b>	<b>15</b>
Capítulo 1 Introducción y contextualización	17
1.1 ¿Qué es <i>deep learning</i> ? . . . . .	17
1.2 Contextualización de las redes neuronales . . .	20
Capítulo 2 Conceptos básicos de aprendizaje automático	23
2.1 Tipología de métodos . . . . .	24
2.2 Tipología de tareas . . . . .	25
2.3 Preprocesamiento de datos . . . . .	30
2.4 Datos de entrenamiento y test . . . . .	33
2.5 Evaluación de modelos . . . . .	36
<b>II Redes neuronales artificiales</b>	<b>45</b>
Capítulo 3 Principios y fundamentos	47
3.1 Las neuronas . . . . .	47
3.2 Arquitectura de una red neuronal . . . . .	56
3.3 Entrenamiento de una red neuronal . . . . .	61
3.4 Ejemplo de aplicación . . . . .	70
3.5 El problema de la desaparición del gradiente .	73

3.6 Resumen . . . . .	75
<b>Capítulo 4 Optimización del proceso de aprendizaje</b>	<b>77</b>
4.1 Técnicas relacionadas con el rendimiento de la red . . . . .	79
4.2 Técnicas relacionadas con la velocidad del proceso de aprendizaje . . . . .	87
4.3 Técnicas relacionadas con el sobreentrenamiento	92
4.4 Resumen . . . . .	98
<b>Capítulo 5 <i>Autoencoders</i></b>	<b>99</b>
5.1 Estructura básica . . . . .	100
5.2 Entrenamiento de un <i>autoencoder</i> . . . . .	103
5.3 Preentrenamiento utilizando <i>autoencoders</i> . . . . .	105
5.4 Tipos de <i>autoencoders</i> . . . . .	108
 <b>III Redes neuronales convolucionales</b>	<b>111</b>
<b>Capítulo 6 Introducción y conceptos básicos</b>	<b>113</b>
6.1 Visión por computador . . . . .	113
6.2 La operación de convolución . . . . .	117
6.3 Ventajas derivadas de la convolución . . . . .	120
6.4 Conclusiones . . . . .	123
<b>Capítulo 7 Componentes y estructura de una CNN</b>	<b>125</b>
7.1 La capa de convolución . . . . .	126
7.2 Otras capas de las CNN . . . . .	130
7.3 Estructura de una red neuronal convolucional	135
<b>Capítulo 8 Arquitecturas de CNN</b>	<b>143</b>
8.1 Redes convolucionales clásicas . . . . .	143
8.2 <i>Residual networks</i> (ResNet) . . . . .	150
8.3 Inception . . . . .	152

<b>Capítulo 9 Consejos prácticos y ejemplos</b>	<b>157</b>
9.1 Consejos prácticos en el uso de las CNN . . . . .	157
9.2 Ejemplos . . . . .	169
 <b>IV Redes neuronales recurrentes</b>	 <b>181</b>
<b>Capítulo 10 Fundamentos de las redes recurrentes</b>	<b>183</b>
10.1 Concepto de recurrencia . . . . .	183
10.2 Tipos de redes neuronales recurrentes . . . . .	189
10.3 Entrenamiento de una red neuronal recurrente	192
<b>Capítulo 11 Tipología de celdas recurrentes</b>	<b>197</b>
11.1 <i>Long short term memory</i> (LSTM) . . . . .	197
11.2 <i>Gated recurrent unit</i> (GRU) . . . . .	204
<b>Capítulo 12 Arquitecturas de redes recurrentes</b>	<b>209</b>
12.1 Redes neuronales recurrentes bidireccionales .	209
12.2 Redes neuronales recurrentes profundas . . . .	212
12.3 Arquitectura codificador-decodificador . . . .	215
12.4 Mecanismo de atención . . . . .	217
<b>Capítulo 13 Consejos prácticos y ejemplos</b>	<b>221</b>
13.1 Consejos prácticos en el uso de RNN . . . . .	221
13.2 Ejemplos . . . . .	225
 <b>V Apéndices</b>	 <b>233</b>
<b>Capítulo A Notación</b>	<b>235</b>
<b>Capítulo B Detalles del <i>backpropagation</i></b>	<b>237</b>
2.1 Notación . . . . .	237
2.2 Caso particular con un único ejemplo . . . . .	238
2.3 Caso general con varios ejemplos . . . . .	245
<b>Bibliografía</b>	<b>248</b>



# Prefacio

El enfoque del libro es claramente descriptivo, con el objetivo de que el lector entienda los conceptos e ideas básicos detrás de cada algoritmo o técnica, sin entrar en excesivos detalles técnicos, tanto matemáticos como algorítmicos. Siempre que es posible, se hace referencia a los trabajos originales que describieron por primera vez el problema a resolver, los cuales pueden ser rastreados para obtener la literatura más actualizada.

Este libro se divide en cuatro partes principales:

- El primer bloque constituye la introducción al aprendizaje profundo (*deep learning*), en general, y a las redes neuronales, en particular. A continuación se presentan algunos conceptos básicos sobre aprendizaje automático que son necesarios para seguir correctamente los capítulos posteriores.
- En la segunda parte se describen los conceptos introductorios de las redes neuronales. Iniciaremos este bloque con un repaso a los conceptos fundamentales, tales como la estructura de una neurona, las principales funciones de activación, etc. A continuación veremos un conjunto de técnicas diseñadas para mejorar el rendimiento de

las redes neuronales e intentar evitar el problema del sobreentrenamiento. Finalizaremos este bloque viendo un tipo particular de redes neuronales, los *autoencoders*.

- La tercera parte presenta los fundamentos y principales arquitecturas de las redes neuronales convolucionales (*convolutional neural networks*, CNN). Veremos sus fundamentos teóricos, su estructura y su aplicación en el procesamiento de imágenes.
- El cuarto bloque de este libro se centra en las redes neuronales recurrentes (*recurrent neural networks*, RNN). De forma similar al bloque anterior, veremos sus fundamentos teóricos, así como sus principales arquitecturas y aplicaciones para el procesamiento de series temporales.

En los distintos capítulos de este libro se describen algunos ejemplos sencillos con el objetivo de que puedan ser interpretados de forma fácil y ayuden en la comprensión de los detalles teóricos de los métodos descritos. En el libro, sin embargo, no se incluye el código asociado a estos ejemplos, dado que este puede sufrir cambios debido a las actualizaciones de librerías y versiones de los lenguajes de programación empleados.

# Parte I

## Introducción



# Capítulo 1

## Introducción y contextualización

En este capítulo discutiremos, de forma escueta, qué entendemos por *deep learning* y haremos un breve repaso histórico para revisar los hechos que nos han permitido llegar hasta el momento actual.

### 1.1. ¿Qué es *deep learning*?

Desde los inicios de la inteligencia artificial (IA) se ha soñado con máquinas capaces de «pensar» de forma similar a como lo hacemos los seres humanos. Lejos de lograr este ambicioso objetivo, hace ya unas décadas se consiguieron superar importantes desafíos en ámbitos o problemas muy concretos. Por ejemplo, en 1997 la computadora Deep Blue de IBM fue capaz de ganar al ajedrez al campeón mundial Garry Kasparov (Hsu, 2002). Aunque saber jugar correctamente al ajedrez es un problema muy complejo para los seres humanos, este se rige por una serie de normas estrictas y en un entorno muy

delimitado y perfectamente especificado. Esto hace que lo que es una tarea compleja (o muy compleja) para una persona, sea una tarea «fácil» de implementar en un computador. Los computadores, por regla general, se manejan bien en los escenarios formales. Es decir, escenarios cerrados, regidos por reglas y con un conjunto de actores finito. A partir de aquí, las computadoras tienen herramientas para inferir situaciones futuras a partir de la situación actual y las reglas que rigen el escenario.

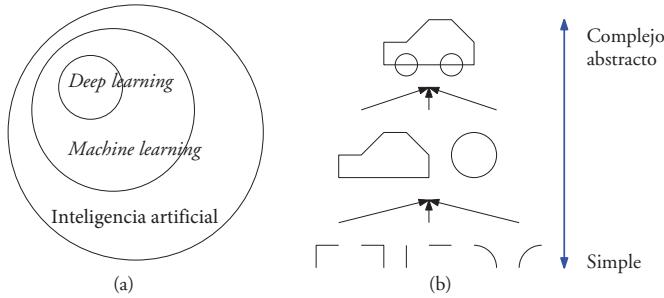
En muchos de estos casos, las reglas y el conocimiento son introducidos en el sistema en el momento de su programación. Es decir, es el programador el encargado de «transferir» el conocimiento al modelo, quien después será capaz de inferir nuevo conocimiento a partir de las reglas. Un caso típico son los sistemas basados en reglas (*rule-based systems*, RBS).

Por el contrario, algunas de las acciones más elementales e intuitivas para los humanos como, por ejemplo, reconocer una persona, un animal o un objeto, pueden ser tareas tremendamente complejas de resolver para un computador. La principal dificultad radica en el hecho de que, en estos casos, no es fácil formalizar los conceptos abstractos que aparecen en estos escenarios. La representación de los conceptos deviene un punto clave en la implementación de modelos y, en el caso de los conceptos abstractos, la complejidad de representar conceptos abstractos de forma explícita dificulta enormemente que pueda ser implementado mediante los modelos de IA clásicos.

Debido a la dificultad para introducir este tipo de conocimiento *a priori* se ha optado por utilizar modelos que sean capaces de extraer su propio conocimiento a partir de los datos. Esta área se ha denominado aprendizaje automático o

*machine learning*. Existen multitud de técnicas y modelos que permiten aprender a partir de un conjunto de datos como, por ejemplo, modelos de regresión, modelos estadísticos o las máquinas de vectores soporte (SVM, *support vector machine*). En muchos casos, los modelos son capaces de aprender patrones a partir de los datos, pero no son capaces de aprender conceptos abstractos y complejos a partir de esos mismos datos.

**Figura 1.1.** Relación del *deep learning* con otros conceptos (a) y ejemplo de jerarquía para la generación de conceptos complejos (b)



Fuente: elaboración propia

El concepto de *deep learning* (o aprendizaje profundo) aparece como un subconjunto, dentro del aprendizaje automático (véase la figura 1.1a), donde se persigue crear modelos que sean capaces de representar conceptos complejos y/o abstractos a partir de otros más sencillos. Es decir, el modelo es capaz de crear de forma automática una jerarquía de conceptos, empezando por conceptos simples, e ir mezclando estos conceptos más simples para ir creando conceptos cada vez más complejos. Esto permite definir conceptos abstractos como composiciones de conceptos mucho más simples. La figura 1.1b muestra un sencillo ejemplo de esta composición de conceptos complejos o abstractos a partir de conceptos mucho

más simples y sencillos. En el ejemplo vemos como, a partir de líneas, cruces y arcos es posible definir el concepto de coche, especificándolo como una composición de estos elementos más simples. Cuando esta jerarquía de conceptos tiene múltiples capas, hablamos de la «profundidad» del modelo. Es aquí donde aparece el concepto de «aprendizaje profundo» o *deep learning*.

Aunque actualmente se asocia el concepto de *deep learning* con las redes neuronales, existen otros modelos de aprendizaje automático que también son capaces de componer conceptos a partir de una jerarquía de conceptos más simples (Bengio, Goodfellow y Courville, 2016). Aun así, en este libro nos centraremos en las distintas variedades de redes neuronales profundas, que actualmente representan el estado del arte en el *deep learning*.

## 1.2. Contextualización de las redes neuronales

Las redes neuronales artificiales (*artificial neural networks* o ANN) (Beale y Jackson, 1990; Haykin, 2009) son un conjunto de algoritmos inspirados en el mecanismo de comunicación de la neurona biológica. Han demostrado ser una buena aproximación a problemas donde el conocimiento es impreciso o variable en el tiempo. Su capacidad de aprender convierte a las redes neuronales en algoritmos adaptativos y elaborados a la vez.

El inicio de las redes neuronales se remonta a los años cuarenta, cuando McCulloch y Pitts (1988) propusieron un modelo para intentar explicar el funcionamiento del cerebro humano. Una década después, en los años cincuenta, se empezaron a implementar modelos de cálculo basados en el concepto

de perceptrón (Rosenblatt, 1962). Posteriormente, Marvin Minsky demostró las limitaciones teóricas de los modelos existentes (Minsky y Papert, 1969), provocando un estancamiento de la investigación y aplicaciones relacionadas con las redes neuronales durante los años setenta y ochenta hasta que en el año 1982 John Hopfield propuso el modelo de la «propagación hacia atrás» (*backpropagation*) (Hopfield, 1984), que permitió superar las limitaciones del perceptrón y, además, facilitó en gran medida la fase de entrenamiento de las redes.

Las grandes expectativas puestas en las redes neuronales durante las décadas de los años ochenta y noventa provocaron una considerable desilusión en estos modelos, que no fueron capaces de cumplir las (altas) expectativas generadas en aquel momento. Aun así, la investigación y el avance en las redes neuronales fue continuo durante estos años, aunque sin una gran atención por parte de la comunidad científica, que se centró en otros modelos que parecían más prometedores en aquel entonces.

El resurgimiento de las redes neuronales, en este caso bajo la etiqueta de *deep learning*, se produjo a partir de mediados de la primera década de este siglo. Hinton, Osindero y col. (2006) presentaron un algoritmo que permitió entrenar redes *deep belief* de forma eficiente, terminando así con un periodo de dificultades relacionadas con el entrenamiento de redes. Otros autores desarrollaron métodos similares para otros tipos de datos, como Bengio, Lamblin y col. (2006); Ranzato, Poultney y col. (2006). Fue a partir de este momento cuando se popularizó el término *deep learning*, haciendo referencia, especialmente, al concepto de ser capaces de entrenar redes más profundas. Actualmente, las redes profundas son el estado del

arte en muchos dominios, donde han superado ampliamente a los demás modelos o algoritmos de aprendizaje automático.

Es importante remarcar que el término moderno *deep learning* va más allá del concepto biológico que inspiró, inicialmente las redes neuronales para centrarse en el principio de los múltiples niveles de composición que forman los conceptos complejos y/o abstractos. Por lo tanto, actualmente la neurociencia se toma solo como una inspiración para los modelos de *deep learning*, pero en ningún caso como una «guía» para el desarrollo de nuevos modelos o arquitecturas. Simplemente porque no tenemos hoy en día suficiente conocimiento de cómo trabaja nuestro cerebro a bajo nivel para poder entender el funcionamiento global de un conjunto de miles o millones de neuronas.

## Capítulo 2

# Conceptos básicos de aprendizaje automático

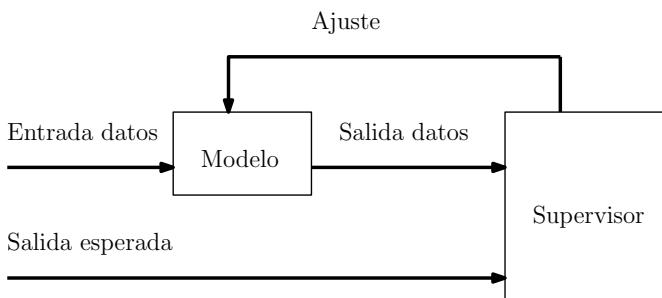
El aprendizaje automático (más conocido por su denominación en inglés, *machine learning*, ML) es el conjunto de métodos y algoritmos que permiten a una máquina aprender de manera automática en base a experiencias pasadas.

En este capítulo veremos algunos conceptos básicos de aprendizaje automático, en general, que son especialmente relevantes para el uso de redes neuronales artificiales. En este sentido, empezaremos viendo la tipología de métodos y tareas existentes en aprendizaje automático. A continuación revisaremos algunas medidas básicas de preprocesamiento de datos y discutiremos muy brevemente la creación del conjunto de entrenamiento y test. Finalizaremos este capítulo revisando algunas medidas de evaluación de modelos que nos resultarán de gran utilidad en los capítulos posteriores de este libro.

## 2.1. Tipología de métodos

Generalmente, un algoritmo de aprendizaje automático debe construir un modelo en base a un conjunto de datos de entrada que representa el conjunto de aprendizaje, lo que se conoce como *conjunto de entrenamiento*. Durante esta fase de aprendizaje, el algoritmo va comparando la salida de los modelos en construcción con la salida ideal que deberían tener estos modelos, para ir ajustándolos y aumentando la precisión. Esta comparación que forma la base del aprendizaje en sí, y éste aprendizaje puede ser *supervisado* o *no supervisado*. En el aprendizaje supervisado (figura 2.1), hay un componente externo que compara los datos obtenidos por el modelo con los datos esperados por este, y le proporciona retroalimentación para que vaya ajustándose. Para ello, pues, será necesario proporcionar al modelo un conjunto de datos de entrenamiento que contenga tanto los datos de entrada como la salida esperada para cada uno de ellos.

**Figura 2.1.** Aprendizaje supervisado



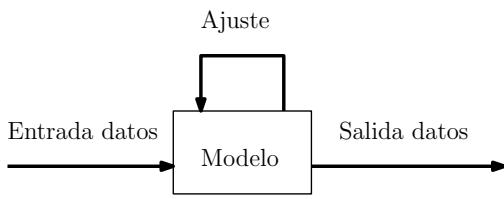
Fuente: elaboración propia

Todas ellas se basan en el paradigma del aprendizaje inductivo. La esencia de cada una de ellas es derivar inductivamente

te a partir de los *datos* (que representan la información del entrenamiento), un *modelo* (que representa el conocimiento) que tiene utilidad predictiva, es decir, que puede aplicarse a nuevos datos.

En el aprendizaje no supervisado (figura 2.2), el algoritmo de entrenamiento aprende sobre los propios datos de entrada, descubriendo y agrupando patrones, características, correlaciones, etc.

**Figura 2.2.** Aprendizaje no supervisado



Fuente: elaboración propia

## 2.2. Tipología de tareas

Según el objetivo de nuestro análisis, podemos distinguir entre tres grandes grupos de tareas, que revisaremos brevemente a continuación.

### 2.2.1. Clasificación

La clasificación (*classification*) es uno de los procesos cognitivos importantes, tanto en la vida cotidiana como en los negocios, donde podemos clasificar clientes, empleados, transacciones, tiendas, fábricas, dispositivos, documentos o cualquier otro tipo de instancias en un conjunto de clases o categorías predefinidas con anterioridad.

La tarea de clasificación consiste en asignar instancias de un dominio dado, descritas por un conjunto de atributos discretos o de valor continuo, a un conjunto de clases, que pueden ser consideradas valores de un atributo discreto seleccionado, generalmente denominado *clase*. Las etiquetas de clase correctas son, en general, desconocidas, pero se proporcionan para un subconjunto del dominio. Por lo tanto, queda claro que es necesario disponer de un subconjunto de datos correctamente etiquetado, y que se usará para la construcción del modelo.

La función de clasificación puede verse como:

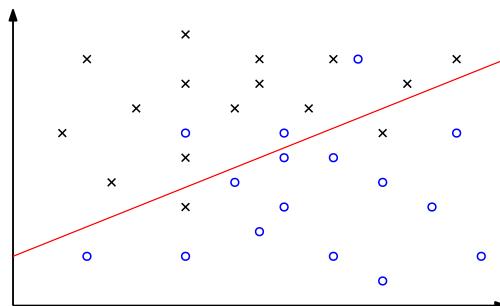
$$c : X \rightarrow C \quad (2.1)$$

donde  $c$  representa la función de clasificación,  $X$  el conjunto de atributos que forman una instancia y  $C$  la etiqueta de clase de dicha instancia.

Un tipo de clasificación particularmente simple, pero muy interesante y ampliamente estudiado, hace referencia a los problemas de clasificación binarios, es decir, problemas con un conjunto de datos pertenecientes a dos clases, por ejemplo,  $C = \{0, 1\}$ . La figura 2.3 muestra un ejemplo de clasificación binaria, donde las cruces y los círculos representan elementos de dos clases, y se pretende dividir el espacio tal que separe a la mayoría de elementos de clases diferentes.

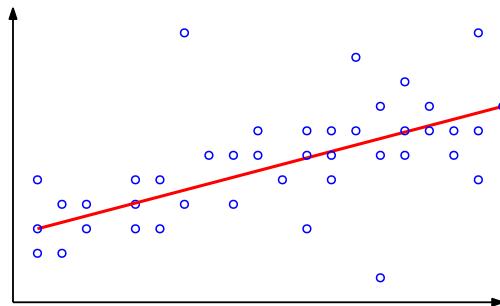
### 2.2.2. Regresión

Al igual que la clasificación, la regresión (*regression*) es una tarea de aprendizaje inductivo que ha sido ampliamente estudiada y utilizada. Se puede definir, de forma informal, como un problema de «clasificación con clases continuas». Es decir, los modelos de regresión predicen valores numéricos en lugar

**Figura 2.3.** Ejemplo de clasificación

Fuente: elaboración propia

de etiquetas de clase discretas. A veces también nos podemos referir a la regresión como «predicción numérica».

**Figura 2.4.** Ejemplo de regresión lineal

Fuente: elaboración propia

La tarea de regresión consiste en asignar valores numéricos a instancias de un dominio dado, descritos por un conjunto de atributos discretos o de valor continuo, como se muestra en la figura 2.4, donde los puntos representan los datos de aprendizaje y la línea representa la predicción sobre futuros eventos. Se supone que esta asignación se aproxima a alguna función objetivo, generalmente desconocida, excepto para un

subconjunto del dominio. Este subconjunto se puede utilizar para crear el modelo de regresión.

En este caso, la función de regresión se puede definir como:

$$f : X \rightarrow \mathbb{R} \quad (2.2)$$

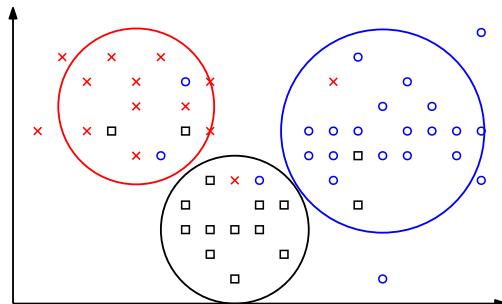
donde  $f$  representa la función de regresión,  $X$  el conjunto de atributos que forman una instancia y  $\mathbb{R}$  un valor en el dominio de los números reales.

Es importante remarcar que una regresión no pretende devolver una predicción exacta sobre un evento futuro, sino una aproximación (como muestra la diferencia entre la línea y los puntos de la figura). Por lo general, datos más dispersos resultarán en predicciones menos ajustadas.

### 2.2.3. Agrupamiento

El agrupamiento (*clustering*) es una tarea de aprendizaje inductiva que, a diferencia de las tareas de clasificación y regresión, no dispone de una etiqueta de clase a predecir. Puede considerarse como un problema de clasificación, pero donde no existen un conjunto de clases predefinidas, y estas se «descubren» de forma autónoma por el método o algoritmo de agrupamiento, basándose en patrones de similitud identificados en los datos.

La tarea de agrupamiento consiste en dividir un conjunto de instancias de un dominio dado, descrito por un número de atributos discretos o de valor continuo, en un conjunto de grupos (*clusters*) basándose en la similitud entre las instancias, y crear un modelo que puede asignar nuevas instancias a uno de estos grupos o *clusters*. La figura 2.5 muestra un ejemplo de agrupamiento donde las cruces, círculos y cuadros pertenecen a tres clases de elementos distintos que pretendemos agrupar.

**Figura 2.5.** Ejemplo de agrupamiento

Fuente: elaboración propia

Un proceso de agrupación puede proporcionar información útil sobre los patrones de similitud presentes en los datos como, por ejemplo, segmentación de clientes o creación de catálogos de documentos. Otra de las principales utilidades del agrupamiento es la detección de anomalías. En este caso, el método de agrupamiento permite distinguir instancias con un patrón absolutamente distinto a las demás instancias «normales» del conjunto de datos, facilitando la detección de anomalías y posibilitando la emisión de alertas automáticas para nuevas instancias que no tienen ningún *cluster* existente.

La función de agrupamiento o *clustering* se puede modelar mediante:

$$h : X \rightarrow C_h \quad (2.3)$$

donde  $h$  representa la función de agrupamiento,  $X$  el conjunto de atributos que forman una instancia y  $C_h$  un conjunto de grupos o *clusters*. Aunque esta definición se parece mucho a la tarea de clasificación, existe una diferencia aparentemente pequeña pero muy importante consistente en que el conjunto

de «clases» no está predeterminado ni es conocido *a priori*, sino que se identifica como parte de la creación del modelo.

## 2.3. Preprocesamiento de datos

Las redes neuronales artificiales, como el resto de modelos de aprendizaje automático, requieren que los datos de entrada se encuentren en un formato específico para poder trabajar de forma correcta.

Veamos las principales características de los datos de entrada de una red neuronal:

- Deben estar en formato numérico. Los atributos categóricos deben ser convertidos a atributos numéricos, empleando alguna de las técnicas existentes.
- Se recomienda que los atributos estén en el rango  $[0, 1]$ , aunque no es imprescindible. En cualquier caso, sí se recomienda que los datos estén en una escala similar, es decir, que no haya atributos en un rango de varias magnitudes superior a otros atributos.

### 2.3.1. Datos estructurados

Generalmente, en los métodos supervisados partiremos de un conjunto de datos correctamente etiquetados ( $D$ ) en el que distinguimos la siguiente estructura:

$$D_{n,m} = \begin{pmatrix} d_1 = & a_{1,1} & a_{1,2} & \cdots & a_{1,m} & c_1 \\ d_2 = & a_{2,1} & a_{2,2} & \cdots & a_{2,m} & c_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ d_n = & a_{n,1} & a_{n,2} & \cdots & a_{n,m} & c_n \end{pmatrix}$$

donde:

- $n$  es número de elementos o instancias,
- $m$  el número de atributos del conjunto de datos (también conocido como dimensionalidad),
- $d_i$  indica la instancia  $i$  del juego de datos,
- $a_{i,j}$  indica el atributo descriptivo  $j$  de la instancia  $i$  y
- $c_i$  indica el atributo objetivo o clase a predecir para la instancia  $d_i$ .

En primer lugar, las redes neuronales, como la mayoría de los algoritmos de aprendizaje automático, prefieren trabajar con *valores numéricos*, así que se debe transformar los atributos categóricos a atributos numéricos.

En segundo lugar, es importante *escalar* los atributos para mejorar el comportamiento y el rendimiento de los modelos de aprendizaje automático.

Hay dos formas básicas de realizar esta operación:

- La normalización (o también conocido como *min-max scaling*) consiste en escalar los valores en el rango  $[0, 1]$ .
- La estandarización se aplica restando el valor medio y dividiendo por la varianza cada uno de los atributos. El resultado no se encuentra limitado en un rango determinado, pero se comporta mejor ante valores extremos (*outliers*).

A diferencia de la normalización, la estandarización no limita los valores a un rango específico, lo que puede ser un problema para algunos algoritmos (por ejemplo, las redes neuronales a menudo esperan un valor de entrada que oscila entre 0 y 1). Sin embargo, la estandarización se ve mucho menos afectada por los valores atípicos o extremos.

### 2.3.2. Imágenes

En el caso de trabajar con imágenes, hay una serie de pasos específicos de preprocesado que debemos llevar a cabo antes de usar los datos en cualquier algoritmo de aprendizaje automático y, en particular, en redes neuronales.

A continuación describimos brevemente los principales:

- Uno de los primeros pasos es garantizar que las imágenes tengan el mismo tamaño y relación de aspecto (*uniform aspect ratio*). La mayoría de los modelos de redes neuronales asumen una imagen de entrada de forma cuadrada, lo que significa que debe verificarse si cada imagen es un cuadrado o no, y recortarse de manera apropiada.
- Una vez que tenemos todas las imágenes en la misma relación de aspecto (ya sean cuadradas o tengan otra relación de aspecto), es el momento de escalar cada imagen de manera adecuada (*image scaling*).
- La normalización de los datos es un paso importante que garantiza que cada parámetro de entrada (píxel, en este caso) tenga una distribución de datos similar. Esto hace que la convergencia sea más rápida mientras se entrena la red.
- Aunque opcional, la reducción de la dimensionalidad es un paso habitual cuando trabajamos con imágenes en color. Generalmente, se opta por colapsar los canales RGB en un solo canal de escala de grises.
- Finalmente, en algunos casos es interesante aumentar el conjunto de datos de forma artificial. Para este fin, se complementa el conjunto de datos con nuevas versiones

perturbadas de las imágenes originales. Las perturbaciones típicas incluyen: escalas, rotaciones y otras transformaciones afines. Esta operación puede ayudar a la red a generalizar mejor, evitando el sobreentrenamiento.

## 2.4. Datos de entrenamiento y test

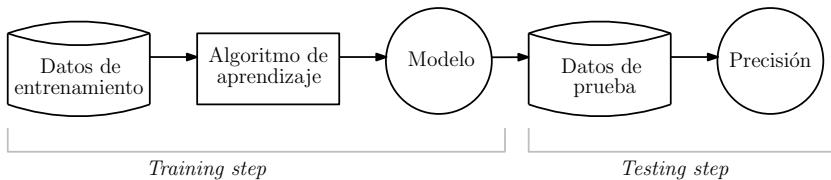
Para validar un algoritmo de aprendizaje o modelo es necesario asegurar que este funcionará correctamente para los datos de prueba o test futuros, de forma que capture la esencia del problema que quiere resolver y generalice correctamente. En esencia se trata de evitar que sea dependiente de los datos utilizados durante su entrenamiento, evitando el problema conocido como «sobreentrenamiento» (en inglés, *overfitting*).

El *sobreentrenamiento* se puede definir, informalmente, como el peligro que corremos al sobreentrenar un modelo de que este acabe respondiendo estrictamente a las propiedades del juego de datos de entrenamiento y que sea incapaz de extrapolarse con niveles de acierto adecuados a otros juegos de datos que puedan aparecer en un futuro.

En la figura 2.6, queremos subrayar que cuando hablamos de datos de entrenamiento y de test, estamos refiriéndonos en exclusiva a los algoritmos de aprendizaje supervisado. En este escenario es conveniente evaluar los resultados obtenidos sobre datos etiquetados nunca vistos anteriormente y comparar los errores cometidos para cada conjunto.

Así, usaremos el conjunto de datos de entrenamiento para crear el modelo supervisado, mientras que el conjunto de datos de test se usará para medir la precisión alcanzada por el modelo. La repetición iterativa de entrenamiento y verificación formará parte del proceso de construcción del modelo

**Figura 2.6.** Proceso de creación y validación de un modelo basado en aprendizaje supervisado



Fuente: elaboración propia

hasta conseguir unos niveles de precisión y de capacidad de predicción aceptables.

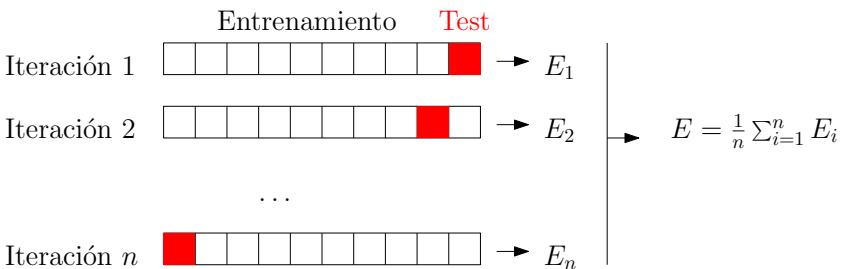
Habitualmente los juegos de datos de entrenamiento y de test suelen ser extracciones aleatorias del juego de datos inicial. En función del número de datos disponibles, existen diferentes técnicas para la construcción de los conjuntos de entrenamiento y de prueba. Se trata de un compromiso entre la robustez del modelo construido (a mayor número de datos usados para el entrenamiento, más robusto será el modelo) y su capacidad de generalización (a mayor número de datos usados para la validación, más fiable será la estimación del error cometido).

#### 2.4.1. *K-fold cross validation*

Para la creación del conjunto de datos de entrenamiento y test, la opción más empleada en la actualidad es conocida como *k-fold cross validation*. Esta técnica consiste en realizar una partición aleatoria del conjunto de datos en  $k$  conjuntos del mismo tamaño, usando  $k - 1$  conjuntos para entrenar el modelo y el conjunto restante para evaluarlo, repitiendo el proceso  $k$  veces y promediando el error estimado.

Valores habituales pueden ser  $k = 5$  o  $k = 10$ , aunque no existe ninguna base teórica que sustente dichos valores, sino que es resultado de la experimentación (Kohavi, 1995). La figura 2.7 muestra el esquema de validación cruzada con  $k = 10$ .

**Figura 2.7.** Esquema de división de conjuntos de entrenamiento y test según el método *k-fold cross validation*



Fuente: elaboración propia

Por otra parte, dado que la partición en  $k$  conjuntos es aleatoria, este proceso podría repetirse un cierto número de ocasiones, promediando todos los errores cometidos (que, de hecho, ya eran promedios del resultado de aplicar *k-fold cross validation*), siempre y cuando se utilice el mismo valor de  $k$ .

Como siempre que hay una partición aleatoria, es importante comprobar que la distribución de los valores de la variable objetivo sea similar, para evitar la creación de modelos muy sesgados. De hecho, la partición aleatoria se debe realizar teniendo en cuenta dicha distribución, para evitar el posible sesgo.

## 2.5. Evaluación de modelos

En esta sección revisaremos los principales indicadores o métricas para evaluar los resultados de un proceso de minería de datos, en general, y de un modelo basado en redes neuronales, en particular. El objetivo es cuantificar el grado o valor de «bondad» de la solución encontrada, permitiendo la comparación entre distintos métodos sobre los mismos conjuntos de datos.

Las métricas para realizar este tipo de evaluación dependen, principalmente, del tipo de problema con el que se está lidiando. En este sentido, veremos métricas específicas para problemas de clasificación y regresión.

### 2.5.1. Modelos de clasificación

Las medidas de calidad de modelos de clasificación se calculan comparando las predicciones generadas por el modelo en un conjunto de datos  $D$  con las etiquetas de clase verdaderas de las instancias de este conjunto de datos.

#### Matriz de confusión

La matriz de confusión (*confusion matrix*, CM) presenta en una tabla una visión gráfica de los errores cometidos por el modelo de clasificación. Se trata de un modelo gráfico para visualizar el nivel de acierto de un modelo de predicción. También es conocido en la literatura como tabla de contingencia o matriz de errores.

La figura 2.8 presenta la matriz de confusión para el caso básico de clasificación binaria. En esencia, esta matriz indica el número de instancias correcta e incorrectamente clasificadas. Los parámetros que nos indica son:

**Figura 2.8.** Matriz de confusión binaria

		Clase predicha	
		P	N
Clase verdadera	P	TP	FN
	N	FP	TN

Fuente: elaboración propia

- Verdadero positivo (*true positive*, TP): número de clasificaciones correctas en la clase positiva (*P*).
- Verdadero negativo (*true negative*, TN): número de clasificaciones correctas en la clase negativa (*N*).
- Falso negativo (*false negative*, FN): número de clasificaciones incorrectas de clase positiva clasificada como negativa.
- Falso positivo (*false positive*, FP): número de clasificaciones incorrectas de clase negativa clasificada como positiva.

Como se puede observar, en la diagonal de la matriz aparecen los aciertos del modelo, ya sean positivos o negativos.

### Matriz de confusión para $k$ clases

La matriz de confusión se puede extender a más de dos clases de forma natural, tal y como podemos ver en la figura 2.9. Esta representación nos permite identificar de forma

rápida el número de instancias correctamente clasificadas, que se corresponde con la diagonal de la tabla.

**Figura 2.9.** Matriz de confusión para  $k$  clases

		Clase predicha		
		$C_1$	$\dots$	$C_k$
Clase verdadera	$C_1$			
	$\dots$			
$C_k$				

Fuente: elaboración propia

Por otro lado, en el caso de problemas de clasificación que impliquen más de dos clases, se puede realizar la evaluación de dos formas distintas:

- *1-vs-1* (OvO): Midiendo la capacidad de discriminar entre instancias de una clase, considerada la clase positiva, frente a las instancias de otra clase, consideradas negativas.
- *1-vs-All* (OvA): Midiendo la capacidad de discriminar entre instancias de una clase, considerada la clase positiva, frente a las instancias de las demás clases, consideradas negativas.

La aproximación de *1-vs-1* nos conduce a una matriz de confusión, con formato  $2 \times 2$ , para cada par de clases existentes. Por otra parte, la aproximación *1-vs-All* produce una matriz de confusión  $2 \times 2$  para cada clase.

## Métricas derivadas de la matriz de confusión

A partir de la matriz de confusión, definimos un conjunto de métricas que permiten cuantificar la bondad de un modelo de clasificación.

- El *error de clasificación* (*misclassification error*, ERR) y la *exactitud* (*accuracy*, ACC) proporcionan información general sobre el número de instancias incorrectamente clasificadas. El error, ecuación 2.4, es la suma de las predicciones incorrectas sobre el número total de predicciones. Por el contrario, la exactitud (*accuracy*) es el número de predicciones correctas sobre el número total de predicciones, como se puede ver en la ecuación 2.5.

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}, \quad (2.4)$$

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR. \quad (2.5)$$

- En algunos problemas nos puede interesar medir el error en los falsos positivos o negativos. Por ejemplo, en un sistema de diagnosis de tumores, nos interesa centrarnos en los casos de tumores malignos que han sido clasificados incorrectamente como tumores benignos. En estos casos, la *tasa de verdaderos positivos* (*true positive rate*, TPR) y la *tasa de falsos positivos* (*false positive rate*, FPR), que definimos a continuación, pueden ser muy útiles:

$$TPR = \frac{TP}{FN + TP}, \quad (2.6)$$

$$FPR = \frac{FP}{FP + TN}. \quad (2.7)$$

- La *precisión* (*precision*, PRE) mide el rendimiento relacionado con las tasas de verdaderos positivos y negativos, tal y como podemos ver a continuación.

$$PRE = \frac{TP}{TP + FP}. \quad (2.8)$$

- El *recall* (*recall*, REC) y la *sensibilidad* (*sensitivity*, SEN) se corresponden con la tasa de verdaderos positivos (TPR), mientras que la *especificidad* (*specificity*, SPE) se define como la tasa de instancias correctamente clasificadas como negativas respecto a todas las instancias negativas:

$$REC = SEN = TPR = \frac{TP}{FN + TP}, \quad (2.9)$$

$$SPE = \frac{TN}{TN + FP} = 1 - FPR. \quad (2.10)$$

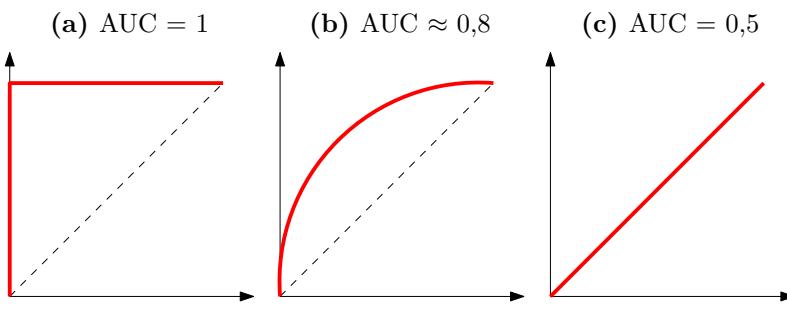
- Finalmente, en la práctica se suelen combinar la precisión y el recall en una métrica llamada *F1* (*F1 score*), que se define de la siguiente forma:

$$F1 = 2 \times \frac{PRE \times REC}{PRE + REC}. \quad (2.11)$$

## Curvas ROC

Una curva ROC (acrónimo de *receiver operating characteristic*) mide el rendimiento respecto a los FP y los TP. La diagonal de la curva ROC se interpreta como un modelo generado aleatoriamente, mientras que valores inferiores se consideran peores que una estimación aleatoria de los nuevos datos.

**Figura 2.10.** Ejemplo de curvas ROC



Fuente: elaboración propia

En esta métrica, un clasificador perfecto ocuparía la posición superior izquierda de la gráfica, con una tasa de TP igual a 1 y una tasa de FP igual a 0. A partir de la curva ROC se calcula el área *bajo la curva* (*area under the curve*, AUC) que permite caracterizar el rendimiento del modelo de clasificación. La figura 2.10 ejemplifica un rendimiento excelente, bueno y malo de una curva ROC.

### 2.5.2. Modelos de regresión

La evaluación de modelos de regresión comparte el mismo principio que la evaluación de modelos de clasificación, en el sentido de que se compara los valores predichos con los valores reales de las instancias del conjunto de datos. En esta sección

veremos algunas de las métricas más empleadas para evaluar el rendimiento de un modelo de regresión.

- El *error absoluto medio* (*mean absolute error*, MAE) es la métrica más simple y directa para la evaluación del grado de divergencia entre dos conjuntos de valores, representado por la ecuación:

$$MAE = \frac{1}{|D|} \sum_{d \in D} |f(d) - h(d)|, \quad (2.12)$$

donde  $D$  es el conjunto de instancias,  $h : X \rightarrow \mathbb{R}$  es la función del modelo y  $f : X \rightarrow \mathbb{R}$  es la función objetivo con las etiquetas correctas de las instancias.

En este caso, todos los residuos tienen la misma contribución al error absoluto final.

- El *error cuadrático medio* (*mean square error*, MSE) es, probablemente, la métrica más empleada para la evaluación de modelos de regresión. Se calcula de la siguiente forma:

$$MSE = \frac{1}{|D|} \sum_{d \in D} (f(d) - h(d))^2. \quad (2.13)$$

Utilizando esta métrica se penalizan los residuos grandes. En este sentido, si el modelo aproxima correctamente a gran parte de las instancias del conjunto de datos, pero comete importantes errores en unos pocos, la penalización en esta métrica será muy superior a la indicada si se emplea la métrica anterior (MAE).

- La *raíz cuadrada del error cuadrático medio* (*root mean square error*, RMSE) se define aplicando la raíz cuadrada al error cuadrático medio (MSE). Tiene las mismas características que este, pero aporta la ventaja adicional de que el error se expresa en la misma escala que los datos originales. En el caso del MSE no era así, ya que la diferencia de valores son elevados al cuadrado antes de realizar la suma ponderada. En algunos casos esta diferencia puede ser importante, mientras que irrelevante en otros.

$$RMSE = \sqrt{MSE}. \quad (2.14)$$

- Finalmente, en algunos modelos de regresión se pueden tolerar diferencias importantes entre los valores predichos y los verdaderos, siempre que el modelo tenga un comportamiento general similar a los valores verdaderos, prestando especial atención a la monotonicidad. En estos casos, las métricas vistas anteriormente pueden no ser representativas del rendimiento de un modelo de regresión. Por el contrario, los *índices de correlación* lineal o de rango, como por ejemplo Pearson o Spearman, pueden ser una buena métrica para medir la bondad del modelo generado.



# **Parte II**

## **Redes neuronales artificiales**



# Capítulo 3

## Principios y fundamentos

Las redes neuronales artificiales permiten realizar tareas de clasificación en juegos de datos etiquetados y tareas de regresión en juegos de 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.

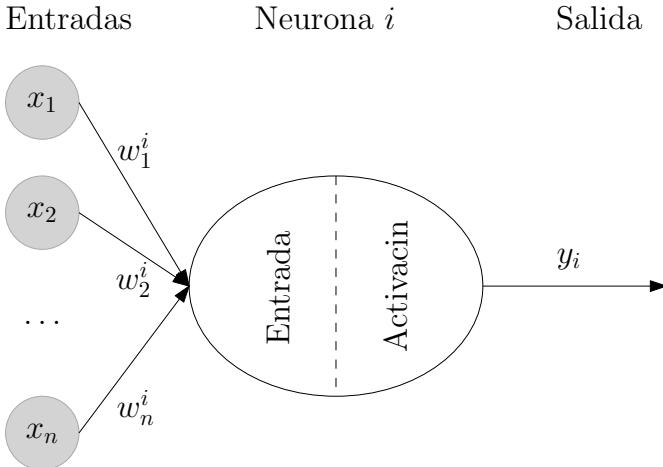
En general las redes neuronales relacionan entradas con salidas, es decir, el juego de datos inicial con la salida o resultado del algoritmo aplicado sobre estos. Algunos autores se refieren a las redes neuronales como «aproximadores universales», porque son capaces de aprender y aproximar con precisión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

### 3.1. 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 determi-

nada 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 3.1.** Esquema básico de una neurona artificial



Fuente: elaboración propia

La figura 3.1 presenta el esquema básico de una neurona artificial. Cada neurona tiene un conjunto de entradas, denotadas como  $X = \{x_1, x_2, \dots, x_n\}$ . Cada una de estas entradas está ponderada por el conjunto de valores  $W^i = \{w_1^i, w_2^i, \dots, w_n^i\}$ . 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.

### 3.1.1. 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:

$$z(x) = \sum_{j=1}^n x_j w_j^i. \quad (3.1)$$

- La función máximo:

$$z(x) = \max(x_1 w_1^i, \dots, x_n w_n^i). \quad (3.2)$$

- La función mínimo:

$$z(x) = \min(x_1 w_1^i, \dots, x_n w_n^i). \quad (3.3)$$

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

$$z(x) = (x_1 w_1^i \wedge \dots \wedge x_n w_n^i), \quad (3.4)$$

$$z(x) = (x_1 w_1^i \vee \dots \vee x_n w_n^i). \quad (3.5)$$

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.

### 3.1.2. 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 solo sería capaz de llevar a cabo aproximaciones lineales.

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

Algunas de las funciones de transferencia más utilizadas son:

- La *función escalón*, que se muestra en la figura 3.2a y cuyo comportamiento es:

$$y(x) = \begin{cases} 1 & \text{si } x \geq \alpha, \\ -1 & \text{si } x \leq \alpha, \end{cases} \quad (3.6)$$

donde  $\alpha$  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 3.2b.

$$y(x) = \beta x. \quad (3.7)$$

- La *función sigmoide* o *función logística*, que se muestra en la figura 3.2c y cuya fórmula incluye un parámetro ( $\rho$ ) para determinar la forma de la curva, pudiendo actuar como un separador más o menos «suave» de las salidas.

$$y(x) = \frac{1}{1 + e^{\frac{-x}{\rho}}}. \quad (3.8)$$

- La *tangente hiperbólica*, que describimos a continuación y podemos ver en la figura 3.2d.

$$y(x) = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{1 - e^{-2x}}{1 + e^{-2x}}. \quad (3.9)$$

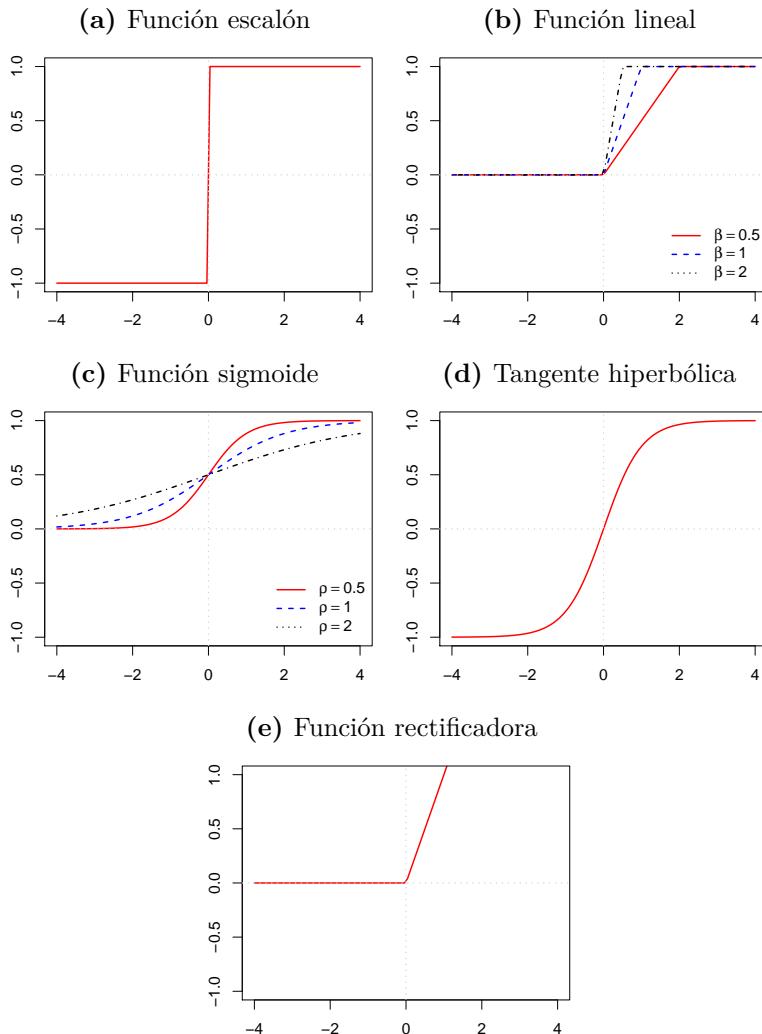
- La *función rectificadora* se calcula obteniendo la parte positiva de su argumento, como podemos ver en la figura 3.2e, y su fórmula de cálculo es:

$$y(x) = \max(0, x). \quad (3.10)$$

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

**Figura 3.2.** Representación de las funciones escalón, lineal, sigmoide, hiperbólica y rectificadora



Fuente: elaboración propia

lo tanto, cuando se utilizan 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 sus coeficientes, de manera que se aproxime a los datos que aparecen en el conjunto de las entradas.

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, Bordes y col., 2011). Actualmente, es la función de activación más popular para redes profundas, como veremos en las secciones posteriores.

### 3.1.3. 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 caracteriza por las siguientes funciones:

- La función de entrada de la neurona  $i$  es la suma ponderada de las entradas ( $x_j$ ) y los pesos ( $w_j^i$ ).

$$f(x) = \sum_{j=1}^n x_j w_j^i. \quad (3.11)$$

- La función de activación se representa mediante la función escalón, presentada en la ecuación 3.6. 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:

$$y_i = \begin{cases} 0 & \text{si } x_1 w_1^i + \dots + x_n w_n^i - \alpha \leq 0, \\ 1 & \text{si } x_1 w_1^i + \dots + x_n w_n^i - \alpha > 0, \end{cases} \quad (3.12)$$

donde el parámetro  $\alpha$  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  $x_1 w_1^i + \dots + x_n w_n^i - \alpha$  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 ( $w_j^i$ ).

$$f(x) = \sum_{j=1}^n x_j w_j^i. \quad (3.13)$$

- La función de activación emplea la función sigmoide (ver ecuación 3.8). 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  $\Delta w_j$  en los pesos y en el sesgo  $\Delta \alpha$  producirán un cambio pequeño  $\Delta y$  en la salida de la neurona. Matemáticamente, se puede expresar de esta forma:

$$\Delta y = \sum_{j=1}^n \frac{\partial y}{\partial w_j} \Delta w_j + \frac{\partial y}{\partial \alpha} \Delta \alpha, \quad (3.14)$$

donde  $\frac{\partial y}{\partial w_j}$  y  $\frac{\partial y}{\partial \alpha}$  denotan la derivada parcial de la salida respecto a  $w_j$  y  $\alpha$ , respectivamente.

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 ( $w_j^i$ ).

$$f(x) = \sum_{j=1}^n x_j w_j^i. \quad (3.15)$$

- La función de activación es la función rectificadora (ver ecuación 3.10).

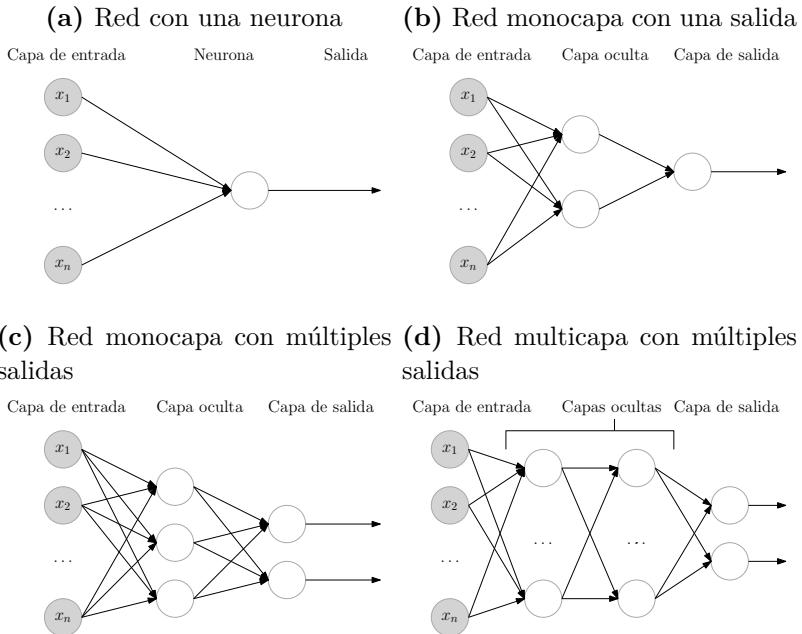
Por lo tanto, su salida no es binaria, si no un valor continuo en el rango  $[0, +\infty]$ .

### 3.2. 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 estas tales como las funciones de entrada o activación. Cada arquitectura o topología puede ser válida para algunos tipos de problemas, y presentar 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 3.3a. 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, formada por un conjunto variable de neuronas y, finalmente, la capa de salida con una o más neuronas. La figura 3.3b muestra la arquitectura monocapa con una única salida, mientras que la figura 3.3c presenta una red monocapa con múltiples salidas (dos en este

**Figura 3.3.** Ejemplos de arquitecturas de redes neuronales

Fuente: elaboración propia

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 una cantidad demasiado grande de neuronas en esta capa aumenta el riesgo de sobre-especialización en el proceso de entrenamiento. 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 3.3d. 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. Aunque también se aumenta su tendencia a la sobreespecialización y se aumenta el coste computacional y temporal 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: la información siempre avanza, nunca se retroalimenta.

Sin embargo, hay otros modelos de redes neuronales artificiales en los que los bucles de retroalimentación son posibles. Estos modelos se llaman redes neuronales recurrentes (*recurrent neural networks*, RNN). La idea 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 con una duración limitada. Es decir, las 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 (Rumelhart, McClelland y col., 1986), que tienen conexiones simétricas, o las máquinas de Boltzmann (Rumelhart, McClelland y col., 1986). Este tipo de redes las presentaremos y discutiremos más adelante, en el capítulo 10.

### 3.2.1. 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 a su vez 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 solo 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.

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 y Nadal, 1989).

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

1. Aumentar el poder predictivo de la red, es decir, su capacidad de reconocimiento y predicción. Pero también aumenta el peligro de sobreajuste a los datos de entrenamiento.
2. Disminuir la posibilidad de caer en un mínimo local (Rumelhart, McClelland y col., 1986).
3. Alterar el tiempo de aprendizaje. Este varía de forma inversa al número de neuronas de las capas ocultas (Plaut y Hinton, 1987).

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

1. Reducir el riesgo de sobreespecialización 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.

### 3.3. Entrenamiento de una red neuronal

En esta sección discutiremos los principales métodos de entrenamiento de una red neuronal. En primer lugar, en la sección 3.3.1, analizaremos el funcionamiento de una neurona tipo perceptrón. En la sección 3.3.2 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 3.3.3.

#### 3.3.1. Funcionamiento y entrenamiento de una neurona

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

$$y = \sigma(wx - \alpha), \quad (3.16)$$

donde  $\sigma$  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  $W^i = \{w_1^i, w_2^i, \dots, w_n^i\}$ .
- El valor umbral de la función de activación, también llamado sesgo, que denotaremos por  $\alpha$ .

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

$$y = \begin{cases} 0 & \text{si } wx - \alpha \leq 0, \\ 1 & \text{si } wx - \alpha > 0. \end{cases} \quad (3.17)$$

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 3.4 y su expresión matemática en la ecuación 3.18.

$$x_2 = \frac{\alpha - x_1 w_1}{w_2}. \quad (3.18)$$

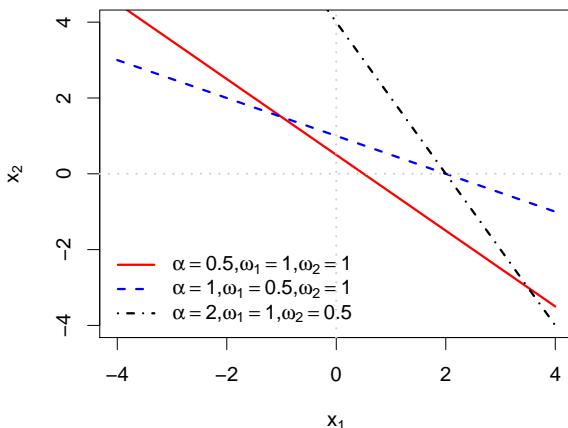
La figura muestra distintos valores para los parámetros  $\alpha$ ,  $w_1$  y  $w_2$ . Si se extiende a más dimensiones, se obtiene el hipерплano separador. Por lo tanto, podemos concluir que una neurona simple con función de entrada basada en suma ponderada y función de activación escalón es un separador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

### 3.3.2. 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:

$$\varepsilon = \frac{1}{2}(c - y)^2, \quad (3.19)$$

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

**Figura 3.4.** Espacio de clasificación de un perceptrón

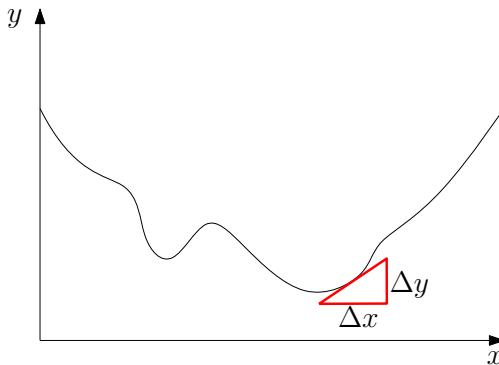
Fuente: elaboración propia

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.

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 3.5 y expresado por la ecuación:

$$\frac{\Delta y}{\Delta x}, \quad (3.20)$$

**Figura 3.5.** Caracterización del mínimo de una función



Fuente: elaboración propia

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.

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:

$$\Delta w = \eta(c - y)d, \quad (3.21)$$

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  $\eta$  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.

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

---

**Algoritmo 1** Seudocódigo del método del descenso del gradiente

---

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

**mientras** ( $y \neq c_i \forall (d_i, c_i) \in D$ ) **hacer**

**para todo** ( $(d_i, c_i) \in D$ ) **hacer**

Calcular la salida  $y$  de la red cuando la entrada es  $d_i$

**si** ( $y \neq c_i$ ) **entonces**

Modificar el vector de pesos  $w' = w + \eta(c - y)d_i$

**fin si**

**fin para**

**fin mientras**

**devolver** El conjunto de vectores de pesos  $W$

---

## 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 tampoco es continua. Por lo tanto, no es posible aplicar el método de descenso del gradiente sobre los valores de salida de la red, pero sí es posible aplicarlo sobre los valores de activación. Esta técnica se conoce como *elementos adaptativos lineales* (*adaptive linear elements* o ADALINE).

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

$$\varepsilon = \frac{1}{2}(c - z)^2, \quad (3.22)$$

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:

$$\Delta w = \eta(c - z)d \quad (3.23)$$

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

---

**Algoritmo 2** Seudocódigo del método ADALINE

---

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

**mientras** ( $a \neq c_i \forall (d_i, c_i) \in D$ ) **hacer**

**para todo** ( $(d_i, c_i) \in D$ ) **hacer**

Calcular el valor de activación  $z$  de la red cuando la entrada es  $d_i$

**si** ( $a \neq c_i$ ) **entonces**

Modificar el vector de pesos  $w' = w - \eta(c - z)d_i$

**fin si**

**fin para**

**fin mientras**

**devolver** El conjunto de vectores de pesos  $W$

---

### 3.3.3. 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:

$$\Delta w_i^j = \eta\sigma'(z^j)(c^j - y^j)x_i^j, \quad (3.24)$$

donde:

- $\sigma'(z^j)$  indica la pendiente (derivada) de la función sigmoidal, 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 3.2c). 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:

$$\delta^j = \sigma'(z^j)(c^j - y^j), \quad (3.25)$$

$$\Delta w_i^j = \eta \delta^j x_i^j. \quad (3.26)$$

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 cómo 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. Para ver el detalle del método y su derivación matemática se puede consultar el apéndice B (Rumelhart, McClelland y col., 1986).

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  $\delta^j$  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).
  - b) A continuación, se modifica el vector de pesos de la neurona de la capa de salida, a partir de la tasa de aprendizaje ( $\eta$ ), el valor delta de la neurona

calculado en el paso anterior ( $\delta^j$ ) y el factor  $x_i^j$  que indica la responsabilidad de la entrada  $i$  de la neurona  $j$  en el error (línea 6).

---

**Algoritmo 3** Seudocódigo del método de retropropagación
 

---

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

```

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

---

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  $\delta^k$  basado en la derivada de la función sigmoide ( $\sigma'(z^k)$ ) y el sumatorio del producto de la delta calculada en el paso anterior ( $\delta^k$ ) 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 se encuentra 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 ( $\eta$ ), el valor delta de la neurona calculado en el paso anterior ( $\delta^k$ ) 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.

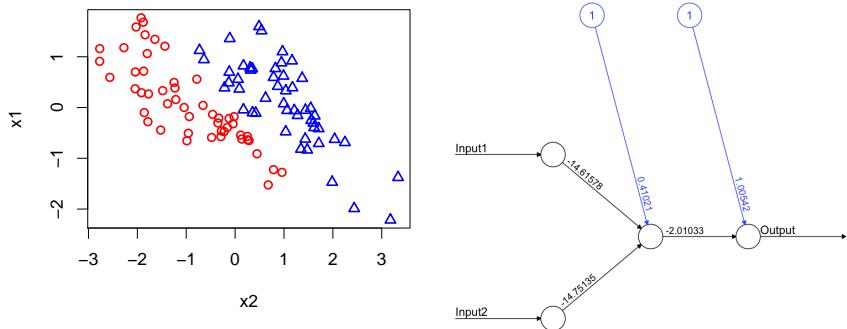
### 3.4. Ejemplo de aplicación

En primer lugar, veremos un ejemplo sencillo con un conjunto de datos linealmente separables (figura 3.6a). 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 3.6b. Este modelo clasifica correctamente todas las instancias del conjunto de entrenamiento y test.

**Figura 3.6.** Ejemplos basado en datos lineales

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



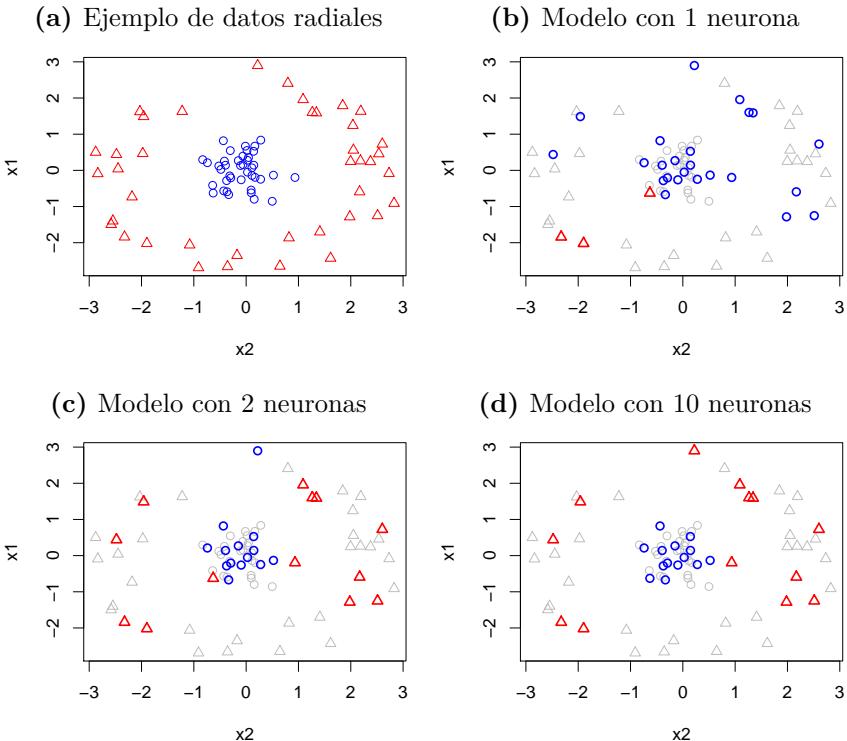
Fuente: elaboración propia

A continuación veremos un ejemplo utilizando datos en un espacio bidimensional, pero a diferencia del caso anterior, estos 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 3.7a. 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 3.7b 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 3.7c, 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 diez neuronas en una única capa oculta nos permite mejorar hasta una precisión de 0,963, es decir, solo una instancia de test incorrectamente clasificada. Los resultados se pueden ver en la figura 3.7d. 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 3.7.** Ejemplos basado en datos radiales

Fuente: elaboración propia

### 3.5. 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. Aun así, se espera que las redes con un número mayor de capas ocultas puedan «aprender» conceptos más abstractos y, por lo tanto, aproximar pro-

blemas 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, Bengio y col., 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 cada una de ellas.

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.

Este problema ha propiciado la aparición de las redes neuronales profundas (*deep neural networks*), que veremos en los siguientes capítulos de este libro.

### 3.6. Resumen

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 explicar 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 de que los datos de entrada se encuentren en el intervalo  $[0, 1]$ , lo cual implica unos procesos previos de transformación de los datos.

Las redes neuronales se utilizan muy a menudo para problemas de clasificación relacionados con imágenes, así como otros problemas en los cuales tenemos datos de muy alta dimensionalidad.



# Capítulo 4

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

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 en 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 tremadamente complejo 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. Este es un campo activo de investigación, donde van apareciendo nuevas técnicas y mejoras sobre las anteriores.

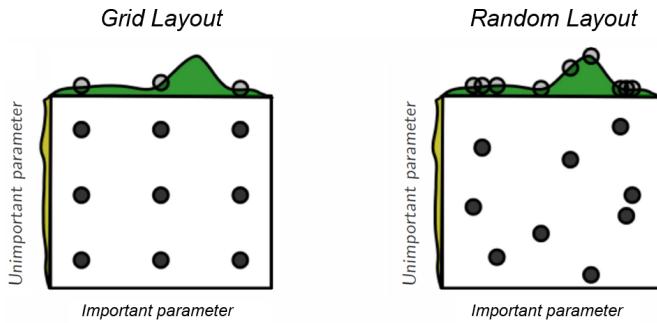
- La técnica conocida como *búsqueda en cuadrícula (grid search)* (Snoek, Larochelle y col., 2012) crea una distribución uniforme de valores entre los distintos parámetros, de forma que se crea una «cuadrícula» uniforme sobre el espacio de posibles combinaciones.
- La *búsqueda aleatoria (random search)* (Bergstra y Bengio, 2012) se basa en escoger combinaciones de valores aleatorios dentro de unos rangos prefijados.

Aunque pueda parecer un contrasentido, la búsqueda aleatoria suele producir resultados similares e incluso superiores en muchas ocasiones (Bergstra y Bengio, 2012). 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 4.1. 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.

#### **4.1. 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.

**Figura 4.1.** Ejemplo de comparación de los métodos *grid* y *random search* con nueve pruebas cada uno



Fuente: Bergstra y Bengio (2012)

#### 4.1.1. Arquitectura de la red

Hemos discutido algunos parámetros referentes al tamaño de las capas de entrada, ocultas y de salida en la sección 3.2.1.

La flexibilidad de las redes neuronales es una 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, solo podamos 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 mucho tiempo, estos hechos convencieron a los investigadores de que no había necesidad de investigar redes neuronales más profundas. Pero pasaron por alto el hecho de que 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 4.1.3) 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.

#### 4.1.2. Épocas, iteraciones y *batch*

En primer lugar, vamos a definir algunos sencillos conceptos importantes que nos resultarán de utilidad en esta sección. 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, intentando 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) divide 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 2.000 ejemplos en lotes de 500, entonces se necesitarán 4 iteraciones para completar 1 época.

#### 4.1.3. *Softmax*

La idea de *softmax* es definir un nuevo tipo de neurona 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, para la neurona  $j$  de la capa  $L$  el valor de entrada es:

$$z_j^L = \sum_k w_{jk}^L y_k^{L-1} + \alpha_j^L, \quad (4.1)$$

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  $\alpha_j^L$  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, es decir,  $\sigma(z_j^L)$ , se aplica la denominada función *softmax*:

$$y_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}, \quad (4.2)$$

donde el denominador suma sobre todas las neuronas de salida.

Los valores de salida de la capa de *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  $j$ .

#### 4.1.4. 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 (Ruder, 2016).

- Momentum (Qian, 1999) es un método de optimización que ayuda a acelerar el *stochastic gradient descent*.

*cent* (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 máxima 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.

- Adagrad (Duchi, Hazan y col., 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.
- 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  $w$ .
- Adam (*adaptive moment estimation*) (Kingma y Ba, 2014), que representa la estimación del momento adaptativo, combina las ideas de optimización de Momentum (seguimiento de un promedio de degradación expo-

nencial decreciente de los gradientes pasados) y además mantiene un seguimiento de un promedio de decaimiento exponencial de los gradientes.

Aunque hasta hace poco se recomendaba el uso del métodos de optimización adaptativos como, por ejemplo, Adam, un estudio reciente de Wilson, Roelofs, Stern y col. (2017) apunta a que 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.

## **4.2. 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.

### **4.2.1. 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 4.2.3) 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.

#### 4.2.2. Velocidad de aprendizaje

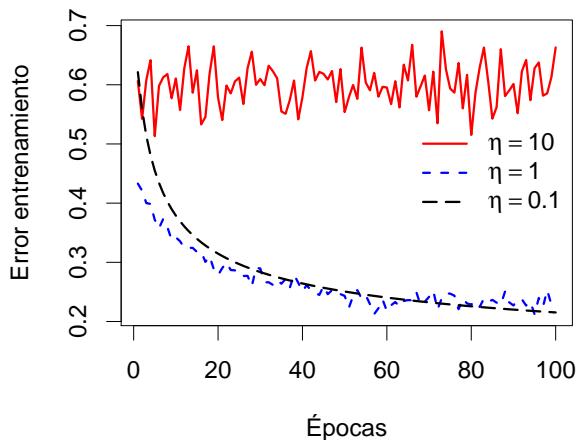
Más allá de las optimizaciones vistas hasta este punto, debemos calibrar algunos parámetros, como por ejemplo la tasa de aprendizaje ( $\eta$ ) o el parámetro de regularización ( $\lambda$ ).

La *velocidad de aprendizaje* (determinada por el parámetro  $\eta$ ) 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 aproxime 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  $\eta$  que sean sensiblemente diferentes entre ellos. Por ejemplo, la figura 4.2 muestra el error en la función de coste durante el entrenamiento utilizando tres valores muy distintos para el parámetro  $\eta$ . 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  $\eta$  que produce un decrecimiento durante las primeras etapas del proceso de aprendizaje. Una vez determinado este valor de umbral, es aconsejable seleccionar valores de  $\eta$  inferiores, en general, una o dos magnitudes inferiores, para asegurar un proceso de entrenamiento adecuado.

**Figura 4.2.** Comparación del error de entrenamiento con múltiples valores de  $\eta$



Fuente: elaboración propia

Una vez determinada la tasa de aprendizaje, esta puede permanecer fija durante todo el proceso. Aun 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* ( $\lambda$ ) se suele desactivar para realizar el calibrado de la tasa de aprendizaje, es decir,  $\lambda = 0$ . 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.

#### 4.2.3. 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, hasta alcanzar un punto donde el aprendizaje se acelera de forma considerable. Cuando usamos la función de coste cuadrática, el aprendizaje es más lento en los casos en lo que 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, aun estando en un estado de saturación. Definimos la función de coste de entropía cruzada (*cross-entropy cost function*) para una neurona como:

$$C = -\frac{1}{n} \sum_d [c \ln(y) + (1 - c)\ln(1 - y)], \quad (4.3)$$

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:

$$C = -\frac{1}{n} \sum_d \sum_j [c_j \ln(y_j^L) + (1 - c_j)\ln(1 - y_j^L)], \quad (4.4)$$

donde  $c = c_1, c_2, \dots$  son los valores deseados y  $y_1^L, y_2^L, \dots$  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 este valor.

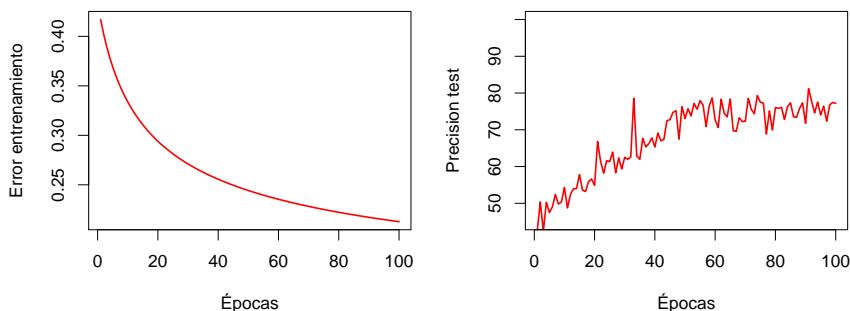
En general, utilizar la función de coste basada en la entropía cruzada suele ser la mejor opción siempre que las neuronas de salida sean neuronas sigmoides, dado que en la inicialización de pesos puede producir la saturación de neuronas de salida cerca de 1, cuando deberían ser 0 o viceversa. En estos 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.

### 4.3. 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*). La figura 4.3a 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 4.3b. En este caso vemos que la precisión de la red mejora hasta llegar al  $epoch=50$ , donde empieza a fluctuar, pero no se aprecia una mejora significativa en la precisión de la red.

**Figura 4.3.** Ejemplo de sobreentrenamiento

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



Fuente: elaboración propia

Por lo tanto, nos encontramos ante un ejemplo donde el error de entrenamiento nos dice que la red está mejorando en cada etapa, mientras 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*).

La red reduce el error porque 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.

#### 4.3.1. Regularización L2

A continuación veremos una de las técnicas de regularización más utilizadas, conocida como regularización L2 (*L<sub>2</sub> regularization* o *weight decay*).

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

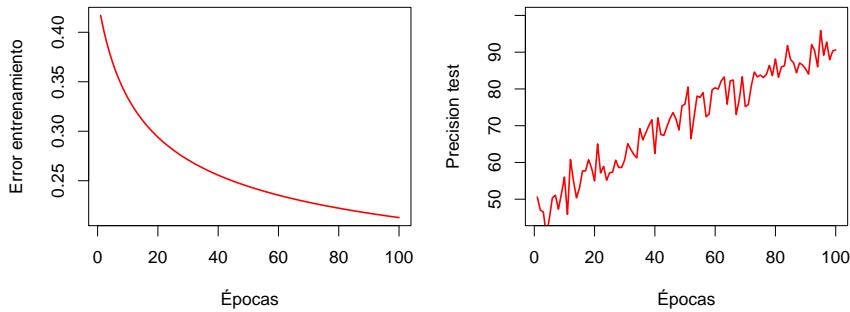
$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2, \quad (4.5)$$

donde  $C_0$  es la función de coste original, en otras palabras, 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  $\lambda$ : cuando este es pequeño, se minimiza la función de coste original; en caso contrario, se opta por premiar los pesos pequeños.

**Figura 4.4.** Ejemplo de regularización

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



Fuente: elaboración propia

La figura 4.4 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,

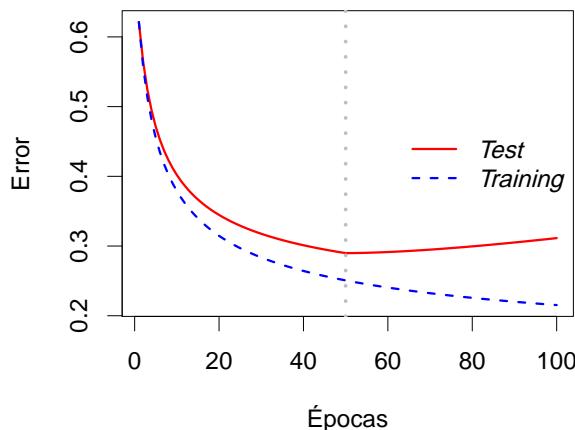
signo de que no se está produciendo sobreentrenamiento. Se ha demostrado, aunque solo de forma empírica, que las redes neuronales regularizadas suelen generalizar mejor que las redes no regularizadas.

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

#### 4.3.2. *Early stopping*

Una técnica simple, pero muy efectiva, consiste en parar el proceso de aprendizaje en el momento en el 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 4.5.** Ejemplo de *early stopping*



Fuente: elaboración propia

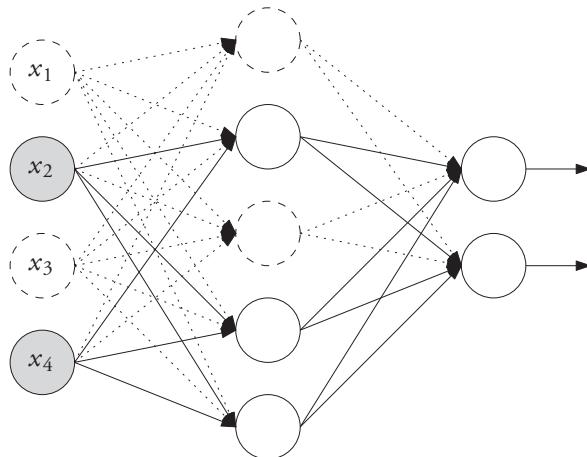
La figura 4.5 muestra un ejemplo en el que un modelo ha sido entrenado durante un cierto periodo. Al inicio, el error en el conjunto de datos de entrenamiento se va reduciendo, al igual que el mismo error en el conjunto de datos de validación. Pero llega un momento en el cual el error sobre el conjunto 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.

#### 4.3.3. *Dropout*

La técnica conocida como *dropout* (Srivastava, Hinton, y col. (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 ???. 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  $p = 0,5$ .

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

**Figura 4.6.** Ejemplo de *dropout*. Las neuronas mostradas en línea discontinua están desactivadas



Fuente: elaboración propia

#### 4.3.4. Expansión artificial del conjunto de datos

Una última técnica de regularización, conocida como expansión artificial del conjunto de datos (*artificially expanding the training data* o *data augmentation*) (Nielsen, 2019), 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.

## 4.4. Resumen

Para finalizar, la tabla 4.1 resume los objetivos de las técnicas de optimización que hemos revisado en este capítulo.

**Tabla 4.1.** Principales objetivos de las técnicas de optimización

Técnicas	Mejoras rendimiento	Velocidad aprendizaje	Overfitting
Arquitectura de la red	X	X	X
Épocas, iteraciones y <i>batch</i>		X	
<i>Softmax</i>	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
<i>Early stopping</i>			X
<i>Dropout</i>			X
Expansión conjunto de datos	X		X

Fuente: elaboración propia

# Capítulo 5

## *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 entrena n 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.
- Preentrenamiento de redes neuronales. Los *autoencoders* pueden facilitar la detección de los atributos más relevantes, y pueden ser empleadas para preentrenamiento 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).

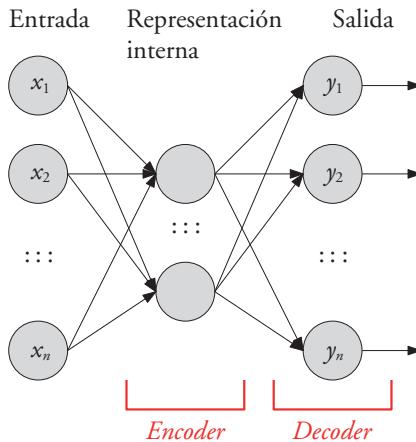
En este capítulo veremos el funcionamiento general de un *autoencoder*, así como sus principales arquitecturas y aplicaciones concretas.

## 5.1. 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».

La figura 5.1 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 este intentará reproducir en la salida la entrada que ha

**Figura 5.1.** Estructura básica de un *autoencoder*

Fuente: elaboración propia

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 5.2.** 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



Fuente: elaboración propia

La figura 5.2 muestra el resultado de la reconstrucción de datos del conjunto de dígitos MNIST<sup>1</sup> 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 solo 32 atributos, para luego volver a expandirse hasta la misma dimensión que los datos originales.

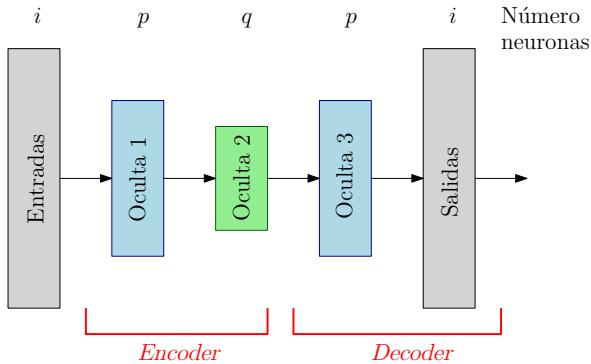
### 5.1.1. *Stacked autoencoders*

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

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, se-

---

<sup>1</sup>[https://en.wikipedia.org/wiki/MNIST\\_database](https://en.wikipedia.org/wiki/MNIST_database)

**Figura 5.3.** Estructura básica de un *stacked autoencoder*

Fuente: elaboración propia

gún la estructura básica representada en la figura 5.3, 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 periodo 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.

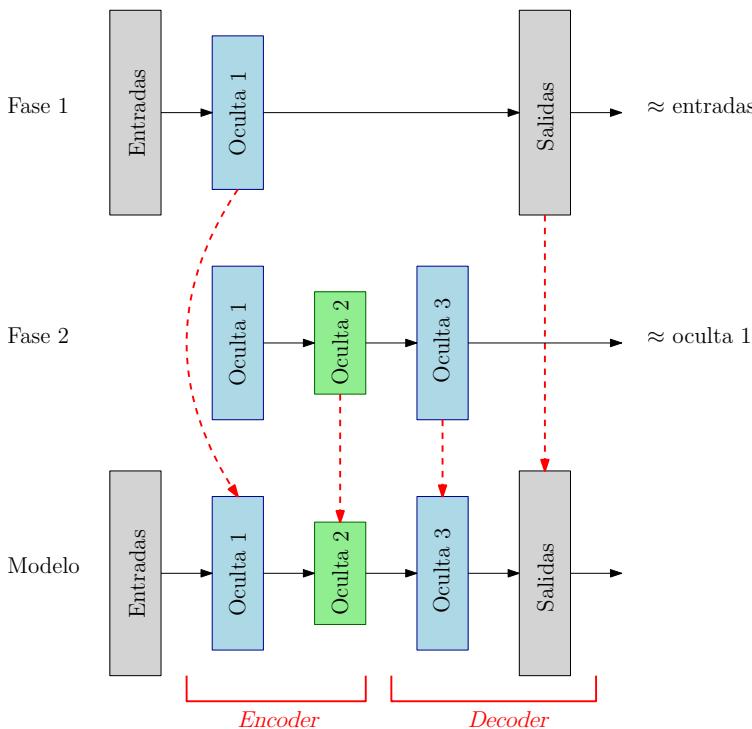
## 5.2. Entrenamiento de un *autoencoder*

El entrenamiento de un *autoencoder* que solo 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 dicho entrenamiento. Es decir, se van entrenando las capas más profundas a partir de los resultados del entrenamiento previo de las capas más superficiales.

**Figura 5.4.** Esquema del entrenamiento iterativo de un *stacked autoencoder*



Fuente: elaboración propia

La figura 5.4 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 solo se incluyen las capas de entrada, la primera capa oculta y la capa de salida. En esta fase el *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 *bias* obtenidos tras el entrenamiento para construir el *autoencoder* final (líneas punteadas en la figura 5.4). De esta forma, se «apilan» las capas previamente entrenadas para construir el modelo final. De aquí el nombre de *stacked autoencoder*.

### 5.3. Preentrenamiento utilizando *autoencoders*

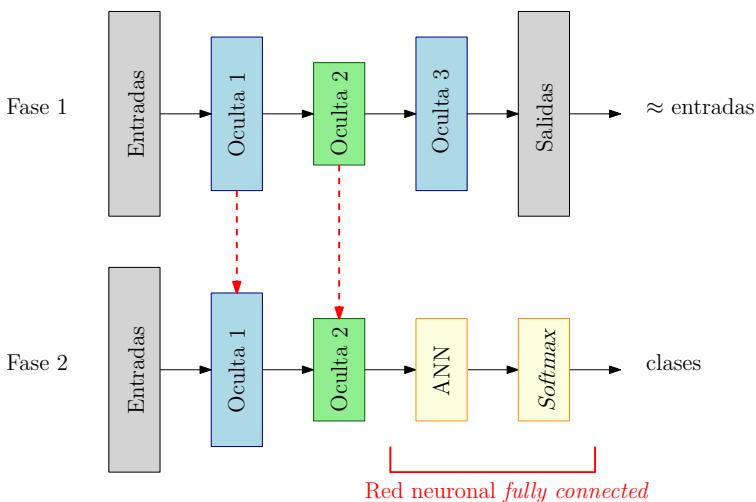
El preentrenamiento con *autoencoders* es una técnica que permite reducir la dimensionalidad de los datos antes del entrenamiento del modelo que se encargará de la tarea de clasificación o regresión.

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 actual-

mente no se utilice demasiado para mejorar la velocidad de los procesos de entrenamiento. Aun así, esta técnica continua siendo muy popular cuando tenemos un conjunto de datos con pocas muestras ( $n$ ) en relación al número de atributos ( $m$ ) y permite mejorar la relación entre registros y atributos.

La idea de este esquema, tal y como se muestra en la figura 5.5, 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 5.5.** Esquema de preentrenamiento utilizando un *autoencoder*



Fuente: elaboración propia

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 neural (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 como 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 preentrenamiento 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 se dispone de un conjunto de datos grande, pero en el cual solo 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 solo los datos etiquetados disponibles.

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

## 5.4. 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 libro una revisión exhaustiva de todos los tipos existentes, veremos algunos de los principales en las siguientes secciones.

### 5.4.1. *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, Larochelle, Bengio y col., 2008; Vincent, Larochelle, Lajoie y col., 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.

### 5.4.2. *Variational autoencoders*

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

- 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 ( $\mu$ ) y una desviación estándar ( $\sigma$ ). Entonces, la codificación se genera a partir de una distribución gaussiana con media  $\mu$  y desviación estándar  $\sigma$ . A partir de aquí el proceso es similar al visto con anterioridad, y se procede a la parte de decodificación según hemos presentado.

#### 5.4.3. 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, Vincent, Muller y col., 2011) intenta forzar al modelo para que genere codificaciones similares para valores de entrada similares.
- *Stacked convolutional autoencoders* (Masci, Meier, Ciresan y col., 2011), especializado en extraer atributos visuales de imágenes a partir de capas convolucionales.
- Las *generative stochastic networks* (GSN) (Alain, Bengio, Yao y col., 2016) son una generalización de los *denoising autoencoders* que permiten la generación de nuevos datos.

# **Parte III**

## **Redes neuronales convolucionales**



# Capítulo 6

## Introducción y conceptos básicos

Iniciaremos este primer capítulo del bloque dedicado a las redes neuronales convolucionales (en inglés, *convolutional neural networks*, CNN) viendo algunos ejemplos de aplicaciones de estos modelos en entornos reales y cotidianos que todos conocemos. En la segunda parte de este capítulo nos centraremos en la operación básica de las CNN, la convolución. Veremos sus fundamentos matemáticos y sus principales propiedades, que nos ayudarán a entender el funcionamiento de este tipo de redes neuronales.

### 6.1. Visión por computador

Las redes convolucionales (LeCun, 1989) son un tipo especial de redes neuronales para procesar datos con tipología cuadriculada, como las imágenes. La visión por computador es una de las áreas que ha avanzado más rápidamente gracias al *deep learning*.

Los modelos de *deep learning* en visión por computador están ayudando y mejorando muchos aspectos de nuestra vida cotidiana como, por ejemplo:

- Están ayudando a los vehículos autónomos (*self-driving cars*) a detectar dónde están los otros coches y personas a su alrededor para evitarlos y no colisionar con ellos.
- Están haciendo que el reconocimiento facial sea considerablemente mejor de lo que había sido hasta hace pocos años. Hoy en día podemos desbloquear nuestro teléfono móvil, e incluso en poco tiempo la puerta de entrada de nuestra casa, simplemente usando nuestra cara.
- Muchos de nosotros tenemos aplicaciones en el teléfono móvil que nos enseñan fotos de nuestro interés, clasifican las que tomamos con nuestra cámara, hacen reconocimiento de objetos en ellas, etc. Muchas de las empresas que construyen estas aplicaciones usan modelos basados en *deep learning* para mostrarnos las imágenes más atractivas, más relevantes o más bonitas.

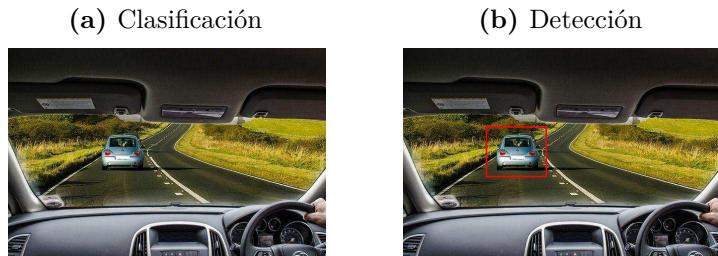
Hay dos razones principales por las que el *deep learning* aplicado en la visión por computador es realmente interesante: (1) primero, los avances tan rápidos en esta área permiten y permitirán crear aplicaciones que nos parecían imposibles hace pocos años; (2) la comunidad que trabaja en visión por computador, ha demostrado ser muy creativa e innovadora en las arquitecturas y algoritmos de *deep learning* que están usando y esto ha creado una sinergias importantes en otras áreas científicas, como por ejemplo en el reconocimiento del habla, que muchas veces se inspira en ideas provenientes del campo de la visión por computador.

Algunos de los problemas de visión por computador que podemos resolver con *deep learning* incluyen:

- Clasificación de imágenes, también llamado reconocimiento de imágenes. Dada una imagen, como por ejemplo la figura 6.1a, queremos responder a la pregunta: ¿hay un coche en esta imagen?
- Detección de objetos. Si estamos construyendo un coche que conduce de forma autónoma, no solamente necesitamos saber si hay coches en la imagen sino que necesitamos obtener la posición de los otros coches, por ejemplo dibujando cajas a su alrededor (figura 6.1b) para no colisionar con ellos.
- Transferencia de estilos. Imaginemos que tenemos una imagen y la queremos pintar con un estilo diferente. En la transferencia neuronal de estilos, tenemos una imagen contenido (figura 6.2a) y una imagen estilo (figura 6.2b), por ejemplo un Van Gogh. La red neuronal une las dos para repintar la imagen contenido obteniendo la imagen de la figura 6.2c.

Uno de los retos más grandes en visión por computador es que las entradas pueden ser realmente grandes. Imaginemos que tenemos imágenes de 64 píxeles de alto por 64 píxeles de ancho ( $64 \times 64$ ). Si son imágenes en color, la dimensión del vector de entrada es de  $64 \times 64 \times 3 = 12.288$  píxeles, porque tenemos 3 canales de color, esto es, RGB. El valor obtenido, 12.288, no parece un vector muy grande para las computadoras modernas, pero las imágenes de  $64 \times 64$  son muy pequeñas. Imaginemos ahora que tenemos imágenes de  $1000 \times 1000 \times 3 = 3 \times 10^6$ . Es decir, necesitamos un vector

**Figura 6.1.** Ejemplos de visión por computador que pueden resolverse mediante *deep learning*: (a) clasificación de imágenes: «¿hay un coche?» y (b) detección de objetos: localización de coches mediante una *bounding box*



Fuente: elaboración propia

**Figura 6.2.** Ejemplos de visión por computador que pueden resolverse mediante *deep learning*: imagen contenido (a) + imagen estilo (b) = imagen resultado (c)



Fuente: elaboración propia

de 3 millones de dimensiones para almacenar dicha imagen. Si en una red neuronal la primera capa oculta (*hidden layer*) tiene 1.000 neuronas (*hidden units*), entonces la dimensión de la primera matriz será de  $1000 \times 3 \times 10^6 = 3 \times 10^9$  posiciones. Si usamos una red conectada completamente (*fully-connected network*), esto implicaría que debemos aprender ¡3 mil millo-

nes de parámetros! No hace falta decir que es poco práctico ya que, aparte de los problemas computacionales y de memoria para entrenar que podamos tener, es muy difícil tener suficientes datos para prevenir el sobreentrenamiento (*overfitting*) de la red.

Las imágenes de  $64 \times 64$  se consideran de baja resolución y las de  $1000 \times 1000$  de alta resolución. Cuando trabajamos en visión por computador no queremos trabajar con imágenes de baja resolución, por lo que si queremos usar imágenes más grandes debemos implementar algún método que nos permita reducir la dimensión de las imágenes conservando el máximo (si es posible, toda) la información que estas contiene. Es aquí donde aparece el concepto de *operación de convolución*, que es el bloque fundamental de las redes convolucionales. La convolución es una operación lineal y las redes convolucionales son redes neuronales que usan esta operación en lugar de la multiplicación de matrices en sus capas.

## 6.2. La operación de convolución

La *convolución* es una operación matemática sobre dos funciones ( $f$  y  $g$ ) que produce una tercera función ( $s$ ) que expresa como la forma de una es modificada por la otra. Típicamente, la operación de convolución se denota con un asterisco (\*):

$$s(t) = (f * g)(t) \int f(\tau)g(t - \tau)d\tau. \quad (6.1)$$

La operación de convolución se puede aplicar en distintas áreas y aplicaciones tales como probabilidad, estadística, visión por computador, procesamiento del lenguaje natural, imagen y procesamiento de la señal, ingeniería y ecuaciones diferenciales.

En la terminología de las redes convolucionales, el primer argumento (la  $f$  en la ecuación 6.1) de la convolución se refiere a la *entrada (input)* y el segundo argumento (la función  $g$ ) es el *filtro o kernel*. La salida de la función ( $s$ ) se refiere al *mapa de características o feature map*. Normalmente, los datos usados por las redes convolucionales son datos discretos y usaremos la convolución discreta:

$$s(t) = (f * g)(t) \sum_{\tau=-\infty}^{\infty} f(\tau)g(t - \tau). \quad (6.2)$$

En aplicaciones de aprendizaje automático (*machine learning*) la entrada es un vector multidimensional de datos y el *kernel* es un vector multidimensional de parámetros. En visión por computador, trabajamos en imágenes y las convoluciones se aplican sobre más de un eje. Por ejemplo, si usamos una imagen ( $I$ ) de dos dimensiones como entrada, también usaremos un *kernel* ( $K$ ) de dos dimensiones y, dado que la convolución cumple con la propiedad conmutativa, podemos usar la siguiente formula:

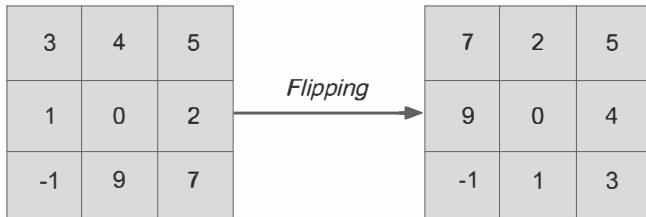
$$S(i, j) = (I * K)(i, j) \sum_m \sum_n I(i - m, j - n)K(m, n). \quad (6.3)$$

La propiedad conmutativa<sup>1</sup> en la operación de convolución se cumple porque hacemos un *flipping* del *kernel* relativo a la entrada, como se muestra en la figura 6.3. No obstante, esta propiedad no es importante para la implementación de una red neuronal y es por ello que para mejorar la eficiencia, la mayoría de librerías que trabajan con redes convolucionales implementan la función de *cross-correlation*, aunque por convención la llaman convolución:

---

<sup>1</sup> $f * g = g * f$

**Figura 6.3.** Ejemplo de *flipping* de un *kernel* que se usa antes de la operación de convolución



Fuente: elaboración propia

$$S(i, j) = (I * K)(i, j) \sum_m \sum_n I(i + m, j + n)K(m, n). \quad (6.4)$$

En resumen, por convención, en el área de *deep learning* no nos importa mucho hacer el *flipping* del *kernel* y, técnicamente, lo que estamos haciendo se llama operación de *cross-correlation*. Sin embargo, la literatura que encontramos relacionada con el área de *deep learning* llama a esta operación «convolución», por convención. En este libro usaremos esta misma convención para mantener la coherencia con otros textos. En otras áreas, como el procesamiento del señal y otras especialidades matemáticas, hacer el *flipping* del *kernel* hace que se cumpla la propiedad asociativa,<sup>2</sup> pero cuando trabajamos en *deep learning* esto realmente no es importante y nos permite simplificar el código de los modelos, mientras que las redes neuronales funcionan igual de bien.

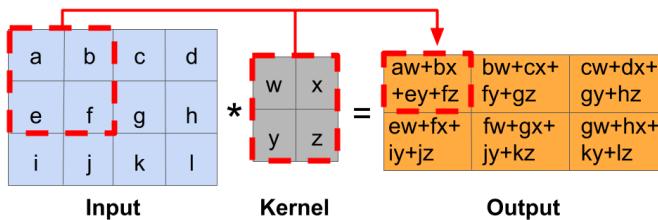
La figura 6.4 muestra un ejemplo de convolución aplicada a un vector de datos de dos dimensiones sin hacer *flipping* del *kernel*. La salida se muestra solamente en las posiciones

---

<sup>2</sup> $(f * g) * h = f * (g * h)$

donde el *kernel* cuadra con la imagen (no sale del borde). Se muestra un cuadrado y flechas para indicar como el elemento superior izquierdo se forma aplicando el *kernel* a la parte correspondiente superior izquierda de la región de la entrada.

**Figura 6.4.** Ejemplo de una convolución 2D sin *flipping* del *kernel*



Fuente: Bengio, Goodfellow y Courville (2016)

## 6.3. Ventajas derivadas de la convolución

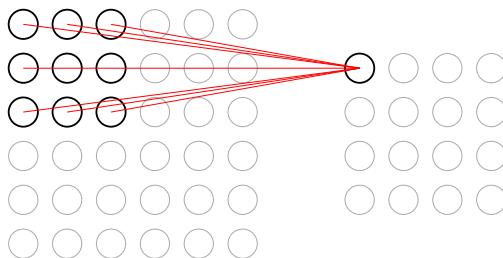
La operación de convolución descrita anteriormente permite a las redes neuronales convolucionales una mejora en dos temas muy importantes para el entrenamiento de las redes (Bengio, Goodfellow y Courville, 2016), que veremos en las siguientes secciones.

### 6.3.1. Interacciones o conexiones dispersas

Las redes neuronales tradicionales usan la multiplicación de matrices para describir la interacción entre cada unidad de entrada y cada unidad de salida. En este tipo de arquitectura, cada neurona de la red está conectada a todas las neuronas de las capas adyacentes (anterior y posterior). Este tipo de redes reciben el nombre de redes totalmente conectadas (*fully*

*connected networks*). Las CNN usan un *kernel* mucho más pequeño que la entrada para trabajar con conexiones dispersas y las neuronas de una capa se conectan solo a un subconjunto de las neuronas de la capa siguiente, asimismo están conectadas solo a un subconjunto de neuronas contiguas de la capa anterior.

**Figura 6.5.** Ejemplo de interacciones dispersas de una red convolucional



Fuente: elaboración propia

La figura 6.5 muestra un ejemplo de esta arquitectura, donde de cada grupo de  $3 \times 3$  neuronas de la capa  $i$  se conecta con una neurona de la capa  $i + 1$ . En particular, el resto de los valores de los píxeles no tienen efecto sobre la salida, y a esto se refieren las interacciones dispersas. Esto implica que trabajemos con menos parámetros reduciendo a la vez los requerimientos de memoria y el tiempo de computación.

La estructura y tamaño del conjunto de neuronas de la capa  $i$  que se conectan con una neurona de la capa  $i + 1$  forma lo que se conoce como el «campo receptivo local» (*local receptive fields*). Por ejemplo, en el ejemplo anterior hemos visto un campo receptivo local de  $3 \times 3$ .

### 6.3.2. Compartición de parámetros

Según la estructura presentada, cada neurona oculta tiene un sesgo y un conjunto de pesos que depende del tamaño de su campo receptivo local. A diferencia de lo visto anteriormente, en este tipo de redes se usan los mismos pesos y sesgos para todas las neuronas ocultas de una misma capa. En otras palabras, para una neurona oculta, su salida es:

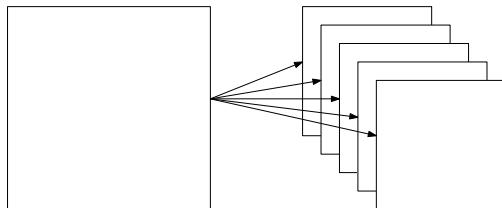
$$\sigma \left( b + \sum_{l=0}^{n-1} \sum_{m=0}^{n-1} w_{l,m} a_{j+l,k+m} \right), \quad (6.5)$$

donde  $\sigma$  representa la función de activación,  $b$  es el valor compartido de sesgo,  $w_{l,m}$  es una matriz de  $n \times n$  que contiene los pesos compartidos de las neuronas de una misma capa, y  $a_{x,y}$  es el valor de entrada de la posición  $(x, y)$ .

Esto significa que todas las neuronas de la capa oculta detectan exactamente la misma característica, solo que realizan esta función en diferentes ubicaciones de la imagen de entrada. Supongamos que los pesos y sesgos son tales que la neurona oculta puede distinguir, digamos, un borde vertical en un campo receptivo local particular. Esta misma habilidad es probable que sea útil en otros lugares de la imagen. Por lo tanto, es útil aplicar el mismo detector de características en todas partes. Para ponerlo en términos un poco más abstractos, las redes convolucionales están bien adaptadas a la invariancia de las características a detectar.

Generalmente, se llama *mapa de características* (*feature map*) a la relación entre las neuronas de la capa de entrada a la capa oculta. Los *pesos compartidos* (*shared weights*) y el *sesgo compartido* son los pesos y sesgo que definen el mapa de características. A menudo se dice que los pesos compartidos y el sesgo definen un *kernel* o filtro.

**Figura 6.6.** Ejemplo de estructura con cinco mapas de características en la capa oculta



Fuente: elaboración propia.

Las redes convolucionales deberán utilizar múltiples mapas de características según el número de patrones a detectar en los datos en entrada. Por ejemplo, la figura 6.6 muestra una estructura con una capa de entrada conectada a cinco mapas de características que forman la primera capa oculta de neuronas.

Una gran ventaja de compartir pesos y sesgos es que reduce en gran medida el número de parámetros involucrados en una red convolucional. Esto, a su vez, resultará en un entrenamiento más rápido para este modelo y, en última instancia, nos ayudará a construir redes profundas usando capas convolucionales.

## 6.4. Conclusiones

Con los mecanismos vistos anteriormente, las redes convolucionales tienen muchos menos parámetros que aprender. Este es un factor clave en el proceso de entrenamiento, ya que permite reducir el tiempo de este y del tamaño del conjunto de datos necesario para «evitar» el sobreentrenamiento.

Otra ventaja importante de las redes convolucionales, que en este caso deriva de la compartición de parámetros, es

una propiedad conocida como *representaciones equivalentes*. Esta propiedad se refiere al hecho de que la salida es invariante a los movimientos de translación de las entradas. La estructura convolucional ayuda a las redes neuronales a codificar el hecho de que una imagen desplazada (*shifted*) unos píxeles debe resultar en una estructura muy similar de características. Esta propiedad puede ser muy útil cuando nos preocupa la presencia de una determinada forma u objeto dentro de una imagen, no su posición. En este sentido, las redes convolucionales son bastante buenas capturando la invariante en translación.

En resumen, el bloque más importante de las CNN son las capas convolucionales. Las neuronas de la primera capa convolucional no están conectadas a cada píxel de la imagen de entrada, sino que solo lo están a los píxeles de sus respectivos campos receptivos locales. En consecuencia, cada neurona en la segunda capa convolucional está conectada solamente con las neuronas localizadas en un pequeño rectángulo de la primera capa. Esta arquitectura permite a la red concentrarse en las características de bajo nivel en las primeras capas ocultas, para ensamblarlas luego en características de más alto nivel en la siguiente capa, y así progresivamente. Esta estructura jerárquica es muy común en las imágenes del mundo real, y es una de las razones por las cuales las CNN funcionan tan bien para resolver problemas de reconocimiento de imágenes.

# Capítulo 7

## Componentes y estructura de una CNN

Iniciaremos este capítulo centrándonos en los detalles de las capas convolucionales (sección 7.1), para entender mejor el funcionamiento de estas operaciones, junto con los parámetros relevantes para su correcto funcionamiento.

A continuación, en la segunda parte de este capítulo (sección 7.2), veremos las principales capas que coexisten, junto con las capas convolucionales, en las CNN. Como veremos, las capas convolucionales son el núcleo de las CNN, pero es necesario añadir otro tipo de capas para obtener un buen rendimiento global de la red.

Finalizaremos viendo un ejemplo completo de red neuronal convolucional, en la sección 7.3, donde veremos una arquitectura básica y analizaremos su funcionamiento.

## 7.1. La capa de convolución

En las siguientes secciones veremos algunos detalles relevantes referentes a la implementación de las capas de convoluciones. En concreto, nos fijaremos en los siguientes conceptos relacionados con las capas de convolución:

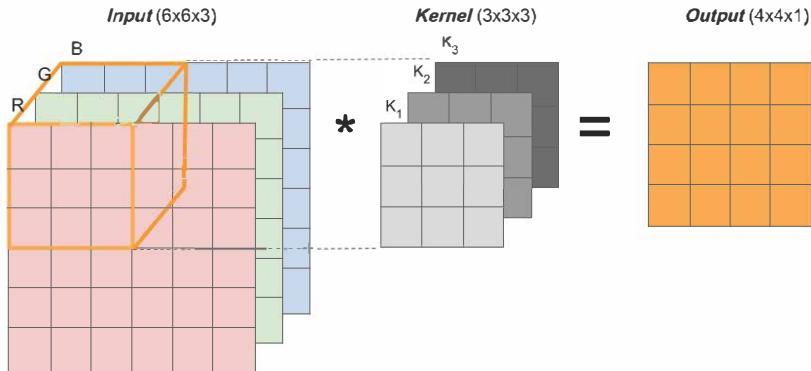
- Filtros o *kernel*.
- *Padding*.
- Convoluciones por pasos *strided convolutions*.

### 7.1.1. Filtros o *kernel*

El *kernel*, también llamado *filtro*, es la matriz de pesos de una capa de convolución dentro de una CNN. Implementa una convolución en toda la matriz de entrada. El *kernel*, normalmente, tiene un tamaño mucho más pequeño que la entrada. La entrada puede ser una imagen en escala de gris o en color, generalmente, RGB. Si es una imagen RGB, a la tercera dimensión se la llama *canal* o *volumen*. La representación mas común es una imagen RGB donde cada canal es una matriz de dos dimensiones (2D) que representa un color. La figura 7.1 muestra un ejemplo donde se puede ver una entrada en RGB de  $6 \times 6$  píxeles, a la cual se le aplica un filtro de  $3 \times 3$ . Para calcular la salida se utiliza la ecuación 6.2 sobre las tres dimensiones o canales, esto es,  $R * K_1 + G * K_2 + B * K_3$ .

Es importante notar que el *kernel* siempre tiene el mismo número de canales que la entrada. Es por ello que se produce una reducción de la dimensión (en todos los canales), excepto en el caso de emplear un *kernel* de  $1 \times 1$ . Podemos tener diferentes *kernels* que capturan diferentes características, tal

**Figura 7.1.** Convolución de un *kernel* sobre una imagen en tres dimensiones

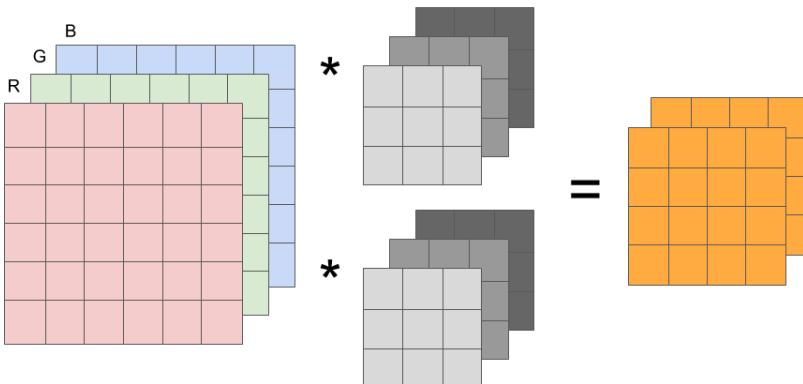


Fuente: elaboración propia

y como muestra la figura 7.2, donde hay dos *kernels*, cada uno con dimensión  $3 \times 3 \times 3$ . Para calcular la salida se usa la ecuación 6.2 sobre las tres dimensiones o canales, esto es,  $R * K_1 + G * K_2 + B * K_3$ . Cabe destacar que en este caso el número de canales de la salida es dos, ya que este valor viene determinado por el número de *kernels* usados para la operación de convolución.

Las dimensiones de la entrada son  $(n_w, n_h, n_{canal})$ . Cuando el canal = 1, es una entrada de dos dimensiones (2D), es decir, una imagen en escala de gris. Las dimensiones del *kernel* son  $(n_k, n_k, n_{canal})$ . Aunque no es muy común, el *kernel* no tiene porque ser siempre cuadrado, pudiendo ser su dimensión  $(n_{k1}, n_{k2}, n_{canal})$ . Entonces, la dimensión de la salida será  $(n_w - n_{k1} + 1, n_h - n_{k2} + 1, 1)$ , y si tenemos  $n$  *kernels* diferentes, entonces la dimensión de la salida será de  $(n_w - n_{k1} + 1, n_h - n_{k2} + 1, n)$ .

**Figura 7.2.** Convolución de dos *kernels* sobre una imagen en tres dimensiones



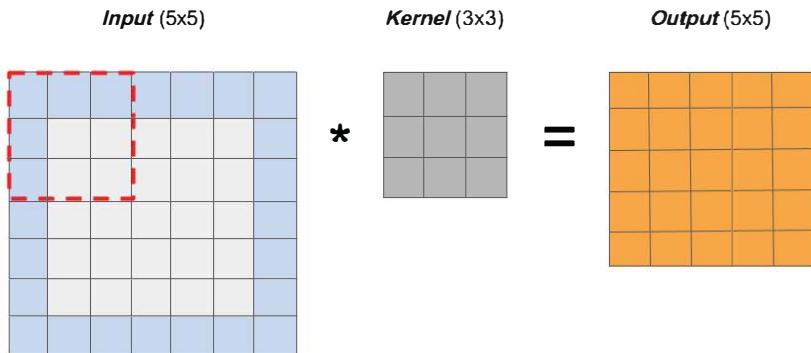
Fuente: elaboración propia

### 7.1.2. *Padding*

El *kernel* y el *stride* (ver sección 7.1.3) son técnicas para reducir la dimensión de la imagen de entrada. Generalmente, la dimensión de la salida de una capa de convolución suele ser más pequeña que la dimensión de la entrada de esta. No obstante, si queremos construir una red neuronal profunda (*deep*), no queremos que esto suceda de forma muy rápida.

Un *kernel* pequeño puede realizar la función que deseamos, pero para mantener la dimensión (en otras palabras, dimensión de la entrada = dimensión de la salida) usamos lo que se llama *zero padding* ( $p = 1$ ). Básicamente se trata de añadir ceros al borde de la entrada, tal y como muestra la figura 7.3. De esta forma, la dimensión de la salida será la misma que la dimensión de la entrada, aun después de aplicar la convolución.

**Figura 7.3.** Ejemplo de aplicación de *cero padding* ( $p = 1$ ) para mantener la dimensión de la salida. El borde exterior (cuadro marcado en azul) de la entrada representa las celdas con ceros añadidos



Fuente: elaboración propia

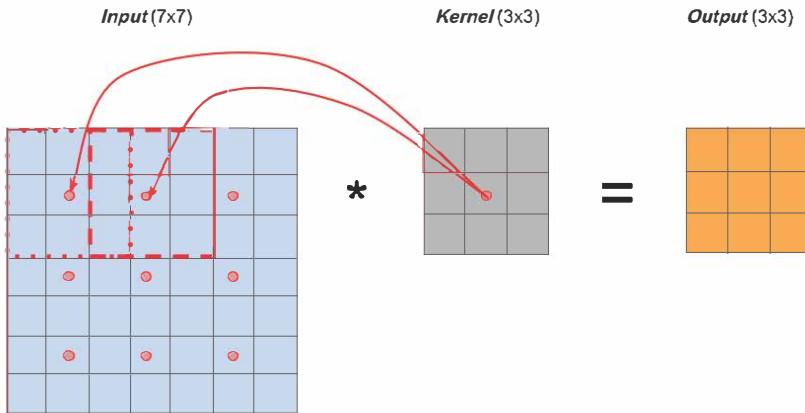
### 7.1.3. Convoluciones por pasos

Como ya se ha mencionado, una de las ventajas de las CNN es que mediante la operación de convolución reducen la dimensión de los datos de entrada y son más eficientes en los tiempos de ejecución. Usando las *convoluciones por pasos* o *strided convolutions* podemos ser todavía mas eficientes, aunque con el coste adicional de perder algunas características en la salida.

Digamos que tenemos una entrada y un filtro, y que en lugar de hacer la convolución de la forma tradicional (en cada celda de la entrada) lo hacemos con un paso = 2 ( $stride = 2$ ). Esto quiere decir que en lugar de aplicar la convolución en celdas consecutivas, la aplicaremos en las celas con un salto de 2, tanto en la dimensión horizontal como en la vertical. La figura 7.4 muestra un ejemplo de convolución con  $stride = 2$ ,

marcando con un punto las celdas de la entrada donde se aplica la convolución.

**Figura 7.4.** Ejemplo de una convolución usando un *stride* de 2 ( $s = 2$ ) y un *padding* de 0 ( $p = 0$ ). Las celdas marcadas con un punto rojo de la entrada son aquellas donde se aplicará el *kernel* de convolución



Fuente: elaboración propia

Las dimensiones de la salida usando *stride* y *padding* vienen definidas por la siguiente formula:

$$\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor, \quad (7.1)$$

donde  $n$  es la dimensión de la entrada,  $f$  la dimensión del filtro (o *kernel*),  $p$  el *padding* y  $s$  el paso (o *stride*).

## 7.2. Otras capas de las CNN

Aunque las capas convolucionales son el factor clave de una CNN, las redes convolucionales no están formadas únicamente

por capas convolucionales. Al contrario, encontramos una notable diversidad de capas de otros tipos que ayudan a generar la información deseada a partir de los datos de entrada.

En esta sección revisaremos las principales capas que pueden coexistir dentro de una CNN con las capas de convolución. En concreto, veremos cuatro tipos de capas:

- Capa de agrupamiento (*pooling*).
- Capa totalmente conectada (*fully connected*).
- Capa ReLU.
- Capa de *dropout*.

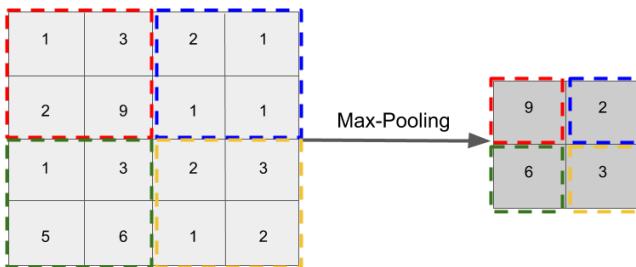
### 7.2.1. Capa de agrupamiento (*pooling*)

A parte de las capas convolucionales, las CNN usan a menudo *capas de agrupamiento* (*pooling layers*) con el doble propósito de hacer que algunas de las características sean más robustas y más eficientes.

Existen diferentes tipos de *pooling*, siendo el *max-pooling* el más usado en las redes convolucionales, que simplemente selecciona el valor máximo del conjunto de valores de entrada.

Imaginemos que tenemos una entrada de  $4 \times 4$  y queremos aplicar *max-pooling*. La salida de esta capa tendrá una dimensión de  $2 \times 2$ . Para ello, cogemos la entrada y lo dividimos en diferentes regiones, cuatro en este caso concreto. La figura 7.5 muestra la división de regiones diferenciadas por colores. Entonces la salida se corresponde al máximo de cada región, ya que estamos aplicando la función *max-pooling*. En el ejemplo en cuestión, el máximo de la región superior izquierda es 9, y así sucesivamente para las demás regiones. Para calcular los

**Figura 7.5.** Ejemplo de agrupamiento basado en la función *max-pooling*



Fuente: elaboración propia

valores de la salida estamos aplicando un filtro  $f$  de  $2 \times 2$  y un *stride* igual a dos ( $s = 2$ ), estos son los hiperparámetros de la capa de *max-pooling*.

La ecuación 7.1 nos sirve también aquí para calcular la dimensión de la salida en las capas de agrupamiento.

La intuición detrás de la operación de *max-pooling* en las redes convolucionales es que nos quedamos con aquellas características más frecuentes y más relevantes para cada cuadrante de la imagen. Una propiedad interesante de esta operación es que tiene un conjunto de hiperparámetros, pero estos no se tienen que aprender. Es decir, no es necesario que el algoritmo de entrenamiento aprenda estos parámetros, y simplemente se fijan los valores de  $f$  y  $s$  al inicio.

Si tenemos una entrada con un *canal*  $> 1$ , las salidas tendrán el mismo número de canales. Por ejemplo, si tenemos una entrada de  $5 \times 5 \times 2$  con  $f = 3$  y  $s = 1$ , la salida será de  $3 \times 3 \times 2$ . La operación de *max-pooling* se aplica de forma independiente sobre cada canal.

Hay otro tipo de operación de agrupamiento que no se usa tan a menudo, conocida com *average-pooling*. En este caso, en lugar de hacer el máximo sobre los valores de la región, hace-

mos el promedio (*average*) de todos los valores. En el ejemplo de la figura 7.5, si se emplea la función *average-pooling*, la salida sería  $\{3,75, 1,25, 3,75, 2\}$ .

Actualmente, la función *max-pooling* es la más utilizada en la mayoría de redes profundas, excepto en redes neuronales muy profundas (veremos un ejemplo en la sección 8.1).

### 7.2.2. Capa totalmente conectada (*fully connected*)

Hemos visto ejemplos de capas totalmente conectadas (*fully connected layers*, FC) en las redes neuronales tradicionales, donde cada salida de la capa  $i$  se conecta con todas las entradas de la capa  $i + 1$ . Esta capa, básicamente, coge un conjunto de entradas (cuálquiera que sea la capa precedente, ya sea una convolución, una capa ReLU o una capa de agrupación) y da como salida un vector  $N$ -dimensional, donde  $N$  es el número de clases que el modelo tiene que escoger.

Por ejemplo, si queremos clasificar dígitos,  $N$  debería ser 10, ya que existen 10 dígitos diferentes, esto es,  $\{0, 1, 2, \dots, 9\}$ . Cada número de este vector  $N$ -dimensional representa una probabilidad de pertenecer a una clase. En el caso de emplear una función de salida *softmax* (ver sección 4.1.3), el vector resultado para el programa de clasificación de dígitos podría ser:  $\{0, 0, 1, 0, 1, 0, 75, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5\}$ , que representa un 10 % que la imagen contenga un 1, un 10 % que la imagen contenga un 2, un 75 % de probabilidad que la imagen contenga un 3 y un 5 % de probabilidad que la imagen sea un 9.

De esta forma, la capa *fully connected* recibe la salida de la capa anterior (que representan mapas de activación (*feature maps*) de características de alto nivel) y determina cuáles son las características que correlacionan mejor con cada clase en particular.

Por ejemplo, si el objetivo es predecir si una imagen contiene un perro, tendrá valores altos en los mapas de activación que representen características de alto nivel como patas, cola, etc. De forma similar, si se quiere predecir si la imagen contiene un pájaro, tendrá valores altos en los mapas de activación que representen características de alto nivel como alas, pico, etc.

Básicamente, la función principal de insertar una o varias capas totalmente conectadas al final de la red convolucional es determinar cuáles son las características de alto nivel que correlacionan con una clase en particular. Entonces, durante el proceso de entrenamiento se deberán fijar estos pesos específicos, de forma que cuando se calcula el producto entre los pesos y la capa anterior, se obtienen las probabilidades correctas para cada clase.

### 7.2.3. Capa ReLU (*rectified linear units*)

Inmediatamente después de cada capa de convolución, es muy habitual aplicar una capa no lineal (o capa de activación). El propósito de esta capa es introducir no linealidad al sistema, que básicamente ha estado realizando operaciones lineales durante las capas de convolución (multiplicaciones elemento a elemento y sumas). En el pasado, se usaban funciones no lineales como *tanh* o *sigmoid*, pero los investigadores han encontrado que la función ReLU funciona mucho mejor, ya que la red es capaz de entrenar más rápido sin que esto afecte a la precisión final del modelo.

También ayuda a aliviar el problema de la desaparición del gradiente (*vanishing gradient*), provocando que el entrenamiento sea muchísimo más lento en las primeras capas de la red que en las últimas a causa del decrecimiento exponencial del gradiente a través de las capas (véase la sección 3.5).

La capa ReLU aplica la función  $f(x) = \max(0, x)$  a todos los valores de la entrada. En términos básicos, esta capa solo cambia los valores negativos por 0, incrementando las propiedades no lineales del modelo y de toda la red, sin que esto afecte a los campos receptivos de la capa de convolución (Nair y Hinton, 2010).

#### 7.2.4. Capa de *dropout*

Las capas de *dropout* tienen una función muy específica en las redes neuronales: prevenir el sobreentrenamiento u *overfitting* (ver sección 4.3). Esta capa desactiva (*drops out*) un número aleatorio de entradas, poniéndolas a un valor igual a 0 (ver sección 4.3.3). El principal beneficio del *dropout* es que está forzando a la red a ser redundante, y esta será capaz de dar la clasificación y salidas correctas a pesar de que algunas entradas estén inactivas. De esta forma prevenimos que la red se ajuste demasiado al conjunto de datos de entrenamiento y ayuda a prevenir el problema de *overfitting*. Esta capa se usa solamente durante el entrenamiento, pero no durante el proceso de test (Krizhevsky, Sutskever y col., 2014).

### 7.3. Estructura de una red neuronal convolucional

A partir de las diferentes funciones, operaciones y capas que hemos visto hasta este punto, veremos como podemos «agrupar» todas estas piezas para construir una CNN.

Como ya hemos comentado, los filtros de las primeras capas de la red detectan características de bajo nivel, tales como aristas y curvas. Como es de imaginar, para predecir si una imagen tiene algún tipo de objeto necesitamos que la red sea

capaz de encontrar características de alto nivel, como pueden ser las manos, los ojos, las patas, etc.

Cuando hablamos de la primera capa, la entrada es la imagen original. No obstante, cuando hablamos de la segunda capa, la entrada es el mapa (o mapas) de activación que resultan de la primera capa. Cada capa de entrada, básicamente, está describiendo las localizaciones de la imagen original donde aparecen determinadas características. Cuando aplicamos un conjunto de filtros sobre estos (pasada la segunda capa) la salida serán activaciones que se corresponden a características de más alto nivel. Algunas de estas características pueden ser semicírculos, cuadrados (combinaciones de varias líneas rectas) o polígonos (combinaciones de curvas y líneas rectas), entre muchos otros. A medida que avanzamos en la red y vamos aplicando más y más capas de convolución, obtenemos mapas de activación que representan características más complejas y abstractas. Al final de la red, podemos tener filtros que se activan cuando hay un determinado dígito en la imagen, filtros que se activan cuando hay objetos de color verde, etc.

Si tenemos una entrada de  $32 \times 32 \times 3$ , la salida de la red después de la primera capa de convolución aplicando tres filtros de  $5 \times 5 \times 3$  tendrá una dimensión de  $28 \times 28 \times 3$ . Si después pasamos por otra capa de convolución, la salida de la primera capa pasa a ser la entrada de la segunda, y esta es un poco más difícil de visualizar. Por ejemplo, la figura 7.6 muestra la visualización de las características (Zeiler y Fergus, 2014) de una red convolucional similar a ImageNet (Deng, Dong y col., 2009). Se muestran los patrones de las *top 9* activaciones de los mapas de características del conjunto de datos de validación. Son patrones reconstruidos de los datos de validación que causan activaciones altas en los mapas de características.

**Figura 7.6.** Ejemplo de filtros en diferentes capas de una red convolucional



Fuente: Zeiler y Fergus (2014)

Otra característica importante es que, a medida que nos internamos más y más profundo en la red, los filtros tienen un campo receptivo más grande. Es decir, consideran información de una área más grande de la imagen original.

Imaginemos que queremos resolver el problema de reconocimiento de dígitos en una imagen y tenemos una entrada de  $32 \times 32 \times 3$ , en otras palabras, una imagen RGB de  $32 \times 32$  píxeles. Vamos a crear una red neuronal inspirada en LeNet-5 (LeCun, Bottou y Haffner, 1998) para resolver este problema. El ejemplo se puede ver en la figura 7.7.

Vamos a usar 6 filtros de  $5 \times 5$ , un *stride* de 1 y sin *padding* en la primera capa. Aplicamos los 6 filtros, añadimos el sesgo (*bias*), aplicamos no linealidad y la salida será de  $28 \times 28 \times 6$ , lo llamaremos `conv1`. Seguidamente aplicamos una capa de agrupación empleando la función *max-pooling* con unos valores de  $f = 2$  y  $s = 2$ . Si no se menciona el *padding*, asumimos que este es 0. Este proceso reduce el alto y ancho de la imagen por un factor de 2, obteniendo una salida de  $14 \times 14 \times 6$ . El número de canales se mantiene igual, esto es, 6. Nos referimos a esta salida como `Pool1`.

En la literatura de las redes neuronales convolucionales, existe una convención en referencia a lo que llamamos «capa»: una capa de convolución + una capa de agrupamiento forman una sola «capa» de la red neuronal. En el ejemplo que estamos viendo, la `conv1` + `pool1` forman la **capa 1** de la red (*layer 1*).

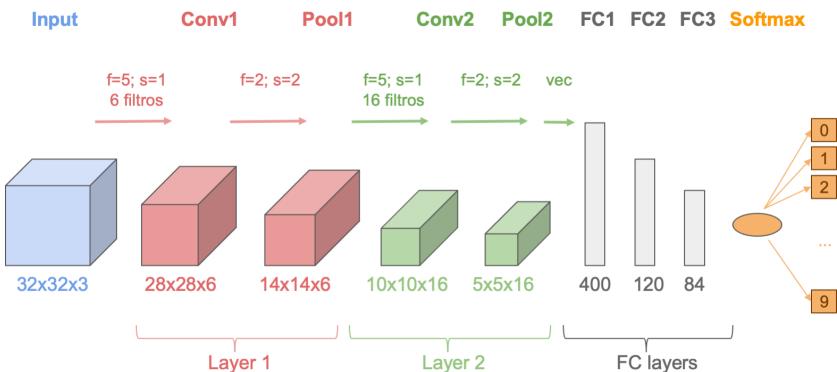
Cuando se reporta el número de capas de una red neuronal, normalmente solo se reporta el número de capas que tiene pesos, es decir, que tiene parámetros. Dado que la capa de agrupamiento no tiene pesos (solo hiperparámetros), no se reporta cuando mencionamos el número de capas que tiene la red.

Volviendo al ejemplo, dado el volumen de entrada de  $14 \times 14 \times 6$ , aplicamos otra capa convolucional. Esta vez con 16 filtros de  $5 \times 5$  y  $s = 1$ , la salida de la cual será de  $10 \times 10 \times 16$ , al que llamaremos `conv2`. Aplicamos después una función *max-pooling* con  $f = 2$  y  $s = 2$ , obteniendo una salida de  $5 \times 5 \times 16$  al que llamamos `pool2`. Estas dos capas forman la **capa 2** de la red neuronal (*layer 2*).

En la parte final de esta red vamos a incorporar las capas *fully connected*. En primer lugar, debemos reestructurar la salida de `pool2` en forma de vector unidimensional, obteniendo un vector de  $400 \times 1$  (a partir del vector multidimensional de  $5 \times 5 \times 16$ ). A continuación, lo que vamos a hacer es conectar estas 400 unidades, estas 400 neuronas, con la siguiente capa de 120 unidades. Esto es la primera capa totalmente conectada que llamaremos `FC2`, donde tenemos 400 unidades conectadas densamente con las 120 unidades, es decir, cada una de las 400 unidades está conectada con cada una de las 120 unidades. Finalmente añadimos otra capa de 84 unidades, a la que llamaremos `FC3`, obteniendo 84 unidades que serán la

entrada de una capa *softmax*. Si lo que queremos hacer es reconocimiento de dígitos, esta será una *softmax* con 10 salidas, esto es,  $\{0, 1, 2, \dots, 9\}$ .

**Figura 7.7.** Ejemplo de una CNN para detectar dígitos inspirada en la arquitectura LeNet-5



Fuente: LeCun y col. (1989)

Una recomendación o guía para el diseño y creación de redes convolucionales es no intentar inventar una arquitectura y configuración de hiperparámetros desde cero, si no investigar en la literatura y seleccionar una arquitectura que funcione bien para aplicaciones similares y, luego, intentar afinar la red para mejorar los resultados con los datos específicos del problema a resolver. Veremos algunas arquitecturas típicas de CNN en la sección 8.1.

Si nos fijamos en el ejemplo de la figura 7.7, observamos que a medida que vamos profundizando en la red neuronal, el ancho y alto de los volúmenes se hacen más pequeños (de 32 a

6) mientras que el número de canales se incrementa (de 3 a 16). Vemos que las capas de convolución (**conv**) van seguidas de una capa de agrupamiento (**pool**) y las capas *fully connected* siempre se encuentran en la parte final de la red seguidas de una capa *softmax* (en los problemas de clasificación). Este es un patrón típico y estándar en las diferentes arquitecturas de redes neuronales convolucionales.

La tabla 7.1 muestra los valores de las capas de activación, sus medidas y el número de parámetros de la red. Para calcular el número de parámetros se usan las siguientes formulas:

- Capa de entrada (*input layer*): lo que hace la capa de entrada es leer la imagen y no hay parámetros que aprender.
- Capas convolucionales (*convolutional layers*): consideramos una capa convolucional que tiene  $l$  canales de entrada,  $k$  canales de salida y un tamaño del filtro de  $f$ . El número total de pesos será igual a  $f \times f \times l \times k$ . Adicionalmente, también debemos considerar el valor del sesgo (*bias*) para cada mapa de características, por lo tanto, el número total de parámetros será de  $(f \times f \times l + 1) \times k$ .
- Capas de agrupamiento (*pooling layers*): no hay parámetros que aprender durante el proceso de entrenamiento de la red.
- Capas totalmente conectadas (*fully connected layers*): en este tipo de capas, cada unidad de entrada tiene pesos separados en cada unidad de salida. Para  $n$  entradas y  $m$  salidas, el número de pesos es  $n \times m$ . También tenemos el sesgo para cada neurona de salida, siendo por tanto el número total de parámetros igual a  $(n + 1) \times m$ .

**Tabla 7.1.** Valores y número de parámetros de la red convolucional del ejemplo de la figura 7.7

Capa	Dimensión	Tamaño	Parámetros
Input	( $32 \times 32 \times 3$ )	3.072	0
Conv1	( $28 \times 28 \times 6$ )	4.704	456
Pool1	( $14 \times 14 \times 6$ )	1.176	0
Conv2	( $10 \times 10 \times 16$ )	1.600	2.416
Pool2	( $5 \times 5 \times 16$ )	400	0
FC2	( $120 \times 1$ )	120	48.120
FC3	( $84 \times 1$ )	84	10.164
Softmax	( $10 \times 1$ )	10	850

Fuente: elaboración propia

- Capa de salida (*output layer*): la capa de salida suele ser una capa *fully connected*, donde el número de parámetros será  $(n + 1) \times m$ .

Un último comentario en referencia a las redes convolucionales y el número de parámetros que estas deben aprender durante el proceso de entrenamiento. Las capas de convolución tienen un número de parámetros relativamente pequeño a aprender, mientras que la mayoría de parámetros que estas redes deben aprender provienen de las capas *fully connected*. El tamaño de las activaciones tiende a disminuir a medida que vamos profundizando en la red, no obstante si disminuye de forma muy brusca no es una buena señal cuando buscamos obtener una buena precisión en las predicciones de la red neuronal.

Son muchos los investigadores que han estudiado cuál es la mejor forma de poner las diferentes piezas de las redes convolucionales, de tal forma que se pueda maximizar la efectividad de las redes neuronales.

En la sección 8.1 veremos que las redes convolucionales que mejor funcionan siguen los patrones del ejemplo descrito en este apartado.

# Capítulo 8

# Arquitecturas de CNN

Este capítulo está dedicado a revisar la arquitectura de las redes neuronales más usadas en visión por computador. Empezaremos viendo tres de las principales redes convolucionales clásicas en la sección 8.1. Continuaremos con la estructura y funcionamiento de las redes residuales (*residual networks*, ResNet), y finalizaremos con un repaso de la red Inception, en la sección 8.3.

## 8.1. Redes convolucionales clásicas

En esta sección detallaremos la estructura y funcionamiento de las tres redes convolucionales clásicas, que han inspirado muchos otros modelos actuales: LeNet-5 (LeCun, Bottou y Haffner, 1998), AlexNet (Krizhevsky, Sutskever y col., 2012) y VGG (Simonyan y Zisserman, 2014).

Si se quieren leer los artículos originales se recomienda empezar por AlexNet, seguido de VGG y finalmente LeNet. LeNet es la red más pequeña de las tres, pero el artículo es un

poco complicado de leer, por lo tanto, recomendamos dejarlo como última lectura.

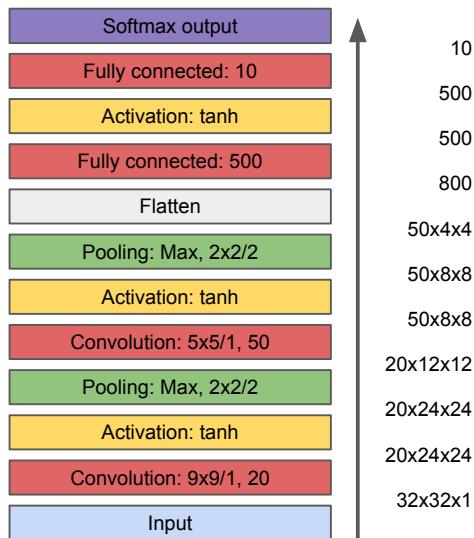
Estas redes, además de tener un interés histórico innegable, han servido de inspiración para otras redes más potentes y avanzadas, como la *residual network* o *inception*, que veremos en las secciones siguientes de este capítulo.

### 8.1.1. LeNet-5

El objetivo de LeNet-5 (LeCun, Bottou y Haffner, 1998) era el reconocimiento óptico de caracteres (*optical character recognition*, OCR) y dígitos escritos manualmente en documentos. La arquitectura LeNet-5 es bastante pequeña y, por lo tanto, es un buen inicio para aprender y hacer las primeras pruebas redes convolucionales. En este caso, dado el tamaño de la red, incluso puede ejecutarse sobre una computadora que no disponga de GPU, y ejecutar todos los cálculos en la misma CPU.

Esta arquitectura es bastante similar a la que hemos mostrado anteriormente, en la sección 7.3. La entrada es una imagen de  $32 \times 32 \times 1$ . Esta red fue entrenada en imágenes en escala de gris, por esto la dimensión del canal es igual a 1. La figura 8.1 muestra la arquitectura LeNet-5.

Las cajas marcadas en rojo son capas convolucionales o completamente conectadas (FC). Para cada capa convolucional se muestra el tamaño del filtro ( $f \times f$ ), el *stride* ( $s$ ) y el número de filtros usados. Para cada capa FC se muestra el número de neuronas que la forman. Las capas de *pooling* se muestran en cajas de color verde junto con el tipo de agrupación usado (ya sea *max* o *average*) y el tamaño de filtro y *stride*. Las cajas amarillas muestran el tipo de activación. Es interesante mencionar que cuando se propuso la arquitectura de LeNet-5

**Figura 8.1.** Arquitectura de la red LeNet-5

Fuente: elaboración propia

las funciones de activación más usadas eran *sigmoid* y *tanh*, mientras que recientemente la función no lineal mas usada es la ReLU. La salida o predicción de esta red neuronal será uno de los 10 posibles valores que corresponden a los dígitos de 0 a 9, tal y como hemos visto en las capas de salida *softmax*. Los números de la derecha hacen referencia a las dimensiones de los volúmenes después de hacer cada operación.

Esta red neuronal contiene unos 60.000 parámetros que se deben ajustar durante el proceso de entrenamiento, mientras que actualmente se diseñan arquitecturas de redes neuronales convolucionales con un volumen de 10 a 100 millones de parámetros.

Una característica muy importante y que tienen en común todas las redes neuronales es que a más profundidad, aumenta el número de canales, mientras que el volumen disminuye. Otro patrón que se mantiene es que las capas de convolución van seguidas de una capa de agrupamiento o *pooling*, a continuación una o varias capas FC y, finalmente, la salida de la red.

### 8.1.2. AlexNet

La red neuronal AlexNet (Krizhevsky, Sutskever y col., 2012) tiene bastantes similitudes con LeNet-5, pero una diferencia importante es que AlexNet es mucho más grande, como se puede pareciar en la figura 8.2. Mientras que LeNet-5 tenía alrededor de 60.000 parámetros que debían ser ajustados durante el proceso de entrenamiento, AlexNet tiene alrededor de 60 millones de parámetros.

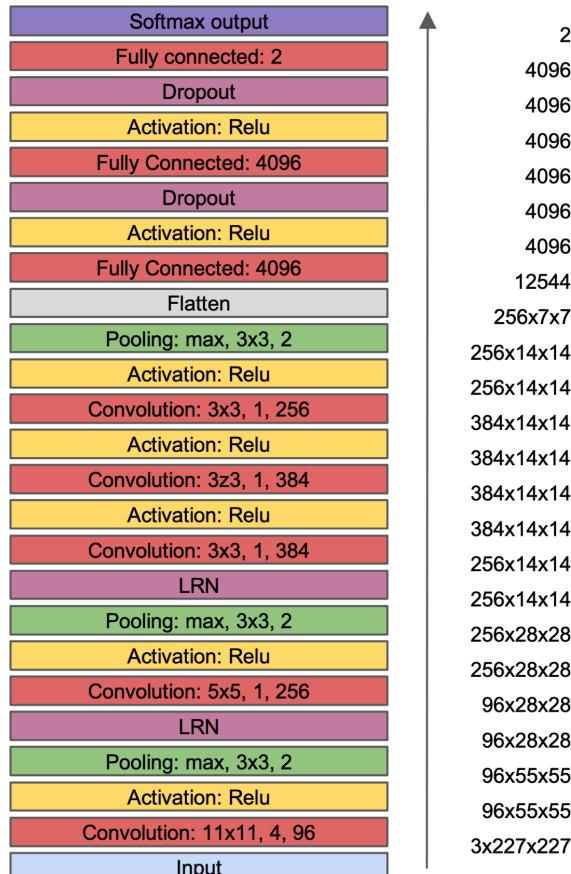
Una consecuencia del tamaño de esta red es que se necesita un volumen de datos bastante grande, dado que durante el entrenamiento se deben ajustar los valores de muchísimas neuronas de las capas ocultas de la red. A cambio, esta red obtiene resultados muy prometedores.

Otro aspecto destacable de esta arquitectura, que le ayuda a mejorar el rendimiento de la red LeNet-5, es el uso de ReLU como función de activación. Esta función de activación es contemporánea a otras funciones clásicas, como la función *sigmoide* o *tanh*, pero ha demostrado ser mejor que la función *tanh*.

Por otra parte, esta arquitectura fue propuesta cuando las GPU todavía eran un poco lentas, y el artículo original (ver Krizhevsky, Sutskever y col. (2012)) propone un método de entrenamiento un poco complejo para entrenar la red utili-

zando dos GPU, con el objetivo de mejorar el tiempo de entrenamiento. La idea es la de dividir las capas en dos GPU diferentes y que estas se podían comunicar entre ellas.

**Figura 8.2.** Arquitectura de la red AlexNet



Fuente: elaboración propia

Si observamos la figura 8.2 veremos que hay una capa llamada LRN (*local response normalization*), que corresponde a las cajas de color morado. Estas capas no se usan demasiado

en actualidad, pero vamos a verlas brevemente. Imaginemos que tenemos un volumen de  $13 \times 13 \times 256$ , las LRN miran a una posición del volumen y normalizan los valores a través de los 256 canales. La motivación de esta capa es que para cada posición del volumen  $13 \times 13$  no queremos muchas neuronas con un valor de activación alto.

Antes de que se propusiera la arquitectura AlexNet, el *deep learning* había empezado a ganar terreno en áreas como el reconocimiento del habla, pero fue esta arquitectura la que realmente convenció la comunidad de visión por computador que mediante las redes neuronales convolucionales se podían conseguir resultados superiores a los obtenidos hasta el momento con otros tipos de algoritmos o modelos de aprendizaje automático, como por ejemplo las máquinas de vectores de soporte (*support-vector machines*, SVM).

### 8.1.3. VGG

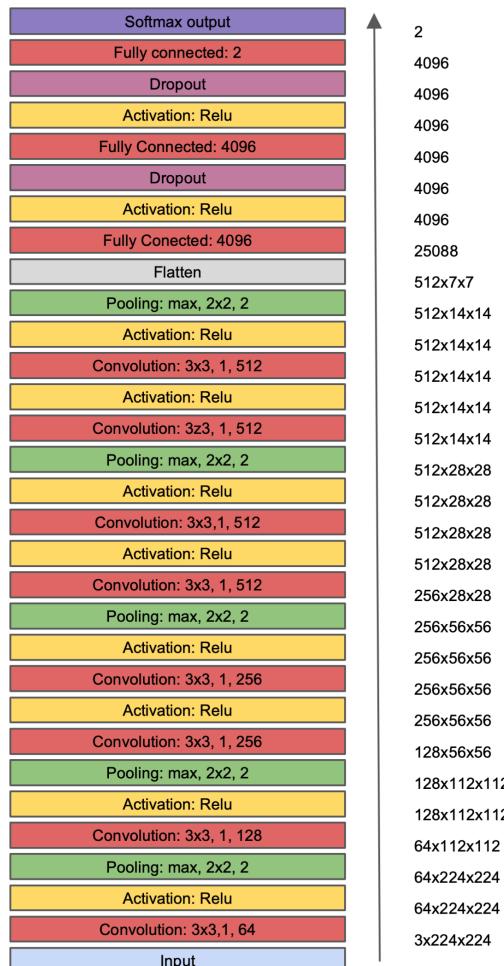
La última de las redes clásicas que veremos en este capítulo es la red conocida como VGG (Simonyan y Zisserman, 2014), y de la cual podemos ver la arquitectura en la figura 8.3.

Una de las propiedades más remarcables de esta red neuronal es que en lugar de tener muchos hiperparámetros, sus autores usaron una red muy simple donde se usan solo capas convolucionales con filtros de  $3 \times 3$  y *stride*  $s = 1$ , y capas de *pooling* con función de agrupamiento *max-pooling* de  $2 \times 2$  y *stride*  $s = 2$ .

Esta arquitectura es bastante profunda y tiene un total de 138 millones de parámetros, pero su simplicidad la hace muy atractiva. VGG tiene una arquitectura bastante uniforme, las capas convolucionales siguen con capas de *pooling* que reducen el alto y ancho de los volúmenes. Además, si nos fijamos en el

número de filtros que se usan en cada convolución, sus valores son: 64, 128, 256 y 512. Es decir, el valor se duplica en cada paso, creando un principio bastante simple en el diseño de la arquitectura.

**Figura 8.3.** Arquitectura de la red VGG



Fuente: elaboración propia

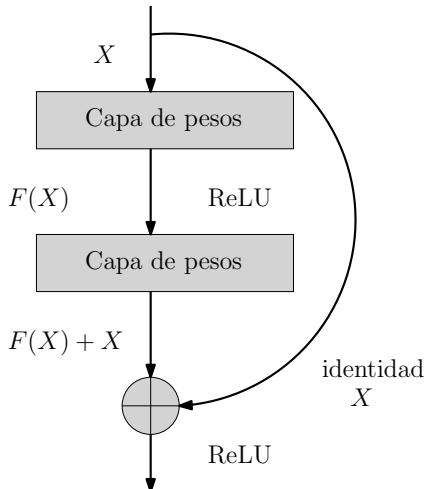
Sin embargo, la principal desventaja de esta red es el gran número de parámetros que contiene y que hay que ajustar durante el proceso de entrenamiento.

En la literatura encontramos referencias a las redes VGG-16 y VGG-19 (Simonyan y Zisserman, 2014). VGG-19 es una versión todavía mas profunda que VGG-16, pero la precisión obtenida por las dos es muy similar y normalmente se usa la red VGG-16.

## 8.2. *Residual networks* (ResNet)

Los autores de ResNet (Zhang, He y Ren, 2016) observaron que, no importa cuán profunda sea una red, no debería ser peor que una red menos profunda. Esto se debe a que si sostengamos que la red neuronal puede aproximar cualquier función complicada, también podría aprender la función de identidad, es decir, entrada = salida, saltando de manera efectiva el progreso de aprendizaje en algunas capas. Pero en el mundo real, este no es el caso debido al problema de la desaparición y explosión del gradiente (*vanishing/exploding gradient*) (Szegedy, Ioffe, Vanhoucke y col., 2017; Huang, Liu, L. v. d. y col., 2017).

Por lo tanto, podría ser útil forzar explícitamente a la red a aprender una asignación de identidad, aprendiendo el residuo de entrada y salida de algunas capas (o subredes), como se muestra en la figura 8.4. Supongamos que la entrada de la subred es  $X$  y el verdadero resultado es  $H(X)$ . El residual es la diferencia entre ellos:  $F(X) = H(X) - X$ . Como estamos interesados en encontrar el verdadero resultado subyacente de la subred, reorganizamos esta ecuación, que queda de la siguiente forma:  $H(X) = F(X) + X$ .

**Figura 8.4.** Residual ResNet

Fuente: elaboración propia

Esta es la principal diferencia entre ResNet y las redes neuronales tradicionales. Donde estas últimas aprenderán  $H(X)$  directamente, ResNet modela las capas para conocer el residuo de entrada y salida de las subredes. Esto le dará a la red una opción para omitir subredes haciendo  $F(X) = 0$  y  $H(X) = X$ . En otras palabras, la salida de una subred en particular es solo la salida de la última subred.

Durante la retropropagación (*backpropagation*), el aprendizaje residual nos da una buena propiedad. Debido a la formulación, la red podría optar por ignorar el gradiente de algunas subredes y simplemente reenviar el gradiente desde capas superiores a capas inferiores sin ninguna modificación. Como ejemplo extremo, esto significa que ResNet podría reenviar el gradiente desde la última capa directamente a la primera capa. Esto le da a ResNet una opción adicional que puede

ser útil, en lugar de simplemente hacer cálculos en todas las capas.

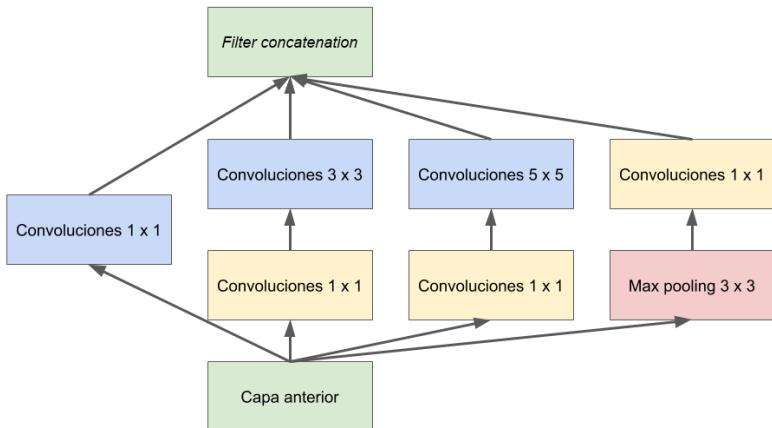
### 8.3. Inception

La inspiración de esta red proviene de un problema de decisión que se plantean los autores en el momento de diseñar la arquitectura de una red convolucional: ¿qué tipo de convolución aplicamos en cada capa?, ¿usaremos un filtro de  $3 \times 3$ ?, ¿o un filtro de  $5 \times 5$ ?

La solución a esta cuestión es el fundamento de esta red: ¿y porque no usarlos todos y hacer que sea el propio modelo quien decida? En este sentido, Inception (Szegedy, Ioffe y Vanhoucke, 2016) hace todas las convoluciones de forma paralela y concatena los resultados en mapas de características antes de ir a la siguiente capa.

Supongamos que la siguiente capa también es un módulo Inception. Cada uno de los mapas de características de las convolución pasará como un conjunto mixto de convoluciones a la siguiente capa. La idea es que no necesitamos decidir qué tipo de convoluciones es mejor,  $3 \times 3$  o  $5 \times 5$ . En su lugar, hacemos todas las convoluciones y dejamos que sea el modelo el que escoja. Además, esta arquitectura permite al modelo recuperar tanto características locales mediante convoluciones pequeñas como características de alto nivel mediante convoluciones más grandes. Ahora que tenemos la idea básica, vamos a ver la arquitectura específica.

La figura 8.5 muestra la arquitectura de un solo modulo Inception. En este caso específico, estamos usando convoluciones  $1 \times 1$ ,  $3 \times 3$  y  $5 \times 5$  junto con un agrupamiento basado en la función de *max-pooling* de  $3 \times 3$ . El *pooling* se añade al módu-

**Figura 8.5.** Módulo Inception

Fuente: elaboración propia

lo Inception porque todas las redes neuronales que funcionan bien tienen esta función (esto se ha probado de forma experimental en artículos de referencia (Szegedy, Ioffe, Vanhoucke y col., 2017; Zhang, He y Ren, 2016; Krizhevsky, Sutskever y col., 2012)). Las convoluciones más grandes tienen un coste computacional y temporal mayor. Por este motivo, se suelen realizar primero las convoluciones  $1 \times 1$ , reduciendo de esta forma la dimensionalidad del mapa de características, pasar el resultado a través de una ReLU y, finalmente, hacer las convoluciones más grandes (en este caso  $3 \times 3$  y  $5 \times 5$ ). Por lo tanto, es importante explicitar que la convolución  $1 \times 1$  es clave en este tipo de redes, ya que permite reducir la dimensionalidad del mapa de características.

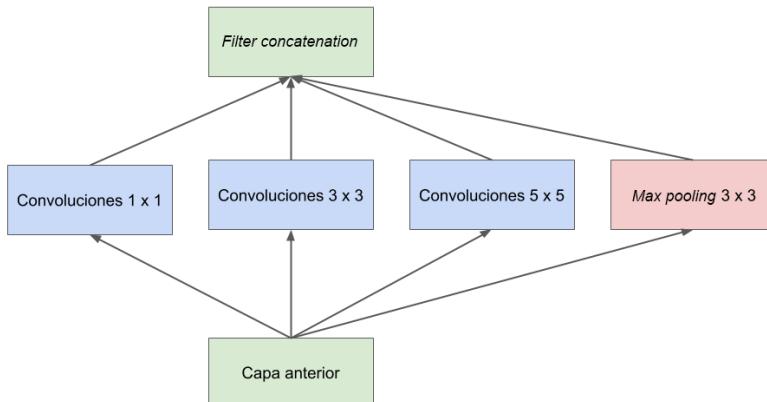
Para ver por qué el uso de las convoluciones  $1 \times 1$  para reducir la dimensionalidad de la entrada permite que los cálculos

sean más razonables, veamos primero el motivo por el que necesitamos afrontar un problema computacional muy costoso si no usamos este método de reducción de dimensionalidad.

Supongamos usamos una implementación *naïve* del módulo Inception. La figura 8.6 muestra el módulo, que es muy similar al de la figura 8.5, pero en este caso no tenemos las convoluciones adicionales  $1 \times 1$  antes de las convoluciones más grandes ( $3 \times 3$  y  $5 \times 5$ ). Vamos a examinar el número de cálculos necesarios para el primer módulo de la red Inception.

Para simplificar un poco los cálculos, podemos asumir que la red usa el mismo *padding* para las convoluciones del mismo módulo, dado que la entrada y la salida tienen una dimensión de  $28 \times 28$ . Vamos a ver el coste de las convoluciones  $5 \times 5$  si no aplicamos previamente la reducción de la dimensionalidad.

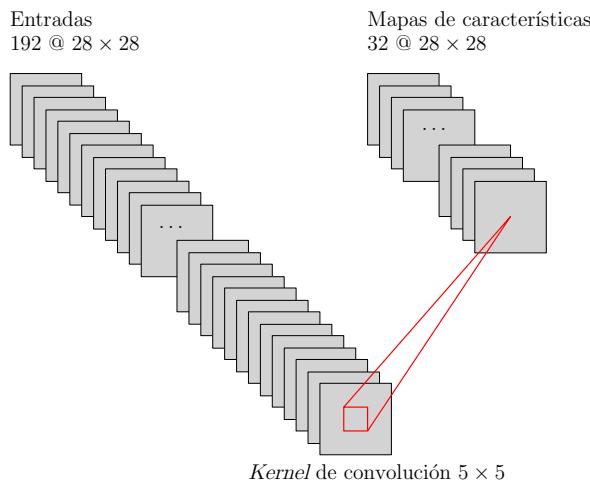
**Figura 8.6.** Módulo Inception *naïve*



Fuente: elaboración propia

La figura 8.7 muestra de forma gráfica estas operaciones. Como se puede verificar, estamos hablando de  $5^2 \times 28^2 \times 192 \times 32 = 120.422.400$  operaciones... ¡eso son muchos cálculos! Ahora podemos entender por qué los autores de esta red necesitan intentar, de alguna forma, reducir este número.

**Figura 8.7.** Convoluciones del módulo Inception usando la versión *naïve*



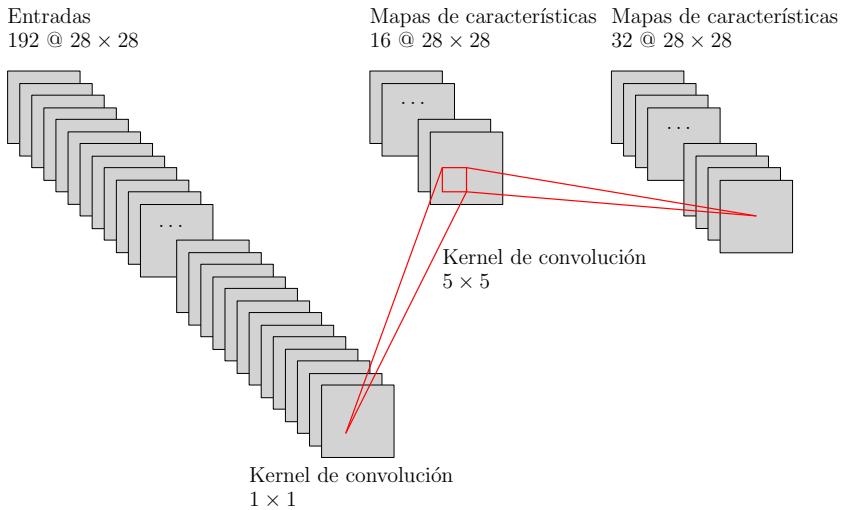
Fuente: elaboración propia

Para tal fin, usamos el modelo de la figura 8.5. Previamente a realizar las convoluciones  $5 \times 5$ , aplicamos una capa anterior con una convolución de  $1 \times 1$ , que devuelve como salida un mapa de características de  $16 @ 28 \times 28$ . A continuación, aplicamos la convolución  $5 \times 5$  en estos mapas de características que dan una salida de  $32 @ 28 \times 28$ . Este proceso se muestra en la figura 8.8.

En este caso habrá  $(1^2 \times 28^2 \times 192 \times 16) + (5^2 \times 28^2 \times 16 \times 32) = 12.443.648$  operaciones. Aunque sea un número todavía grande, se reduce el número de cálculos con respecto al modelo *naïve* en un factor de 10. Queda como ejercicio para

el lector realizar los cálculos para las convoluciones de  $3 \times 3$ , que siguen el mismo proceso.

**Figura 8.8.** Convoluciones  $1 \times 1$  que reducen de la dimensionalidad con el objetivo de reducir el número de operaciones en las convoluciones  $5 \times 5$



Fuente: elaboración propia

# Capítulo 9

## Consejos prácticos y ejemplos

En este último capítulo en el bloque de redes neuronales convolucionales, nos centraremos en ver algunos consejos prácticos a la hora de trabajar con CNN e imágenes para la creación de modelos predictivos, en la sección 9.1. Finalmente, veremos algunos ejemplos de problemas que se pueden resolver empleando las CNN (sección 9.2).

### 9.1. Consejos prácticos en el uso de las CNN

En esta sección resumiremos algunos consejos prácticos, fruto de varios años trabajando con redes neuronales convolucionales en temas de procesamiento de imágenes y visión por computador, que pueden resultar muy útiles a la hora de abordar diferentes proyectos relacionados con la creación de modelos predictivos empleando las CNN.

### 9.1.1. Implementaciones *open source*

Hemos visto hasta el momento diferentes arquitecturas muy efectivas de redes convolucionales. En esta sección vamos a ver algunos consejos prácticos de como usarlas, empezando primero con las implementaciones de código abierto (*open source*).

Muchas de las redes neuronales convolucionales conocidas y empleadas actualmente son muy difíciles de replicar, debido principalmente a dos motivos: (1) los detalles de implementación no son conocidos o no son fáciles de reproducir, y (2) el ajuste (*tunning*) de los hiperparámetros como, por ejemplo, el *learning decay* que pueden marcar una diferencia importante en el rendimiento del modelo. Replicar una CNN a partir de un artículo científico (o similar) es una tarea compleja y difícil, incluso para muchos expertos, simplemente a partir de la información descrita en el artículo del trabajo de referencia.

Por suerte, muchos investigadores en el área de *deep learning* abren su código a internet. Es decir, publican el código de forma abierta en ciertos repositorios, como por ejemplo GitHub<sup>1</sup> similares.

Entonces, si vemos un artículo de investigación, los resultados del cual queremos replicar, una primera opción que debemos considerar es buscar una implementación *open source* disponible en la red del modelo en cuestión. Si podemos obtener la implementación de los autores avanzaremos mucho más rápido que si queremos implementar el mismo método desde cero. Aunque también hay que tener en cuenta que implementar desde cero un modelo es un muy buen ejercicio para entenderlo.

Veamos un ejemplo. Imaginemos que queremos usar las *residual networks* (ResNets) y hacemos una búsqueda en internet

---

<sup>1</sup><https://github.com/>

con los conceptos «ResNet `github`», a partir de la cual nos aparecen muchos enlaces a diferentes implementaciones.

Uno de los enlaces que probablemente nos aparecerá (y probablemente en primera posición) es la implementación original de los autores del artículo ResNet<sup>2</sup> (Zhang, He y Ren, 2016). Si accedemos al código de GitHub veremos la descripción del trabajo de la implementación en particular. Veremos también la licencia del código, en este caso concreto, corresponde a una licencia MIT para poder ver las implicaciones de usarlo. La licencia MIT es una de las más permisivas y abiertas. A partir del repositorio podemos clonarlo de forma local, es decir, en el sistema de ficheros de nuestra computadora. En este caso particular, la implementación usa el *framework* de computación Caffe,<sup>3</sup> pero si se precisa una implementación de este modelo en otro *framework* (por ejemplo, Keras)<sup>4</sup> es muy probable que también esté disponible de forma abierta en algún repositorio público de código.

Si estamos desarrollando una aplicación de visión por computador, un flujo de trabajo (*workflow*) muy común podría ser:

1. seleccionar la arquitectura que nos gustaría probar,
2. buscar una implementación de código abierto en cualquier repositorio de código *open source*,
3. y empezar a construir el modelo a partir de la implementación seleccionada.

Una de las ventajas de trabajar de esta forma es que las redes neuronales profundas tardan mucho tiempo en entrenar,

---

<sup>2</sup><https://github.com/KaimingHe/deep-residual-networks>

<sup>3</sup><http://caffe.berkeleyvision.org>

<sup>4</sup><https://keras.io/>

y quizá alguien haya entrenado el mismo modelo usando GPU con un conjunto de datos lo suficientemente grande.

Esto nos permitiría usar el modelo ya entrenado o bien usar técnicas de *transfer learning* (ver sección 9.1.2). Por supuesto, si estamos haciendo investigación en redes neuronales lo que queremos es hacer una implementación desde cero y el flujo de trabajo sería muy diferente.

### 9.1.2. *Transfer learning*

La *transferencia del aprendizaje* (o *transfer learning*) es una técnica de aprendizaje automático que consiste en transferir un modelo que ha sido entrenado para una tarea determinada para ser usado y resolver otra tarea parecida.

Esta optimización permite progresar de forma más rápida y mejorar el rendimiento del modelo. Las técnicas de *transfer learning* son muy populares en el área de *deep learning*, debido a la gran cantidad de datos y recursos computacionales que se requieren para entrenar un modelo.

Sin embargo, es importante destacar que esta técnica solo funciona si las características que ha aprendido el modelo durante el entrenamiento de la primera tarea son lo suficientemente generales para ser útiles y aplicables en otros entornos similares. Esta forma de transferir el aprendizaje que se usa en *deep learning* se llama *transferencia inductiva* (*inductive transfer*).

En general, un modelo se puede ajustar de forma beneficiosa a una tarea diferente, siempre que esta esté relacionada con la tarea original. Como hemos visto anteriormente, el entrenamiento de las redes neuronales consiste, en esencia, en ajustar los pesos de la red empleando los datos etiquetados del conjunto de datos. Estos pesos, una vez «aprendidos»

se pueden exportar y transferir a otra red neuronal, en lugar de entrenar los pesos de la red desde cero.

Hay diferentes maneras de usar el *transfer learning*, la más común es utilizar un modelo preentrenado. Un modelo preentrenado es un modelo creado por una tercera persona que resuelve un problema similar al que queremos resolver nosotros. En lugar de construir un modelo desde cero vamos a usar el modelo preentrenado como punto de partida. El funcionamiento es el siguiente:

1. Seleccionar el modelo fuente a partir de los modelos disponibles. Muchas instituciones de investigación ponen a disposición de terceros modelos ya entrenados a partir de grandes volúmenes de datos.
2. Reusar el modelo. El modelo seleccionado puede ser usado como punto de partida para resolver la tarea de interés. Esto puede implicar utilizar todas o algunas de las partes del modelo original, dependiendo de la técnica de modelado usada.
3. Ajuste (*tunning*) del modelo. Opcionalmente, el modelo puede necesitar ser adaptado o refinado según los datos requeridos para resolver la tarea de interés.

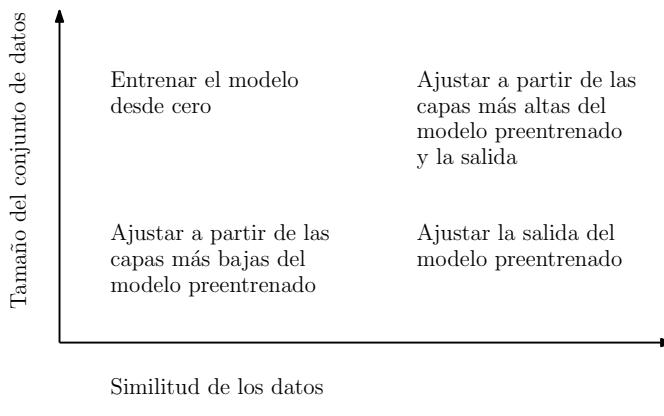
El proceso de ajuste (habitualmente conocido como *tunning*) del modelo suele ser complejo y bastante costoso. Existen múltiples formas de realizar este proceso, siendo algunas de las más habituales, las siguientes:

- Extracción de características. Podemos usar el modelo preentrenado como un método de selección de características. Si borramos la última capa de la red neuronal

(la salida que nos da las probabilidades de una instancia de pertenecer a cada una de las clases) y usamos la red entera como un extractor de características fijo para los datos de la nueva tarea.

- Usar la arquitectura del modelo preentrenado. Usamos la arquitectura del modelo e inicializamos los pesos de forma aleatoria para entrenar el modelo de nuevo con un nuevo conjunto de datos.
- Entrenar algunas capas y congelar las otras. Esto se refiere a reentrenar el modelo de forma parcial. Para ello mantenemos los pesos de las capas iniciales (es decir, «congelamos» estos pesos) y reentrenamos solamente los pesos de las capas más altas.

**Figura 9.1.** Diagrama de casos de uso de *transfer learning*



Fuente: elaboración propia

El diagrama de la figura 9.1 puede servir de ayuda a la hora de decidir qué forma de ajuste es mejor según el volumen de datos disponible y la similitud entre las tareas a realizar. En

ella se muestran cuatro escenarios diferentes, que pasamos a comentar de forma individual:

1. El tamaño del conjunto de datos es pequeño, pero la similitud de los datos es muy alta. En este caso, debido a que la similitud de los datos es muy alta, no necesitamos reentrenar el modelo de nuevo. Lo que tenemos que hacer es personalizar y modificar la salida para adaptarla al nuevo problema. Se usa el modelo preentrenado como un extracto de características.

Imaginemos que queremos usar un modelo entrenado en ImageNet para identificar un nuevo conjunto de datos que contiene, solamente, imágenes de perros y gatos. Las imágenes de los dos conjuntos son muy parecidas, no obstante ahora solo tenemos dos categorías en la salida. Por lo tanto, lo que haremos es modificar la capa FC y la capa *softmax* para que tenga únicamente dos categorías (o clases) en lugar de las 1.000 del conjunto de datos original.

2. El tamaño de los datos es pequeño y la similitud de los datos es muy baja. En este caso podemos congelar los pesos de las  $k$  capas iniciales del modelo preentrenado y entrenar las siguientes capas ( $n - k$ ). Las capas más altas serán personalizadas y adaptadas al nuevo conjunto de datos. Como el nuevo conjunto de datos tiene una muy baja similitud, es importante ajustar las capas altas según los datos contenidos en el nuevo conjunto.

El tamaño pequeño del conjunto de datos se puede ver compensado por el hecho que las capas iniciales mantienen los pesos del modelo inicial, que ha sido entrenado utilizando un conjunto de datos mucho más grande.

3. El tamaño del conjunto de datos es grande, pero la similitud de los datos es baja. En este caso, debido a que tenemos un conjunto de datos grande, el entrenamiento de la red neuronal será efectivo. No obstante, debido a que los datos son diferentes comparados con los datos iniciales que se usaron para entrenar la red, los pesos aprendidos quizás no sean del todo óptimos y es mejor entrenar la red desde cero.
4. El tamaño del conjunto de datos es grande y la similitud de los datos es alta. Esta es la situación ideal. En este caso el modelo preentrenado debería de ser muy efectivo. La mejor manera de usarlo es reentrenar la arquitectura del modelo y los pesos iniciales de este. Entonces podemos reentrenar el modelo usando los pesos del modelo preentrenado como método de inicialización.

### **9.1.3. *Data augmentation***

Existe una relación casi lineal entre la cantidad de datos que se requieren para entrenar un modelo y el tamaño de este. El modelo debe de ser lo suficientemente grande para capturar las relaciones existentes en los datos (texturas, formas, etc.) junto con las especificidades del problema (por ejemplo, número de categorías o clases).

Las capas más iniciales del modelo capturan relaciones de alto nivel entre las diferentes partes de la entrada, tales como aristas o patrones. Las capas posteriores capturan información que ayuda al modelo a tomar la decisión final. Es decir, la información que ayuda a discriminar entre las diferentes salidas que puede generar del modelo. Si el problema es complejo, como la clasificación de imágenes, el número de parámetros

y la cantidad de datos requeridos es muy alto. Por lo tanto, en estos casos, disponer de un conjunto de datos grande (o muy grande) es crucial para obtener buenos resultados en *deep learning*. Sin embargo, no siempre es posible disponer de grandes conjuntos de datos etiquetados, y este se ha convertido en uno de los grandes problemas del *deep learning* y del aprendizaje automático, en general.

### **¿Porque es importante tener muchos datos?**

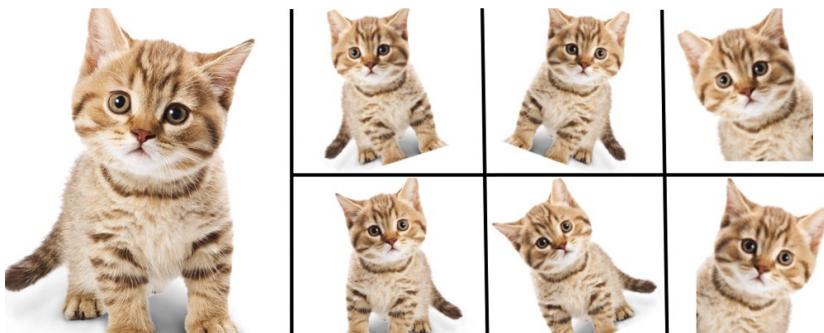
Cuando entrenamos un modelo de *deep learning*, lo que realmente estamos haciendo es ajustar los parámetros para que puedan expresar la relación entre la entrada (una imagen) y la salida (una clase) de la red. Las arquitecturas de los modelos empleados en el estado del arte de *deep learning* tienen un orden de millones de parámetros que se deben entrenar y ajustar. En consecuencia, se necesita proporcionar muchos ejemplos al modelo para que sea capaz de aprender los mejores parámetros y obtener un buen rendimiento.

### **¿Cómo obtenemos más datos si nuestro conjunto es pequeño?**

Una opción ampliamente utilizada para obtener más datos de los inicialmente disponibles, consiste en hacer alteraciones en las imágenes existentes de nuestro conjunto de datos. Mediante cambios pequeños en la imágenes originales como, por ejemplo traslaciones, rotaciones o volteos (*flip*), obtendremos más imágenes que podemos emplear para entrenar la red, ya que la red neuronal las interpretará como distintas.

La figura 9.2 muestra como con pequeñas alteraciones en la imagen original es posible obtener un conjunto de imágenes nuevas.

**Figura 9.2.** Alteraciones de la imagen original mediante técnicas de expansión artificial de datos



Fuente: <https://medium.com>

Las CNN son invariantes a traslaciones, punto de vista, tamaño e iluminación. Esta es la premisa de la expansión artificial de datos (o *data augmentation*). Es decir, generamos imágenes de forma artificial (sintéticamente) para obtener datos en una variedad de condiciones que nos acerque a la realidad que nos encontraremos a la hora de resolver el problema en el mundo real.

Esta práctica no se usa solamente cuando tenemos conjuntos de datos pequeños, si no que se ha demostrado que, aunque tengamos un conjunto de datos lo suficientemente grande, ayuda a tener más datos relevantes para el entrenamiento del modelo. La red neuronal será tan buena (o mala) como lo son los datos que utilizamos para entrenarla.

Antes de empezar con las diferentes técnicas de *data augmentation* vamos a ver que hay dos técnicas principales para aplicarla:

- *Offline augmentation.* Todas las transformaciones se hacen antes de entrenar el modelo, aumentando de esta forma el tamaño del conjunto de datos. Esta forma de trabajar es la recomendada si tenemos conjuntos de datos pequeños, ya que el tamaño del *dataset* se incrementará proporcionalmente al número de transformaciones que realicemos.
- *Online augmentation* o *augmentation on the fly*. Las transformaciones se hacen en los *mini-batch*, antes de alimentar el modelo. Este método es preferible cuando tenemos conjuntos de datos grandes, dado que no se puede afrontar el incremento de imágenes desde el inicio.

Ya sea de forma *offline* o *online*, las técnicas más populares para expandir o aumentar artificialmente el conjunto de datos son:

- Dar la vuelta (*flip*). A las imágenes se les puede dar la vuelta vertical o horizontalmente. Algunos *frameworks* no tienen la función de dar la vuelta vertical, pero esto es equivalente a rotar la imagen 180 grados y luego dar la vuelta de forma horizontal.
- Rotación. Hay que tener en cuenta que después de realizar esta operación las dimensiones de la imagen se pueden ver alteradas. Si la imagen es cuadrada las dimensiones se mantendrán, pero si es un rectángulo solo se mantendrá si la rotación es de 180 grados.

- Escalado. La imagen se puede escalar haciéndola más grande o más pequeña.
- Recortar (*crop*). De forma aleatoria seleccionamos una parte de la imagen original y la cortamos. Posteriormente se redimensiona la imagen para que mantenga el tamaño de la imagen original.
- Translación. Esta operación se refiere a desplazar la imagen a lo largo del eje vertical (X) o horizontal (Y). Este método es muy útil porque los objetos se pueden encontrar en casi cualquier parte de la imagen y fuerza a las redes a aprender y buscar los objetos en cualquier parte de la imagen.
- Añadir ruido. El *overfitting* normalmente se produce cuando la red neuronal intenta aprender características que se dan con mucha frecuencia. El ruido gaussiano (mediana 0) genera puntos de datos en todas las frecuencias, distorsionando de esta forma las características más usuales.

En el mundo real, las imágenes naturales existen en una variedad de condiciones que no se pueden reproducir con los métodos anteriormente comentados. Por ejemplo, imaginemos la tarea de identificar un paisaje en una fotografía. El paisaje puede ser cualquier cosa: bosques, montañas, nieve, etc. Parece una tarea sencilla si no tenemos en cuenta las diferentes estaciones del año. Si la red neuronal no aprende el hecho de que ciertos paisajes pueden existir en diferentes condiciones (nieve, humedad, sol, etc.), podría llegar a clasificar paisajes nevados como glaciares.

En este tipo de casos, se emplean métodos como el *GAN convolucional*<sup>5</sup> (Zhu, Park, Isola y col., 2017) (figura 9.3) u otras redes neuronales de transferencia de estilos para generar nuevas imágenes (Luan, Paris y col., 2017) (figura 9.4).

**Figura 9.3.** Cambio de estaciones usando CycleGAN)



Fuente: <https://www.kdnuggets.com>

## 9.2. Ejemplos

En esta segunda parte de este capítulo nos centraremos en discutir un par de ejemplos relevantes en el contexto del *deep learning* y de la visión por computador. En concreto, veremos un caso práctico de reconocimiento facial y otro de transferencia de estilo entre imágenes.

<sup>5</sup><https://junyanz.github.io/CycleGAN/>

**Figura 9.4.** Transferencia de estilos usando *deep learning* para aumentar los datos de un conjunto de datos



Fuente: elaboración propia

### 9.2.1. Reconocimiento facial

En esta sección vamos a ver un ejemplo de cómo resolver el problema de reconocimiento facial mediante redes neuronales convolucionales.

Vamos a empezar primero con la terminología usada en el reconocimiento facial. En la literatura, los autores hablan a menudo de verificación facial y reconocimiento facial. La *verificación facial* consiste en, dados una imagen de entrada y una identificación de una persona (ya sea un nombre, ID o cualquier identificación única), el sistema debe determinar si la imagen de entrada se corresponde con la persona indicada. Este proceso se conoce como un problema *one-to-one*, donde solo nos interesa saber si la persona se corresponde con quien dice que es. El problema del *reconocimiento facial* consiste en, a partir de una imagen de entrada, determinar la identifica-

ción de la persona en cuestión (presuntamente, dentro de un conjunto finito de personas conocidas).

Una de las principales problemáticas o complejidades relacionadas con un sistema de verificación se debe, precisamente, a que necesitamos resolver un problema de aprendizaje singular. Es decir, necesitamos reconocer a una persona usando solo una imagen o un solo ejemplo de la cara de la persona. Como ya hemos visto y comentado, los algoritmos de *deep learning* no funcionan demasiado bien con solo una imagen de entrenamiento por cada persona.

Supongamos que tenemos una base de datos de imágenes de empleados de una organización. Uno de ellos aparece en la oficina y quiere que el sistema de identificación lo deje pasar. Aunque el sistema haya visto solamente una imagen del empleado, este debe ser capaz de reconocer positivamente a la persona y permitirle el acceso a las instalaciones. Por el contrario, si aparece una persona que no está en la base de datos, el sistema debe ser capaz de negar el acceso advirtiendo que esta persona no es un empleado de la organización. Esto es así para la mayoría de aplicaciones de reconocimiento facial reales, porque normalmente solo tenemos en el sistema una imagen de los empleados o las personas que queremos reconocer.

Una primera aproximación sería pasar la imagen de la persona por una CNN y dejar que esta nos dé una etiqueta de salida usando una unidad *softmax* con un número de salidas igual al número de empleados. Pero esto realmente no funciona, debido principalmente al número reducido de imágenes de entrenamiento de que disponemos, y porque cada vez que alguien nuevo se uniera a la organización deberíamos de reentrenar el sistema.

Una segunda aproximación más eficaz consiste en aprender una función de «similitud». En particular, queremos una red neuronal que aprenda una función  $s(x_i, x_j)$  que dadas dos imágenes de entrada nos indique el grado de diferencia entre las dos imágenes. En particular, si las dos imágenes son la misma persona, queremos que la salida sea un número pequeño; y si las imágenes son de diferentes personas, queremos que la salida sea un número más grande. Es decir, queremos que la red nos devuelva una métrica de la divergencia entre dos imágenes.

Durante el proceso de reconocimiento, si el grado de diferencia es más pequeño que un cierto umbral  $\tau$  (*threshold*, que es un hiperparámetro) entonces la predicción será que las dos imágenes son de la misma persona; y si el valor devuelto por la red es mayor que  $\tau$ , entonces la predicción será que son personas distintas.

Así pues, el proceso es el siguiente: dada una imagen  $x_i$ , usamos la función  $s(x_i, x_j)$  para compararla con cada una de las imágenes que tenemos en nuestra base de datos, esto es,  $x_j \in \{x_1, \dots, x_p\} \setminus x_i$ . Si la función nos da un número pequeño, tal que  $s < \tau$ , entonces determinaos que se trata de la misma persona. Si finalmente ninguna de las comparaciones nos da un valor inferior a  $\tau$  es porque la persona de la imagen de entrada no se corresponde con ninguna de las que tenemos en la base de datos y, por lo tanto, no la podemos reconocer.

Para entrenar una red neuronal que aprenda la función  $s$  se usan lo que conocemos como las *redes siamesas*.

Tenemos una imagen de entrada  $x_1$  y la pasamos a través de la red convolucional para terminar con una capa FC que finaliza con un vector de ciertas características. Normalmente este vector de características es usado como entrada de la función

*softmax* para hacer la clasificación. Para el reconocimiento facial nos vamos a focalizar en el vector de características y no en la salida del clasificador. Vamos a llamar a este vector  $f(x_1)$ , representa una codificación de  $x_1$ .

La forma de construir el sistema de reconocimiento facial es pasar una segunda imagen  $x_2$  a través de la misma red neuronal con los mismos parámetros obteniendo un vector de salida diferente, que representará la codificación de esta segunda imagen  $f(x_2)$ .

Finalmente, debemos escoger una función que nos permite calcular la distancia  $d$  entre los vectores  $x_1$  y  $x_2$ . Para tal propósito, podemos emplear, por ejemplo, la norma de la diferencia de ambos vectores:

$$d = \|f(x_1) - f(x_2)\|^2. \quad (9.1)$$

Si  $d < \tau$ , entonces  $x_1$  y  $x_2$  son la misma persona. En caso contrario, el sistema determina que  $x_1$  y  $x_2$  son personas diferentes. La idea es pasar dos entradas diferentes sobre la misma red neuronal y comparar las salidas. Esto es lo que se conoce como *redes neuronales siamesas* (Taigman, Yang y col., 2014).

Vamos a ver ahora cómo se entrena este tipo de redes neuronales. Formalmente, queremos aprender los parámetros de tal forma que, si dos imágenes de entrada  $x_i$  y  $x_j$  pertenecen a la misma persona, la distancia entre su codificación sea pequeña y, por el contrario, si  $x_i$  y  $x_j$  son personas diferentes, entonces queremos que la distancia sea grande.

Si variamos los parámetros en las capas de la red neuronal terminamos con diferentes codificaciones. Podemos usar el algoritmo de entrenamiento *backpropagation* y modificar todos los parámetros para asegurar que se satisfacen estas condicio-

nes. Para que la red aprenda lo que acabamos de describir, debemos definir una función objetivo, que en este caso será la función *triplet loss*.

### **Triplet loss**

Una buena forma de aprender los parámetros de la red neuronal, para que nos de una buena codificación para las imágenes de las caras, es aplicar el algoritmo del descenso del gradiente sobre la función *triplet loss*.

Para aplicar el *triplet loss* necesitamos comparar pares de imágenes. En la terminología de *triplet loss* identificamos una imagen como *anchor* y buscamos que la distancia entre el *anchor* y la imagen positiva sea pequeña, mientras que queremos que la distancia entre la imagen *anchor* y la imagen negativa sea más grande. Esto es lo que da nombre al término *triplet loss*, que siempre estaremos mirando a tres imágenes al mismo tiempo: *anchor* (A), positiva (P) y negativa (N). Para formalizar este principio que acabamos de comentar, queremos que los parámetros de la red neuronal tengan la siguiente propiedad para las codificaciones que produce:

$$\|f(P) - f(A)\|^2 \leq \|f(P) - f(N)\|^2. \quad (9.2)$$

Para asegurar que se cumple la ecuación anterior y que la salida de las codificaciones no es siempre 0, vamos a aprender y optimizar la siguiente ecuación:

$$\|f(P) - f(A)\|^2 - \|f(P) - f(N)\|^2 + \alpha \leq 0, \quad (9.3)$$

donde  $\alpha$  es un hiperparámetro (*margin*) que previene que las salidas no sean optimizadas de forma eficiente, intentando separar los pares  $A - P$  de  $A - N$ .

Para formalizar todo esto vamos a definir el *loss* de la siguiente forma:

$$\mathcal{L}(A, N, P) = \max(\|f(P) - f(A)\|^2 - \|f(P) - f(N)v^2 + \alpha, 0). \quad (9.4)$$

Y la función de coste global de la red neuronal se puede definir como:

$$J = \sum_{i=1}^m \mathcal{L}(A_i, P_i, N_i). \quad (9.5)$$

Si, por ejemplo, tenemos un conjunto de datos de entrenamiento de 10.000 imágenes con 1.000 personas diferentes, lo que haremos es usar las 10.000 imágenes para generar *triplets*. A continuación, entrenaremos la red utilizando estos *triplets* y usando el método del descenso del gradiente sobre la función de coste que hemos definido anteriormente.

El conjunto de datos de entrenamiento debe contener varias imágenes de la misma persona, ya que debemos generar los pares  $A - P$ . Sin embargo, después de la fase de entrenamiento, la red solo necesita una imagen de cada individuo.

En este punto, nos surge la duda de cómo seleccionar las imágenes que formarán los *triplets*. Si seleccionamos  $A$ ,  $P$  y  $N$  de forma aleatoria (siendo  $P$  una imagen que contiene la misma persona que  $A$ , y  $N$  una persona distinta), la restricción es muy fácil de satisfacer. Lo que queremos es seleccionar el *triplet* para que sea difícil de entrenar. En concreto, queremos que para todos los *triplets*, la distancia  $d(A, P)$  sea muy próxima a  $d(A, N)$ . Si no los escogemos de esta forma, habrá muchos *triplets* que pueden satisfacer la función de coste y, por lo tanto, el método del descenso del gradiente no hará

nada debido a qué la red neuronal generará la salida deseada la mayoría de las veces (Schroff y Kalenichenko, 2015).

### 9.2.2. Transferencia de estilos

Imaginemos que tenemos una imagen de un gato y queremos obtener esta misma imagen, pero con un estilo diferente. Por ejemplo, queremos obtener la misma imagen del gato, pero como si hubiese sido pintado por Van Gogh. Lo que nos permite la transferencia de estilos es, precisamente, esto: generar una imagen nueva a partir de una imagen de origen, pero aplicando el estilo de una segunda imagen. La figura 9.5 muestra el ejemplo que acabamos de comentar. Para lo que sigue de sección vamos a denominar  $C$  la imagen contenido,  $S$  la imagen de estilo y  $G$  la imagen que se genera.

**Figura 9.5.** Ejemplos de transferencia de estilo



Fuente: elaboración propia

Para poder hacer este proceso, necesitamos mirar las características extraídas por la CNN en varias capas, de menos a más profundas.

En primer lugar, vamos a definir una función de coste  $J(G)$  que nos dirá cómo de buena es una imagen generada

en particular. Vamos a usar el método del descenso del gradiente para minimizar la función  $J(G)$ . Vamos a dividir esta función en dos partes:

- Coste de contenido:  $J_{content}(C, G)$ . Esta función mide el grado de similitud entre el contenido de la imagen generada ( $G$ ) y la imagen contenido ( $C$ ).
- Coste de estilo:  $J_{style}(S, G)$ . Calcula el grado de similitud en el estilo entre la imagen de estilo ( $S$ ) y la imagen generada ( $G$ ).

Estas dos funciones de coste se suman y ponderan con dos hiperparámetros,  $\alpha$  y  $\beta$ , que especifican el peso relativo de ambas funciones de coste para determinar el coste total (Leon y Matthias, 2015):

$$J(G) = \alpha J_{content}(C, G) + \beta J_{style}(S, G). \quad (9.6)$$

El algoritmo funciona de la siguiente forma: inicializamos de forma aleatoria la imagen  $G$  y usamos el método del descenso del gradiente para minimizar la función  $J(G)$ , de manera que se actualizan los valores de los píxeles. Vamos a ver con más detalle como se definen las dos funciones de coste.

## Coste de contenido

Supongamos que vamos a utilizar una capa oculta ( $l$ ) para calcular el coste de contenido. Entonces, si  $l$  es una capa inicial de la red, es decir, próxima a los datos de entrada, estamos forzando que la imagen generada tenga valores de píxeles muy similares a los de la imagen de contenido. Mientras que si usamos una capa más profunda, es decir, próxima a la salida

de la red, estaremos forzando que si hay un gato contenido en la imagen, deberá aparecer también en la imagen generada.

Una forma de conseguir este valor de coste es usar una red neuronal preentrenada como, por ejemplo VGG, y medir el grado de similitud de contenido entre la imagen de contenido y la imagen generada. Supongamos que  $a_C^{[l]}$  y  $a_G^{[l]}$  son las activaciones para la capa  $l$ . Si estas dos activaciones son similares, implica que las dos imágenes tiene un contenido similar. Vamos pues a definir la función de coste  $J_{content}(C, G)$  para que nos calcule el grado de disimilitud entre los valores de las activaciones:

$$J_{content}(C, G) = \frac{1}{2} \|a_C^{[l]} - a_G^{[l]}\|^2. \quad (9.7)$$

## Coste de estilo

Imaginemos que hemos escogido una capa oculta ( $l$ ) para medir el estilo de la imagen. Lo que debemos hacer es definir este como la correlación entre las activaciones a través de los diferentes canales de esta capa de activación  $l$ . La intuición nos sugiere usar el grado de correlación entre canales como medida del estilo. Si calculamos, por ejemplo, el grado de correlación entre el primer y el segundo canal de la imagen generada, esto nos dará información sobre la frecuencia de aparición de una textura concreta y, por lo tanto, podemos medir el grado de similitud en el estilo entre la imagen generada y la imagen de estilo.

Para formalizar esta intuición vamos a generar una matriz, a la que llamaremos la *matriz de estilo*, que medirá todas las correlaciones que hemos comentado anteriormente.

Nos referiremos a  $a_{i,j,k}^{[l]}$  como el valor de activación en la posición  $i, j, k$  de la capa de activación  $l$ , de tal forma que  $i$

indexa en altura,  $j$  indexa en el ancho y  $k$  indexa a través de los diferentes canales.

A continuación, vamos a calcular una matriz  $G^{[l]}$ , que será de dimensión  $n_c \times n_c$ , donde  $n_c$  es el número de canales existentes en la capa  $l$ , y que medirá el grado de correlación entre cada par de canales. En particular,  $G_{kk'}^{[l]}$  medirá como de correlacionados están los valores de activación del canal  $k$  con el canal  $k'$ , donde  $k$  y  $k' \in \{1, \dots, n_c\}$ .

Esta matriz la calculamos para la imagen estilo ( $S$ ) y para la imagen generada ( $G$ ) de la siguiente forma:

$$G_{kk'}^{[l](S)} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{i,j,k}^{[l](S)} a_{i,j,k'}^{[l](S)}, \quad (9.8)$$

$$G_{kk'}^{[l](G)} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{i,j,k}^{[l](G)} a_{i,j,k'}^{[l](G)}. \quad (9.9)$$

Si los valores de activación están correlacionados, entonces el valor de  $G_{kk'}$  será alto; mientras que si no lo están, será un valor pequeño.

Finalmente, la función de coste sobre una capa  $l$  se define de la siguiente forma:

$$J_{style}^{[l]}(S, G) = \frac{1}{(2n_H^l n_W^l n_C^l)^2} \sum_k \sum_{k'} (G_{kk'}^{[l](S)} - G_{kk'}^{[l](G)}). \quad (9.10)$$

Y la función de coste de estilo global se define como:

$$J_{style}(S, G) = \sum_l \lambda^{[l]} J_{style}^{[l]}(S, G), \quad (9.11)$$

donde  $\lambda^{[l]}$  es un hiperparámetro.



# **Parte IV**

## **Redes neuronales recurrentes**



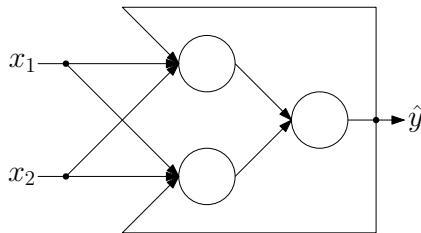
# Capítulo 10

## Fundamentos de las redes recurrentes

En este capítulo introduciremos los principios y conceptos fundamentales de las redes neuronales recurrentes. En concreto, iniciaremos este capítulo revisando el concepto de «recurrencia» que se aplica en este tipo de redes y, a continuación veremos los principales tipos de redes neuronales recurrentes. Finalizaremos el capítulo discutiendo el método de entrenamiento en el caso de las redes recurrentes.

### 10.1. Concepto de recurrencia

Hasta ahora, en todas las redes neuronales que hemos visto la información fluye siempre en una única dirección: desde las neuronas de entrada hacia las neuronas de salida. Este tipo de redes neuronales se conocen en inglés con el término *feed-forward*, pero, en general, en una red neuronal puede haber conexiones que vayan desde una capa posterior hacia una capa anterior.

**Figura 10.1.** Ejemplo de red neuronal recurrente

Fuente: elaboración propia

En la figura 10.1 podemos ver un ejemplo en el que la capa de neuronas ocultas está conectada a los datos de entrada y, a su vez, a la salida de las neuronas de la propia capa. De esta forma, cuando la red procesa un registro concreto está utilizando, a la vez, información del propio registro e información que generó la red con el registro anterior. Esto crea un estado interno en la red que le permite variar la respuesta a un cierto registro de entrada dependiendo de los registros que ha procesado la red anteriormente. Como veremos más adelante, en muchos casos al estado de la red también se le denomina *memoria* de la red.

Este tipo de conexiones pueden ser muy útiles para tratar secuencias de datos, es el caso de las series temporales o los textos. Por ejemplo, si queremos hacer análisis de sentimiento de una frase, podemos introducir las palabras de la frase una a una en la red neuronal, de forma que en cada paso de la red el estado va cambiando hasta que introducimos la última palabra de la frase, momento en el cual observamos la salida de la red para determinar si la frase tiene sentimiento positivo o negativo.

Observemos que, en principio, nada impide tratar secuencias de datos con una red completamente conectada como las que hemos visto en el capítulo 3. Para ello deberíamos fijar *a priori* la longitud de las secuencias a tratar e iríamos introduciendo el primer valor de la secuencia en la primera neurona, el segundo valor en la segunda neurona, etc. Uno de los mayores problemas con esta solución aparece cuando existen secuencias que tienen los mismos valores en distinto orden, pero aun así tienen el mismo significado. Un ejemplo de esto serían las frases «La película que vi ayer no me gustó mucho» y «No me gustó mucho la película que vi ayer». En una red completamente conectada, las neuronas tienen un orden fijo, por lo que reaccionan a los datos de entrada en un orden concreto. Esto quiere decir que, en este tipo de problemas donde el orden de las palabras no es lo más importante, deberíamos entrenar la red neuronal completamente conectada utilizando todas las variaciones posibles de cada frase, lo cual, en la mayoría de los casos, es inviable.

En general, las conexiones recurrentes en una red neuronal pueden ser de muchos tipos. Por esta razón, y para poder hacer un estudio unificado de las redes neuronales recurrentes, se define el concepto de «celda» (en inglés, *cell*). Una *celda* es simplemente una operación con dos entradas y dos salidas que se va componiendo consigo misma a medida que se aplica la operación a los diferentes valores de la secuencia. Los puntos de entrada de la celda corresponden, por un lado, a los valores de la secuencia para cada registro y, por otro lado, al estado de la red neuronal en el paso anterior. Los puntos de salida suelen ser el estado de la red en paso concreto en el que se ejecuta la celda y la respuesta de la red para ese paso.

Es decir, supongamos que denotamos el paso concreto de la secuencia con la letra  $t$ . Si los valores de la secuencia son vectores de dimensión  $m$  tenemos que, en el tiempo  $t$ , podemos denotar el vector de entrada a la red como  $x_t \in \mathbb{R}^m$ . De la misma forma, la salida en cada paso de la recurrencia puede ser un vector de dimensión  $l$ , por lo que tendremos  $h_t \in \mathbb{R}^l$ . Por último, debemos fijar un valor  $n$  para la dimensión del vector que representa el estado de la red, de forma que tendremos  $s_t \in \mathbb{R}^n$ . Con estas consideraciones, una celda es una operación  $f$  como la siguiente:

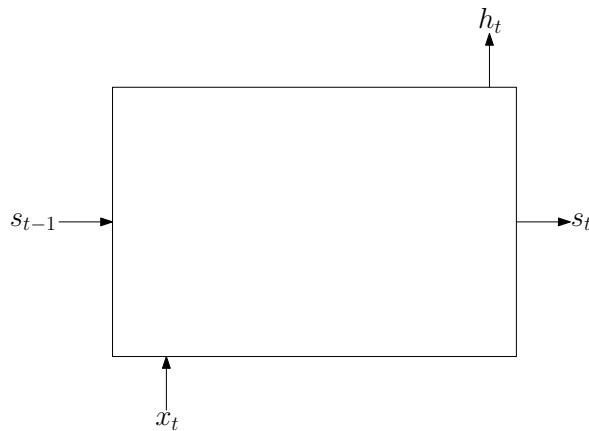
$$\begin{aligned} f : \quad \mathbb{R}^{m+n} &\rightarrow \mathbb{R}^{n+l}, \\ (x_t, s_{t-1}) &\mapsto (h_t, s_t). \end{aligned} \tag{10.1}$$

La operación que calcula la celda,  $f$ , se suele representar internamente con neuronas y conexiones no recurrentes entre las neuronas. En la figura 10.2 podemos observar una celda genérica, donde solo mostramos las entradas y salidas, mientras que en la figura 10.3 podemos visualizar la red de la figura 10.1 representada con el formato de celda.

En el ejemplo concreto de la figura 10.3, cada conexión representa un único valor escalar, pero, en general, las conexiones internas de una celda recurrente pueden trabajar con vectores, de forma que las salidas de la celda están formadas también por vectores, como se muestra en la ecuación 10.1. En este caso, normalmente se añaden una o varias capas completamente conectadas a la salida de la última celda recurrente para adaptar el tamaño de la salida al conjunto de datos con el que trabajamos.

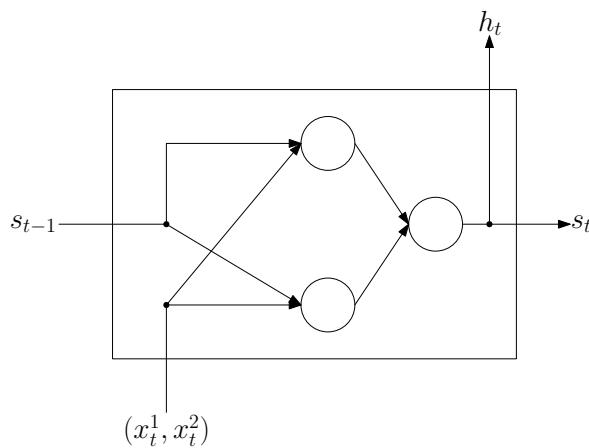
Si consideramos una red neuronal recurrente en su máxima generalidad, la salida de la red en un momento dado puede depender de datos tan antiguos como larga sea la secuencia. En la práctica, para hacer más eficiente el uso de las redes neu-

**Figura 10.2.** Diagrama de una celda, sin especificar las operaciones internas



Fuente: elaboración propia

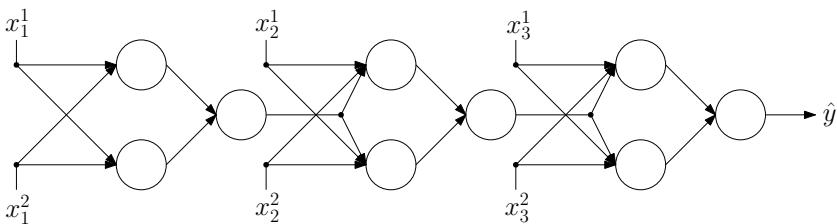
**Figura 10.3.** Representación en formato celda de la red recurrente de la figura 10.1



Fuente: elaboración propia

ronales recurrentes, se escoge un valor  $k$  que fija la longitud de las secuencias que tratará la red. Si las secuencias a tratar tienen longitud menor que  $k$  se añaden valores especiales de relleno (al principio o al final de la secuencia). En cambio, si las secuencias son más largas que el valor escogido se cortan y se descarta parte de la secuencia (se debe escoger también si descartar el principio o el final de la secuencia). Una vez fijado el valor de  $k$  podemos «desenrollar» la red para convertir la red neuronal recurrente en una red neuronal *feed-forward*, como podemos ver en la figura 10.4.

**Figura 10.4.** Red neuronal recurrente desenrollada con  $k = 3$



Fuente: elaboración propia

Es importante notar que los parámetros internos de las celdas son compartidos en todos los pasos, por lo que los valores de los parámetros son iguales en cada paso. La diferencia entre un paso y el siguiente estará únicamente en las entradas a la celda que, como hemos dicho, se corresponden con el estado de la red y los datos de entrada. Concretamente, los parámetros internos de la red fijan la función  $f : \mathbb{R}^{m+n} \rightarrow \mathbb{R}^{n+l}$ , de forma que, si fijamos inicial para  $s_0$ , podemos escribir la recurrencia como:

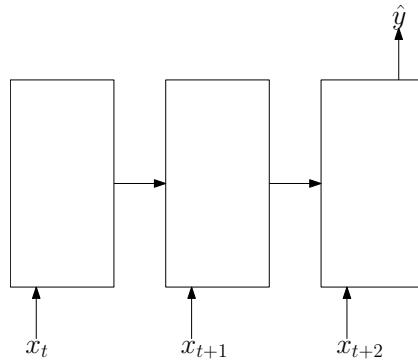
$$\begin{aligned}(h_1, s_1) &= f(x_1, s_0), \\ (h_2, s_2) &= f(x_2, s_1), \\ &\vdots \\ (h_t, s_t) &= f(x_t, s_{t-1}).\end{aligned}$$

## 10.2. Tipos de redes neuronales recurrentes

Como hemos comentado, las redes neuronales recurrentes son especialmente útiles para tratar con secuencias de datos, pero aun así podemos distinguir entre distintos tipos. Dependiendo del problema en el que estemos trabajando, puede interesarnos considerar únicamente algunas de las entradas o salidas de las celdas de la red, lo que da lugar a los siguientes tipos de redes recurrentes:

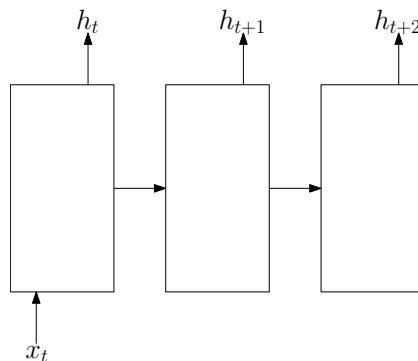
- Si las secuencias que tratamos tienen varios datos de entrada pero solo nos interesa un único valor que observamos al final de la recurrencia, obtenemos un diagrama como el de la figura 10.5. Un ejemplo de este caso de uso sería hacer un análisis de sentimiento sobre textos.
- Si trabajamos con secuencias que contienen un único valor pero nos interesa generar una salida con varios pasos, podemos introducir un valor nulo a partir del segundo paso de la recurrencia e ir observando la salida de la red, como aparece en la figura 10.6. Este tipo de red recurrente se puede utilizar por ejemplo para obtener descripciones dada una única imagen de entrada.

**Figura 10.5.** Red neuronal recurrente con varios datos de entrada y una única salida



Fuente: elaboración propia

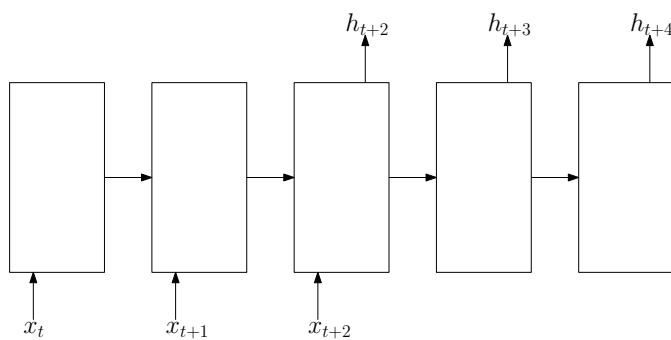
**Figura 10.6.** Red neuronal recurrente con un único dato de entrada y varios datos de salida



Fuente: elaboración propia

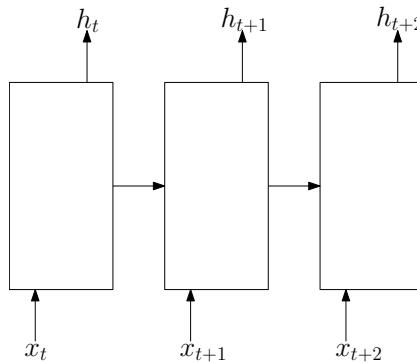
- En el caso en que tratemos secuencias con varios valores y estemos interesados en los valores que genera la red neuronal una vez tratada la secuencia de entrada, podemos proceder como en el caso anterior introduciendo valores nulos una vez procesada la secuencia y observando la salida de la red, como podemos ver en la figura 10.7. Si trabajamos con series temporales podemos utilizar este tipo de red recurrente para introducir valores pasados a la red y obtener la predicción de varios valores a futuro.
- Es posible que estemos interesados en trabajar con secuencias que tienen varios valores y necesitemos obtener una respuesta sincronizada para cada uno de los valores de entrada, en este caso tendríamos un diagrama como el de la figura 10.8. Un caso de uso de este tipo de red se da, por ejemplo, cuando se trabaja con un vídeo fotograma a fotograma y se desean clasificar objetos en cada uno de ellos.

**Figura 10.7.** Red neuronal recurrente con un varios datos de entrada y varios datos de salida



Fuente: elaboración propia

**Figura 10.8.** Red neuronal recurrente con un varios datos de entrada y varios datos de salida sincronizados



Fuente: elaboración propia

Es importante observar que las todas las celdas tienen siempre dos entradas y dos salidas. En cada una de las figuras 10.5, 10.6, 10.7 y 10.8, si no aparece alguna de las salidas es porque esa salida en concreto se ignora, mientras que si no aparece alguna de las entradas es porque en esos puntos se introduce un valor concreto que actúa como valor nulo.

### 10.3. Entrenamiento de una red neuronal recurrente

En el capítulo 3 hemos visto el algoritmo de retropropagación (*backpropagation*), que permite calcular de una forma eficiente el gradiente de una función de coste para una red neuronal *feed-forward*. En el caso de una red neuronal recu-

rrente no se puede aplicar directamente el algoritmo de retropropagación debido a las conexiones entre neuronas de capas posteriores con neuronas de capas anteriores.

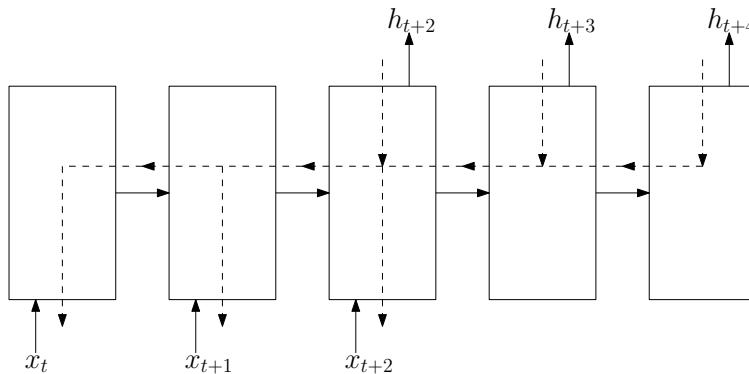
En general, hemos visto que entrenar una red neuronal es encontrar unos valores para los parámetros de la red que minimicen una determinada función de coste, y esto es igual en el caso de las redes recurrentes. Por esta razón, se pueden entrenar las redes neuronales recurrentes con cualquier algoritmo de optimización como, por ejemplo, los algoritmos genéticos. Aun así, dado el alto rendimiento del algoritmo de retropropagación para otros tipos de redes neuronales, es lógico pensar que una adaptación del mismo algoritmo puede resultar útil también en el caso de las redes recurrentes.

Esta adaptación se conoce con el nombre de retropropagación en el tiempo (en inglés, *backpropagation through time*), y no es más que desenrollar la red recurrente para convertirla en una red *feed-forward* y aplicar el algoritmo de retropropagación. Hay que tener en cuenta, eso sí, el tipo de red que estemos utilizando ya que, según hemos visto en la sección anterior, en ocasiones nos puede interesar trabajar con la salida de la red en diferentes pasos, lo cual afecta directamente a la función de coste. Recordemos el hecho de que en una red recurrente todos los parámetros internos de la celda son compartidos en todos los pasos, por lo que el gradiente de la función de coste puede afectar a un mismo parámetro de la red en repetidas ocasiones, dependiendo de en qué pasos se estén considerando las salidas de la red para calcular la función de coste.

Si nos fijamos en la figura 10.4, podemos intuir que el método de retropropagación calculará el error en cada una de las salidas que estemos considerando y lo propagará hacia atrás

en cada uno de los pasos, tal y como se muestra en la figura 10.9. Esto quiere decir que, si consideramos la salida de la red a cada paso, cada parámetro de la red neuronal contribuirá al gradiente de la función de coste una vez por cada uno de los pasos en los que hayamos desenrollado la red. Cuánto contribuirá exactamente cada parámetro en cada momento dependerá del paso concreto del que se trate y de los valores de entrada de la red.

**Figura 10.9.** Flujo del gradiente de la función de coste en una red recurrente



Fuente: elaboración propia

En una red neuronal en la que haya conexiones recurrentes que vayan directamente de capas posteriores a capas anteriores se puede demostrar, haciendo el cálculo analítico, que el gradiente de la función de coste con respecto a los parámetros en un paso concreto de la red depende del producto del gradiente en los pasos posteriores. Esto implica que, si calculamos cuánto contribuye al gradiente un parámetro concreto en uno de los pasos iniciales de la red, el gradiente aparecerá elevado a una potencia, que puede ser muy alta si hemos

desenrollado la red en un gran número de pasos intentando capturar dependencias lo más lejanas posibles.

El hecho de que el gradiente se vea multiplicado consigo mismo en el cálculo de cómo afectan los parámetros en cada paso provoca dos problemas diferentes pero relacionados, que ya hemos comentando anteriormente en la sección 3.5, pero que volvemos a resumir a continuación:

- Desaparición del gradiente: si la norma del gradiente es menor que 1, al multiplicar el gradiente consigo mismo muchas veces el resultado es prácticamente 0, por lo que los primeros pasos de la red apenas aportan nada al cálculo del gradiente. Esto implica que la red no será capaz de aprender dependencias entre valores alejados de una secuencia, ya que el valor de los parámetros de la red se verá influenciado únicamente por los últimos pasos de la recurrencia.
- Explosión del gradiente: si la norma del gradiente es mayor que 1, al multiplicar el gradiente consigo mismo muchas veces el resultado es inmensamente grande. Esto provoca que el entrenamiento de la red sea completamente inestable, ya que las modificaciones que se aplican a los parámetros utilizando el método de retropropagación hacen que haya demasiadas fluctuaciones y la red no consigue aprender nada.



# Capítulo 11

## Tipología de celdas recurrentes

La solución más utilizada para resolver los problemas de explosión y desaparición del gradiente consiste en utilizar celdas en las que existe una conexión entre el estado anterior y la salida de la celda a la cual no se aplican funciones no lineales. En este tipo de celdas se controla la información que fluye por esta conexión directa utilizando puertas lógicas modeladas con neuronas completamente conectadas.

En este capítulo veremos las dos arquitecturas de celdas más utilizadas de este tipo, la arquitectura LSTM y la arquitectura GRU.

### 11.1. *Long short term memory (LSTM)*

En muchos casos las secuencias que queremos analizar con una red neuronal recurrente pueden presentar dependencias entre puntos de la secuencia muy separados. Por ejemplo, si utilizamos una red neuronal recurrente para analizar un texto,

es posible que dentro de este se hagan referencias a conceptos que se han mencionado cientos de palabras antes, por lo que, si nuestro objetivo es que la red sea capaz de extraer información del texto, es posible que necesitemos que trate cientos de pasos.

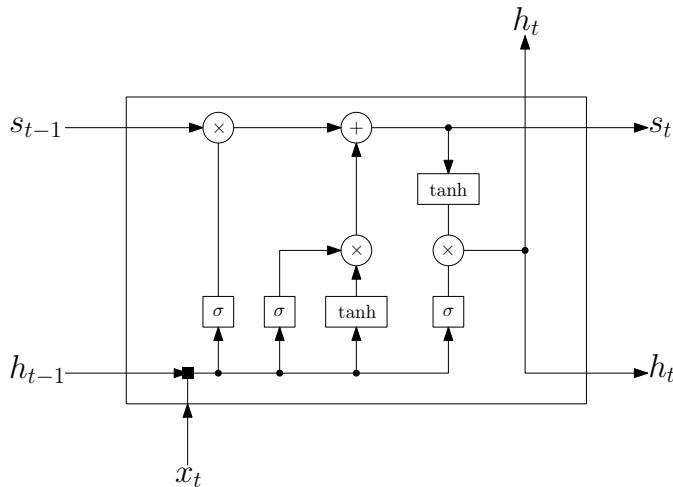
### 11.1.1. Estructura de las celdas

Este tipo de modelos presentan siempre problemas como la desaparición o explosión del gradiente. Para intentar mitigar estos problemas, en 1997 se propuso el uso de una arquitectura de celda muy concreta llamada *long short term memory* (LSTM) (Hochreiter y Schmidhuber, 1997). Las entradas de una celda LSTM son, por una parte, el valor de la secuencia en el paso correspondiente y, por otra parte, la concatenación del estado y la salida de la red en el paso anterior. Normalmente, para evitar tener que concatenar vectores a la salida de un paso para separarlos justo al inicio del paso siguiente se suelen representar las celdas LSTM con tres entradas, como podemos observar en la figura 11.1.

Una red LSTM es capaz de aprender tanto dependencias entre puntos alejados de una secuencia, como dependencias entre puntos cercanos. Los puntos claves para que la red tenga esta habilidad son dos:

1. El *canal de memoria* que se puede observar como una línea horizontal en la parte superior de la celda.
2. El *control del flujo de información* mediante puertas.

Como podemos ver en el diagrama, la arquitectura de la celda LSTM se basa en controlar qué información se guarda

**Figura 11.1.** Diagrama de una celda LSTM

Fuente: elaboración propia

en el *estado* (también llamado *memoria*) y fluye directamente por la parte superior y qué información se produce como respuesta de la red. Este control se ejerce mediante las siguientes puertas:

### Puerta de olvido

La puerta de olvido (*forget gates*) observa el valor de entrada a la celda junto con la salida de la red en el paso anterior y decide qué parte de la memoria se debe conservar. Concretamente, la puerta de olvido calcula un vector  $f_t$ , de la misma longitud que los vectores de estado, con la siguiente fórmula:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f), \quad (11.1)$$

donde  $W_f$  es una matriz de parámetros que multiplica a la concatenación de  $h_{t-1}$  y  $x_t$ , y  $b_f$  es un vector de sesgos.

Para imaginar el uso de la puerta de olvido, supongamos que tenemos un conjunto de datos con el consumo diario de un cierto producto y queremos predecir el consumo durante los próximos treinta días. Asumamos, además, que por las características del producto, el consumo en un día concreto únicamente depende del consumo que se haya producido en la misma semana. En este caso, si en cada paso introducimos en la red el consumo de un día y añadimos información sobre el día de la semana del que se trata, es posible que la puerta de olvido aprenda a borrar toda la memoria cuando la entrada indique que está tratando con el consumo de un lunes. De esta forma, toda la información pasada será eliminada del estado y los pasos siguientes no tendrán información sobre el consumo la semana anterior, lo cual puede ser útil en un problema como el descrito.

## Puerta de entrada

La puerta de entrada controla qué información se añade a la memoria de la red. Es una puerta completamente análoga a la puerta de olvido, pero no comparten parámetros:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i), \quad (11.2)$$

donde  $W_i$  es una matriz de parámetros y  $b_i$  es un vector de sesgos.

Como caso de uso simple, imaginemos que estamos trabajando con diferentes frases y queremos extraer si el objeto principal de la frase es masculino o femenino. En este caso, habrá palabras que den información relevante y otras que no, por lo que es posible que la red neuronal aprenda a recono-

cer las palabras que aportan información y deje pasar únicamente la información que corresponda a las palabras útiles para el problema en cuestión.

A la vez que se calculan los valores de la puerta de entrada y, utilizando exactamente la misma información pero con diferentes parámetros, se calcula qué información concretamente es candidata a añadirse a la memoria con la siguiente fórmula:

$$\tilde{s}_t = \tanh(W_s \cdot [h_{t-1}, x_t] + b_s), \quad (11.3)$$

donde  $W_s$  vuelve a ser otra matriz diferente de parámetros y  $b_s$  es otro vector diferente de sesgos.

Por último, una vez se ha calculado la información candidata a añadirse a la red y los valores de la puerta de entrada y salida, el estado de la red se actualiza de la siguiente forma:

$$s_t = f_t \odot s_{t-1} + i_t \odot \tilde{s}_t, \quad (11.4)$$

donde  $\odot$  denota el producto de vectores componente a componente.

Como se puede apreciar en la última fórmula, el nuevo estado de la red viene dado por una suma ponderada entre el estado anterior y la nueva información, donde los pesos de cada componente vienen dados por la puerta de olvido y la puerta de entrada, respectivamente.

## Puerta de salida

Una vez se ha calculado el nuevo estado que tendrá la red, hay que decidir cuál será la salida de la red en el paso en el que nos encontramos. Para eso utilizamos la puerta de salida, cuyo proceso es similar al de la puerta de entrada. Concretamente, la puerta de salida observa también el valor de entrada a la

celda junto con la salida anterior de la red y calcula un vector  $o_t$  de la siguiente forma:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o). \quad (11.5)$$

Este vector marcará qué información se da como respuesta, la cual vendrá dada por la siguiente fórmula:

$$h_t = o_t \odot \tanh(s_t). \quad (11.6)$$

Como podemos observar, la salida de la red consistirá en una transformación no lineal del estado actual de la red, modulada según el valor de la puerta de salida.

Enlazando con los ejemplos anteriores, es intuitivo pensar que la memoria de la red contendrá información útil para dar una respuesta en el paso concreto en el que se encuentra la red, pero también contendrá información útil para los pasos siguientes, por lo que la puerta de salida se encarga de dejar pasar únicamente la información que resulta útil en el momento actual.

### 11.1.2. Consideraciones sobre las celdas LSTM

Observemos que todas las fórmulas anteriores describen exactamente diferentes capas de una red neuronal, ya sea con la función sigmoide o con la tangente hiperbólica como funciones de activación. Esta forma de construir las celdas es lo que da el nombre de redes neuronales recurrentes.

El punto más interesante de controlar una memoria con puertas, como en el caso de una LSTM, es que el control se hace de forma difusa, ya que las puertas pueden tomar cualquier valor real entre 0 y 1, de tal manera que pueden dejar pasar información solo en parte.

Por ejemplo, en el caso de intentar predecir el consumo de un producto, imaginemos que la dependencia no es únicamente con los días de la misma semana, pero sí sabemos que la tendencia en el consumo disminuye un 20 % los domingos. En este caso concreto, la red neuronal podría aprender a mantener el valor de la tendencia en alguna componente del vector de estado  $y$ , de esta forma, cuando la entrada a una celda indique que se está considerando un domingo, es posible que la puerta de olvido multiplique por 0,8 la componente concreta de la memoria que guarda la tendencia del consumo.

De forma parecida, en el ejemplo en el que queremos extraer información de una frase, es posible que no haya una única palabra que contenga toda la información, si no que para lograr el objetivo del modelo se necesite información de varias palabras. En este caso, la puerta de entrada de la red podría aprender a detectar cuánta información relevante contiene cada palabra y modular el paso de información hacia la memoria de la red.

Anteriormente vimos que las redes neuronales recurrentes simples no son capaces de modelar largas dependencias por culpa de los problemas de desaparición y explosión del gradiente. En el caso de las LSTM, se puede comprobar que, gracias a las fórmulas utilizadas para calcular el estado, la información del gradiente de la función de coste puede fluir hacia atrás sin multiplicarse consigo misma un número elevado de veces. Este hecho permite que valores de entrada en los primeros pasos puedan tener un efecto importante en el entrenamiento de la red y aprender así dependencias entre puntos alejados de una secuencia.

Otra técnica muy utilizada que ayuda a evitar el problema de la explosión del gradiente es el recorte del gradiente (en

inglés, *gradient clipping*). El *recorte del gradiente* consiste en fijar un valor máximo para cada componente del gradiente, de forma que si alguna de estas componentes es superior al valor fijado se le asigna directamente el valor máximo.

## 11.2. *Gated recurrent unit (GRU)*

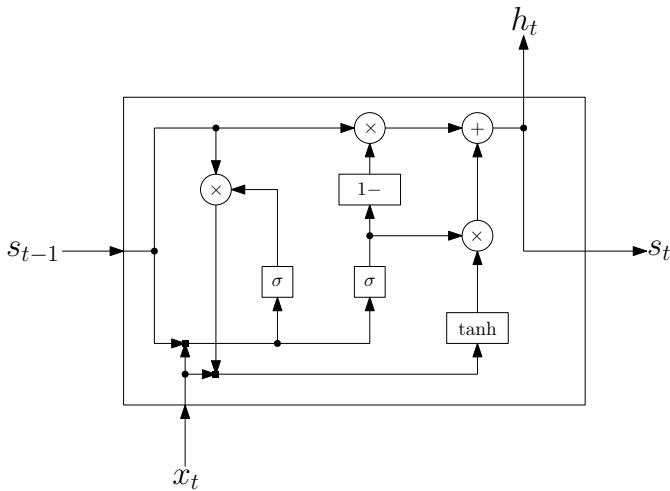
Dado el alto rendimiento de las redes LSTM se han investigado diferentes variantes de celdas en las que se controla el flujo de información utilizando capas de neuronas como puertas que dejan pasar más o menos información hacia la memoria de la red. Una de las variantes que se ha demostrado más útil se introdujo en 2014 con el nombre de *gated recurrent unit* (GRU) (Cho, van Merriënboer y col., 2014a).

### 11.2.1. Estructura de las celdas

En la figura 11.2 podemos ver el diagrama de una celda GRU, en la que podemos observar que las dos salidas que normalmente hay en una celda son exactamente iguales.

Una celda GRU se puede ver como una simplificación de una celda LSTM, en la que se fusiona el estado con la respuesta de la red y además se fuerza a que el estado de la red sea una media ponderada entre el estado anterior y la nueva información, pero también existen ligeras diferencias en el cálculo concreto que se realiza para obtener la respuesta de la red.

Las redes recurrentes con celdas GRU han demostrado en varios casos dar rendimientos muy similares a las redes recurrentes con celdas LSTM. Por esta razón, las celdas GRU son muy utilizadas cuando el conjunto de datos no es extremadamente grande, ya que en una celda GRU hay menos

**Figura 11.2.** Diagrama de una celda GRU

Fuente: elaboración propia

parámetros y por lo tanto se necesitan menos datos para el entrenamiento.

Concretamente, en una celda GRU el flujo de información se controla con las puertas de *reset* y las puertas de actualización, que pasamos a describir a continuación.

### Puerta de *reset*

La puerta de *reset* permite seleccionar qué información de la memoria va a ser utilizada en un paso concreto. Para ello, observa el estado anterior de la red y los datos de entrada y obtiene un vector que será multiplicado posteriormente con el estado de la memoria. En este caso concreto la fórmula utilizada es la siguiente:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r), \quad (11.7)$$

donde  $W_r$  es una matriz de parámetros que multiplica a la concatenación de  $h_{t-1}$  y  $x_t$ , y  $b_r$  es un vector de sesgos.

## Puerta de actualización

La puerta de actualización es análoga a las puertas de olvido y entrada de una celda LSTM. En particular, esta celda controla qué información se mantiene en la memoria y qué información de la memoria se olvida. En primer lugar se calcula un vector de la misma longitud que el vector de estado con la siguiente fórmula:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z), \quad (11.8)$$

donde  $W_z$  es una matriz de parámetros y  $b_z$  es un vector de sesgos.

A continuación se calcula la información candidata a añadirse a la memoria (recordemos que en el caso de una celda GRU el vector de estado de la red y el vector de salida coinciden). En una celda GRU este cálculo utiliza los datos de entrada, el estado anterior de la red y la salida de la puerta de *reset*, tal y como muestra la fórmula siguiente:

$$\tilde{s}_t = \tanh(W_s \cdot [r_t \odot h_{t-1}, x_t] + b_s), \quad (11.9)$$

donde  $W_s$  es una matriz de parámetros y  $b_s$  es un vector de sesgos.

Por último, se actualiza el estado de la red utilizando una media ponderada entre el estado anterior y la información recién calculada:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{s}_t \quad (11.10)$$

### 11.2.2. Consideraciones sobre las celdas GRU

Como hemos comentado, una celda GRU es, en cierto modo, una simplificación de una celda LSTM, pero aun así existen otras variantes de celdas que simplifican aún más la arquitectura para necesitar el mínimo número de parámetros posible, y poder así entrenar la red recurrente con un conjunto de datos relativamente pequeño.



# Capítulo 12

## Arquitecturas de redes recurrentes

Una vez sabemos que tipo de celda queremos utilizar necesitamos decidir cómo se van a organizar las celdas. En este capítulo veremos las técnicas mas comúnmente utilizadas para conectar celdas recurrentes de forma que podamos obtener el máximo rendimiento dependiendo del problema que queramos resolver.

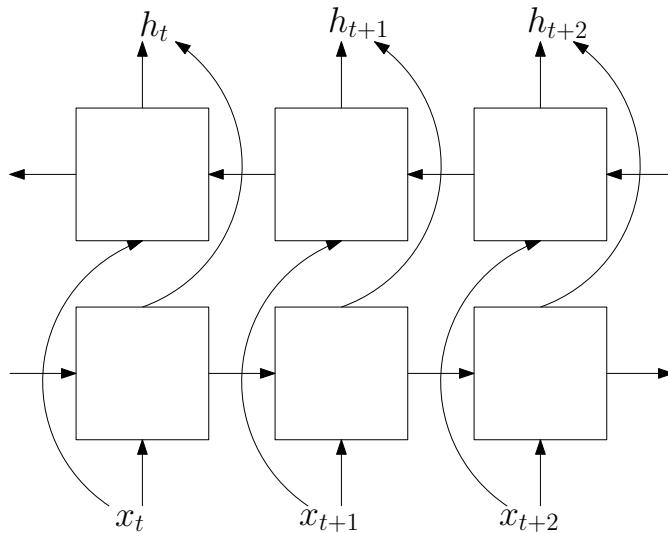
### 12.1. Redes neuronales recurrentes bidireccionales

En muchos casos de uso de una red recurrente, la secuencia de datos tiene una clara componente temporal como, por ejemplo, en las series temporales. En otros casos de uso, sin embargo, los datos de entrenamiento están formados por secuencias que no tienen una componente temporal subyacente como, por ejemplo, en el caso de los textos.

En estos casos en los que trabajamos con secuencias aisladas sin componente temporal es posible introducir los valores de entrada a la red, tanto en el orden original de los datos como en el orden inverso. Un ejemplo claro de que es importante el orden en el que se introducen los datos en la red se da cuando se trabaja con textos. En efecto, dependiendo del idioma de un texto las palabras más relevantes pueden estar al principio o al final de la frase. De la misma forma, las construcciones en las que ciertas palabras se ven modificadas por palabras adyacentes pueden tener diferente orden en diferentes idiomas.

En general, en estos casos sin componente temporal subyacente, para obtener el máximo posible de información es posible utilizar *redes neuronales recurrentes bidireccionales* Schuster y Paliwal (1997). En este tipo de redes primero se aplica una recurrencia en un orden y luego se aplica otra recurrencia en el orden inverso, para después aglutinar toda la información recogida en una capa final. La figura 12.1 muestra un diagrama de una red neuronal recurrente bidireccional que utiliza una capa completamente conectada para aglutinar toda la información.

Es importante observar en dicha figura cuáles son las conexiones entre las diferentes partes de la arquitectura de la red. En particular, podemos ver que las dos recurrencias de la red son independientes, y que reciben cada parte de la recurrencia el estado anterior respectivo y los valores de entrada (en orden diferente). La información que proporciona cada celda en cada uno de los pasos se concatena una vez finalizado el análisis de la secuencia, tanto en un orden como en orden inverso, y se utiliza una capa completamente conectada para extraer la información útil según el problema. En la sección

**Figura 12.1.** Diagrama de una red neuronal recurrente bidireccional

Fuente: elaboración propia

12.4 veremos una alternativa a la concatenación directa de todas las salidas de las celdas.

Este tipo de arquitecturas son muy útiles, por ejemplo, en problemas de clasificación de texto. En este caso podemos entender la red neuronal como dos agentes que extraen el máximo posible de información del texto, uno leyendo el texto en el orden original y el otro leyendo el texto en el orden inverso, para luego compartir la información y hacer una clasificación definitiva utilizando una capa completamente conectada.

## 12.2. Redes neuronales recurrentes profundas

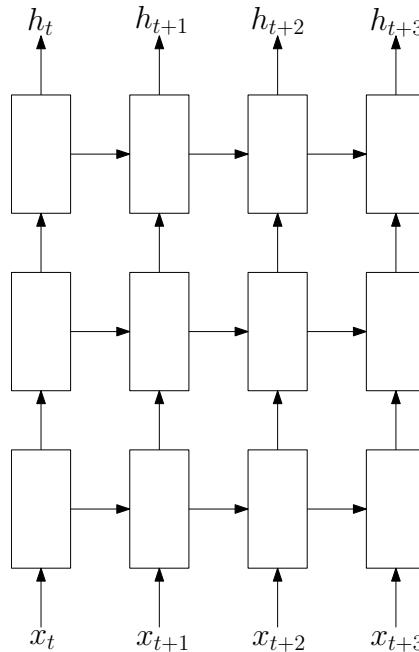
Si observamos la arquitectura de una celda LSTM o una celda GRU, en las figuras 11.1 y 11.2, podemos ver que las operaciones que se realizan sobre los datos de entrada son como si tuviéramos una red neuronal con una única capa. Es cierto que existe un cierto grado de composición entre las operaciones dada la recurrencia de la red, pero esta composición se hace siempre utilizando los mismos parámetros y cambiando únicamente los datos de entrada en cada paso. Esto, como en el caso de las redes neuronales completamente conectadas, provoca que la red no tenga una gran capacidad de abstracción.

Para intentar dotar de más flexibilidad a una red recurrente podemos proceder como en el caso de las redes completamente conectadas y apilar capas de celdas recurrentes unas encima de otras. En la figura 12.2 podemos ver exactamente cómo podemos conectar las entradas y salidas de varias celdas para crear una *red neuronal recurrente profunda* Graves, Mohamed y col. (2013).

Es importante tener en cuenta qué parámetros son compartidos entre las celdas de la red. En la figura 12.3 podemos ver la misma red recurrente que aparece desenrollada en la figura 12.2. Como podemos ver, las celdas recurrentes de una misma capa comparten parámetros, pero cada una de las capas es independiente, por lo que no comparten parámetros.

Si observamos de nuevo la figura 12.2, podemos apreciar que, a diferencia de lo que ocurre en una red recurrente como la de la figura 10.4, la información que entra a la red pasa por diferentes capas con diferentes parámetros. Esto permite

**Figura 12.2.** Ejemplo de una red neuronal recurrente profunda desenrollada

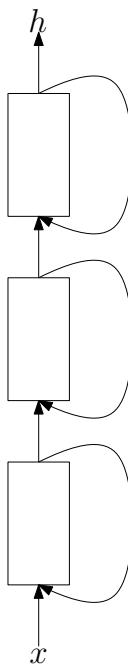


Fuente: elaboración propia

a la red especializar cada capa en conceptos ligeramente más complejos que los que trata la capa anterior, por lo que, de la misma forma que pasa con las CNN (ver capítulo 7), la red es capaz de aprender conceptos muy abstractos. Pero, como en todas las redes neuronales, este aumento de la capacidad de abstracción viene asociado a una mayor complejidad del proceso de entrenamiento.

En una red neuronal recurrente profunda se unen el problema de la desaparición y la explosión del gradiente a la complejidad de entrenar una red neuronal con varias capas.

**Figura 12.3.** Red neuronal recurrente de la figura 12.2 sin desenrollar



Fuente: elaboración propia

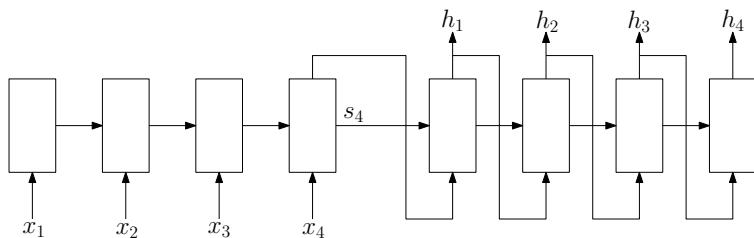
Es completamente indispensable disponer de un conjunto de datos muy grande para poder entrenar una red neuronal recurrente profunda.

Además, es muy importante ajustar correctamente los diferentes hiperparámetros del entrenamiento para obtener una red neuronal estable. Esto puede provocar que debamos probar diferentes combinaciones de hiperparámetros buscando la estabilidad, lo que junto con el tiempo de entrenamiento de la red, puede alargar mucho el tiempo de desarrollo de una red de este tipo.

### 12.3. Arquitectura codificador-decodificador

En la figura 10.7 hemos visto que en una red neuronal recurrente podemos tener, a la vez, secuencias de entrada y secuencias de salida. Un caso particular de esta combinación de entradas y salidas lo podemos ver en la figura 12.4, en la que primero la red neuronal consume completamente la secuencia de entrada y, posteriormente, genera la secuencia de respuesta (Cho, van Merriënboer y col., 2014a; Sutskever, Vinyals y col., 2014).

**Figura 12.4.** Red neuronal recurrente con arquitectura *encoder-decoder/sequence-to-sequence*



Fuente: elaboración propia

En una red neuronal con este tipo de arquitectura se denombra *codificador* a la primera parte de la red y *decodificador* a la segunda parte.

La información que pasa del codificador al decodificador está toda comprimida en el *vector de estado* del último paso del codificador. En este sentido, el trabajo del codificador consiste en guardar toda la información de la secuencia de entrada que sea relevante para el problema que estamos tratando en un vector que luego el decodificador sea capaz de tratar.

Por otra parte, el trabajo del decodificador consiste en crear una secuencia de salida basada únicamente en la información que recibe del codificador.

El caso de uso típico de este tipo de arquitectura se da en los problemas de traducción automática. En estos casos, la red primero consume toda la secuencia de entrada para obtener toda la información de una frase y después genera la fase traducida palabra a palabra. El trabajo del codificador en este caso consiste en comprimir toda la información contenida en una frase en un vector, de forma que después el decodificador sea capaz de interpretar el contenido del vector y generar una frase en un nuevo idioma que tenga el mismo significado que la frase original.

Como en muchos otros casos, el punto clave de este tipo de redes neuronales está en conseguir representar la información relevante en un vector numérico. Para obtener unos vectores lo más representativos posibles se pueden combinar las técnicas que hemos visto en las secciones anteriores sobre redes bidireccionales y redes recurrentes profundas.

En este sentido, dependiendo del problema, es posible que una red neuronal bidireccional sea capaz de obtener una mejor representación vectorial de la secuencia de entrada, de manera que tener un codificador bidireccional hará que el decodificador pueda obtener mejores resultados.

De la misma forma, si las secuencias de entrada contienen conceptos abstractos, es muy posible que utilizar diversas capas en el codificador y en el decodificador permita obtener mejores resultados. Estas mejoras llevan, como es de esperar, una mayor complejidad en el entrenamiento de la red neuronal, por lo que normalmente es necesario disponer de

una gran cantidad de datos para entrenar una red neuronal con arquitectura codificador-decodificador.

## 12.4. Mecanismo de atención

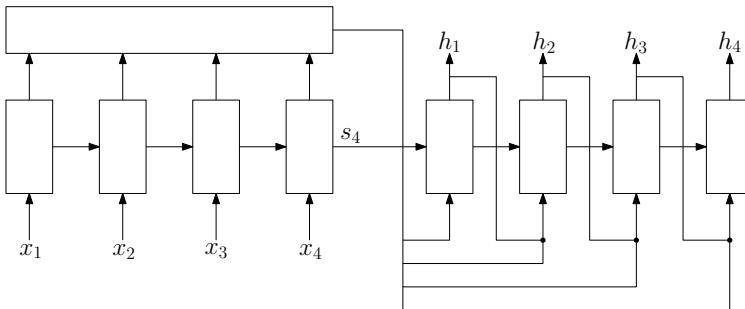
Como hemos visto anteriormente, con una arquitectura codificador-decodificador se puede entrenar un modelo que permita traducir frases entre idiomas. En este tipo de arquitecturas, es intuitivo pensar que la parte más compleja del modelo está en sintetizar en un único vector toda la información contenida en una frase, para después, poder generar una frase en otro idioma.

Para mejorar en este proceso de compresión de la información, en 2015 se introdujo el mecanismo de atención (Bahdanau, Cho y col., 2015), cuyo objetivo es ayudar a construir un vector que condense la información contenida en una frase de la mejor forma posible.

Concretamente, con el *mecanismo de atención*, en lugar de utilizar el vector de estado del último paso de la red recurrente como representación de toda la secuencia, se utiliza una suma ponderada de las salidas de cada paso. En la figura 12.5 podemos ver una representación del mecanismo de atención.

La idea fundamental es utilizar la salida de la red en cada paso para estimar unos pesos que se utilizarán después para ponderar estas salidas y sumarlas. Esta suma ponderada de todas las salidas se introduce después en cada paso del decodificador, lo que permite a la red centrar la atención en ciertos puntos de la secuencia en cada momento. De esta forma, es posible representar más fidedignamente las partes importantes de la secuencia en el vector que se introduce en cada paso del decodificador.

**Figura 12.5.** Mecanismo de atención en una red neuronal recurrente con arquitectura *encoder-decoder/sequence-to-sequence*

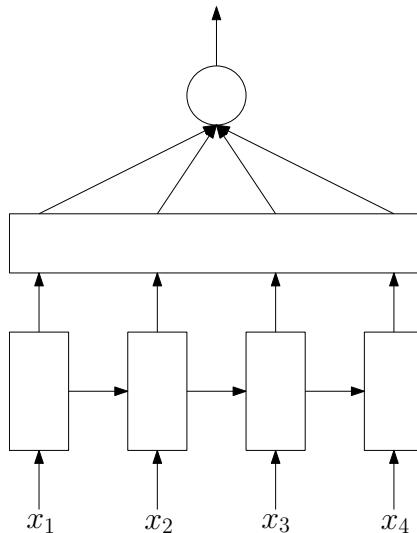


Fuente: elaboración propia

El mecanismo de atención también puede ser utilizado con una red recurrente en un problema de regresión o clasificación habitual. El proceso es exactamente el mismo que con una arquitectura codificador-decodificador, pero en lugar de calcular una suma ponderada de las salidas del codificador para cada paso del decodificador, se realiza una única suma ponderada al finalizar de tratar cada secuencia de entrada. En la figura 12.6 podemos ver un diagrama con una red neuronal con mecanismo de atención aplicada a un problema de clasificación.

Este mecanismo puede ser muy útil en casos en los que una parte de una secuencia es la que lleva toda la información relevante para el problema. Por ejemplo, si queremos extraer el tiempo verbal de una frase es probable que la información relevante se encuentre en una zona muy concreta de la frase que no tiene por qué estar al final. Utilizando un mecanismo de atención, la red puede aprender a detectar los verbos, aislarlos

**Figura 12.6.** Mecanismo de atención en una red neuronal recurrente para un problema de clasificación



Fuente: elaboración propia

y luego clasificar el tiempo verbal sin prestar atención a otras partes de la frase que pueden aportar ruido.



# Capítulo 13

## Consejos prácticos y ejemplos

En este último capítulo en el bloque de redes neuronales recurrentes, nos centraremos en ver algunos consejos prácticos a la hora de trabajar con RNN y series temporales para la creación de modelos predictivos, en la sección 13.1. Finalmente, veremos algunos ejemplos de problemas que se pueden resolver empleando las RNN, en la sección 13.2.

### 13.1. Consejos prácticos en el uso de RNN

Durante esta parte dedicada a las RNN hemos visto algunos ejemplos de problemas que se pueden resolver con redes neuronales recurrentes. Sin embargo, es importante tener presente que cada problema tiene unas características especiales que requerirán diferentes soluciones, ya sea en la arquitectura de la red neuronal, o en el procesado de los datos.

En el caso de trabajar con series temporales, por ejemplo, es importante utilizar técnicas estadísticas clásicas para pre-

procesar los datos, como la normalización o la eliminación de tendencias, para obtener los mejores resultados posibles del entrenamiento.

Una de las técnicas más utilizadas es la descomposición de una serie temporal en tres componentes: tendencia, estacionalidad y componente irregular. Esta descomposición se puede realizar de dos formas diferentes: aditiva y multiplicativa. Si tenemos una serie temporal  $y_t$ , aplicando una descomposición aditiva obtenemos

$$y_t = T_t + S_t + I_t, \quad (13.1)$$

mientras que si aplicamos una descomposición multiplicativa obtenemos

$$y_t = T_t \cdot S_t \cdot I_t, \quad (13.2)$$

donde  $T_t$  denota la tendencia,  $S_t$  la estacionalidad e  $I_t$  la componente irregular. En la figura 13.1 podemos ver un ejemplo de una serie temporal descompuesta de forma aditiva.

Utilizando la descomposición en tres componentes transformamos el problema de entrenar una red neuronal para predecir una serie temporal a tener que predecir tres series temporales diferentes. La ventaja en este caso es que las componentes serán más fáciles de predecir, especialmente la parte de la tendencia y de la estacionalidad.

Posteriormente, cuando queramos predecir un nuevo valor de la serie original simplemente deberemos predecir los valores de las tres componentes y agregarlos utilizando el mismo método que utilizamos para la descomposición, ya sea aditiva o multiplicativa.

En el caso de que estemos utilizando una red neuronal recurrente para trabajar con textos, es muy importante selec-

cionar una forma adecuada para representar numéricamente las palabras que forman cada registro.

La forma más simple de representar un texto consiste en fijar primero un diccionario con  $N$  palabras (por ejemplo, 1.000, 2.000 o 5.000 palabras). Entonces, para cada texto se construye un vector de  $N$  dimensiones para cada una de las palabras. Así, un texto se asocia con un número de vectores igual a la longitud del texto, y donde cada vector tiene todas las componentes igual a 0 excepto una componente, que tiene el valor 1, e indica qué palabra del diccionario estamos representando con el vector.

Esta forma de representar textos tiene varios inconvenientes, como por ejemplo:

- Cada palabra está descrita por un vector de  $N$  dimensiones, donde  $N$  puede ser un valor muy grande si el diccionario contiene muchas palabras. Por lo tanto, no es adecuado en la mayoría de los casos porque incrementa de forma considerable el tamaño de la red neuronal, aumentando la complejidad del entrenamiento.
- Las palabras tienen relaciones de similitud que no se ven reflejadas en los vectores. Es decir, los vectores correspondientes a dos palabras que tengan significado parecido tendrán la misma relación que los vectores correspondientes a dos palabras completamente diferentes.

Esto vuelve a incrementar la complejidad del entrenamiento, ya que dos textos que tengan significados diferentes tendrán representaciones vectoriales sin ninguna relación, por lo que la red tendrá que aprender por separado sobre los dos textos y no podrá utilizar infor-

mación aprendida con un ejemplo a la hora de tratar con ejemplos relacionados.

Para solucionar estos dos problemas se suele utilizar una capa de *embedding* justo antes de la entrada de la red neuronal.

Esta capa es simplemente una matriz de tamaño  $N \times k$ , donde  $N$  es el número de palabras que tenemos en el diccionario y  $k$  es la dimensión del *embedding*, que suele estar entre 50 y 300, esto es,  $k \in \{50, \dots, 300\}$ .

Esta capa de *embedding* asigna a cada palabra un vector con  $k$  valores reales, por lo que es posible representar las  $N$  palabras del diccionario con solo  $k$  componentes. Para que este proceso sea útil, la representación de cada palabra con  $k$  valores debe cumplir que palabras con significado similar tengan valores similares, mientras que palabras con significado diferente tengan valores diferentes. Esto se puede conseguir de dos formas principalmente:

- Crear la matriz de *embedding* de forma no supervisada como preentrenamiento de la red neuronal recurrente.

En este caso, el proceso consiste en utilizar un algoritmo como Word2Vec (Mikolov, Sutskever y col., 2013), que es capaz de extraer relaciones semánticas entre palabras de forma no supervisada observando únicamente los textos disponibles para el problema que intentamos resolver, sin ninguna otra forma de supervisión.

- Crear la matriz de *embedding* a la vez que se entrena la red neuronal recurrente.

En este caso se empieza con una matriz de *embedding* con valores aleatorios y se consideran las entradas de la

matriz como parámetros de la red neuronal durante el entrenamiento.

El propio proceso de entrenamiento de la red neuronal ajustará los valores de los vectores para que cada palabra quede representada en  $k$  dimensiones de tal forma que la respuesta final de la red sea lo mejor posible.

En el caso de optar por una matriz de *embedding* preentrenada es posible utilizar *embeddings* disponibles en Internet que ya están preentrenados con grandes cantidades de textos como, por ejemplo, la Wikipedia.

Para escoger entre una forma u otra de crear la matriz de *embedding* deberemos tener en cuenta diferentes puntos como, por ejemplo, si disponemos de suficientes textos como para crear un *embedding* de forma no supervisada o si el vocabulario de nuestros textos es demasiado específico como para utilizar un *embedding* genérico obtenido de internet.

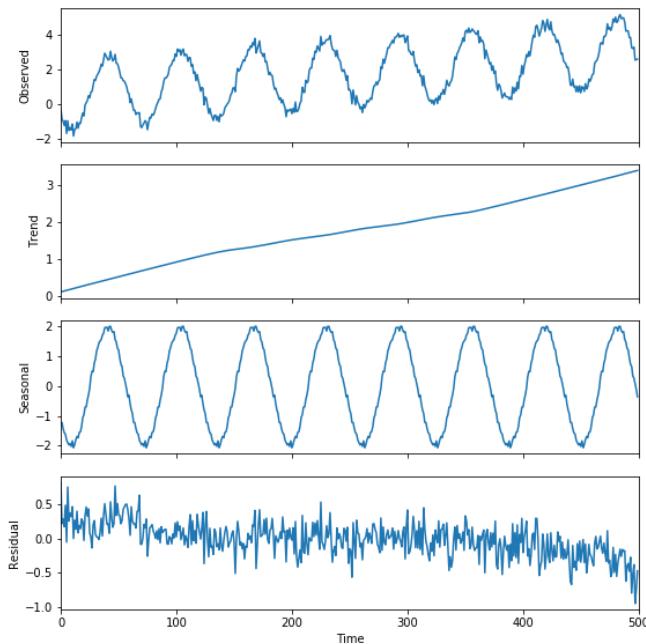
También puede ser interesante mezclar las dos aproximaciones y crear la matriz de *embedding* utilizando primero un preentrenamiento no supervisado y afinando después los valores de los vectores durante el entrenamiento de la red neuronal.

## 13.2. Ejemplos

A continuación veremos cómo podemos predecir valores futuros de la serie temporal de la figura 13.1 utilizando su descomposición.

Esta serie temporal se ha generado sintéticamente, por lo que sabemos que todo aquello que no pertenece a la tendencia o a la estacionalidad es ruido blanco. En este sentido, el siguiente ejemplo muestra como se podrían predecir valores

**Figura 13.1.** Serie temporal y su descomposición en tendencia, estacionalidad y componente irregular

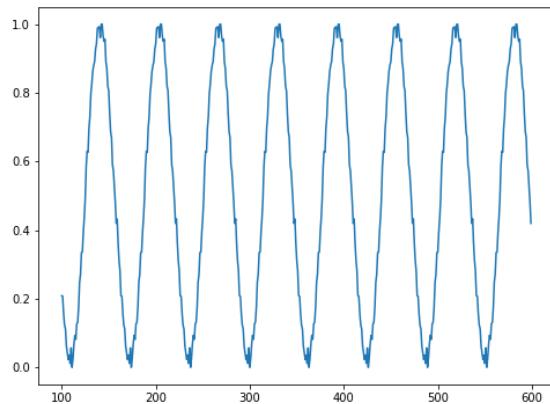


Fuente: elaboración propia

futuros utilizando una red neuronal recurrente en la parte estacionaria de la serie. En un caso con datos reales se podría utilizar un modelo más simple para la parte estacionaria y entrenar una red recurrente para intentar detectar patrones en la componente irregular, pero en el caso de la serie de la figura 13.1 sabemos que, por construcción, sufriríamos de sobreajuste (estaríamos entrenando un modelo con un conjunto de datos formado completamente por ruido).

El primer paso del proceso consiste en entrenar una red neuronal recurrente que capture la estacionalidad de los datos. Para ello, es necesario normalizar los valores de la parte estacional de la serie, como podemos ver en la figura 13.2.

**Figura 13.2.** Parte estacional de la serie normalizada

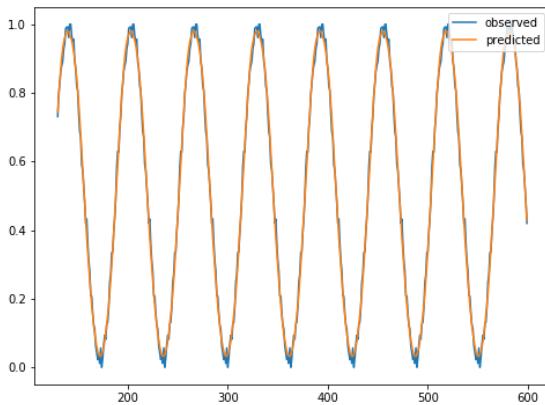


Fuente: elaboración propia

Para predecir la parte estacional de la serie podemos utilizar una red LSTM con 50 unidades en cada celda y que utilice los 30 últimos valores de la serie para predecir el siguiente. Con una serie tan periódica y una red neuronal recurrente se pueden obtener resultados muy buenos, como se muestra en la figura 13.3.

Para predecir la tendencia podríamos utilizar también una red neuronal, pero es mucho mejor utilizar un modelo más simple para evitar el sobreajuste. En este caso, dado que la tendencia es prácticamente lineal, utilizaremos un modelo

**Figura 13.3.** Predicción de los datos de entrenamiento de la parte estacional de la serie normalizada



Fuente: elaboración propia

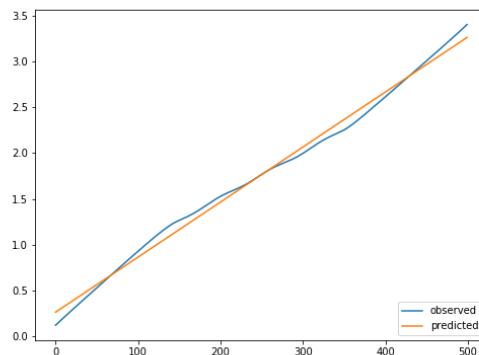
lineal de regresión simple. En la figura 13.4 se puede ver el resultado de ajustar la tendencia de los datos de entrenamiento a un modelo lineal.

Si sumamos ahora las predicciones del modelo lineal con las predicciones de la red neuronal recurrente obtenemos una predicción para los datos de entrenamiento, que se puede ver en la figura 13.5.

Dado que la red parece haber captado la tendencia y la estacionalidad de los datos podemos utilizarla para la predicción de datos futuros. En la figura 13.6 podemos ver los siguientes valores de la serie.

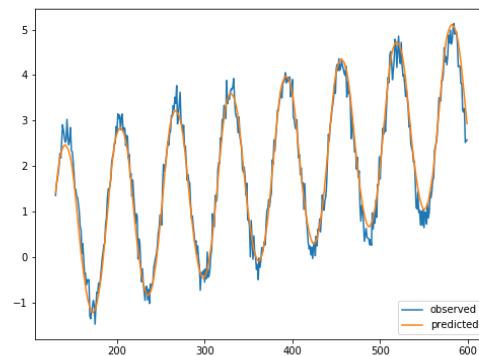
Si aplicamos los modelos ya entrenados para predecir los nuevos datos obtenemos los resultados que muestra la figura 13.7.

**Figura 13.4.** Predicción de la tendencia de los datos de entrenamiento utilizando un modelo lineal



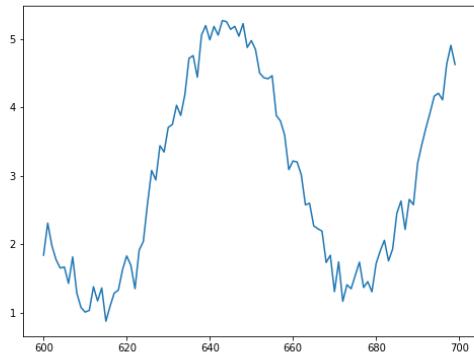
Fuente: elaboración propia

**Figura 13.5.** Suma de las predicciones del modelo lineal y la red neuronal recurrente para los datos de entrenamiento



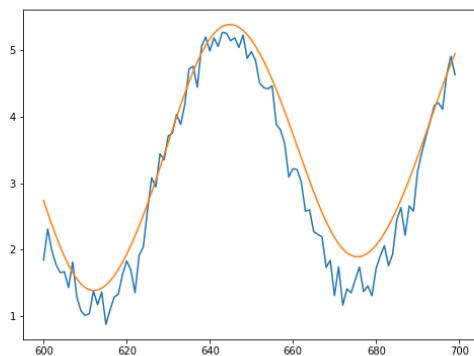
Fuente: elaboración propia

**Figura 13.6.** Valores de la serie posteriores a los utilizados para el entrenamiento



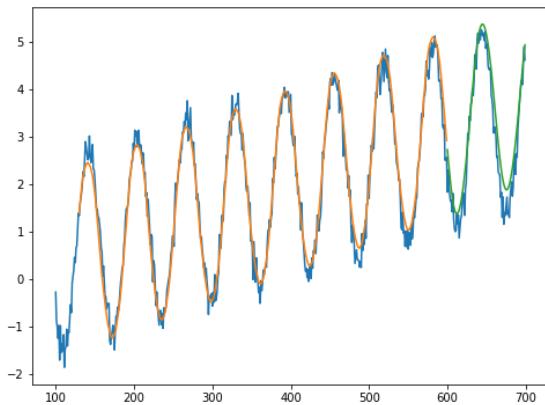
Fuente: elaboración propia

**Figura 13.7.** Predicción de los modelos para los datos de test



Fuente: elaboración propia

**Figura 13.8.** Resumen de predicciones para la serie temporal (datos de entrenamiento y de test)



Fuente: elaboración propia

Por último, podemos ver todos los datos y predicciones de forma conjunta en la figura 13.8.

En este caso en particular los datos de la serie temporal han sido generados sintéticamente, por lo que sabemos seguro que la parte residual de la descomposición está formada principalmente de valores aleatorios. Esto es así en general, por lo que, en principio, no se debe entrenar ningún modelo sobre la parte residual, ya que sino, por definición, sufriríamos de sobreajuste.

Hay casos en cambio, en los que es posible que haya algún patrón escondido en la parte residual de la descomposición, especialmente si los valores de la serie temporal se ven afectados por variables externas que no se están considerando a la hora de hacer la descomposición. En estos casos puede ser

interesante entrenar una red neuronal con todas las variables que se consideren convenientes para intentar extraer algo de información de los residuos. Por ejemplo, si la serie temporal representa el consumo de energía de un pueblo turístico, es posible que en la descomposición se detecten los fines de semana o los meses de verano, pero hay festividades como la Semana Santa que cambian de fecha cada año. En este caso, un modelo al que se indique qué días serán festivos y cuales no es posible que consiga predecir con sentido parte de los residuos sin caer en el sobreajuste.

# **Parte V**

## **Apéndices**



# Apéndice A

## Notación

Generalmente, en los métodos supervisados partiremos de un conjunto de datos correctamente etiquetados,  $D$ , en el que distinguimos la siguiente estructura:

$$D_{n,m} = \begin{pmatrix} d_1 = & a_{1,1} & a_{1,2} & \cdots & a_{1,m} & c_1 \\ d_2 = & a_{2,1} & a_{2,2} & \cdots & a_{2,m} & c_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ d_n = & a_{n,1} & a_{n,2} & \cdots & a_{n,m} & c_n \end{pmatrix},$$

donde:

- $n$  es número de elementos o instancias,
- $m$  el número de atributos del conjunto de datos o también su dimensionalidad,
- $d_i$  indica cada una de las instancias del juego de datos,
- $a_i$  indica cada uno de los atributos descriptivos y
- $c_i$  indica el atributo objetivo o clase a predecir para cada elemento.

En el caso de los métodos no supervisados, el conjunto de datos sigue el mismo patrón y notación, excepto para el atributo de clase,  $c_i$ , que no existe en el caso de los conjuntos no etiquetados.

La tabla 1.1 muestra un resumen de la notación empleada en los distintos capítulos de este libro.

**Tabla 1.1.** Resumen de la notación empleada en el libro

Símbolo	Descripción
$D_{n,m}$	Conjunto de datos
$n$	Número de instancias
$m$	Número de atributos
$k$	Número de clases
$d_i$	Instancia $i$ del conjunto de datos $D$
$a_i$	Conjunto del atributo $i$ en $D$
$c_i$	Clase de la instancia $d_i$
$p_i(t)$	Proporción de elementos de la clase $i$ en la hoja $t$

Fuente: elaboración propia

# Apéndice B

## Detalles del *backpropagation*

Algoritmo de propagación hacia atrás para el cálculo del gradiente de la función de coste en una red neuronal completamente conectada

### 2.1. Notación

Supongamos que tenemos una red neuronal de  $L$  capas completamente conectadas. Durante este texto utilizaremos 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  $w_{jk}^{[l]}$ , 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  $b_j^{[l]}$ .
- $z^{[l]} \in \mathcal{M}_{n_l \times 1}(\mathbb{R})$  denota la combinación lineal de la entrada a la capa  $l$  con los parámetros  $W^{[l]}$  y  $b^{[l]}$ .
- $g : \mathbb{R} \rightarrow \mathbb{R}$  es una función no lineal (como relu o sigmoid). 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  $\odot$ .

## 2.2. Caso particular con un único ejemplo

### 2.2.1. 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:

$$z_j^{[l]} = \sum_{k=1}^{n_{l-1}} w_{jk}^{[l]} \cdot a_k^{[l-1]} + b_j^{[l]}, \quad (2.1)$$

$$a_j^{[l]} = g(z_j^{[l]}). \quad (2.2)$$

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

$$z^{[l]} = W^{[l]} * a^{[l-1]} + b^{[l]}, \quad (2.3)$$

$$a^{[l]} = g(z^{[l]}). \quad (2.4)$$

Una vez hemos aplicado las fórmulas anteriores a todas las capas obtenemos la salida de la red,  $a^{[L]}$ . Para saber si la salida de la red es adecuada podemos definir una función que mida qué error 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:

$$\mathcal{L}(y, a^{[L]}) = -(y \cdot \log(a^{[L]}) + (1 - y) \cdot \log(1 - a^{[L]})). \quad (2.5)$$

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.

### 2.2.2. Propagación hacia atrás

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

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:

$$\begin{array}{ll} f : \mathbb{R} \rightarrow \mathbb{R}, & g : \mathbb{R} \rightarrow \mathbb{R}, \\ x \mapsto f(x), & y \mapsto g(y). \end{array}$$

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

$$\frac{\partial(g \circ f)}{\partial x} = \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial x}. \quad (2.6)$$

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

$$\begin{aligned} f &: \mathbb{R} \rightarrow \mathbb{R}^d, \\ x &\mapsto (f_1(x), \dots, f_d(x)). \end{aligned}$$

$$\begin{aligned} g &: \mathbb{R}^d \rightarrow \mathbb{R}, \\ (y_1, \dots, y_d) &\mapsto g(y_1, \dots, y_d). \end{aligned}$$

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:

$$\frac{\partial(g \circ f)}{\partial x} = \sum_{c=1}^d \frac{\partial g}{\partial f_c} \cdot \frac{\partial f_c}{\partial x}. \quad (2.7)$$

Utilizando la regla de la cadena podemos calcular entonces 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:

$$\frac{\partial \mathcal{L}}{\partial a^{[L]}} = - \left( \frac{y}{a^{[L]}} - \frac{1-y}{1-a^{[L]}} \right). \quad (2.8)$$

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:

$$\begin{aligned} \mathbb{R} &\rightarrow \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R}, \\ w_{jk}^{[l]} &\mapsto z_j^{[l]} \mapsto a_j^{[l]} \mapsto \mathcal{L}. \end{aligned}$$

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

$$\frac{\partial \mathcal{L}}{\partial w_{jk}^{[l]}} = \frac{\partial \mathcal{L}}{\partial a_j^{[l]}} \cdot \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \cdot \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}}. \quad (2.9)$$

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 2.1 y 2.2 tenemos:

$$\frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} = g'(z_j^{[l]}), \quad (2.10)$$

$$\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = a_k^{[l-1]}. \quad (2.11)$$

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:

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

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

$$\frac{\partial \mathcal{L}}{\partial b_j^{[l]}} = \frac{\partial \mathcal{L}}{\partial a_j^{[l]}} \cdot \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} \cdot \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}}. \quad (2.12)$$

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:

$$\frac{\partial a_j^{[l]}}{\partial z_j^{[l]}} = g'(z_j^{[l]}), \quad (2.13)$$

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1. \quad (2.14)$$

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

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \left( \frac{\partial \mathcal{L}}{\partial a^{[l]}} \odot g'(z^{[l]}) \right) * (a^{[l-1]})^T. \quad (2.15)$$

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 *bias*:

$$\frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{\partial \mathcal{L}}{\partial a^{[l]}} \odot g'(z^{[l]}) = \frac{\partial \mathcal{L}}{\partial z^{[l]}}. \quad (2.16)$$

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:

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

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

$$\frac{\partial \mathcal{L}}{\partial a_j^{[l]}} = \sum_{c=1}^{n_{l+1}} \frac{\partial \mathcal{L}}{\partial a_c^{[l+1]}} \cdot \frac{\partial a_c^{[l+1]}}{\partial z_c^{[l+1]}} \cdot \frac{\partial z_c^{[l+1]}}{\partial a_j^{[l]}}. \quad (2.17)$$

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

$$\frac{\partial a_c^{[l+1]}}{\partial z_c^{[l+1]}} = g'(z_c^{[l+1]}), \quad (2.18)$$

$$\frac{\partial z_c^{[l+1]}}{\partial a_j^{[l]}} = w_{cj}^{[l+1]}. \quad (2.19)$$

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:

$$\frac{\partial \mathcal{L}}{\partial a^{[l]}} = (W^{[l+1]})^T * \left( \frac{\partial \mathcal{L}}{\partial a^{[l+1]}} \odot g'(z^{[l+1]}) \right). \quad (2.20)$$

Donde

$$(W^{[l+1]})^T \in \mathcal{M}_{n_l \times n_{l+1}}(\mathbb{R}) \text{ y } \frac{\partial \mathcal{L}}{\partial a^{[l+1]}} \odot g'(z^{[l+1]}) \in \mathcal{M}_{n_{l+1} \times 1}(\mathbb{R})$$

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  $l = 1, \dots, L$ , que se han calculado anteriormente durante la propagación hacia delante.

A continuación resumimos los pasos para calcular el gradiante 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  $l = 1, \dots, L$ .

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.

## 2.3. Caso general con varios ejemplos

### 2.3.1. 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:

$$Z^{[l]} = (z^{[l](1)} \quad z^{[l](2)} \quad \dots \quad z^{[l](m)}), \quad (2.21)$$

$$A^{[l]} = (a^{[l](1)} \quad a^{[l](2)} \quad \dots \quad a^{[l](m)}). \quad (2.22)$$

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:

$$Z^{[l]} = W^{[l]} * A^{[l-1]} + B^{[l]}, \quad (2.23)$$

$$A^{[l]} = g(Z^{[l]}). \quad (2.24)$$

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:

$$J = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(y^{(i)}, a^{[L](i)}). \quad (2.25)$$

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.

### 2.3.2. 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 fórmulas 2.15, 2.16 y 2.20 y adecuarlas a las sustituciones  $a^{[l]} \rightarrow A^{[l]}$  y  $z^{[l]} \rightarrow Z^{[l]}$ .

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

$$\frac{\partial J}{\partial W^{[l]}} = \frac{1}{m} \left( \frac{\partial \mathcal{L}}{\partial A^{[l]}} \odot g'(Z^{[l]}) \right) * (A^{[l-1]})^T. \quad (2.26)$$

Donde

$$\frac{\partial \mathcal{L}}{\partial A^{[l]}} , g'(Z^{[l]}) \in \mathcal{M}_{n_l \times m}(\mathbb{R}) \text{ y } (A^{[l-1]})^T \in \mathcal{M}_{m \times n_{l-1}}(\mathbb{R}),$$

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:

$$\frac{\partial J}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^m \frac{\partial \mathcal{L}}{\partial A^{[l](i)}} \odot g'(Z^{[l](i)}) = \frac{1}{m} \sum_{i=1}^m \frac{\partial \mathcal{L}}{\partial Z^{[l](i)}}. \quad (2.27)$$

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 fórmula 2.20 y considerar:

$$\frac{\partial \mathcal{L}}{\partial A^{[l]}} = (W^{[l+1]})^T * \left( \frac{\partial \mathcal{L}}{\partial A^{[l+1]}} \odot g'(Z^{[l+1]}) \right). \quad (2.28)$$

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  $\frac{\partial J}{\partial b^{[l]}}$  para todo  $l = 1, \dots, L$ .

# Bibliografía

**Alain, G.; Bengio, Y.; Yao, L.; Yosinski, J.; Thibodeau-Laufer, E.; Zhang, S.; Vincent, P.** (2016). «GSNs: Generative Stochastic Networks». *Information and Inference*.

**Bahdanau, D.; Cho, K.; Bengio, Y.** (2015). «Neural Machine Translation by Jointly Learning to Align and Translate». *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

**Beale, R.; Jackson, T.** (1990). *Neural Computing: An Introduction*. Bristol: IOP Publishing Ltd.

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

**Bengio, Y.; Lamblin, P.; Popovici, D.; Larochelle, H.** (2006). «Greedy Layer-Wise Training of Deep Networks». *Proceedings of the 19th International Conference on Neural Information Processing Systems, NIPS'06* (págs. 153-160).

**Bengio, Y.; Lamblin, P.; Popovici, D.; Larochelle, H.** (2007). «Greedy Layer-Wise Training of Deep Networks». B. Schölkopf; J. C. Platt; T. Hoffman; (eds.), *Advances in*

*Neural Information Processing Systems*, (n.<sup>o</sup> 19, págs. 153-160).

**Bergstra, J.; Bengio, Y.** (2012). «Random search for hyper-parameter optimization». *J. Mach. Learn. Res.* (vol. 13, n.<sup>o</sup> 1, págs. 281-305).

**Cho, K.; Merriënboer, van B.; Gülcühre, Ç.; Bahdanau, D.; Bougares, F.; Schwenk, H.; Bengio Y.** (2014). «Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation». *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing* (págs. 1724-1734).

**Deng, J.; Dong, W.; Socher, R.; Li, L.-J.; Li, K.; Fei-Fei, L.** (2009). «ImageNet: A Large-Scale Hierarchical Image Database». *Proceedings of The IEEE Computer Society Conference on Computer Vision and Pattern Recognition CVPR09*.

**Duchi, J.; Hazan E.; Singer, Y.** (2011). «Adaptive Sub-gradient Methods for Online Learning and Stochastic Optimization». *J. Mach. Learn. Res.* (n.<sup>o</sup> 12, págs. 2121-2159).

**Gatys, L. A.; Ecker, A. S.; Bethge, M.** (2015). «Image Style Transfer Using Convolutional Neural Networks». *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*» (págs. 2414-2423).

**Glorot, X.; Bordes, A.; Bengio, Y.** (2011). «Deep Sparse Rectifier Neural Networks». *AISTATS*, volume 15 of *JMLR Proceedings* (págs. 315-323).

- Graves, A.; Mohamed, A.; Hinton, E. G.** (2013). «Speech Recognition with Deep Recurrent Neural Networks». *IEEE International Conference on Acoustics, Speech and Signal Processing, Vancouver, BC, Canada, May 26-31, 2013* (págs. 6645-6649).
- Haykin, S. O.** (2009). *Neural Networks and Learning Machines, 3rd Edition*. Pearson.
- Hinton, G. E.; Osindero, S.; Teh, Y.-W.** (2006). «A Fast Learning Algorithm for Deep Belief Nets». *Neural Comput.* (vol. 18, n.<sup>o</sup> 7, págs. 1524-1554).
- Hochreiter, S.; Bengio, Y.; Frasconi, P.** (2001). «Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies». J. Kolen; S. Kremer, (ed.). *Field Guide to Dynamical Recurrent Networks*, IEEE Press.
- Hochreiter, S.; Schmidhuber, J.** (1997). «Long Short-Term Memory». *Neural Computation* (vol. 9, n.<sup>o</sup> 8, págs. 1735-1780).
- Hopfield, J. J.** (1984). «Neurons with Graded Response Have Collective Computational Properties Like Those of Two-State Neurons». *Proceedings of the National Academy of Sciences* (vol. 81, n.<sup>o</sup> 10, págs. 3088-3092).
- Hsu, F.-H.** (2002). *Behind Deep Blue: Building the Computer That Defeated the World Chess Champion*. Princeton: Princeton University Press.
- Huang G.; Liu, Z.; L. v. d. Maaten; Weinberger, K. Q.** (2017). «Densely Connected Convolutional Networks». *2017 IEEE Conference on Computer Vision and Pattern Recognition*.

- Kingma, D. P.; Ba, J.** (2014). «Adam: A Method for Stochastic Optimization». *CoRR*, abs/1412.6980.
- Kingma, D. P.; Welling, M.** (2013). «Auto-Encoding Variational Bayes». *CoRR*, abs/1312.6114.
- Kohavi, R.** (1995). «A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection». *Proceedings of the 14th International Joint Conference on Artificial Intelligence* (págs. 1137-1143).
- Krizhevsky, A.; Sutskever, I.; Hinton, G. E.** (2012). «Imagenet Classification with Deep Convolutional Neural Networks». F. Pereira; C. J. C. Burges; L. Bottou; K. Q. Weinberger, (eds.). *Advances in Neural Information Processing Systems* (n.<sup>o</sup> 25, págs. 1097-1105).
- Krizhevsky, A.; Sutskever, I.; Strivastava, N.; Hinton, G.; Salakhutdinov, R.** (2014). «Learning Sets of Filters Using Back-Propagation». *Journal of Machine Learning Research* (n.<sup>o</sup> 15, págs. 1929-1958).
- LeCun, Y. et al.** (1989). «Backpropagation Applied to Handwritten Zip Code Recognition». *Neural Computation* (vol. 1, n.<sup>o</sup> 2, págs. 541-551).
- LeCun, Y.** (1989). «Generalization and Network Design Strategies». *Technical Report CRG-TR-89-4*.
- LeCun, Y.; Bottou, L.; Haffner, P.** (1998). «Gradient-Based Learning Applied to Document Recognition». *Proceedings of the IEEE* (vol. 86, n.<sup>o</sup> 11, págs. 2248-2324).
- Luan, F.; Paris, S.; Shechtman, E.; Bala, K.** (2017). «Deep Photo Style Transfer». *2017 IEEE Conference on*

- Computer Vision and Pattern Recognition (CVPR)* (págs. 6997-7005).
- Masci, J.; Meier, U.; Cireşan, D.; Schmidhuber, J.** (2011). «Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction». *Proceedings of the 21th International Conference on Artificial Neural Networks* (págs. 52-59).
- McCulloch, W. S.; Pitts, W.** (1988). «A Logical Calculus of the Ideas Immanent in Nervous Activity». J. A. Anderson; E. Rosenfeld, editors, *Neurocomputing: Foundations of Research* (págs. 15-27).
- Mezard, M.; Nadal, J.-P.** (1989). «Learning in Feedforward Layered Networks: The Tiling Algorithm». *Journal of Physics A: Mathematical and General* (vol. 22, n.º 12, págs. 2191).
- Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.; Dean, J.** (2013). «Distributed Representations of Words and Phrases and Their Compositionality». *Proceedings of the 26th International Conference on Neural Information Processing Systems* (págs. 3111-3119).
- Minsky, M.; Papert, S.** (1969). *Perceptrons: An Introduction to Computational Geometry*. Cambridge: MIT Press.
- Nair, V.; Hinton, G. E.** (2010). «Rectified Linear Units Improve Restricted Boltzmann Machines». *Proceedings of the 27 th International Conference on Machine Learning*.
- Nielsen, M.** (2019). «Neural Networks and Deep Learning». [artículo en línea]. Fecha de consulta 13 de febrero de 2019.

- Plaut, D. C.; Hinton, G. E.** (1987). «Learning Sets of Filters Using Back-Propagation». *Computer Speech & Language* (vol. 2, n.<sup>o</sup> 1, págs. 35-61).
- Qian, N.** (1999). «On the Momentum Term in Gradient Descent Learning Algorithms». *Neural Networks* (vol. 12, n.<sup>o</sup> 1, págs. 145-151).
- Ranzato, M. A.; Poultney, C.; Chopra, S.; LeCun, Y.** (2006). «Efficient Learning of Sparse Representations with an Energy-Based Model». *Proceedings of the 19th International Conference on Neural Information Processing Systems* (págs. 1137-1144).
- Rifai, S.; Vincent, P.; Muller, X.; Glorot, X.; Bengio, Y.** (2011). «Contractive Auto-Encoders: Explicit Invariance During Feature Extraction». *Proceedings of the 28th International Conference on International Conference on Machine Learning* (págs. 833-840).
- Rosenblattd, F.** (1962). *Principles of Neurodynamics; Perceptrons and the Theory of Brain Mechanisms*. Spartan Books.
- Ruder, S.** (2016). «An Overview of Gradient Descent Optimization Algorithms». *CoRR*, abs/1609.04747.
- Rumelhart, David E.; McClelland, James L.** (1986). *Parallel Distributed Processing*. MIT press.
- Schroff, F., Kalenichenko, D.; J. P.** (2015). «Facenet: A Unified Embedding for Face Recognition and Clustering». *IEEE Conference on Computer Vision and Pattern Recognition* (págs. 815-823).

- Schuster, M.; Paliwal, K.** (1997). «Bidirectional Recurrent Neural Networks». *Trans. Sig. Proc.* (vol. 45, n.<sup>o</sup> 11, págs. 2673-2681).
- Simonyan, K.; Zisserman, A.** (2014). «Very Deep Convolutional Networks for Large-Scale Image Recognition». *CoRR*, abs/1409.1556.
- Snoek, J.; Larochelle, H.; Adams, R. P.** (2012). «Practical Bayesian Optimization of Machine Learning Algorithms». *Proceedings of the 25th International Conference on Neural Information Processing Systems* (págs. 2951-2959).
- Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R.** (2014). «Dropout: A Simple Way to Prevent Neural Networks From Overfitting». *Journal of Machine Learning Research* (n.<sup>o</sup> 15, págs. 1929-1958).
- Sutskever, I.; Vinyals, O.; Quoc, V. Le** (2014). «Sequence to Sequence Learning with Neural Networks». *Proceedings of the 27th International Conference on Neural Information Processing Systems* (págs. 3104-3112).
- Szegedy, C.; Ioffe, S.; Vanhoucke, V.** (2016). «Inception-V4, Inception-Resnet and the Impact of Residual Connections on Learning». *CoRR*, abs/1602.07261.
- Szegedy, C.; Iofee, S.; Vanhoucke, V.; Alemi A.** (2017). «Inception-V4, Inception-Resnet and the Impact of Residual Connections on Learning». *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (págs. 4271-4284).

**Taigman, Y.; Yang, M.; Ranzato, M. A.; Wolf, L.** (2014). «Deepface: Closing the Gap to Human-Level Performance in Face Verification». *IEEE Conference on Computer Vision and Pattern Recognition* (págs. 1701-1708).

**Vincent, P.; Larochelle, H.; Bengio, Y.; Manzagol, P.-A.** (2008). «Extracting and Composing Robust Features with Denoising Autoencoders». *Proceedings of the 25th International Conference on Machine Learning* (págs. 1096-1103).

**Vincent, P.; Larochelle, H.; Lajoie, I.; Bengio, Y.; Manzagol, P.-A.** (2010). «Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion». *J. Mach. Learn. Res.* (n.º 11, págs. 3371-3408).

**Wilson, A. C.; Roelofs, R.; Stern, M.; Srebro, N.; Recht, B.** (2017). «The Marginal Value of Adaptive Gradient Methods in Machine Learning». I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30* (págs. 4148-4158).

**Zhang, X.; He K.; Ren, S.** (2016). «Deep Residual Learning for Image Recognition». *2016 IEEE Conference on Computer Vision and Pattern Recognition* (págs. 2951-2959).

**Zeiler, M. D.** (2012). «ADADELTA: An Adaptive Learning Rate Method». *CoRR*, abs/1212.5701, 2012.

**Zeiler, M. D.; Fergus, R.** (2014). «Visualizing and Understanding Convolutional Networks». En: Fleet D.; Pajdla T.;

Schiele B.; Tuytelaars T. (eds.). *Computer Vision – ECCV 2014. ECCV 2014». Lecture Notes in Computer Science, vol. 8689*. Cham: Springer.

**Zhu, J.-Y.; Park, T.; Isola, P.; Efros, A. A.** (2017). «Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks». *IEEE International Conference on Computer Vision, Venice, Italy, October 22-29, 2017* (págs. 2242-2251).





En este libro se introducen los conceptos fundamentales del aprendizaje profundo (*deep learning*) mediante el uso de redes neuronales artificiales (*artificial neural networks*, ANN). El lector podrá encontrar una revisión completa de las técnicas avanzadas más usadas en estos campos. El enfoque del libro es claramente descriptivo, con el objetivo de que el lector entienda los conceptos e ideas básicos detrás de cada algoritmo o técnica.

La primera parte del libro constituye una introducción al aprendizaje profundo, en general, y a las redes neuronales, en particular. En la segunda parte se describe el funcionamiento de las redes neuronales, partiendo de conceptos básicos (como la estructura de una neurona, las principales funciones de activación, etc.) hasta alcanzar conceptos avanzados (optimización del rendimiento de las redes neuronales o estrategias para evitar el problema del sobreentrenamiento). La tercera parte presenta los fundamentos teóricos, estructura y principales arquitecturas de las redes neuronales convolucionales (*convolutional neural networks*, CNN) y su aplicación en el procesamiento de imágenes. Finalmente, el cuarto bloque de este texto se centra los fundamentos teóricos, estructura y principales arquitecturas de las redes neuronales recurrentes (*recurrent neural networks*, RNN) y sus aplicaciones para el procesamiento de series temporales y textos.

Con este libro aprenderás sobre:

- 
- ✓ minería de datos; ✓ aprendizaje automático; ✓ inteligencia artificial; ✓ ciencia de datos; ✓ aprendizaje profundo; ✓ *data mining*; ✓ *machine learning*; ✓ *deep learning*

